Vrije Universiteit Brussel – Belgium
Faculty of Sciences
In Collaboration with Ecole des Mines de Nantes – France

2005

IPSComp: Intelligent Portal for Searching Components

A Thesis submitted in partial fulfillment of the requirements
for the degree of Master of Science in Computer Science
(Thesis research conducted in the EMOOSE exchange)

By: Javier Aguirre

Promoter: Prof. Theo D'Hondt (Vrije Universiteit Brussel)
Co-Promoters: Annya Réquilé (Ecole des Mines de Nantes)

Abstract

The software development industry is seeing as an inaccurate science, always dealing with low product quality and over run in time, cost and effort factors. Component Based Software Development (CBSD) has emerged as an approach aiming to improve a number of drawbacks found in the software development industry. The main idea is the reuse of well-tested software elements that will be assembly together in order to develop larger systems. This approach will bring as a consequence the reduction in time development, a more stable final product and a reduction in time, effort, cost and testing process. Despite the goal CBSD is aiming, software industry has not been able to accomplish such objectives.

The purpose of this research is to present means to provide users with the appropriate tools to describe components, in order to be able to share them among a wide consumer's community. On top of such component description, tools to discover and search for components can be implemented. The set of users in this case can be identified as different actors, such as software developers, software architects, software producers and systems integrators, among others.

In this context, my research focuses on two main parts: the first one is related to component ontology, and the second on code and design transformation for the integration of software component repositories.

Acknowledgements

I would like to thank my advisor Professor Annya Réquilé for her advice and guidance in the past five months. I would also like to acknowledge the assistance of Professor Mourad Oussalah and Gustavo Bobeff for their valuable comments during the research. A special word of thanks is due to all the professors who were responsible for such a good academic experience.

Special thanks to my EMOOSE mates: Richa, Jorge, Daniel and Harmin for their friendship and collaboration. A note of thanks to Sylvie Poizac who was always there for us.

Thanks to my parents and family for their love and support.

Contents

1 Introduction 1
2 State of the Art 3
2.1 Ontology Manager Systems 4
2.1.1 ONTOMANAGER 4
2.1.2 SymOntoX 5
2.1.3 Building Domain Ontology Based on Web Data and Generic Ontology 6
2.1.4 PLIB 7
2.2 Component Retrieval Schemes 8
2.2.1 Keywords Technique 11
2.2.1.1 INSEAS – Keyword – Faceted – Browsing 11
2.2.2 Faceted Technique 12
2.2.2.1 InterLegis Project based on Odyssey Search Engine - Faceted 12
2.2.2.2 ADIPS Framework - Faceted - Browsing 13
2.2.3 Signature Matching Technique 14
2.2.3.1 AGORA - Signature Matching 14
2.2.3.2 COMPONENTEXCHANGE - Signature Matching 15
2.2.4 Behavioral Matching Technique 16
2.2.4.1 Behavior sampling - Behavioral Matching 16
2.2.5 Semantic-Based Technique 17
2.2.5.1 Towards a Semantic-based Approach for Software Reusable Component Classification and Retrieval - Semantic-based 17
2.2.5.2 A Semantic-Based Approach to Component Retrieval - Semantic-based 18
2.2.6 Browsing Technique 19
2.2.6.1 CompoNex – Browsing 19
2.2.7 Users Web Mining Technique 20
2.2.7.1 RASCAL - Users Web Mining 20
2.3 Model Driven Architecture - MDA 20
3 The eCots Association 23
4 Contribution 25
4.1 Component Ontology for IPSComp Specification 25
4.1.1 XCM Component Ontology 26
4.1.2 Component Ontology for IPSComp Specification 27
4.1.2.1 Domain 27
4.1.2.2 Price 28
4.1.2.3 Quality Attributes 28
4.1.2.4 License 31
4.1.2.5 Publisher Description 31
4.1.2.6 Specialization Scenarios 31
4.2 Component Ontology for IPSComp Design 33
4.3 Component Ontology for IPSComp Implementation 37
4.3.1 Java Code Implementation 37
4.3.2 IPSComp Java Code Generation from a XML file 39
4.3.3 IPSComp Ontology Implementation PLIB 40
4.4 Integrating Software Component Repositories 42
5 Conclusions 63
6 Future Work 67
Appendix A - IPSComp Ontology UML Class Diagram 69
Appendix B - IPSComp Ontology component Package UML Class Diagram 71
Appendix C - IPSComp Ontology qualityAttribute Package UML Class Diagram 73
Appendix D - IPSComp Ontology metric Package UML Class Diagram 75

Appendix E - IPSComp Ontology xmlParser Package UML Class Diagram 77
Appendix F - IPSComp XML Meta-Model - XSD Schema 79
Appendix G - IPSComp Component Description – XML Example..... 87
References 93

List of Figures

Figure 3-1 IPSComp System Architecture [48] 24
Figure 4-1 Contribution to the IPSComp Project 25
Figure 4-2 XCM Hierarchical Component Structure [51] 27
Figure 4-3 IPSComp Hierarchical Component Structure 32
Figure 4-4 IPSComp Component Ontology UML Class 34
Figure 4-5 IPSComp Metric Concept UML Class Diagram 35
Figure 4-6 IPSComp Quality Attribute Concept UML Class Diagram 36
Figure 4-7 Visitor Pattern with Reflection to load a XML file – UML Class Diagram 40
Figure 4-8 PLIB Editor IPSComp Ontology – Screen Shot 42
Figure 4-9 Elements for Software Component Repository Integration 44
Figure 4-10 Elements Examples for Software Component Repository Integration 45
Figure 4-11 Vendor Respository {2} – Vendor Component Description {4} Example 46
Figure 4-12 http://www.componentsource.com *Vendor Component Description Meta-Model* 48
Figure 4-13 http://devcatalog.com *Vendor Component Description Meta-Model* 49
Figure 4-14 http://www.ecots.org *Vendor Component Description Meta-Model* 50
Figure 4-15 *Essence Component Description Meta-Model* 51
Figure 4-16 Component Repositories Domain Layers 55
Figure 4-17 Obtaining a Component Description Image in the IPSComp Ontology 56
Figure 4-18 Structure of an Enterprise Bean JAR [67] 57
Figure 5-1 IPSComp System Architecture [48] 65
Figure 5-2 IPSComp System Architecture Analyzed 66

List of Tables

Table 2-1 Summary of Component Retrieval Schemes 10
Table 2-2 InterLegis Component Description [39] 13
Table 4-1 Quality Model for COTS components [52] 29
Table 4-2 Implementation of the `IValue` interface 38
Table 4-3 Abstract Class Metric 38
Table 4-4 Concrete Metric Class 38
Table 4-5 Abstract Class `QualityAttribute` 39
Table 4-6 Concrete Quality Attribute Class 39
Table 4-7 Collections in the Component Ontology 41
Table 4-8 IPSComp Java Implementation Example Item {7} 45
Table 4-9 IPSComp XML Meta-Model - XSD Schema Example {12} 46
Table 4-10 IPSComp Component Description – XML Example {13} 47
Table 4-11 Example Scenario Model Transformation Using the *IPSComp Transformation API* - Steps 1, 2 and 3. 54
Table 4-12 Example Scenario Model Transformation Using the *IPSComp Transformation API* – Step 4. 54

1 Introduction

The extensive use of Internet has brought as a consequence an overload in the web content. Internet has become a huge storage device on which we can find any sort of information. But because of the amount of resources being published on the World Wide Web, users have been experiencing a lack of accuracy when searching for specific topics. Researchers and industry have identified this drawback, and are moving forward to solve it.

Two approaches addressed to help users to find relevant items have emerged. The first one asks the user to explicitly register information of what he likes and what he dislikes. The recommender system then suggests items that are similar to the one the user likes and rejects those that are alike from the ones the user does not like. This is called content-dependent approach, because the system must rate what a similar item is. The second approach classifies the users in profiles; in such profiles are included users with similar characteristics (age, sex, tastes, educational background, etc). For users within the same profile there is a high probability they will like and dislike the same things, so recommendations are made based on such profiles, it is called collaborative filtering. Additionally, the content-based and collaborative filtering approaches have been combined to obtain better results. These approaches and the combination of them work relatively well to find or suggest users about simple matters such as movies, books, shopping, etc. As a matter of fact there have been lots of suggested algorithms in order to redefine the pages ranking and to gather the user preferences.

The amount of information is not the only problem faced at the moment, but also the wide variety and complexity of content. In other words on World Wide Web it is possible to find a wide variety of items starting from static content up to complete applications. It is necessary to come up with a solution that will allow classifying any source of information or content, in order to be able to provide users with means to take advantage of the published resources. The aim of this study is to offer a complete software component description, as well as the base to implement a retrieval tool for such components. Furthermore, it is not possible to leave aside the fact that there are already developed software components as well as repositories which classify them; it is necessary to find means to integrate the elements found in those existing repositories.

Additionally this research is included as a first step in the definition of the functional architecture of a larger project. The Intelligent Portal for Searching Components, called IPSComp project, aims at developing an open information portal for commercial off-the-shelf (COTS) components (software and non software components), in which we deal with information about products, and possibly between their users, or between users and producers. The IPSComp project is in the specification phase.

The idea of building software systems from pre-fabricated software components that could be exchanged in software component markets has been around at least since 1968 expressed by McIlroy in [72]. His basic proposal is to be able to combine components from different vendors to create applications. The composition is made up from plug-and-play-like reuse of black box components, which enables software component markets.

It is important to remark that at this moment there are several definitions of what a software component is, there is still no general consensus about precisely what constitutes a software component. As a matter of fact it is worth it to provide a software component definition that allows enclosing the concepts that will be handled in the present job. As stated in [17]: "A component consists of different (software) artefacts. It is reusable, self-contained and marketable, provides services through well-defined interfaces, hides its implementation and can be deployed in configurations unknown at the time of development. A business component is a component that implements a certain set of services out of a given business domain. In order to be operable, components need a basic infrastructure, e.g. Enterprise Java Beans (EJB) or .NET".

This definition is according to the objective of the IPSComp project, which is intended to become a

component's market place. The fact that a component is reusable and self-contained shows that the component is independent; it can be used without other components present. But it is also feasible to combine with other components or into a system by the set of well defined interfaces. This component definition supports the black box approach, which is necessary in the IPSComp project scope where the software components will be marketable and used by customers in different implementations at later times. The business components definition fits into the IPSComp project items to be handle by the system, as well as the platforms there named.

To achieve the use components based on the given definition, it is necessary to standardize the component description. The interface and behavior of a component has to be described in a precise way. Specification becomes a key point in the composition of business components, since the specification might be the only source of information available for a composer who combines business components from different vendors to an application system.

With the improvements in software components development, a set of platforms have emerged, J2EE .NET, CORBA. All these platforms have means to connect components based on a syntactic description, which is determined by the interfaces each component provides or requires. But it is possible to observe a lack of semantic as well as behavioral description, which should provide the component's characteristics that will encourage and facilitates its reuse. As a consequence, one of the problems Component Based Software Development is facing regarding reuse of components is that finding the right component is a complex task. Despite this fact the software component development has not decreased.

As a matter of fact we can not even state that this is a new issue because in the literature we can find papers dated from 1995 [3] which are proposing methods for COTS selection. In [3] the authors identify as main issues in COTS selection the following points:

- Lack of a defined, systematic, and repeatable process in the COTS selection.
- There can be a potential disregard of the application requirements.
- There is a misuse of data consolidation methods in decision making for the COTS selection.

The hard-to-identify mismatches are largely due to the fact that the capability of the components are not clearly described or understood through their interfaces. Most commercially available software components are delivered in binary form. It is necessary to rely on the components' interface description to understand their exact capability. Even with the components' development documentation available, people would certainly prefer or can only afford to explore their interface descriptions rather than digesting their development details.

Despite the fact that this is an old issue, nowadays it is still an open topic of research. It has not been possible to overcome this issue because it deals with a vast number of possible candidates quite a lot of already developed software components and also with a set of unstructured information trying to describe them, which is in addition difficult to analyze. As such there is not a clear perception of what a Software Component is able to provide.

It is necessary to overcome those issues in order to take advantage of the component-based software development objectives, create a marketable place for software components and be able to incorporate already developed components.

In chapter 2 the State of the Art is presented, in this chapter is possible to find information regarding some techniques used to handle component's description, retrieval and model transformation. In chapter 3 a brief introduction to the IPSComp project initiated by the eCots association is presented. Chapter 4 describes the contribution given to the domain under research as well as the prototype developed. The complete code is found in the annexed document "IPSComp: Intelligeng Portal for Searching Components Prototype Source Code". Chapter 5 states the conclusions, to close in Chapter 6 with the future work.

2 State of the Art

To overcome the issues present in the component selection process the authors in [3] propose the OTSO (Of The Shelf Option) method which supports the search, evaluation and selection of reusable software and provides specific techniques for defining the evaluation criteria, comparing costs and benefits of the different alternatives [10]. The method states that in order to evaluate software components it is necessary to define the evaluation criteria which can be categorized in four areas: Functional requirements, product quality characteristics, strategic concerns, and domain architecture compatibility. The aspects to be evaluated on each category can be defined by the (Goal-Question-Metric model) GQM which will also provide a well-defined template for documenting the evaluation goals. The objective pursue is the decomposition criteria into a set of concrete, measurable, observable or testable evaluation criteria. Then the method relies on the use of the Analytic Hierarchy Process (AHP) for consolidating the evaluation data for decision-making purposes. As stated by the authors each COTS selection process differs from each other, and they are providing a method for considering a set of elements to be taken into account when selecting components. But it is up to the evaluation team to define the goals and to find out the components that will fulfill the requirements pursued. As such they are not classifying components, but at least they are giving a group of parameters to consider. By formalizing this criteria definition process, it is possible to reuse the OTS software selection experiences better, leading to a more efficient and reliable selection process.

We can infer from the OTSO method that we need a complete component specification in order to be able to perform a deep analysis that will drive us to software reuse. Several researches are being performed in the field of components semantic description, and they are based on the utilization of an ontology.

As stated by T. Gruber "An ontology is an explicit specification of a conceptualization" [49]. What is important is what an ontology is for. Gruber's team has been designing ontologies for the purpose of enabling knowledge sharing and reuse. In that context, an ontology is a specification used for making ontological commitments. An ontological commitment is an agreement to use a vocabulary (i.e., ask queries and make assertions) in a way that is consistent (but not complete) with respect to the theory specified by an ontology. Gruber's team builds agents that commit to ontologies. They design ontologies so they can share knowledge with and among these agents.

A number of other knowledge representations (taxonomies, thesauri and controlled vocabulary) are often described as being ontologies. While this is strictly true according the broadest definition of an ontology, the scope and power of ontologies is fully realized when they express a richer set of relationships between concepts. Those terms will briefly describe:[1]

- Controlled Vocabulary. Is the list of a set of terms that in order to be included in the vocabulary must be approved by the vocabulary authority. Each term must be unambiguously defined. A controlled vocabulary follows two rules: (i) If the same term is commonly used to mean different concepts in different contexts, then its name is explicitly qualified to resolve this ambiguity. (ii) If multiple terms are used to mean the same thing, one of the terms is identified as the preferred term in the controlled vocabulary and the other terms are listed as synonyms or aliases.
- Taxonomy. It is a collection of controlled vocabulary organized in a hierarchical structure. There may be different type of relationships in the taxonomy (e.g., whole-part, genus-species, type-instance, etc). Some taxonomies allow poly-hierarchy, in such a case a term can have multiple parents. This means that if a term appears in multiple places in a taxonomy, then it is the same term.
- Thesaurus. It is a networked collection of controlled vocabulary terms. This means that a thesaurus uses associative relationships in addition to parent-child relationships. The expressiveness of the

[1] Based on the article "What is the difference between a vocabulary, a taxonomy, a thesaurus, an ontology, and a meta-model?" written by Johannes Ernst, NetMesh Inc CEO. (http://www.metamodel.com/article.php?story=20030115211223271).

associative relationships in a thesaurus varies and can be as simple as “related to term” as in term A is related to term B.

- Ontology. It is a controlled vocabulary expressed in an ontology representation language. The language has a grammar for using vocabulary terms to express something meaningful within a specified domain of interest. The grammar contains formal constraints (e.g., specifies what it means to be a well-formed statement, assertion, query, etc.) on how a term in the ontology’s controlled vocabulary can be used together. On the other hand the ontology allows the definition of richer and more descriptive relationships between concepts (e.g. is-targeted-for, is-regulated-by, affects, is-expressed-in, etc).
- Meta-model. It is an explicit model of the constructs and rules needed to build specific models within a domain of interest. It can be seen from three perspectives: (i) as a set of building blocks and rules used to build models. (ii) As a model of a domain of interest. (iii) As an instance of another model. A valid meta-model is an ontology, but not all ontologies are modeled explicitly as meta-models. The perspective (ii) allows comparing Meta-models to ontologies.

After taking a look to the previous definition, it is important to point out that it is possible to define relationships between the different concepts. As a matter of fact a Controlled vocabulary can be part of an Ontology, a Taxonomy can be part-of an Ontology, and a Thesaurus can be a part of a Ontology. Finally, if you create an ontology, which is a set of terms naming concepts (classes) and relations, and you use that vocabulary to create a set of data (instances of the classes, and assertions that the instances are related to each other according to the specific relations in the vocabulary), and you think of the set of data you create as the model of your domain, then the ontology is the meta-model and the set of data created is the model. Consequently to enclose this set of definitions, this can be classified as a taxonomy.

2.1 Ontology Manager Systems

The process to create an ontology is not a simple one. It has to go throughout a set of evolution and refinement, and it has to be performed by a specialist in the target field. Even if we are able to define a quite good ontology, this approach must be able to evolve with the domain evolution, and to adapt to new requirements, specifications and context. As a matter of fact some effort has been addressed in order to develop Ontology Management Systems. Among these approaches we can find PLIB [30], ONTOMANAGER [12], ONTOSHARE [14]. Those will be briefly described to get an idea of what they provide. It is important to point out that certain systems will provide a dual function, on the one hand it will handle the ontology, and on the other it will provide means to apply such ontology on a particular set of items.

2.1.1 ONTOMANAGER

The evolution of the ontology will be according to the users’ needs. In order to gather the user needs a special log is being recorded in the web servers. Each user’s activity or event is recorded, specific information such as the type of interaction (query, browse, read, etcetera), date, time, user identity. The dependency between events is represented using the “previous event” relation. This analysis is applicable in ontology-based information portals, in which the ontology supports the process of indexing content of an information resource.

OntoManager consists of three components: the Data Integration Module, the Visualization Module and the Analysis Module [12].

The Data Integration Module performs three main tasks.

- Collect Data from different servers if it is a distributed system and creates a central ontology log, because all distributed logs are supported by the same domain ontology there will not be

heterogeneity problem.

- Pre – Process Data by terms of: (i) Data Abstraction because the log will record words but not concepts, for example if any user is looking for more details regarding the “ABC” project in the log we will find “ABC”, but not the ontology concept project. So the “ABC” occurrences must be replaced with the project ontology concept. (ii) Extracting Links: It is important to find out the frequency of browsing relations between to concepts by analyzing successive events (previous event information recorded in the log). Because of the amount of information that can be store the log is being transferred into OLAP cubes, which enables the analysis of the information.
- Organize logs in a way that enables a fast and efficient access.

The Visualization Module combines the integrated ontology usage data with the ontology itself. It enables presentation of the same information in different ways: Graph-based representation of the ontology where the nodes represent the concepts in the ontology and links correspond to the direct hierarchy; table-based representation; bar-based representation (Pareto diagram). For instance the graph representation allows performing queries on the OLAP cubes, to see the usage of the ontology concepts, among other things.

The Analysis Module makes suggestions to the ontology manager on how to improve the ontology. This improvement should be guided by the users’ needs. The module performs two tasks:

- Ontology Evolution which supports the process of modifying and updated the ontology. Besides it allows undoing changes made to the ontology.
- When adding new concepts to an already existing ontology, some instances of it are not classified. OntoCrawler provides semantic capabilities for identifying and extracting new instances whereby existing knowledge about concepts, relations and instances can be used as background knowledge for the crawling process.

2.1.2 SymOntoX

SymOntoX (Symbolic Ontology XML-based management system) aims to provide identification of business concepts. As a matter of fact it offers some native modeling options called meta-concepts, such as Business Process, Business Object and Business Actor, which help enterprise experts to better categorize and identify concepts. The ontology model in SymOntoX is referred as OPAL (Object, Process, and Actor modelling Language) [24].

A meta-concept has a double nature (i) it defines a template that is used to model a concept in the ontology; (ii) it partition the ontology in a natural and intuitive way. [24]. The entities of the OPAL are defined as:
Actor_Kind: Models any relevant entity of the domain that is able to activate or perform a process (e.g., Tourist, Travel Agency).
Process_Kind: Models the activity that is performed by an actor to achieve a given goal (e.g., making a reservation).
Object_Kind: Models a passive entity, on which a process operates (e.g., hotel, flight), typically to modify its state.

The system also defines a set of meta-concepts in order to facilitate the process of enriching the ontology. Among these meta-concepts we find:
Goal_kind a desired state of the affairs that an Actor seeks to reach (e.g., Go_vacation);
State_Kind a characteristic pattern of values that instance variables of an entity can assume (e.g., Flight_full);
Rule_Kind an expression that is aimed at restraining the possible values of an instance of a concept (constraint rule) or that allows to derive (production rule) new information (e.g., Ticket purchase 30 days before departure);
Information_Element_Kind atomic attribute (e.g., Flight_number, Nr_of_rooms);
Information_Component_Kind a cluster of attributes pertaining to the information structure of a domain

concept (e.g., Flight_info, Hotel_address);
Action_Kind a process component, i.e., an activity that is further decomposable (e.g., Room_Requesting);
Elementary_Action_Kind a process component (activity) that is not further decomposable (e.g., Cancel_reservation).

Then SymOntoX is able to define ontological relations in order to structure the knowledge and to achieve reasoning. These relations are:

Specialization: a binary relation that denotes the IsA refinement between concepts. Generalization is the inverse relation.

Decomposition: connects a composite entity with its parts. PartOf is the inverse relation.

Predication: the relation that allows an Information Element or Component to be associated to primary concepts. The concept that plays the role of an attribute must be either an Information Element or an Information Component.

Relatedness: this notion represents a generic domain relation between two concepts. It is generally refined into a specific domain relation, associating a specific label to it. E.g., rel(hotel, station) is refined into: near(hotel, station).

Similarity: the binary relation that allows a similarity degree between two concepts to be expressed. The similarity degree must belong to the interval [0.4, 1].

On the other hand the system defines three kinds of users: User who can read the ontology; Super User with read and write rights; Ontology Master is responsible for the ontology content, it has to validate the concepts proposed by the super users.

The SymOntoX system also enhances: Querying Capabilities that allows users to retrieve concepts, but it is important to point out that the query is performed by filling out guided forms. References annotation, ontology export to RDF, and there will be provided and import export utility to OWL.

SymOntoX has a three-tier architecture:
User Interface: It runs on a thin client, which is a web browser, and is developed using JSP.
Server Side: it manages the communication with the clients through HTTP by using Java Servlet technology and the SymOntoX application logic.
Storage tier: It is composed by a database containing the concepts (ontology content), a log database containing the history of the activities performed by users, a factual database containing the sample instances and a database for administrative purposes, which handles users and existing ontologies.

2.1.3 Building Domain Ontology Based on Web Data and Generic Ontology

This tool does not have a specific name. It aims to the construction of Domain Ontologies by first extracting a topic three and then evolving it into a domain ontology. The authors state that web pages are semi-structured, and by applying a wrapper technique the web pages are filtered out the HTML tags and useless information such as advertisement. Then the conjunctions, adverb, exclamations are eliminated. Each word is weighted it in the page. The words are also filtered by weight. The data in the web pages are represented as points.

Afterwards a Hierarchical Agglomerative Clustering (HAC) algorithm is applied, which produces a hierarchical grouping of the data featured as points. The algorithm starts with all points in the same cluster, and then per iteration it merges the two clusters that are most similar. In [4] the distance between two points is calculated by the Euclidean distance and the similarity between two clusters is determined by

single-link. A node from the binary tree can also be characterized as a data point.

Later to identify topics in the binary tree, the variance node is calculated. A smaller variance value of the node indicates a higher intra-node similarity, in which the words are syntactically similar. Then the points of inflection are identified. A node is point of inflection is the variance of this node is smaller than that of its parent node and that of all its children nodes. This means that the data in the children nodes are about the same topic, while the parent nodes cover more general issues.

The topic tree is then analyzed in order to:
Figure out the concept that the topic implies. The topic has been chosen by the word with the highest weight, but the word might not reflect the topic
Clarify the semantic meaning of the hierarchical structure.
Extract possible concepts expressed by the words within a topic and building relations between them.

Finally they match the concepts found with a generic ontology, specifically HowNet. In HowNet word about the same subject have similar sememes (The meaning expressed by a morpheme, which is a minimal meaningful language unit) definitions. Central topics of the concepts will be formed by the common sememes of the words. The relations between concepts can be deduced from their sememes definitions.

2.1.4 PLIB

The result of this project is to create an ontology manager which provides user means to define and evolve an ontology. It is based in three basic concepts:

- On any domain there is a specific vocabulary that belongs to the domain and is able to express well define properties which allow person to person communication (context explicit ontology).
- Human beings are continually evolving the concepts and practices so it is necessary to have a tool to allow users to create its own ontology as a specialization of a shared ontology.
- It is necessary to provide human and/or computer understanding of data meaning.

As a result of that they have created a new data base model called Ontology-Based Database (OBDB), where each database contains an ontology.

It is remarkable to point out that in [30] the author distinguishes two kinds of ontologies, one of them is document oriented called linguistic ontology (LO) and the other structured-data-oriented called concept ontology (CO). A linguistic ontology is focused in the meaning of the words for specifics Universe of Discourse (UoD) in a particular language. On the other hand concept ontology is addressed to represent the categories of objects and of objects properties that are in some part of the word. The concept ontology needs only to describe those primitive concepts that can not be derived from other concepts. It can also be property oriented, in the sense that the concepts must be kept minimal and specialized by means of properties, for example instead of defining the concept "10-HP engine", "25-HP engine", "50-HP engine", "100-HP engine", you should define only two concepts that will handle all the meanings a class "engine" and an integer-valued property "power in HP". As a result only those classes which can not be represented by restricting existing classes by means of properties values need to belong to a property-oriented concept ontology.

The author defines a single PLIB ontology as a 6-tuple where:
"O = <C; P; IsA; PropCont;ClassCont; ValCont>, where: (1) C is the set of classes used to describe the concepts of a given domain; (2) P is the set of properties used to describe the instances of C. P is partitioned into P_{val} (characteristics properties), P_{fonc} (context dependent properties) and P_{cont} (context parameters). When p∈P is a physical measure, its definition includes its measure unit; (3) IsA : C → C is a partial function, the semantic of which is subsumption; (4) PropCont : P → C associates to each property the higher class where it is meaningful (the property is said to be visible for this class); (5) ClassCont : C → 2^{p} associates which each class all the properties that are applicable to every instances of this class (rigid properties); (6) ValCont : P_{fonc} → 2^{pcont} associates to each context dependent properties the context

parameters of which its value depends. Axioms specify that: (1) IsA defines a single hierarchy, (2) visible and applicable properties are both inherited, and (3) only visible property may become applicable." [29]

If we want to define an ontology which is mapped to another ontology, the formal definition is O_m = <O, M> where O is a single PLIB ontology and M = {m_i} is a mapping object with four attributes:
"m =< domain; range; import; map >, where: (1) domain Є C defines the class that is mapped onto an external class by a case-of relationship; (2) range Є GUI C contents {string} is the globally unique identifier of the external class onto which the m.domain class is mapped; (3) import Є 2^p is a set of properties visible or applicable in the m.range class that are imported in ClassCont(m.domain); (4) map C {(p, id) | p belongs to P ^ id belongs to GUI C {string}} defines the mapping of properties defined in the m.domain class with equivalent properties visible or applicable in the m.range class. The latter are identified by their GUIs."

As a conclusion, having in mind that the ontologies should react to all changes in the modeled domain, if the underlying ontology supporting a specific domain, is not up-to-date or the annotation of knowledge resources is inconsistent, redundant or incomplete, then the reliability, accuracy and effectiveness of the system decrease significantly [12] [5]. Ontology manager systems should provide the means not only to create but also to evolve an ontology.

2.2 Component Retrieval Schemes

There are several factors that impact the search and retrieval process for software components such as the scope of the repository, query representation, asset representation, storage structure, navigation scheme, size of the repository. Different techniques are used to accomplish component retrieval. In [42] the authors have identified 5 different component retrieval schemes from a repository. I have added two new categories to that classification. The new categories are called users web mining and browsing.

- Keyword Search. It is based in an indexing technology. The keywords provided to perform the search are compared to software documentation and items descriptions. This approach is simple to implement and the indexing task can be an automatic process. But it is limited by a lack of semantic information between the query, the set of keywords describing each item and the relation among items [41]. As a consequence keyword based searching is not efficient and it will have as result either too many or too few hits. If the result set contains too many hits, the number of non-relevant hits is likely to be very high. If the result set contains too few hits, items relevant to the search can be left out of it basically by the lack of semantics. Furthermore this approach does not take into account additional information such as relationships among objects for instance synonymous names between different concepts that might be applied.

- Faceted Classification. It provides a classification for the items it is willing with. The idea is that this classification must be build by domain experts. Some keywords will describe the components, and such keywords will be placed in the classification schema. The classification schema is used as a standard descriptor for the software components. In order to solve ambiguities a thesaurus is derived for each facet to make sure the keyword matched can only be within the facet context. This technique is useful for objects that can clearly fit into such categories, but it losses quality for objects whose classification is not explicit, or objects that can belong to different classifications regarding specific conditions. Besides it is a labor intensive approach to maintain the classification and description. It requires a domain analyst in order to define the facet.

- Signature Matching. It is based in the type and number of arguments defined for the different methods. The idea is that the search is defined in terms of the method's parameters and return type. It has as drawbacks that the requester must have a deep technical knowledge on the software component he is looking for, and the search can retrieve a lot of items not related with the one the user is expecting to retrieve. Two methods can have exactly the same signature and accomplish

completely different task. For instance, the signature for the strcpy and strcat in the C language, even though the signature is the same the result accomplished is totally different. Furthermore this technique does not take into account the domain or the search context information.

- Behavioral Matching. It takes into account the functional behavior of the objects. In this technique objects are provided with input vectors and the output vectors. The input vector represents the state of the system before the execution; the output vector represents the state of the system afterwards. The object method is executed and the output vector is generated for each object. By comparing the generated vector to the expected outputs the objects that show certain behavior are retrieved.

- Semantic-Based Method. This approach employs domain ontologies. The queries provided by the users are expressed in natural language. The components have a description also expressed in natural language. A semantic analysis algorithm is applied to the user's query as well as to the component description. This semantic analysis uses the different domain ontologies. The query semantic analysis is matched against the component description semantic analysis to perform the retrieval process. It is important to point out that for this approach the Ontology construction is time demanding and requires a domain specialist in order to accomplish it. Furthermore, the ontology has to evolve along with the domain.

- Browsing. The items belonging to the sample to be searched must be classified or categorized. The system provides at least one interface, which allows traversing the classification. The interface might offer different visualization schemas (trees, tables, etcetera).

- Users Web Mining. Some search systems are built up by continuously monitoring the user's behavior, in order to learn from them and later on apply such knowledge at the moment of present recommendations. This technology is comparable with the Collaborative Filtering technique. They do not use any kind of ontology to accomplish its task. The idea is that base on the users' behavior and items monitoring tools, the system is able to create user profiles, which will be used at the moment of providing recommendations.

Table 2-1 shows a comparative table between the different component retrieval schemes, which has been taken from [42], but augmented with some retrieval schemas and with a column to include some examples.

Retrieval Scheme	Underlying Approach	Comments	Example
Keyword Search	Search for the occurrence of string patterns specified by the user in component attributes and descriptions.	• May result in too many or too few items retrieved because only keywords are used for searching • May result in many unrelated items • It is not precise; it has a lack of semantic. • It is simple and can be accomplished in an automatic way	http://ww.componentsource.com http://devcatalog.com http://www.devonestop.com/ INSEAS[8]
Faceted classification	Classify components based on facets (taxonomies) such as function the software performs, medium used, type of system, functional area, etc.	• Components must fit the classification scheme • Some components may overlap categories • Difficulty in managing the classification scheme when domain knowledge evolves • Only guided search – no augmentation	ONTOLOGER [5] ONOTOSHARE [14] LawBot [9] INSEAS [8] InterLegis [23] ADIPS [13]

Retrieval Scheme	Underlying Approach	Comments	Example
Signature Matching	Matching of function types and argument types to the query specified by the user. Signature matching could be one at the function level or module level (set of functions).	• Difficult to map user requirements to function and module signatures • Signature match does not guarantee expected behavior of component • Multiple components may have similar signatures • Limited support for query relaxation	Agora[11] COMPONENTEXCHANGE [35]
Behavioral Matching	Execute each library component with random input vectors and generate output vectors. Compare expected output to actual output and select components.	• May have low recall • Difficult to use when components have complex behaviors or involve side effects • Difficult to express required behaviors • No support for query augmentation	Hall R. J. [50] Behavior sampling [55]
Semantic-Based Method	User requirements expressed as simple imperative or nominal sentences. NLP used for generating initial queries and augmented with domain information. Components selected based on closeness measure (query frame vis-à-vis component frame)	• Domain model provides context information • Ontology ensures use of appropriate terms • Query augmentation to improve recall and precision • Natural (flexible) way for the user to specify requirements for components	Yao et al [41] Sugumaran et al [42]
Browsing	Based on components classification it shows different ways to navigate through out the elements composing the repository	• All the items must be classified following a standard taxonomy, ontology, or classification scheme • An item may not be well classified or not fixed in a specific category.	INSEAS [8] CompoNex [15] ADIPS [13]
Users Web Mining	The system monitors user's behaviors and traversals to learn about their preferences. It Classifies users by common characteristics, in different users profiles in order to make recommendations. It is based in collaborative filtering techniques.	• User behavior provides context information • Accuracy depends on the users profiles classification • The system is continuously updating information automatically • Not based in an ontology	LEOPARD [40] RASCAL [20] SUGGEST [2]

Table 2-1 Summary of Component Retrieval Schemes

In the following sections there is a brief description of some of the systems applying the techniques described in the previous paragraphs; this gives an idea of how they work. It is important to mark out that some of them combine several techniques. That is the reason why next to the system name the most relevant techniques applied are specified.

2.2.1 Keywords Technique

2.2.1.1 INSEAS – Keyword – Faceted – Browsing

INSEAS stands for Intelligent Search Agent System. The system is based in XML and agent Technologies. Component Agent and User Agent convert the inputs into XML documents, which are stored in the repository for later retrieval.

The User Interface Agent is responsible to provide a convenient and efficient search environment. It represents the interfaces for the different search methods, and shows a reasonable number of results. The user can weight to extend similar words, perform combined searches, and give priorities to certain facets. The user can demand the help of an agent, so the user interface agent shows him specific questions provided by the intelligent search helper agent.

Component providers utilize the Component Agent in order to store components in the INSEAS repository. The component Agent represents the component information in an XML based format. INSEAS supports CORBA, JavaBeans, and COM/ActiveX components [8].

The Component Search Agent executes four different search methods: keyword search, facet-based search, browsing search and interactive search with a helper agent. The Component Search Agent is composed by 4 independent agents:

- Keyword Search Agent: Taken the words provided by the user, it retrieves similar words based on the relevance and relationships between concepts. If the result set is big the agent provides to the user methods to reduce it. The threshold of the result set is determined by the user's behavior.
- Facet-based Search Agent: The agent provides a set of facets that the user need to input. The user does not have to fill out all the fields. The agent also observes the fuzzy concept relationship matrices for each tag. Weights between concepts (words) and relevance between concepts in tags and component description documents are used for the retrieving process.
- Browsing Search Agent: The search is performed in the classified category tree by experts. It can search by domain, implementation language, system type, operating system, among others. It is possible to apply the other agents on the query result set if the user wants to.
- Intelligent Search Helper Agent: Regarding different factors such as user's preference, user's environment, user's level, domain-related knowledge, search goal, and the search results at each step, the agent decides the order of the questions about the facets to conduct the search. This process is based in a rule-based reasoning technology.

The User Agent: It manages and stores the user information in the repository, and collaborates in the component search using the repository. It helps users to provide information regarding currents project's domain, experience, environment, and etcetera.

The Repository stores information about software components, users, and the knowledge and rules. The users and components information is stored in XML.

- Expert Knowledge and Rules repository:
    - Fuzzy Concept Network Matrices, which stores the relevance and relationship matrices for word concepts and for tag concepts.
    - The Rules and Expert Knowledge. Uses the rules for intelligent search, results representation, and user interface representation. It takes into account the user information tags, component information tags and user behavior tags.
- User Information repository: stores information such as user's id, current project domain, user's role, development experience, domain experience, dominant language, system type, operating system, satisfaction degree of resulting components, search process, user preferences.

- Component Information Repository: The information in this repository is managed by the Component Agent. The data such, as functionality, environment, interfaces, service level, component type, is stored as XML format.
- Case Repository: It sores information such as user query, user search steps, search time and satisfaction degree, which is used to perform system upgrade, component classification upgrade, and user preferences upgrade. It also uses an XML format.

The System Management Agent: It uses the case information about user behaviors for the system upgrade. The user's feedback changes the component information of service level, categorization, and weights of the fuzzy relation matrix in order to improve the system performance.

INSEAS takes advantage of XML in order to give a semantic meaning to the actors involved. It defines three XML DTDs: Component Information DTD, User Information DTD and Search Case DTD.

XML Specification for Component Information: It is composed by several concepts such as functionality, environment (operating system, language, system, etcetera), service level (performance, limitations, database usage), it includes component type, size, domain, understanding level, price, and user's feedback.

XML Specification for User Information: It stores user id, project domain, development experience, language, system, operating system, user's role, preference, and degree of search satisfaction.

XML Specification for Case Information of Search Process: It holds the information of the relationship between user inputs, results, search time, user selection, and user's satisfaction level. For the searches it stores the user id, user characteristics (project domain, expertise, among others), the keywords used, the date, the number and ids of selected components, the number and ids of non selected components by the user.

INSEAS uses the fuzzy similar relationship, the fuzzy generalization and specialization relationship. To model the extended fuzzy concept network it uses the relation matrices and relevance matrices. The System Management Agent makes a relation matrix and a relevance matrix for the total tag group. Furthermore many matrices exist in the repository, for the component classification, the repository has generalization relation and specialization relation matrices for word concepts.

2.2.2 Faceted Technique

2.2.2.1 InterLegis Project based on Odyssey Search Engine - Faceted

In [23] the authors state that component based software reuse is affected because components are distributed and heterogeneous, and there is not a domain ontology by which the users can refer to the components they are willing to use. On the other hand, the multi-database or Heterogeneous and Distributed Database System (HDDS) are related with distribution, heterogeneity (and ontology). They propose to apply HDDS technology to achieve software component retrieval. The legacy database will be replaced for the components repository. The use of mediators will represent and integrate domain information repositories (distributed and/or heterogeneous). The metadata stored in the mediators describes the components repositories, presenting their domain, semantics and components architecture.

By extending the Odyssey Search Engine it allows the publication of components in the internet using comPublish, and by associating each component to a specific domain based on ontologies and XML. The system allows publishing, describing, storing and retrieving software components.

A mediator is created for each domain. The GOA server stores metadata and components locally. In the

mediation layer, each mediator represents an ontology domain. The ontology provides identification of components and the mediator helps in the mapping to the component repository. Each domain Ontology will define Ontology terms, for instance “a proposal” can be an Ontology term in the legislative domain, or “a regulation” could be an Ontology Term in the judiciary domain. The mediator layer will also enable the application to define relationships between ontology terms in different domains, so we could state synonymous among ontology terms, which belong to distinct domains. This relationship will help to outperform the search methods.

Components are described through XML, and it will contain relevant domain information. It will also define the type of component it is. For instance in the code observed in Table 2-2, the specified component belongs to the Legislative domain, it is use in the analysis phase, it is a Use Case, and its implementation language is UML.

```
<component>
<domain> Legislative </domain>
<phase> analysis </phase>
<type> use case </type>
<language> UML </language>
<author> Robson Pinheiro </author>
...
</component>
```

Table 2-2 InterLegis Component Description [39]

The overall architecture for the system described in [39] is as follows:

Search Agent: The user interacts with the interface to define a query that is handled by the Search Agent. This will send the query in the Web Search Engine, which is based on Google results, but it will also send a message to the ComPublish. The message contains the application domain so the ComPublish will use the appropriate mediators, the user profile and the component features to retrieve a set of components.

Machine Learning Module: It is responsible to gather user information to create user profiles. It will be monitoring the users, and it will update the user profile based on the web pages the user visits, the number of occurrences that the words appear in the different pages. It will have a list of user stereotypes that will help to limit the search results.

Filtering Agent and Collaborative Agent: This two agents work together. Once the web search engine, and the ComPublish return the set of components found by a specific query the Filtering Agent and the Collaborative agent will organize and rate the results based on the information stored in the Hot Links and User Profile (that has been gathered by the Machine Learning Module). Once this task is done the rated query results will be presented to the user.

2.2.2.2 ADIPS Framework - Faceted - Browsing

The framework has three main components: agent virtual machine, ADIPS Repository and design support environment. Software components are stored in the repository as agent-based components called repository agents. A repository agent works in the agent virtual machine. ADIPS repository designs a multi-agent system working in an agent virtual machine automatically using the repository agents according to a specification given by an interface agent in an agent virtual machine.

Repository agents are created by component programmers using a design support interface. Repository agents carry out the application design, which has design knowledge concerning agent-based components. The repository agent has knowledge on design specification of the component including a

functional specification an interface specification, cooperation protocols [13].

On the other hand the repository agent has the following two capabilities: (i) Recognition of requirement: if the agent is reusable for the requirement specification, it replies to the message with functions and performance which can be applied. (ii) Retrieving components: carried out by the repository agent.

In order to create new agent-based components the programmer analyzes component specifications designed by an application system designer. Then he creates repository agents to fulfill the specification. Agent-based components can recognize a specification and retrieve other components. The programmer can reuse existing components, modify them and store them as new ones.

When a specification is sent to the repository as a message, all the agents which manage the component in the repository check each specification and requirement in automatically. Furthermore, the agent which manages other agents decomposes the specification to new specification. This will enable it to retrieve the component without exact match between the specifications described by the application designer and the sub-module specification. This is done throughout an interface, which receives the host, repository and specification. The specification is written in a text area, and in [13] they do not describe such specification. Components that perform the task of sub-module specification will reply to the message with a specification that is presented to the designer in a new window. This allow the application designer decide if there is a lack of components to fulfill its needs.

All the components have an attribute for classification called category. The framework allows browsing components by category, and shows the components' specification. Using this tool the user is able to modify the agent-based component specification.

2.2.3 Signature Matching Technique

2.2.3.1 AGORA - Signature Matching

This search engine automatically generates and indexes a worldwide database of software products, classified by component model (JavaBean, ActiveX, and etcetera). The system implements the Internet JavaBeans agent as a meta-search engine on top of the Alta Vista Internet service. This decision was based in the fact that the search for applet:class can locate HTML pages containing applet tags where the code parameter is equal to a specified Java applet class.

In order to index the components found they rely on the JavaBeans Introspector class, by which they gather information into five fields regarding associated with the document. In [11] we can find the description of the fields:

- Component: In this case it will be assigned the string "JavaBean".
- Name: Contains the fully qualified name of the class or interface represented by the JavaBean. It will empower searches by name.
- Property: It is a list of properties descriptors, obtained from the JavaBean's info. Agora processes the properties descriptor to get the property name, type and the names of the methods to read and write the property. It is possible to index the property descriptor, which describes a property that acts like an array and has an indexed read and/or indexed write method to access specific array elements.
- Event: An event set descriptor describes a group of events that a JavaBean fires. The system retrieves the name, the add listener method, and the remove listener method for each event set and adds them to the tokens associated with the event field. The list of target methods within the target listener interface is retrieved, and the method names are added to the event field.
- Method: Method descriptor describes a particular method that a JavaBean supports for external access from other components.

To reduce redundant information Agora maintains exclusion tables with a list of properties, event sets, and methods common to all JavaBeans.

In [11] they also explain the CORBA agent. This agent communicates with CORBA naming services, to get objects, and on the object it reads the interface. To index the CORBA interface information six field values associated with the document are stored:

- Component: It always stores the value “CORBA”.
- Name: It holds the interface name. Because the interface information is stored in a separately interface repository, from times to times Agora might not be able to connect to it. So the information will be gathered directly from the interface by calling the CORBA describe_interface() call which has the interface name, its operations and its attributes.
- Operations: It has the interface operations. Operation description records contain the name of the operation as well as parameters and exceptions.
- Attributes: It has the interface attributes.
- Parameters: It contains the operation’s parameters.
- Exceptions: It contains the operation’s exceptions.

To perform the search AGORA offers a set of keywords, depending on the component platform the user is interested in. For instance for JavaBeans there is the word method: method-Name, it will retrieve all the methods with the specific name.

2.2.3.2 COMPONENTEXCHANGE - Signature Matching

It is an E-Exchange for software components. The components are described using a Component Description Markup Language (CDML) based on XML. The component’s characteristics are partitioned in four categories: syntactic, behavioral, synchronization and quality.

Syntactic Aspects of a component is also known as the interface signature. It shows the component functionality. A language that can be used to describe this aspect is CORBA IDL.

Behavioral Specifications define the outcome operations. It can be described using non-formal languages. It comprises non-functional properties (quality attributes), which can be Quality-of-Services (QoS) properties such as performance, reliability, availability and global attributes of a component such as portability, adaptability. It is possible to use the QoS Modeling Language (QML) to represent various QoS properties.

CMDL describes components in different aspects. In [35] aspects can be seen as horizontal slices of a system’s functional and non-functional properties. Different aspects can be grouped in different aspect categories: Syntactic aspects, Functional aspects, non-functional aspects, and Licensing and Commerce aspects.

- Syntactic Aspects: It is similar to the one provided in the CORBA Component Model. It specifies the following:
  - Provided Interfaces: The services that the component exposes to the client. It has an interface name, a set of methods and a set of attributes. The methods are specified by a method name, the type of the returned value, the parameters and the exception thrown by the method. A name and its data type specify an attribute.

  - Required Interfaces: Are the services needed by the component in order to provide its functionality.

    - Events: It is the set of events that the component either generates or responds to. An event is specified by name and direction (out if the event is generated, in if the component receives the event from another component).

- Functional Aspects: set of properties represented by a name a value pair.

- Non Functional Aspects: it uses QML to define them. Non functional aspects are specified by contracts, which are specified by constraints along multiple dimensions. A constrain consists of a name, operator and value. The name refers to the name of the dimension, or a property of dimension. Dimension properties allow for more complex characterizations of constraints. They can be used for characterizing measured values over some time period.

- Licensing and Commerce Aspects: it defines the scope and use of a specific software component.

The system is architecture is implemented as a Fat Butterfly Model. On each wing of the butterfly we have a module, one for Components Integrators, the other for Components Vendors. The component integrator module provides a Query Interface, meanwhile the Component Vendor Module provides a Publish Interface.

The Component Description Repository, The License and the Matchmakers compose the central part of the system. For the searching process only those components that satisfy all the specified constrains in the query are retrieved. The user's query is organized in a set of aspect categories. The matchmaking process is performed by multiple matchmaker components. Each matchmaker is specialized in a particular aspect category. The matchmaker component compares the client queries and component specifications with respect to its aspect category. There is a dispatcher component that splits the client query into multiple sub-queries, which are sent to their respective matchmakers. Finally the dispatcher determines the final result by computing the intersection of the results return by individual matchmakers. The query is typed using the interface provided by the system but from the example given in [35] it can be inferred that the user needs a deep knowledge of the components, because it is looking for a component with an exact method name. If the interface does not provide any help for that it will become a quite complicated task. They do not show any details on the interface so it is not possible to describe it any deeper.

2.2.4 Behavioral Matching Technique

2.2.4.1 Behavior sampling - Behavioral Matching

The base of this approach is that software components have a functional behavior that can be executed on given inputs to produce certain outputs [55]. The idea behind Behavior Sampling's is as follows: the system generates random input vectors and the user computes the desired outputs. The system then executes each of the library components on the selected inputs, comparing the computed output with the expected output. All components correct on all samples are then presented to the user. The key advantage of this approach is that the semantics of components and queries are captured precisely and canonically by extensional input/output behavior.

This technique was improved in [50] Generalized-behavior based retrieval. It allows retrieving not only the complete component but also a sub-component. In order to test a decided behavior, the user must construct a model to use. It introduces a step in the process but eliminates side effects, because the system executes the behavior in the model. The authors state that this is design to provide reuse in the large, so the construction of the model is worth it and it will bring benefits.

2.2.5 Semantic-Based Technique

2.2.5.1 Towards a Semantic-based Approach for Software Reusable Component Classification and Retrieval - Semantic-based

The paper describes an application that will be developed in order to improve searching and retrieving in large software component repositories, and also in the World Wide Web. In [41] the authors express that the system will have two main tasks:

- It will improve the search capabilities of software reuse libraries through annotating software components and packages in these libraries with a semantic description of the services provided by the software. [41]. This will be accomplished using the following techniques: natural languages processing on queries, reuse metrics to evaluate reusability, semantic service description, and domain knowledge base applied for whole process for semantic description and retrieval.
- It will also improve searching for software on the World Wide Web through the use of program understanding.

The system will be composed by a set of subsystems which will be oriented to accomplish specific tasks, the subsystems are:

- Intelligent, natural language-based user interface. The user's queries expressed in natural language will be transformed in a conceptual graph semantic representation within a knowledge base, and also translated into semantic web based representation. DAM+OIL ontologies are employed to support domain knowledge for the semantic web based representation.
- Analysis and annotation tool, which by means of program understanding it identifies and describes the functionality of the software components using semantic representation. The semantic representation will be in a conceptual graph knowledge base, and will also be translated into a semantic web representation, supported by DAML+OIL ontologies.
- Semantic matchmaker that compares a user query in a conceptual graph with component service description in conceptual graphs. It is based on a domain knowledge base.
- Intelligent Internet search, which automatically will search and download software components from the Internet based on the user requests. They will be annotated by the analysis and annotation tool.
- Software components repository. It will use UDDI as its infrastructure, and WSDL and DAML-S as service description languages. Components in the repository will be annotated with WSDL/RDF service descriptions.

The system will wok as follows: On the user's queries expressed in a natural language a natural language processing technology will be applied to analyze such a query based on semantics. The query will be translated into conceptual graphs, then into WSDL/RDF.

A program understanding tool will be applied to analyze downloaded software packages. The software services and features will also be translated into conceptual graphs and WSDL/RDF.

Finally a semantic matchmaker will match the user query conceptual graph to the component conceptual graphs, and then WSDL/RDF representation of the user query and the component are matched.

WSDL is used to describe web services in terms of interfaces information, public methods, data type, information for messages, binding information for transport protocol, and address information for locating the service. In this sense WSDL is applicable in a software component description domain. But WSDL has a lack of semantic description, which will be enhanced using Semantic Web ontologies to annotate WSDL description.

2.2.5.2 A Semantic-Based Approach to Component Retrieval - Semantic-based

The System proposed in [42] uses a natural language interface to provide component retrieval, which utilizes the domain knowledge embedded in ontologies and the domain model. The process to retrieve a component has three main steps:

- Initial Query Generation. The user specifies the requirements for the component using natural language. Using a heuristic-based approach keywords and concepts are identified. The query is specified simple imperative or nominal sentences. An imperative sentence consists of a verb phrase with an embedded noun phrase and possibly some prepositional phrases. For instance “Give me details about the biding process”.

- Query Refinement. Keywords and Concepts from the user’s query are mapped to the domain ontology. Related terms based on the context are also identified for expansion. The context of the retrieval is established through the domain model. When no matching terms are found in the domain model, the system checks the ontology for synonyms and uses those synonymous to search the domain model.

- Component Retrieval and Feedback. The functional requirements specified by the user are decomposed into specific processes and actions using the domain model. Those are then compared to the object’s methods. The user establishes a threshold value, and the objects which percentage of actions supported is greater than the threshold are retrieved. The reuse repository contains components with methods capable of providing some features. It is necessary to match the functionality required with the functionalities supported by the component. Components are described using simple imperative sentences. When parsing the user’s query a frame query is created. Component description is also parsed. For retrieving components, the query frame is matched against the component frame. The query frame contains the features that must be satisfied. The frame structure consists of terms and synonymous, which are inferred from the query and the ontology. The conceptual distance is calculated based on the number of terms in both frames that matches or is related to.

The success of the system depends on how the repository is managed, contents indexed and the level of detail components is described.

The system architecture consists of a web-based interface to a domain model, an ontology and a reuse repository. It is implemented with server-client architecture. The client is comprised of a web browser interface. In the server side we find the query interface module, query refinement module and repository.

The query interface module has three components, which are responsible for capturing the users’ query requirements, generating the preliminary database query and displaying the results in an appropriate format.

The query refinement module makes use of the domain specific information contained in the ontology and in the domain model to enhance the initial query. Simple natural language processing techniques translates the user’s query from natural language into a structured query language.

The domain model is organized into objectiveness, processes, actions, actors, and components. This classification has been empirically validated in a sales domain application development. The rest of the domain-specific knowledge is found in the ontology. The ontology is composed by the set of terms, information about terms, and relations among terms.

2.2.6 Browsing Technique

2.2.6.1 CompoNex – Browsing

This approach came out after performing a market maturity study of the software component field. The outcome of the study reflected the means to facilitate the exchange of components between sellers and buyers. Nowadays there is not precise information regarding components, and they must be treated as experimental goods, whose characteristics (usability, compatibility, performance, etc) cannot be assessed until after buying [15]. The testing process should be use to validate the component characteristics rather than to determine them. There should exist and appropriate and automatically verifiable component specifications, this will differ from a test version, because it will explicitly describe component characteristics.

It proposes a component classification based in a thematic grouping into several pages [16]:

- White pages: provide general and commercial information about components. It is expressed in natural language. But it proposes the use of taxonomy during specification. It will store general information such as component name, unique identifier, version, description, producer, administrative contacts, and dependencies to other components. As far as the commercial information is regarding it holds conditions of purchase, distribution channel (distribution form, price, accepted payments, scope of supply), and license agreement.
- Yellow pages: specify the domain that a component belongs to. It also contains information about the underlying architecture and technology of the component. The framework provides different taxonomies for the domain the component belongs to such as UNSPSC, NAICS, and Microsoft GEO. It also provides a taxonomy which list implementation technologies (EJB, COM, .NET, XML Web Services, etcetera).
- Blue pages: summarize domain-related information about the component functionality. It describes a domain lexicon. It provides three concepts: objects (entities), operations (tasks) and processes. It is possible to relate concepts by abstraction or composition. Typical abstractions are the is-synonym-to (is-identical-to), is-specialization-of, and is-generalization-of, which are use to relate concepts to each other. Compositions are used to combine concepts. Typical compositions are order relationships, the is-part-of, and consist-of. Concept definitions give an impression of what a component or an interface-method does.
- Green pages: provide the provided and required interfaces specification. It uses OMG IDL. It supports for each interface specification of invariant, pre-conditions and post-conditions by means of OCL (Object Constraint Language), which is extended with temporal logic to provide flow information, regarding the predetermined order on which methods should be invoked.
- Grey pages: provide components quality attributes description either to the component or to the interface methods. The idea is to describe quality components regarding the ISO 9126 quality model, which comprises usability, maintainability, functionality, reliability, and efficiency of a component implementation. It should be specified using QML, but after using the system this module it is not being validated, so it accepts any test in the description.

The component specification languages and different levels to be specified are explained in detail in [36], which is a Standardized Specification of Business Components.

Further references to this project are targeting this approach to describe the Web Services specification in order to provide means to describe and retrieve web services. It shows compatibility with UDDI, as stated in [33] it can be used as a wrapper to UDDI but taking advantages of the complete service description provided by the thematic grouping and component specification into pages.

2.2.7 Users Web Mining Technique

2.2.7.1 RASCAL - Users Web Mining

It is a recommender agent system for software components. It has 2 main objectives firstly it is interested in recommending software components that the user is searching for. Secondly, it is intended to recommend components that the system believes a user actually requires but is unaware of such components existence or the need for such components [20]. In the system the user, which will mainly be a developer is considered a java class and the components employed by a class are items. Specifically the components referred to are java methods.

The system runs in the background to monitor and update the user's usage history. The rate method use for RASCAL is implicit; it means the user does not have to explicit rate the component. It is implicit because it automatically deduces the user vote for an item by monitoring how often the user has used such component and usage histories of components are automatically collected and stored in a user-preference database. Then the recommendation will be based upon such rate and a collaborative filter technique, which states that the users can be grouped together in a set of users alike. So for a specific user it is highly probable that he will use the components used for the users belonging to the same set.

The architecture of the system is composed by three main elements: the code repository, the usage history collector and the recommender agent.

Code Repository: As new components are developed they must be stored somewhere. "The repository is effectively a user preference database, a user is a java class and the components employed by a class are items." [20].

Usage History Collector: It will automatically mine the code repository to extract usage histories for all classes. This will need to be done once initially for each class and subsequently anytime a class is added to the repository. The information is extracted using the Byte-code Engineering Library (BCEL). Component usage histories for all the users are then transformed into a user-item preference database. By the moment the paper was written the database contained a user-item preference matrix for all users. It also contained information for each individual user a list of components based on their actual usage order.

Recommender Agent: The tasks performed by the agent are: monitors the current user and updates the user preference; attempts to create the set of users similar to the active one by searching the user-item matrix produced by the usage history collector; finally recommends a set of ordered components to the current user.

As a conclusion on the different retrieval schemas it is possible to say that each one of them have their strengths and drawbacks. The combination of the different schemas has shown an improvement in the retrieval process. Nevertheless research in this topic is still being carried out nowadays. The retrieval process is highly related to the description of the resources being retrieved, and software components do not have a specific notation to describe them. The retrieval scheme is tailored to the specific repository where the search takes place.

2.3 Model Driven Architecture - MDA

MDA stands for Model Driven Architecture. It is a trademark of the Object Management Group (OMG). As stated in the MDA specification web site [64]: "The MDA is a new way of developing applications and writing specifications, based on a platform-independent model (PIM) of the application or specification's

business functionality and behavior. A complete MDA specification consists of a definitive platform-independent base model, plus one or more platform-specific models (PSM) and sets of interface definitions, each describing how the base model is implemented on a different middleware platform. A complete MDA application consists of a definitive PIM, plus one or more PSMs and complete implementations, one on each platform that the application developer decides to support."

As inferred from the MDA specification, it is based in models. Some people define a model as a visual representation of a system. But on the other hand, many people refer to a set of IDL interfaces as a "CORBA object model." Besides, an UML can be rendered into an XML document using the OMG's XMI DTD for UML, such representation is not a visual artifact. Thus, a more precise definition is needed. In [71] the following definition is given: "A model is a formal specification of the function, structure and/or behavior of a system".

This definition has the following underlying concepts: "A specification is said to be formal when it is based on some well defined language that has well defined meaning associated with each of its constructs" [71]. As a matter of fact if a specification is not formal in this sense, is not a model. Consequently a diagram with boxes and lines and arrows that does not have behind it a definition of the meaning of a box and the meaning of a line and of an arrow is not a model, it is just an informal diagram. Under this model definition the subsequent are models examples "Source code is a model that has the salient characteristic that it can be executed by a machine. A set of IDL interfaces is a model that can be used with any CORBA implementation and that specifies the signature of operations and attributes of which the interfaces are composed. A UML-based specification is a model whose properties can be expressed visually or via an XML document" [71].

A PIM is a model of a software system that does not incorporate any implementation choice. It stands for Platform Independent Model. PIMs describe the system independently of the chosen implementation technology. On the other hand a PSM is a model of a software system that incorporates choices for certain implementation technology/technologies. It stands for Platform Specific Model. PSMs describe the system taking into account the chosen implementation technology.

The aim of this approach is to let software development process concentrate in the specific domain it is trying to model. At the higher design level, the design process should only care about the software functionality. There must be a clear separation between the required functionality and the middleware platform on which such functionality is going to be implemented. In the MDA, middleware-specific models and implementations are secondary artifacts. A specification's PIM is the primary artifact. It defines one or more PSMs and sets of interface definition, each specifying how the base model is implemented on a different middleware platform. It separates the fundamental logic behind a specification from the specifics of the particular middleware that implements it.

MDA is on the use of models in software development [70]. In order to accomplish that goal from an abstract model of the system a more concrete model should be generated. From that model in turn an even more concrete model can be generated until finally the source code is produced. Source code is considered to be the most concrete representation/model of the software system. Key to this process is that each generation step will be automated as far as possible. The ultimate MDA goal is to generate automatically a complete software system from a model with as less human work in the process as possible.

A Model transformation is the process of converting one model to another model of the same system. Transformations can use different mixtures of manual and automatic transformation. There are 4 different transformation approaches: manual transformation, transforming a PIM that is prepared using a profile, transformation using patterns and markings, and automatic transformation [70]:

- Manual transformation. When the design decision to make the transformation from PIM to PSM are made during the process of developing a design that conforms to engineering requirements on the implementation. The decisions are considered in the context of a specific implementation design.

The MDA adds value in two ways: there is an explicit distinction between a PIM and the transformed PSM, the transformation is recorded.

- Transforming a PIM that is prepared using a profile. The PIM and the PSM are expressed using UML profiles. The transformation may involve marking the PIM using marks provided with the platform specific profile.

  The UML 2 profile extension mechanism may include the specification of operations; then transformation rules may be specified using operations, enabling the specification of a transformation by a UML profile.

- Transformation using patterns and markings. This applies a pattern to the transformation. In order to apply the pattern elements from the PIM are marked. Those marked elements are transformed according to the pattern to produce the PSM. For instance a class marked in the PIM with a role from the pattern, once the transformation is applied can produce in the PSM the original class with some extra attributes and operations, new classes corresponding to other roles in the pattern and associations between those classes.

- Automatic transformation. In some cases it is not necessary to provide to add marks or use data from additional profiles in order to be able to generate code. The decisions are implemented in tools, development processes, templates, program libraries and code generators. The PIM contains all the information necessary to produce computer program code.

As a conclusion on The Model-Driven Architecture (MDA), it basically defines an approach to modeling. In the modeling process it separates the specification of system functionalities from the specification of its implementation on a specific technology platform. The MDA promotes an approach where the same model specifying system functionality can be realized on multiple platforms through auxiliary mapping standards, or through point mappings to specific platforms.

Summary and Conclusions

After describing the different schemes used to retrieve components, it can be inferred that each one of them can help to resolve different type of queries. For instance, faceted and classification using natural language will provide means to retrieve components based on external information provided as natural human language. As a consequence it can be also used by non technical users. On the other hand a scheme like signature matching provides a more deeply technical description and as such it can be thought more oriented to technical users. Anyways, researches are combining different schemes in order to take advantage of their strengths and diminish their weakness.

Components retrieval schemes based on classified components have been evolving in the classification techniques, from simple faceted to ontology-based classification. Ontologies appear as a helping element in the classification and modeling of complex relationships. Ontology manager systems have been developed to create and manage ontologies. For my research I propose the use of an ontology to describe software components.

The use of classification schemas has brought a proliferation of models because there is not a common description model for software components. That model is still an open issue. There is not a definitive answer to questions like: what kind of information will characterize a software component? Should this information be based on properties of software components? Can software components be generalized? How quality information will be assessed?

Then, what can be done to tackle down the proliferation of models? In this research I propose as an answer to this question an integration of software component repositories.

3 The eCots Association

eCots is the name for an inter-industrial association founded in January 2004 by Thales, EDF R&D and Bull. The association has specified a project to create an Intelligent Portal for Searching Components called IPSComp. The IPSComp project is in the specification phase and it will become a proposal in a European Integrated Project. The project aims at developing an open information portal for commercial of-the-shelf (COTS) software and non software components, in which we deal with information about products, and possible between their users, or between users and producers. As expected from any industrial project, the main aim is economical: the project is addressed to provide its users with a maximum of quality-controlled information at the lowest possible price.

This research is included as a first step for the definition of the functional architecture of the IPSComp project. The objective of this project is to use the potential offered by Internet portals to federate the community of users of commercial off-the-shelf software components. Thus giving them means of obtaining the information they need from COTS component producers, facilitating access to such information and supplementing it by pooling – through cooperative generation of content – the information on use that it possesses, by setting up a dedicated thematic portal, freely accessible on Internet.

In [48] the authors have identified a set of elements that will need further discussion in order to achieve the project's aim. Among other questions they formulate for instance, what kind of information will characterize a COTS product? Should this information be based on properties of COTS products? Are there many COTS products sharing similar properties? Can they be generalized? How quality of information will be assessed?

The paper also identifies some key elements that have been group into three categories: management, development and knowledge base. The interactions between them are depicted in Figure 3-1. The key elements are:

- Management:
  - Procedure to specify how the portal should be used.
  - Legal issues on the use of the portal.
  - Quality assessment procedures and measurements for software deliverables, ontology specifications, third-party information, and software and ontology development procedures and methodologies.
  - COTS versioning management, software configuration management, procedures for ontology and portal evolution.
  - Standard information to be supplied by vendors and/or private organizations.

- Development
  - Recommender system (personalization and support).
  - Software specifications.
  - Ontology specifications.
  - Multilingual definitions for GUI and ontology specifications.

- Knowledge-based Support
  - Taxonomy and classification.
  - Global ontologies (COTS and domain-oriented).
  - COTS component ontologies (COTS quality attributes, COTS implementations, etc).
  - Domain-oriented ontologies (I&C, E-business, Health Care, etc).

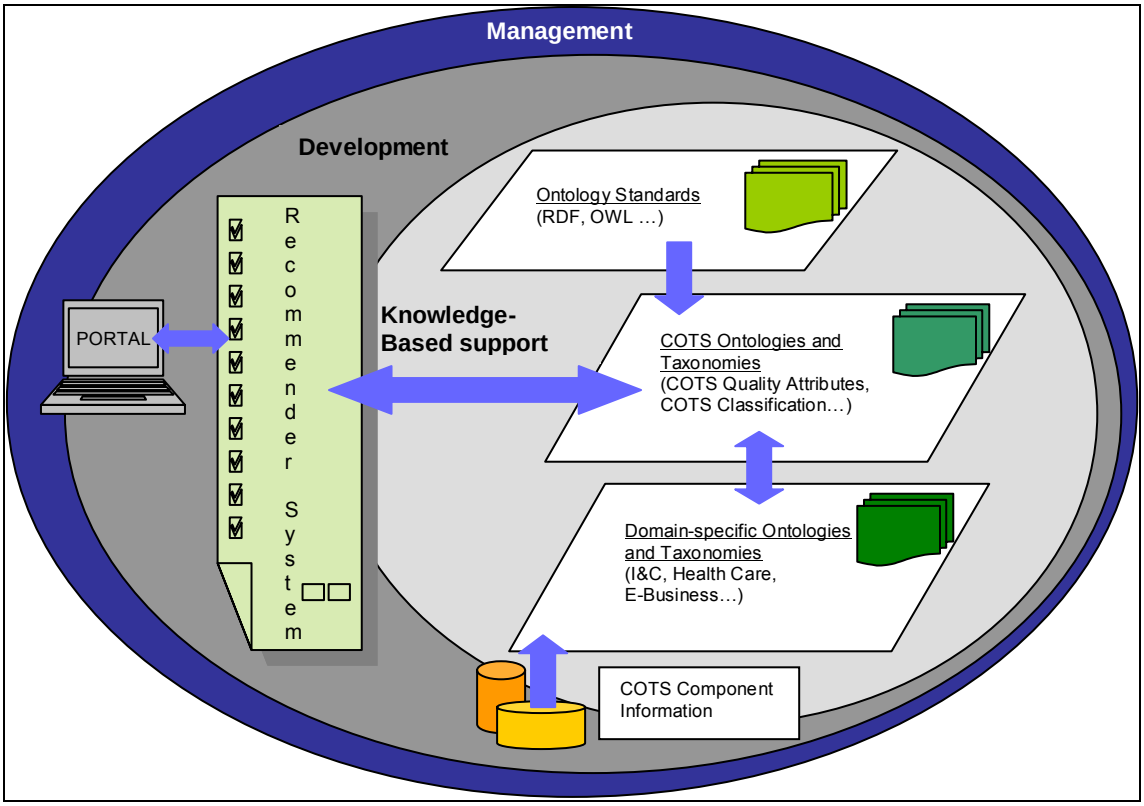


Figure 3-1 IPSComp System Architecture [48]

The end user will have access to the system through a web browser. The knowledge-Based support which is the base of the system has three layers. The Ontology Standards layer will hold the set of tools used to define the ontologies included in the system. For instance, Web Ontology Language (OWL)[2] can be used to explicitly formally describe an ontology[3]. The other two layers correspond to ontologies. One of them is the ontology that supports the set of items include in the system in this particular case software components. The other layer also contains ontologies. Software products are in contact with a wide range of domains. This layer provides ontologies for those domains. For instance, there can be an ontology for the health care domain, for the e-business domain, for the telecommunications domain, components in other domains, etc.

Taking advantage of the knowledge-based module, supported by the different ontologies, the portal will also provide a Recommender System, which will be in charge of helping users in the identification and retrieval of software components that they may be interested in.

My contribution to the IPSComp is explained in the next chapter.

[2] Web Ontology Language (OWL) is a revision of the DAML+OIL web ontology language. It has more facilities for expressing meaning and semantics than XML, RDF, and RDF-S, and thus OWL goes beyond these languages in its ability to represent machine interpretable content on the Web. (http://www.w3.org/TR/owl-features). An ontology language is required for the Semantic Web vision in which information is given explicit meaning, making it easier for machines to automatically process and integrate information available on the Web

[3] Accordingly to W3C an ontology is the representation of the meaning of terms in vocabularies and the relationships between those terms (http://www.w3.org/TR/owl-features).

4 Contribution

My contribution on the IPSComp project is to create the component ontology specification for IPSComp (Section 4.1. Item {2} in Figure 4-1), produce the component ontology design for IPSComp (Section 4.2. Item {2} in Figure 4-1), implement the component ontology for IPSComp (Section 4.3. Item {1} in Figure 4-1) and provide a component repository integration (Section 4.4. Item {3} in Figure 4-1). The previous points aim to provide the means to achieve a qualified Recommender System for the IPSComp project (Item {4} in Figure 4-1). My contribution in the system architecture is red highlighted in Figure 4-1.

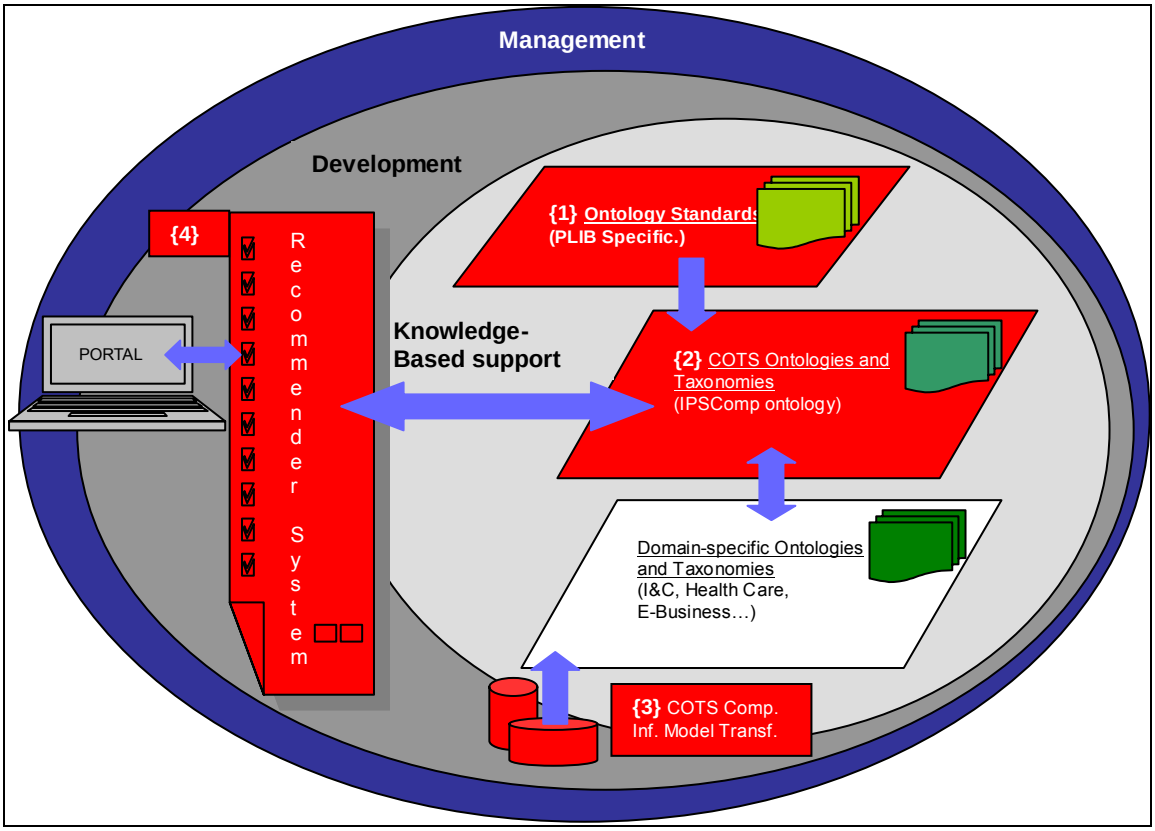


Figure 4-1 Contribution to the IPSComp Project

4.1 Component Ontology for IPSComp Specification

Ontologies are emerging as a key solution to allow different applications to exchange and to reason about information in the system. Ontologies provide a mechanism to represent and store domain specific knowledge. An ontology usually refers to a set of concepts or terms that can describe some area of knowledge or build a representation of it. Ontologies provide a set of well defined, structured and agreed terms in order to disambiguate communication exchange between applications (software agents, programs, etc). An ontology based component retrieval method should be able to exploit the additional knowledge embedded in domain ontologies to augment or revise a user's initial query. This use of ontologies to take into account the semantics of the application domain should result in greater query flexibility, augmentation and user satisfaction.

4.1.1 XCM Component Ontology

The aim of this component Ontology presented in [28] is to provide (i) a standard for the definition of components that unifies the differences between different models (ii) a standard interface for component searching. For each component it defines 2 dimensions: features that are composed by the set of properties, methods and events; and design which describes how a component is constructed by using existing components. As a result XCM is able to hold information for each component regarding:

- **General Information**: i.e. component name, version, package, language, component model, **domain**, operating system, and publisher.
- **Features**: The set of features describes how the component interacts with other components. It is composed by properties, methods and events.
  - Property: Is the named attribute of the component. It is described by:
    - Syntax: pType: is the domain type; access: it can be readWrite, readOnly, writeOnly; Style: it can be simple, indexed, bound, constraint.
    - Specification: pName: is the property name; desc: holds the property description.
    - Introspection: writeMethod: is the method name to set the property value; readMethod: is the method name to get the property value.
  - Method: It holds the interfaces, provided (behavior that can be triggered to other objects) required (from the other components to complete its functionality). It is described by:
    - Syntax: returnType: the return domain type; paraType: ordered list of the parameters domain type; status: can be provided or required.
    - Specification: mName: the method name; desc: a textual description; pre: the pre-condition; post: the post-condition.
  - Event: It is the message used by a component to communicate with other. It is classified as published (a component publishes to its recipients to notify something has happened and an action must be taken) or consumed (an event that a component subscribes to in others components). It is described by:
    - Syntax: eType: the event type; delivery: the event delivery (unicast or multicast); status: published or consumed.
    - Introspection: addListenerMethod: method name that registers one or more listener components based on the event; removeListenerMethod: method name that removes listener components from the event; listenerType: the type of the listener component, represented by the listener interface, that are allow to register for the event; listenerMethods: set of one or more listener methods that the listener components registering for the event must implement. Each listener method is specified by: mName (the method name), returnType (the type of value returned from the method) and paraType (ordered list of parameter type required for the method).
- **Design**: it describes how to construct a composite component connecting pre-existing components.
  - Underlying Component: It is a component use to build up components It is described by:
    - Syntax: comp: the component domain type.
    - Specification: cid: the component instance level; desc: the component instance description; role: can be master, client or support.
  - Connection Oriented Composition: It describes how components are connected using events or pipe and filtering mechanisms. Components can be classified either as Event Components (that fire events) or Listener Components (that listen for events and subsequently trigger specified methods in a well-defined manner). This connection is described by:
    - Syntax: Event: the fired event.
    - Specification: rid eCompInstance: the label of an event source component; rid lCompInstance: the set of labels of event listener components; eAction: the event action defined under a fired event listener interface; lcomposition: the composition of methods: inv (the composition invariant), pre (the composition pre-condition), post (the composition post-condition).
  - Aggregation based Composition: It describes aggregation of components into higher level

components. For the aggregation components can be classified as Container Component (that provide the containment for the containee components) or Containee Component (that is aggregated into a container component in the specified position). It is described by:

- Specification: container: the label of the container component; containee: the label of a containee component; location: the location where the containee component is positioned in a container component.

Figure 4-2 taken from [51] represents the XCM hierarchical component structure. Here a component is defined via (i) general information, (ii) features that contains the component's set of properties, methods and events; (iii) design that encapsulates how a composite component is constructed from other components either by connection-oriented and/or aggregation-based compositions. This hierarchical structure can be represented as an XML document, while the general structure of the description model - the XCM concepts - can be described as an XML schema, as proposed in [28].

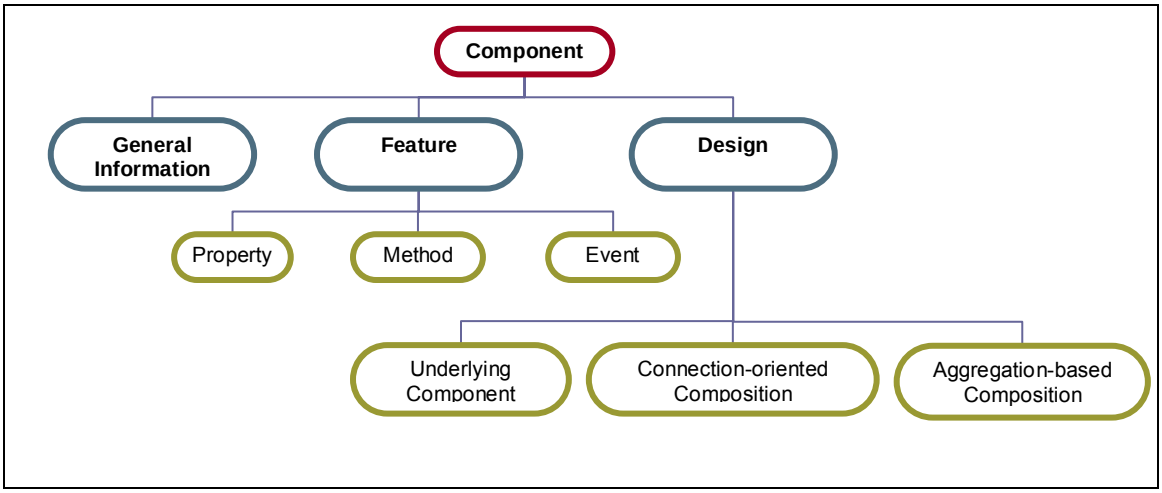


Figure 4-2 XCM Hierarchical Component Structure [51]

The XCM ontology provides a component ontology which gathers the information that is relevant for the IPSComp project component ontology. As a matter of fact the XCM ontology is taken as a base to create the component ontology for IPSComp.

4.1.2 Component Ontology for IPSComp Specification

In order to create the Component Ontology for the IPSComp project I took as a base the XCM ontology and modified some concepts. Moreover I added to the Component Ontology for the IPSComp project quality attributes, the license concept, the price concept and the publisher description concept. The modifications applied to the XMC ontology to create the IPSComp ontology as well as the adding to it are described in the following sections.

4.1.2.1 Domain

The General Information in the XCM ontology contains the domain concept, but it is represented as a String. This concept was change to become a multi value field. The aim behind this change is to avoid the problem presented in the faceted-based search scheme with the items that can not fit into one specific category or classification. As a matter of fact a component can be related or belong to different domains.

4.1.2.2 Price

The price concept has been added to it. It will help to develop the marketing area of the IPSComp project. This concept has been added to the General Information description. It is remarkable to point out that the same component can have different prices depending on characteristics such as the number of licenses, or even the functionality provided by the component.

4.1.2.3 Quality Attributes

I added to the IPSComp ontology a set of quality attributes. This is important to extend the component description. A software component in the IPSComp project scope will most probably be a black box software artifact (as stated in the component definition given in the introduction). Besides, the software components will be marketable and used by customers in different implementations at later times. The set of quality attributes will help in the identification and retrieval of software components.

In [52], the authors identify that most of the software engineering community has been mainly focused on the functional aspects of components. As a consequence the quality and extra-functional attributes have been left aside. Nevertheless, it is worth it to pay attention to these factors because they can become a key point in any commercial evaluation.

There are four main issues when considering quality and extra-functional attributes of software components.

- There are several proposed classifications regarding component's quality attributes. But there is not a general consensus on the quality attributes that should be considered.
- There is a lack of information about quality attributes among the different component's providers and vendors.
- There is an absence of metrics that could help evaluating quality attributes objectively.
- Finally the international standards provide very general quality models and guidelines, which are difficult to apply to certain domains such as Component Based Software Development (CBSD) and Components Off-The-Shelf (COTS).

To overcome these issues Bertoa et al [52] propose a quality model for CBSD based on ISO 9126. The international standard ISO 9126 provide definitions and classifications of the quality characteristics of software products. In ISO 9126 a quality characteristic is a set of properties of a software product by which its quality can be described and evaluated. An attribute is a quality property to which a metric can be assigned. A metric is a procedure for examining a component to produce a single data. The quality model proposed by the authors defines a set of quality attributes and their associated metrics for the effective evaluation of COTS components. This approach is tailored to software components.

Three considerations have been taken into account by Bertoa et al [52] in order to produce the quality model:

- The moment at which a characteristic can be observed or measured, either at runtime (e.g. performance) or during the product life cycle (e.g. maintainability).
- The target users of the model are software developers and software designers.
- A component is considered as a black box software artifact, so even though the targets for this classification are software developers and software designers, the idea is that the specific implementation is hidden and can not be modified by them (This is according with the component definition the IPSComp project has taken).

Basically three types of transformations were applied by Bertoa et al [52] to the original ISO 9126 to be tailored to software components. First of all, the Portability characteristic and the Fault Tolerance, Stability and Analyzability sub-characteristic disappeared. Second, two new sub-characteristics appeared

Compatibility and Complexity. Third, The Usability characteristics and the Learnability, Understandability and Operability sub-characteristics changed their meaning. The quality model characteristics for the modified ISO 9126 are briefly explained:

- Functionality: It tries to express the components ability to provide the required services. Its definition has not been changed. On the other hand the sub-characteristic Compatibility was added to the model, to indicate if former versions of the component are compatible with its current version.
- Reliability: It keeps the original meaning. The maturity sub-characteristic is used to measure the number of commercial versions and the time intervals between them. Furthermore, recoverability measures if the component is able to recover from failure and how it does it.
- Usability: This characteristic has completely changed its definition. The reason behind it is that the component's end users are developers and application designers rather than regular end users. This characteristic measures the component's ability to be used by the application developer during the construction of a software product. The Complexity sub-characteristic when integrating and using the component within a software product or system has been added.
- Efficiency: It keeps the original definition, which distinguishes between Time Behavior and Resource Behavior. Some people call this characteristic performance.
- Maintainability: It describes the characteristic of a software product to be modified. Even though on a black box component it is not possible to make modifications, the developer must adapt it, configure it, and tested to include it in a final application. As a consequence the Changeability and testability are sub-characteristic defined.
- Portability: This characteristic was eliminated, because for software components the ability of a product to be transferred from one environment to another must be intrinsic.

The Table 4-1 extracted from [52] shows the quality attributes defined for software components and also a complete description of them can be found there.

Characteristics	Sub-Characteristic - Runtime	Sub-Characteristic – Life Cycle
Functionality	Accuracy Security	Suitability Interoperability Compliance Compatibility
Reliability	Recoverability	Maturity
Usability		Learnability Understandability Operability Complexity
Efficiency	Time behavior Resource behavior	
Maintainability		Changeability Testability

Table 4-1 Quality Model for COTS components [52]

In order to measure these characteristics the authors also propose specific metrics. Each quality attribute will have a specific metric associated to it. Those metrics are:

- Presence: It identifies whether an attribute is present or not in a component. It is measured by a Boolean indicating if the attribute is present and a String, which states how the characteristic is implemented.
- Time: It measures time intervals. It uses an integer indicating the absolute value and a String indicating the units.
- Level: It is used to indicate the intensity in which an attribute is present it is described by an integer in a scale from 0 (very low), 1 (low), 2 (medium), 3(high) and 4 (very high).
- Ratio: It is used to describe percentages (0 – 100).
- Indexes: Are defined as derived measures calculated from basics attributes. For instance, the Complexity Ratio compares the number of configurable of the component with the number of its

provided interfaces.

This section describes the quality attributes and the proposed metrics for quality attributes taken from [52]. I think the quality attributes are necessary to complete the IPSComp component ontology as stated at the beginning of this section. I took this research because it is based on a standard ISO 9126, it is tailored to software components and it proposes the metrics for the quality attributes. Nevertheless, concerning to the quality attributes these are some issues I want to point out:

- The characteristic Functionality, sub-characteristic Suitability tries to measure how well the component fits the user requirements. It is obtained by dividing the number of user required interfaces by the total number of interfaces provided by the component. As this attribute is directly related with the end user needs, the component provider can not measure it. So it is up to the end user to provide this metric. But this attribute will be dependent not only on the component but also on the application on which the component is being used, and even worst, on the developer needs. There is not a standard, so an application designer might be looking for a component that performs a wide set of interfaces. It is a matter of how the application designer defines the set of services he wants to obtain from a component. As a consequence even though the quality attribute is clear and the operation to be performed in order to obtain the value is also clear and easy to perform, the value depends on the user requirements for the specific application.
- For the characteristic Usability, sub-characteristic Learnability, which tries to measure the time and effort needed to master tasks such as usage, configuration, parameterization, or administration of the component. This measures is provided by the component provider but should be validated by the end user. This is a really subjective metric, because it depends on the knowledge and skills from the user who is working with the component.

As a consequence some metrics can become really subjective values, so I think it is necessary to monitor them in order to determine the accuracy of its value.

As far as metrics are concern, in [52] each type of metric is defined by certain attributes, for instance Presence is measured by a Boolean indicating if the attribute is present and a String, which states how the characteristic is implemented; meanwhile Time uses an integer indicating the elapsed time and a String indicating the units. The Presence metric does not have units, the Time metric does not have the feature as the Presence metric does.

In order to generalize the metric concept for the IPSComp component Ontology, I change the definition of the metric to be represented by three attributes:

- Feature which stores the metric name or characteristic to be measured. It might store a relevant value for the metric, for instance in a presence metric the feature stores how a particular attribute is incorporated by the component. In the security case it could have a value “SSL” which means that the security for the component is implemented using SSL certificates. It can be seen as a qualitative value.
- Value which is the amount of the feature being measured. This can be a number, a Boolean, a Scale system, etc. It can be seen as a quantitative value. For instance if the metric is a time, it can be a number representing the amount of elapsed time.
- Unit which is the unit used in the metric.

The idea behind these three attributes defining a metric is that new metrics can be incorporated to the IPSComp component ontology. Furthermore it provides a context to define concepts to include in the ontological analysis of the component description. For instance the feature defined by the metric can be included in as an ontological concept that can be related to other concepts. On the other hand, the unit concept will fit in the context of a taxonomy to be able to perform comparisons. This standardization facilitates the creation of a grammar for the metrics.

Besides I made the following changes to the Time and the Number metrics:

- Time: It will be measured by a Float instead of an integer. This value represents the elapsed time absolute value.
- Number: In [52] they present the integer metric for some quality attributes, which is only a number. I represent this concept with the number metric, but it is important to remark that this metric has the other 2 attributes addressed to provide context to the metric. Actually the time metric is a specialization of this metric, that has been defined independently and I kept this definition because I consider time is a specific domain, which it is worth it to be handle apart.

Finally, after those remarks about quality attributes and changes to the metrics I added the quality model IPSComp component description. In the scope of the whole project it is necessary to arrive to some standards, which will be adopted for the different actors involved in the project. This quality model has been created from the ISO 9126 standard, tailored for software components what makes it a good starting point to achieve a well accepted norm. Furthermore, the quality model does not define only quality attributes, but also the metric applied to each one of them. On the other hand, the aim behind assigning quality attributes for a software component is to provide a set of extra-functional attributes for it. These set of properties must help in the retrieval and evaluation of software components.

4.1.2.4 License

Because the IPSComp project belongs to a commercial initiative, there is information that should be included in the IPSComp ontology in order to facilitate the commercialization of components, and to reduce legal issues. The concept license is added to the component ontology, it will provide a name and the description of the license. A component can have several licenses associated to it. “A software license is a type of proprietary or gratuitous license as well as a memorandum of contract between a producer and a user of computer software — sometimes called an End User License Agreement (EULA) — that specifies the perimeters of the permission granted by the owner to the user”[4].

4.1.2.5 Publisher Description

The information stored by this concept is addressed to provide software providers’ information. The component producer can be described, in order to gather more information that can help users in the retrieval process, but in gender in a marketable place as the IPSComp project is intended to be this information offers additional value to customers, helping identifying producers.

4.1.2.6 Specialization Scenarios

Component specialization is a technique presented in [38]. The idea is to give to the component producer, who has access to the details of the implementation, the means to identify helpful specialization opportunities and to publish them as part of the component interface. These are seen as specialization scenarios.

The objective of a component producer is to provide software components applicable to the widest possible range of context, having in mind that maximizing reuse, minimizes use. By analyzing the code the producer can offer to the consumer, specialization scenarios in order to provide more efficient alternatives to the generic version of the component. The scenarios are only written in terms of services specified in the port interfaces, because it is the only information available at assembly time. These specialization scenarios are defined providing an extra annotation to the method’s signature in order to indicate whether

[4] This definition has been taken from http://en.wikipedia.org/wiki/Software_license

the return type and parameters are considered static or dynamic. A complete description of the research can be found in [38]. This notation on the interfaces has been included in the component ontology for IPSComp.

The inclusion of this element in the component description pretends to join the IPSComp project to a PhD research [38]. It is addressed to a specific component repository, which differs from the commercial standard. It has been included to show that tailoring the proposed ontology will not have an impact over the model transformations that I will be explaining further down in the document. It will only add a set of methods to the proposed API. It shows the flexibility to include different component types into the component description. Further studies should be performed in order to include different information until the component description standard is reached.

After creating the IPSComp ontology description taking as a base the XCM ontology, modifying it and adding concepts it is worth to show the result in the hierarchical component structure to highlight the differences between the IPSComp ontology and the XCM ontology. Now a component is defined not only by General Information, Feature and Design, but it has been added the Non-Functional Characteristics as shown in Figure 4-3 with the green colored element. The other changes are perceivable at General Information level where the domain concept (Section 4.1.2.1) was modified, the domain price (Section 4.1.2.2), license (section 4.1.2.4) and the publisher description (section 4.1.2.5) were added to the IPSComp ontology. It is shown in Figure 4-3 with the green diagonal lines. Finally at the method level the Specialization Scenarios were added. It is depicted in Figure 4-3 with the green vertical lines.

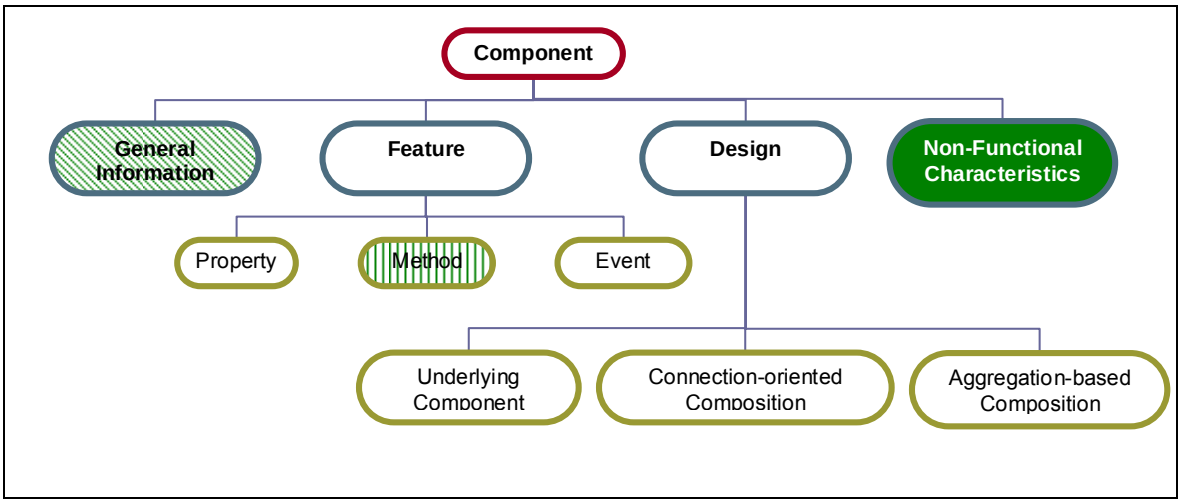


Figure 4-3 IPSComp Hierarchical Component Structure

The IPSComp ontology provides means to describe software component in a syntactic, semantic and behavioral way. For instance, the syntactic definition can be seen in the methods signature representation. The methods represent the set of interfaces the software component offers. The signature of a component interface is a syntactic description. It is necessary to add constraints regarding their use. It can be achieved by a semantic description. The IPSComp ontology description is able to hold the method's pre-condition and post-condition, this allows defining some semantic information. As far as the behavioral description is concerned the idea is to store this information in natural language in the description fields that belongs to the concepts (Method, Property, Component) in the IPSComp ontology. Additionally some of the non-functional characteristics (quality attributes) can store behavioral information (e.g. response time can be seen as a behavioral characteristic).

4.2 Component Ontology for IPSComp Design

The following UML class diagram[5] represents the component description ontology for IPSComp detailed in the previous section. Because the complete model does not fit properly in the page I have selected some classes that will allow illustrating the main ideas, the complete UML class diagram can be found in Appendix A - IPSComp Ontology UML Class Diagram.

Figure 4-4 models the IPSComp component ontology. A component has general information and a list of quality attributes, which are normal associations. Besides the Component have aggregation associations with the Method class, which represents the component's interfaces; the Property class, which represents the component's state; and the events, which are used to model how a component communicates with other components.

On the other hand IPSComp ontology also handles the component's design (Figure 4-3). It has 2 classes to accomplish this task. Each class represents the way a component can be composed by other components. These classes are the AggregationBased and ConnectionOriented (Figure 4-4).

[5] The UML class diagrams were depicted using 'Poseidon for UML' (http://www.gentleware.com/index.php).

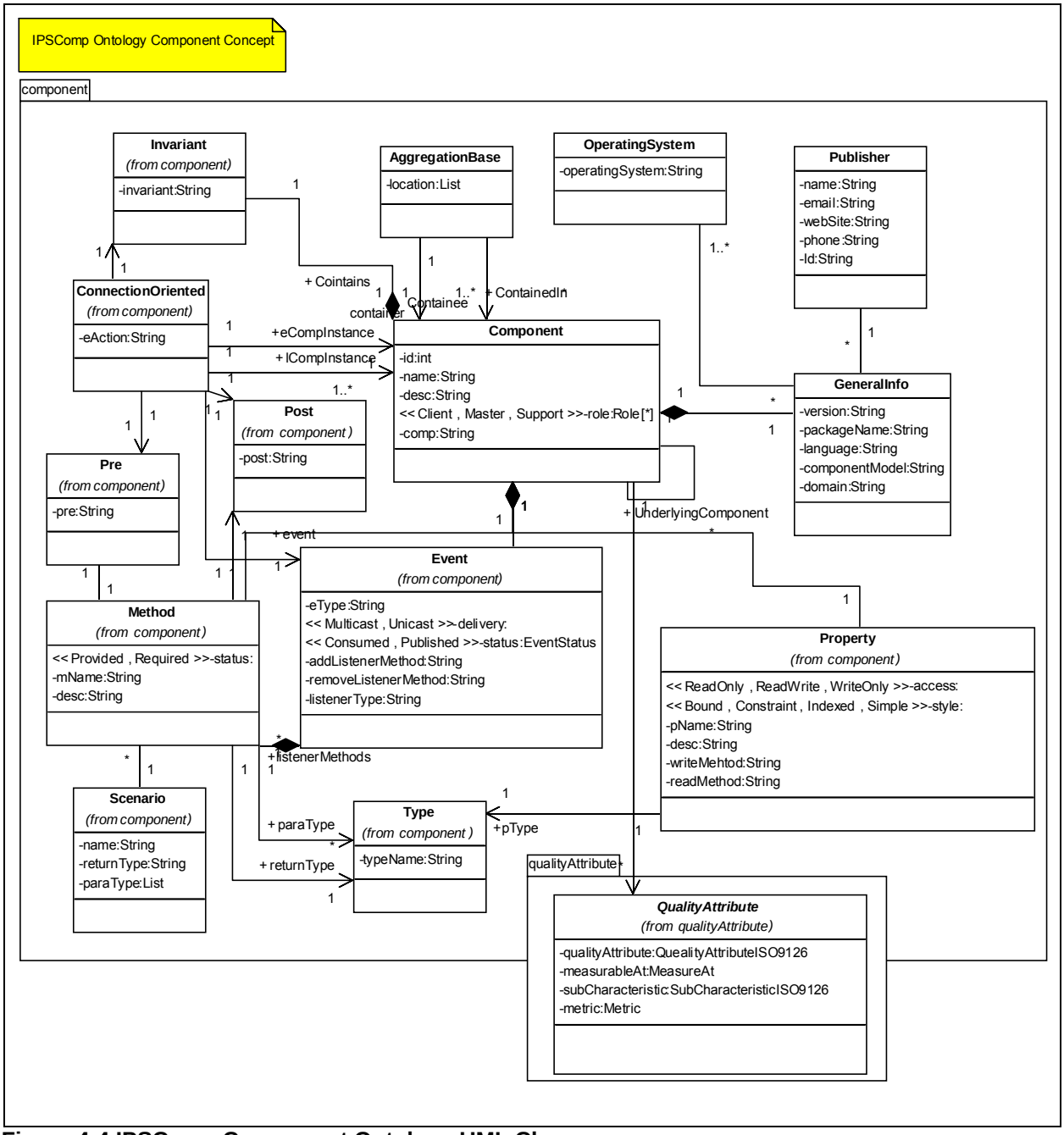


Figure 4-4 IPSComp Component Ontology UML Class

In order to model the Metric concept with the three attributes as explained in section 4.1.2.3 I took the following fact into consideration: The value attribute might have different data types among different metrics, for instance for a presence metric it is stored as a Boolean data type, meanwhile in a time metric it is a float data type.

To handle that behavior and also looking forward to be able to provide a tool for grammatical metric comparison an interface **`IValue`** has been created. This interface defines two abstract methods to manage values (one to get the value the other to set the value). A concrete class to deal with a specific data type has an attribute, which data type is equal to the specific data type the class is willing to handle

and must implement the **`IValue`** interface.

Having explained the **`IValue`** concept, let's consider the Metric concept. In order to model metrics an abstract class **`Metric`** has been created. The three attributes defining a **`Metric`** class **`feature`**, **`value`** and **`unit`** are of type **`IValue`**. This abstract class provides three methods one to handle the setting of each attribute from a **`String`**. It takes advantage that all the class attributes implement the **`IValue`** interface. The UML Class diagram supporting this model is show in Figure 4-5. There is a Metric class Factory **`MetricFactory`** responsible for creating the concrete **`Metric`** classes.

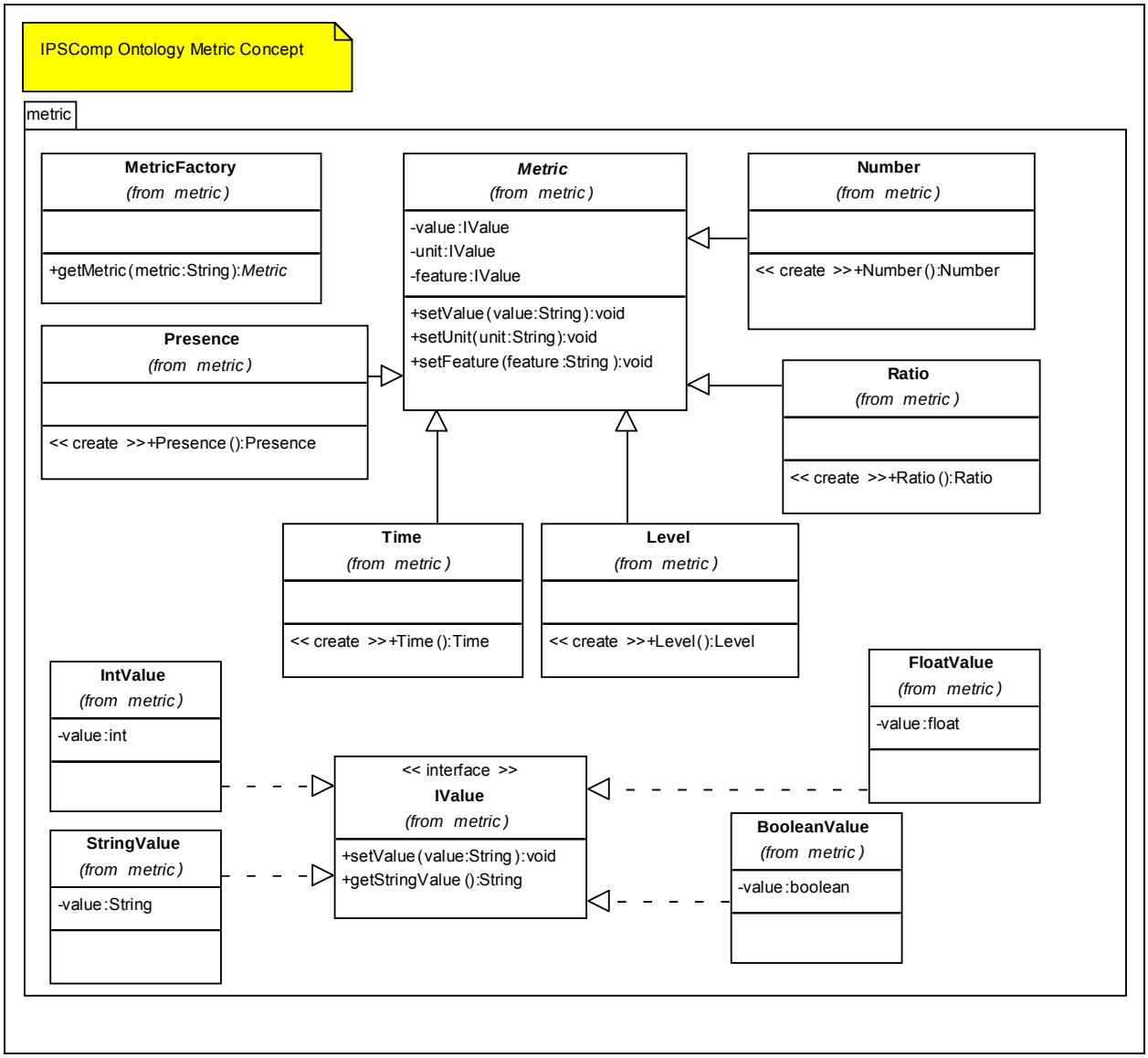


Figure 4-5 IPSComp Metric Concept UML Class Diagram

Turning now to the quality attribute in order to model it, an abstract class **`QualityAttribute`** has been created. This class has 4 attributes. Three of them correspond to the ISO 9126 classification, they store the quality attribute characteristic, sub-characteristic and the instance at which the quality attributes can be measured (Runtime or Life Cycle), which were explained in section 4.1.2.3. The forth attribute is of type Metric; it represents the metric that can be applied to a quality attribute.

To model a concrete quality attribute it is necessary to create a new class that extends the **`QualityAttribute`** abstract class. Figure 4-6 depicts the UML class diagram that models the quality concept for the IPSComp ontology. In the picture only a few quality attributes are shown, but all of them work in the same way. Some classes have been added to model list of values using the enumeration pattern. These classes will hold values for the quality attributes characteristics, sub-characteristics, and measure moment, among others. A factory class is also modeled in order to create concrete quality attribute classes.

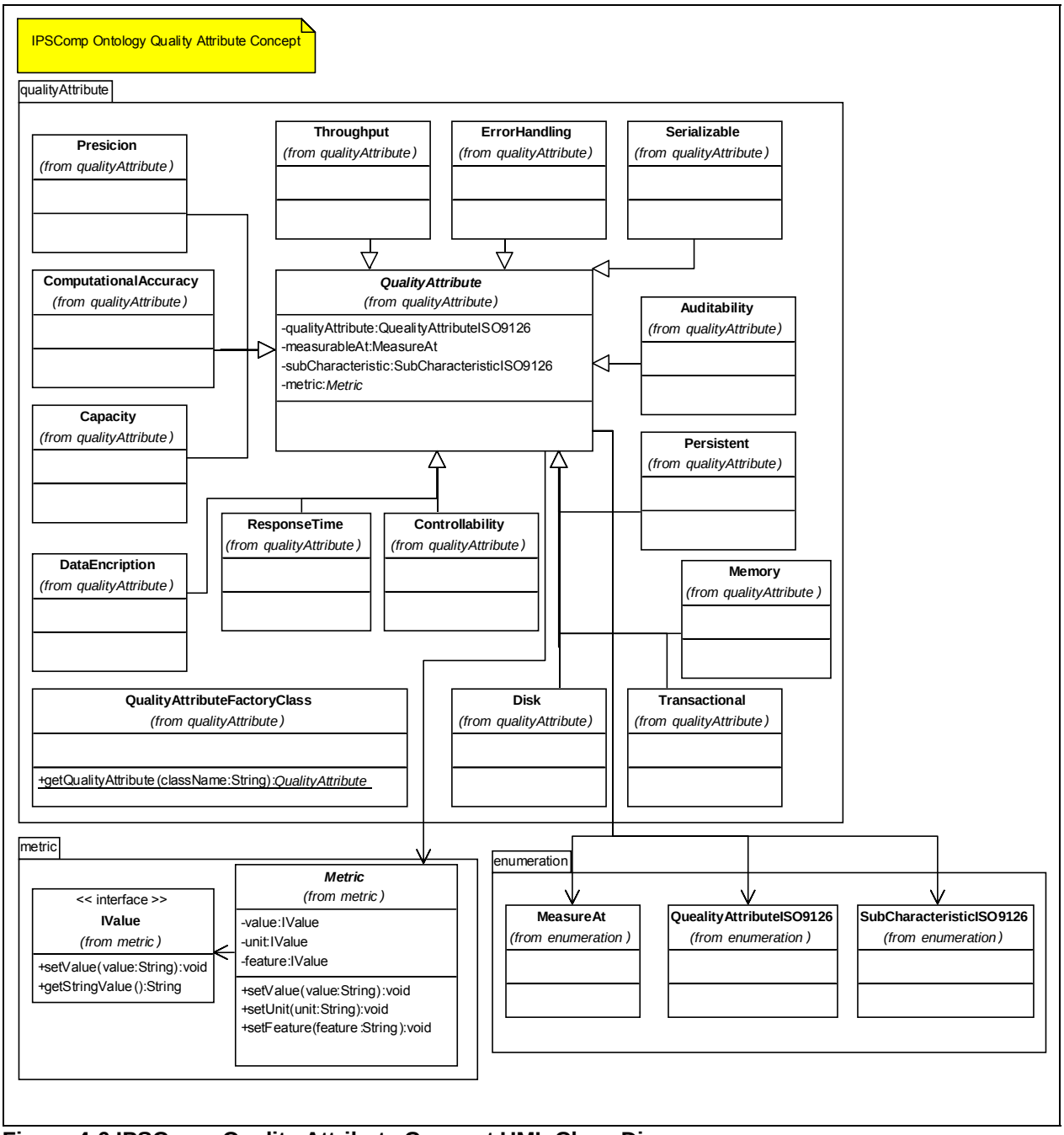


Figure 4-6 IPSComp Quality Attribute Concept UML Class Diagram

The UML class diagrams presented in this section model the IPSComp ontology. The complete UML class

diagram can be found in Appendix A - IPSComp Ontology UML Class Diagram.

4.3 Component Ontology for IPSComp Implementation

In this section three main subjects are explained. First of all, based on the UML model for the IPSComp ontology presented in the previous section, I implemented the java code for it (Section 4.3.1). Second I implemented a Java code generation from a XML file to load component ontology into the java implementation (Section 4.3.2). Finally, based on the component ontology for IPSComp Ontology design presented in the previous section I implemented IPSComp ontology in PLIB [30] which is an ontology manager system based on PLIB specification as explained in section 2.1.4.

4.3.1 Java Code Implementation

The complete UML class diagram (Shown in Appendix A - IPSComp Ontology UML Class Diagram) that models the IPSComp ontology was implemented in Java. Based on the IPSComp ontology design presented in section 4.2 the considerations I took for the implementation are explained in this section.

Regarding the metric concept and I stated that it is defined by three attributes: feature, value and unit (see section 4.1.2.3). Furthermore, I noted that each attribute might have different data types among different metrics, for instance for a **`Presence`** metric the **`value`** attribute is as a Boolean data type, meanwhile in a **`Time`** metric the **`value`** attribute is a float data type.

As shown in Figure 4-5 model I implemented the **`IValue`** interface to handle different data types for the metric attributes. This interface defines two abstract methods to manage values.

- **`public abstract void setValue (String value)`**. In the classes that implement the **`IValue`** interface, this method must set the **`String`** that receives as parameter as the value for a specific data type or object (**`float`**, **`Integer`**, **`String`**, etc).
- **`public abstract String getStringValue ()`**. In the classes that implement the **`IValue`** interface, this method must return the value as a **`String`** data type.

To implement a concrete class to deal with a specific data type has an attribute, which data type is equal to the specific data type the class is willing to handle and must implement the IValue interface. As shown in Table 4-2, the class **`FloatValue`** handles the **`float`** data type. It has an attribute of type **`float`** and the implementation or the methods defined in the **`IValue`** interface. The **`public void setValue(String value)`** method assigns to the **`value`** attribute the **`float`** representation of the **`String`** that receives as parameter. Meanwhile the **`public String getStringValue()`** returns the **`String`** representation of the value attribute which is a **`float`**.

```
public class FloatValue implements IValue {
      private float value;

      public void setValue(String value) {
            Float aFloat;
            aFloat = Float.valueOf(value);
            this.value = aFloat.floatValue();
      }

      public String getStringValue() {
            return String.valueOf(this.value);
      }
```

```
}
```

Table 4-2 Implementation of the `IValue` interface

Afterwards, each concrete class implementing a particular data type should also provide methods to compare values. Each class must be independent and should be responsible of knowing how to be compared. The implementation of such behavior will provide the means to create the metric grammar.

After I implemented the **`IValue`** interface, I implemented the Metric concept. In order to implement metrics I created an **`abstract class Metric`**. The three attributes defining a **`Metric`** class **`feature`**, **`value`** and **`unit`** are of type **`IValue`**. This abstract class provides three methods one to handle the setting of each attribute from a **`String`** (as shown in the snipped code in Table 4-3). It takes advantage that all the class attributes implement the **`IValue`** interface.

```
Public abstract class Metric {
      protected IValue value;
      protected IValue unit;
      protected IValue feature;

      public void setFeature (String feature){
            this.feature.setValue(feature);
      }
      public void setUnit (String unit){
            this.unit.setValue(unit);
      }
      public void setValue (String value){
            this.value.setValue(value);
      }
}
```

Table 4-3 Abstract Class Metric

The class that implements a concrete metric must extend the **`Abstract Metric`** class. In the constructor of the concrete class the specific value type for the metric attributes, which is a class, implementing the **`IValue`** interface must be specified. For instance, to implement the Time metric the **`feature`** attribute must be a **`StringValue`** class, the **`value`** attribute must be a **`FloatValue`** and the **`unit`** attribute must be a **`FloatValue`**, as observed in Table 4-4.

```
Public class Time extends Metric {
      public Time() {
            this.feature = new StringValue();
            this.value = new FloatValue();
            this.unit = new StringValue();
      }
}
```

Table 4-4 Concrete Metric Class

Additionally I added to the metric concept implementation a Factory Pattern combined with an Enumeration Pattern. The Enumeration holds the different metrics names defined for the component domain (presence, level, time, ratio, number, which are explained in section 4.1.2.3). The **`MetricFactory`** class receives a metric name and returns an instance of the concrete class representing the desired metric.

Turning now to the quality attribute implementation, I created an abstract class **`QualityAttribute`**. This class has 4 attributes (Table 4-5). Three of them correspond to the ISO 9126 classification, they store the quality attribute characteristic, sub-characteristic and the instance at which the quality attributes can be

measured (Runtime or Life Cycle). The forth attribute is of type **`Metric`**; it represents the metric that can be applied to the quality attribute.

```
Public abstract class QualityAttribute {
      protected QualityAttributeISO9126 qualityAttribute;
      protected MeasureAt measurableAt;
      protected SubCharacteristicISO9126 subCharacteristic;
      protected Metric metric;
      .
      .
}
```

Table 4-5 Abstract Class `QualityAttribute`

In order to implement a concrete quality attribute it is necessary to create a new class that extends the abstract class **`QualityAttribute`**. In the constructor of the concrete class the specific Metric has to be initialized as shown in Table 4-6, were the **`Auditability`** quality attribute uses a **`Presence`** metric to be measured.

```
Public class Auditability extends QualityAttribute {
      public Auditability() {
            this.qualityAttribute = QualityAttributeISO9126.FUNCTIONALITY;
            this.subCharacteristic = SubCharacteristicISO9126.SECURITY;
            this.metric = new Presence();
            this.determineMeasurableAt();
      }
}
```

Table 4-6 Concrete Quality Attribute Class

I added some classes to the implementation to define list of values following the enumerate pattern. These classes will hold values for the quality attributes characteristics, sub-characteristics, and measure moment, among others. A factory class pattern is implemented in order to create concrete quality attribute classes. A concrete class for each quality attribute must be implemented, but the structure is simple it has a constructor that initializes its ISO 9126 classification as well as the concrete metric that has be defined to handle the quality attribute measure.

I have provided the Java implementation for the IPSComp ontology. This implementation can be extended to include a grammar for the quality attributes as well as include new quality attributes when needed. It is necessary to monitor the IPSComp ontology to incorporate changes as the domain evolves.

4.3.2 IPSComp Java Code Generation from a XML file

The IPSComp ontology has a XML representation. In order to load a component which is an instance of this ontology represented by an XML file I am using the JDOM[6] parser. I implemented the module responsible for loading a component description into the Java implementation ontology model with a visitor pattern. This pattern allows creating different visitors depending on the task that must be performed on the element that belongs to the collection. I had into account two extra considerations:

- The visitor pattern is able to act in a specific way according to the element it is visiting. But for this case when the JDOM parser traverses the XML file all the nodes are of the same type, so base on the XML tag that is being visited a class will be created. This class will represent an explicit XML tag. To implement this class instantiation process I implemented a factory pattern. It works as follows a Java

[6] JDOM http://www.jdom.org

class has been created for each XML tag it is worth it to process. The **`XMLFactory`** class receives the XML tag name and according to it, it creates a class that handles the component description instantiations retrieving the right values from the xml file.

- To allow easy modification in the XML file, and also to handle different XML files the visitor pattern is implemented using reflection. The class **`MethodFinder`** provides means to handle the reflection in order to bind to the correct visit method at run time.

A part of the UML representation of the model is shown in Figure 4-7. This figure only depicts one specific visitor and shows only some of the classes implemented a few XML tags, but the whole idea can be inferred from there because the mechanism behind the other tags is exactly the same.

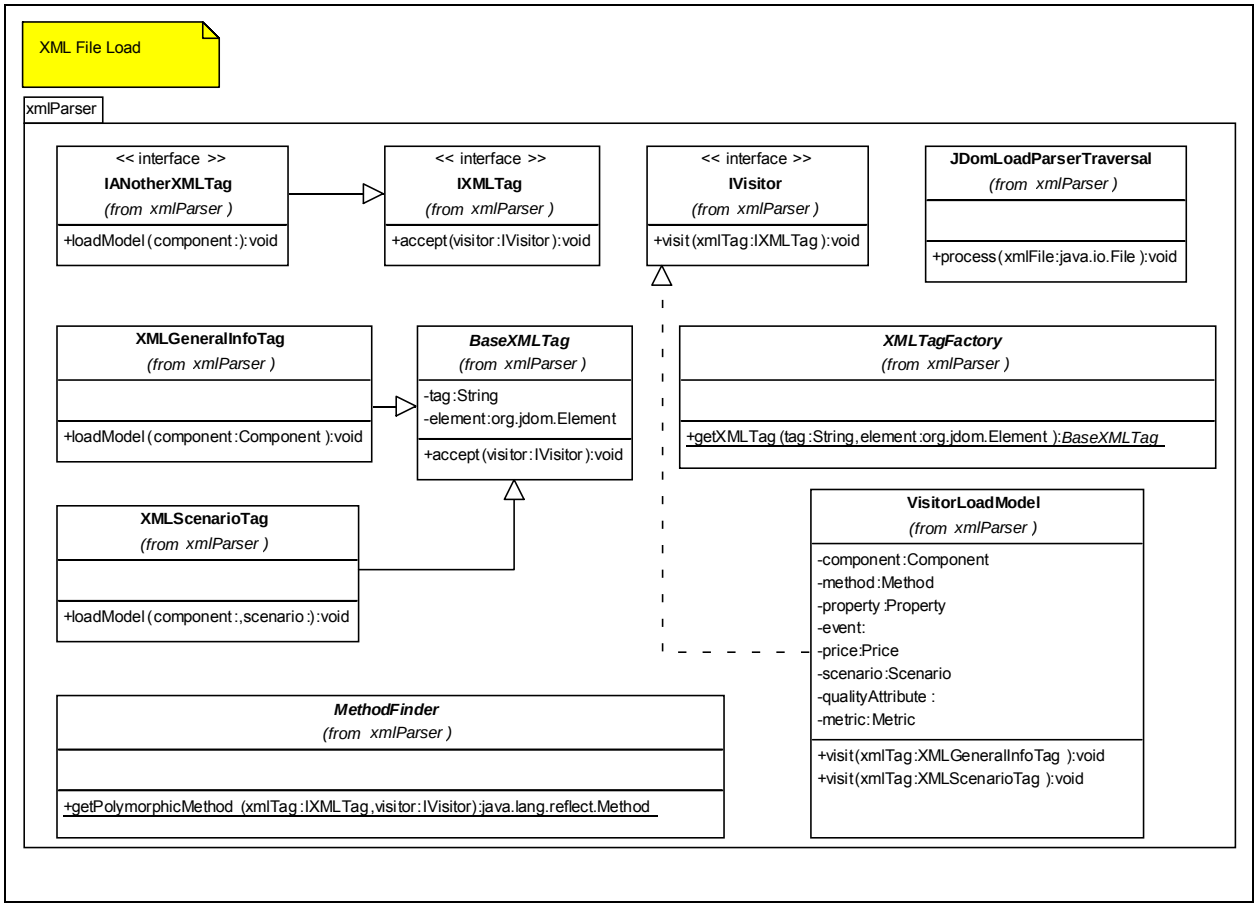


Figure 4-7 Visitor Pattern with Reflection to load a XML file – UML Class Diagram

4.3.3 IPSComp Ontology Implementation PLIB

Once the IPSComp ontology has been defined, it has been included into PLIB. The idea behind this point was to test the inclusion of the IPSComp ontology into an ontology manager system, and to join the ontology manager system with the Java implementation. PLIB provides a Java API to interact with the ontologies defined in it. There was not a specific reason to use PLIB, actually it was available, but later in the project we realized that it was not completely tested so we could introduce the IPSComp ontology in PLIB but could no finalize the test. This section describes the details of the test.

The data model used to describe ontologies in PLIB is an OO data model known as the PLIB data model.

Thus, and in order to comply with the 6-tuple ontology based definition (Section 2.1.4), the first step is to define a set of classes (C) all gathered in a classification hierarchy. For that purpose (and according to the underlying PLIB data model) PLIBEditor makes possible to describe each class on the basis of four classes' categories:

- Item: It enables the modeling of any type of entity of the application domain that corresponds to an autonomous and stand-alone abstraction as a class. It is a super type intended to be sub-typed to define the nature of the objects. Nevertheless, it is not defined as ABSTRACT to enable its instantiation to model the classes that are super-classes of two classes corresponding to two different kinds of objects (e.g., components and materials).
- Component: It captures the dictionary description of a class of items that represent, at some level of abstraction, parts or components. A property of which the data type is defined by a component_class stands for the aggregation relationship.
- Material: It captures the dictionary description of a class of materials. Materials are used to define properties of parts or components. Materials are associated with an idea of amount, they may not be counted. A property of which the data type is defined by a material_class captures that some (part of a) product is made of, or contains, some material.
- Feature: It captures the dictionary description of items that represent one aspect of another item and that are themselves associated with properties.

All these definitions come from ISO13584-42 and ISO13584-24. In the next ISO13584 release, it is expected that all this stuff will be simplified and only one category will remain items. PLIB tools will be updated consequently.

In PLIB Editor, these categories appear explicitly through category containers. For the component ontology description, it was necessary only to focus on the "items" category, and then deploy a classification hierarchy under this particular category from the UML class diagram that defines the IPSComp ontology.

While introducing the ontology in PLIB it was necessary to define lists. This fact will be illustrated with an example. The component IPSComp ontology has a class Type that represents the user defined data types and primitive data types (Table 4-7).

On the other hand the class **`Method`** models an interface belonging to a component. This method has as internal collaborators (attributes) a return type, a method name and a list of parameters. These attributes will represent the method signature (Table 4-7).

The attribute **`parametersType`** is a list of instances of the **`Type`** class. PLIB Editor handles this property data type as aggregate data types. Unfortunately the aggregate type definition has not been tested very intensively, but at the minimum, it is possible to find all the instances required for describing an aggregate structure in the physical file.

```
Class Type {
   String dataTypeName;
}

Class Method{
   Type returnType;
   String methodName;
   ArrayList parametersType;
}
```

Table 4-7 Collections in the Component Ontology

Figure 4-8 shows the inclusion of the IPSComp ontology in PLIB, and it illustrates the example explained in the previous paragraphs. The Left panel shows the concepts included in the ontology. The specific

example is related with the Method concept (It is red highlighted in the picture). The list of properties for the method includes a paraType which is the list of parameters (It is green highlighted in the picture). It shows it is a list, and the element type the list contains is another concept in the ontology, the Type concept (It is yellow highlighted in the picture).

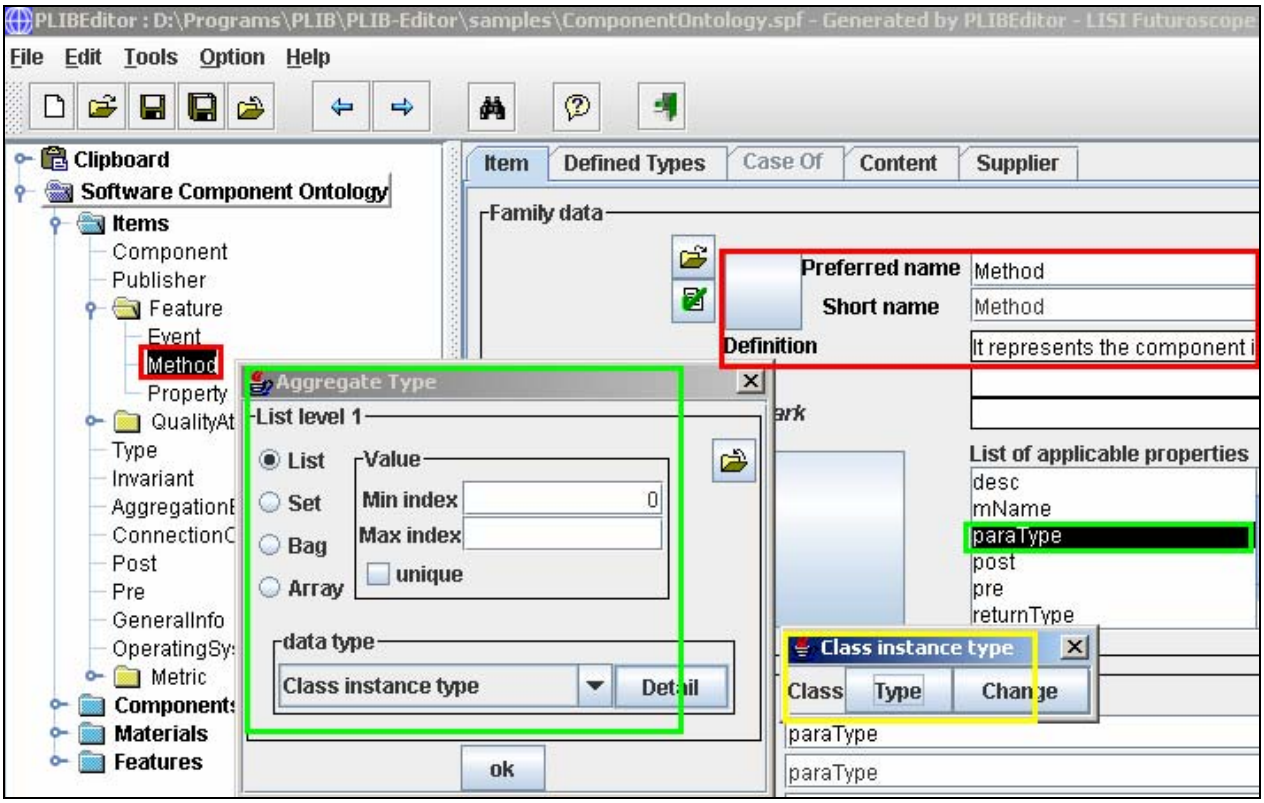


Figure 4-8 PLIB Editor IPSComp Ontology – Screen Shot

Once the IPSComp ontology has been included the next step is to describe instances of the given ontology. PLIB Editor is currently being improved (not sure that the aggregate data type is supported at the content level). As soon as a new stable release (maybe with some restrictions) will be available, this can be tested.

Concluding, it was possible to include the IPSComp ontology into an ontology management tool in this case PLIB it indicates that the ontology can be handle by the system. Because it was not possible to instantiate an instance of the IPSComp ontology, the test regarding the connection through its Java API with the Java implementation could not be performed. Nevertheless, PLIB allows the integration of already defined ontologies; the idea is that in such system the related domain ontologies should be specified in order to accomplish the IPSComp specification goals that pursues to relate the component ontology with other domain ontologies to create a market place for software components.

4.4 Integrating Software Component Repositories

It is important to highlight that nowadays there are some component repositories already developed available through the web, and in the IPSComp project scope it will be really helpful to find means to incorporate or interact with those existing repositories and not only develop and post new components in our IPSComp platform. This issue will be undertaken with a Model Driven Architecture (MDA) perspective. Before going into the detail of the software component repositories integration, it is necessary to provide

some terminology that is used throughout the chapter.

- **Component Repository**: It is a repository that stores software components. In such repository each software component has a description associated to it. There are Vendor Repositories (Item {2} Figure 4-9) and an **IPSComp Repository** (Item {1} Figure 4-9). All of them are compliant in the definition.
- **IPSComp Component Description** (Item {3} Figure 4-9): It is the textual description of a component in the IPSComp repository. The IPSComp Component Description is compliant to the IPSComp ontology described in section 4.1.
- **Vendor Component Description** (Item {4} Figure 4-9): It is the textual description that a vendor provides for any component included in the vendor repository.
- **IPSComp Component Description Meta-Model** (Item {5} Figure 4-9): It is the UML model representing the IPSComp ontology.
- **Essence Component Description** (Item {17} Figure 4-9): It is the textual component description for a component essence description.
- **Vendor Component Description Meta-Model** (Item {6} Figure 4-9): It is the UML model for the Vendor Component Description.
- **Essence Component Description Meta-Model** (Item {15} Figure 4-9): It is the UML model for the Essence Component Description.
- **IPSComp Java Implementation** (Item {7} Figure 4-9): It is the Java program that implements the representation of an IPSComp compliant component description based on the IPSComp Component Description Meta-Model.
- **Vendor Java Implementation** (Item {8} Figure 4-9): It is the java program that implements the representation of a compliant Vendor Component Description Meta-Model.
- **Essence Java Implementation** (Item {16} Figure 4-9): It is the java program that implements the representation of a compliant Essence Component Description Meta-Model.
- **IPSComp Transformation API** (Item {9} Figure 4-9): It is the java implementation that provides the means (methods) to create and populate an IPSComp Component Description compliant to the IPSComp ontology. It is a jar file that a vendor has to import in the Vendor Framework in order to integrate with the IPSComp Framework.
- **Vendor Framework** (Item {10} Figure 4-9): It is the java program that implements the representation of the Vendor Component Description Meta-Model adding the IPSComp Transformation API.
- **IPSComp Framework** (Item {11} Figure 4-9): It is the java program that implements the representation of the IPSComp Component Description Meta-Model adding the IPSComp Transformation API.
- **IPSComp XML Model** (Item {12} Figure 4-9): It is the .xsd file which defines the XML schema Definition for the IPSComp Ontology.
- **IPSComp XML Component Description** (Item {13} Figure 4-9): It is an instance[7] of IPSComp XML Meta-Model. This means that the IPSComp XML Component Description is an xml file that conforms to the Scheme defined by the IPSComp XML Component Description.
- **IPSComp xml file parser** (Item {14} Figure 4-9): It is the executable java program that allows transforming an IPSComp XML Component Description into an IPSComp Java Implementation in the IPSComp Framework (Section 4.3.2).

Additionally in Figure 4-9 there are three different colored arrows, the colors means:
Blue: From the element at the origin it is possible to instantiate an element at the end.
Red: From the element at the origin it is possible implement (code generation) an element at the end.
Green: It is possible to apply a model transformation from the origin to the end.

Finally, the picture is divided in two layers; the base level has concrete information of the components stored in different repositories. The elements in that layer represent a real component description. It holds

[7] According to the W3 Org the purpose of a schema is to define a class of XML documents, and so the term "instance document" is often used to describe an XML document that conforms to a particular schema [74].

the model. On the other hand the upper level contains the Meta-Models, which are the models for the models present in the information layer.

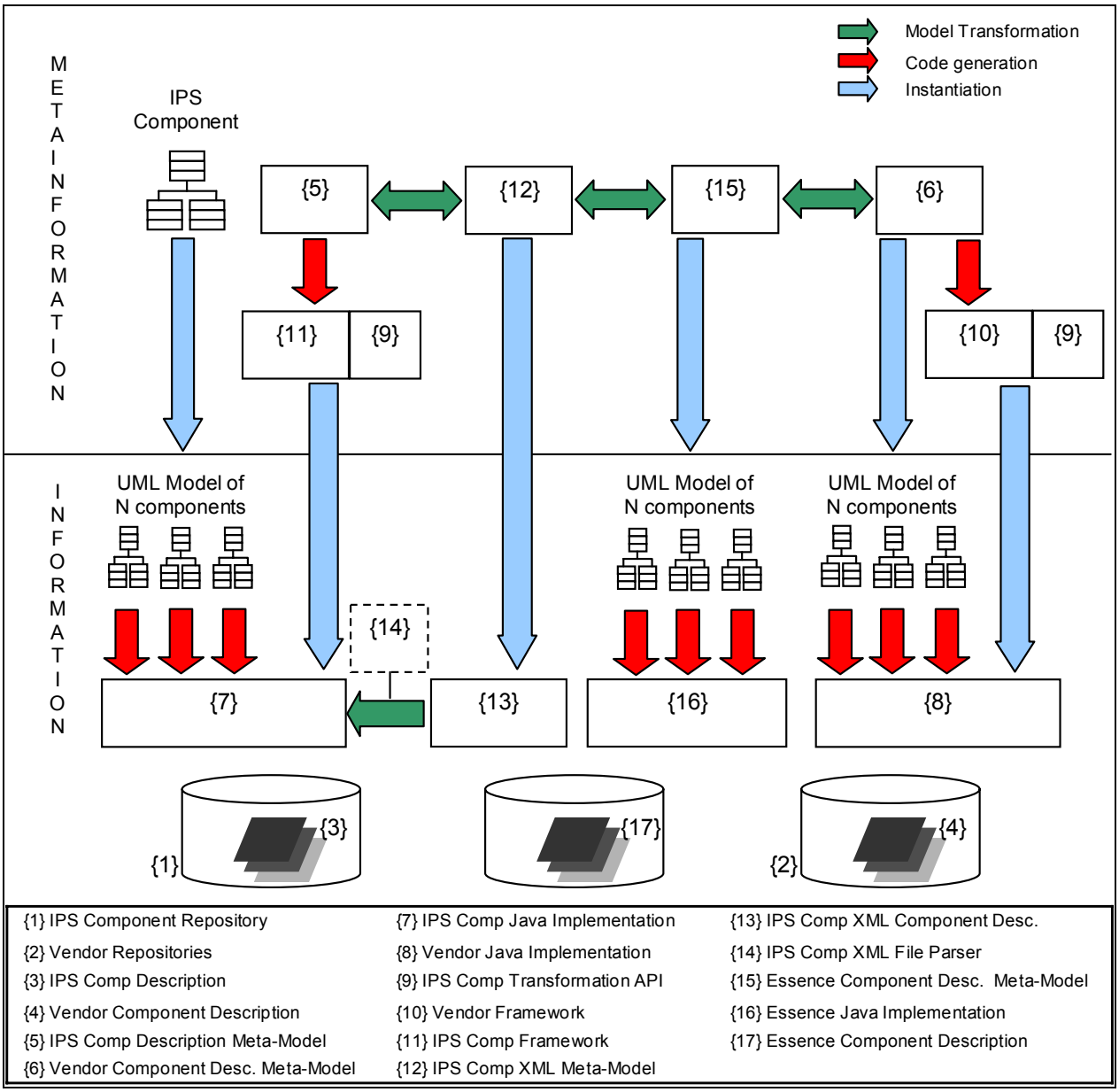


Figure 4-9 Elements for Software Component Repository Integration

The terminology defined in the previous paragraphs and depicted in Figure 4-9 is going to be used throughout the chapter. I will provide a specific example of some elements intended to clarify the explanation. In Figure 4-10 there are text boxes containing a number and a description. The number is the same number used in Figure 4-9 for the element, the description is a concrete example. For instance, items {5} (Appendix A - IPSComp Ontology UML Class Diagram), {6} (Figure 4-12 Figure 4-13 Figure 4-14) and {15} (Figure 4-15) are UML diagrams representing the Meta-Models. Item {2} is web *Vendor Repository* and item {4} is a *Vendor Component Description* in a commercial web site (Figure 4-11). Item {12} is the .xsd file schema that defines the Meta-Model (Table 4-9 or Appendix F - IPSComp XML Meta-Model - XSD Schema). Item {13} is an xml file instance of the xsd schema (Table 4-10 or Appendix G - IPSComp Component Description – XML Example).

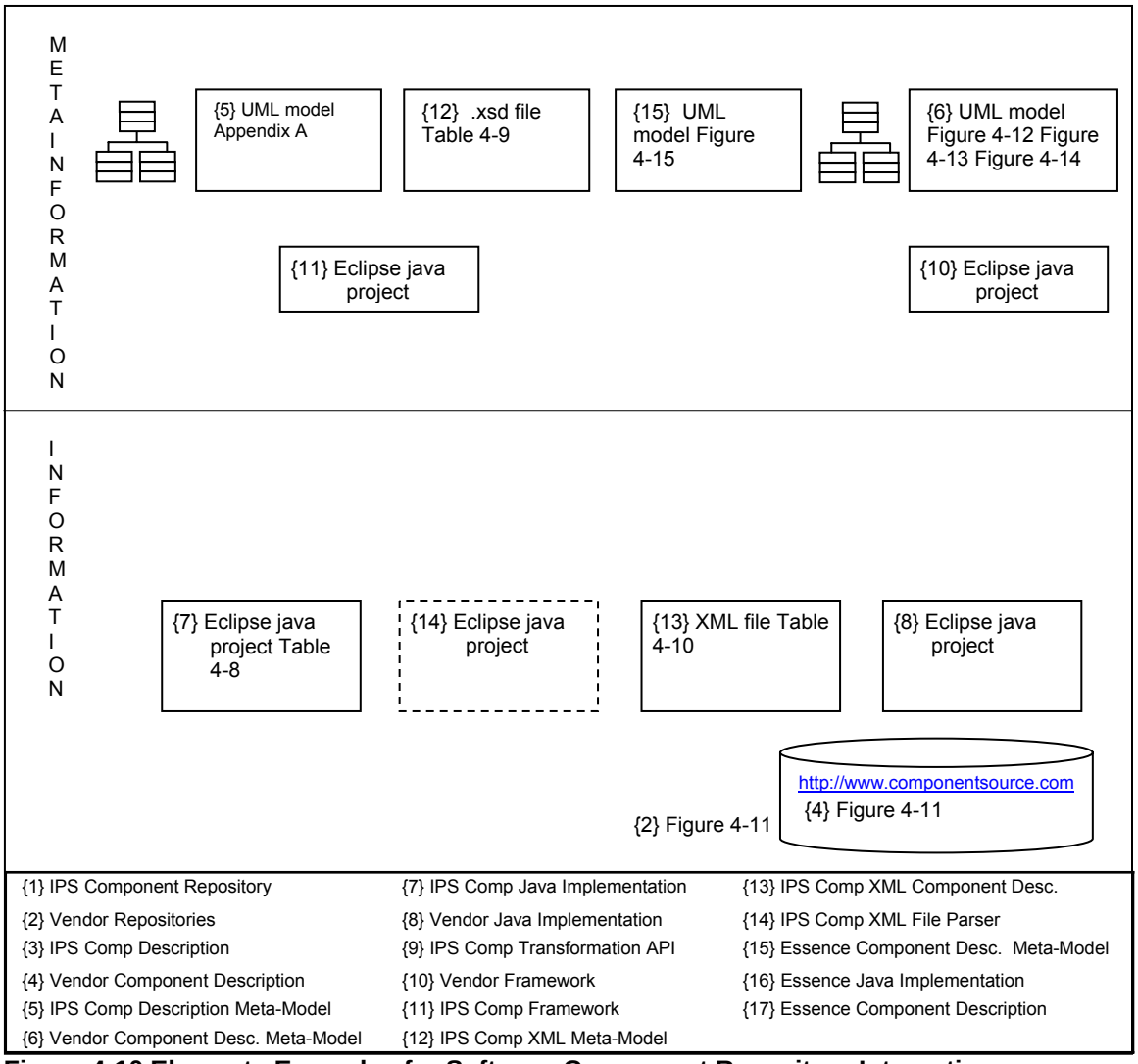


Figure 4-10 Elements Examples for Software Component Repository Integration

```
public class Component {
      private String id;
      private String name;
      private String desc;
      private GeneralInfo generalInfo;
      private Role role;
      private String comp;
      private List properties;
      private List events;
      private List methods;
      private List qualityAttributes;
      public Component(String id, String name, String desc, Role role,
                  String comp, GeneralInfo generalInfo) {
      ……
}
```

Table 4-8 IPSComp Java Implementation Example Item {7}

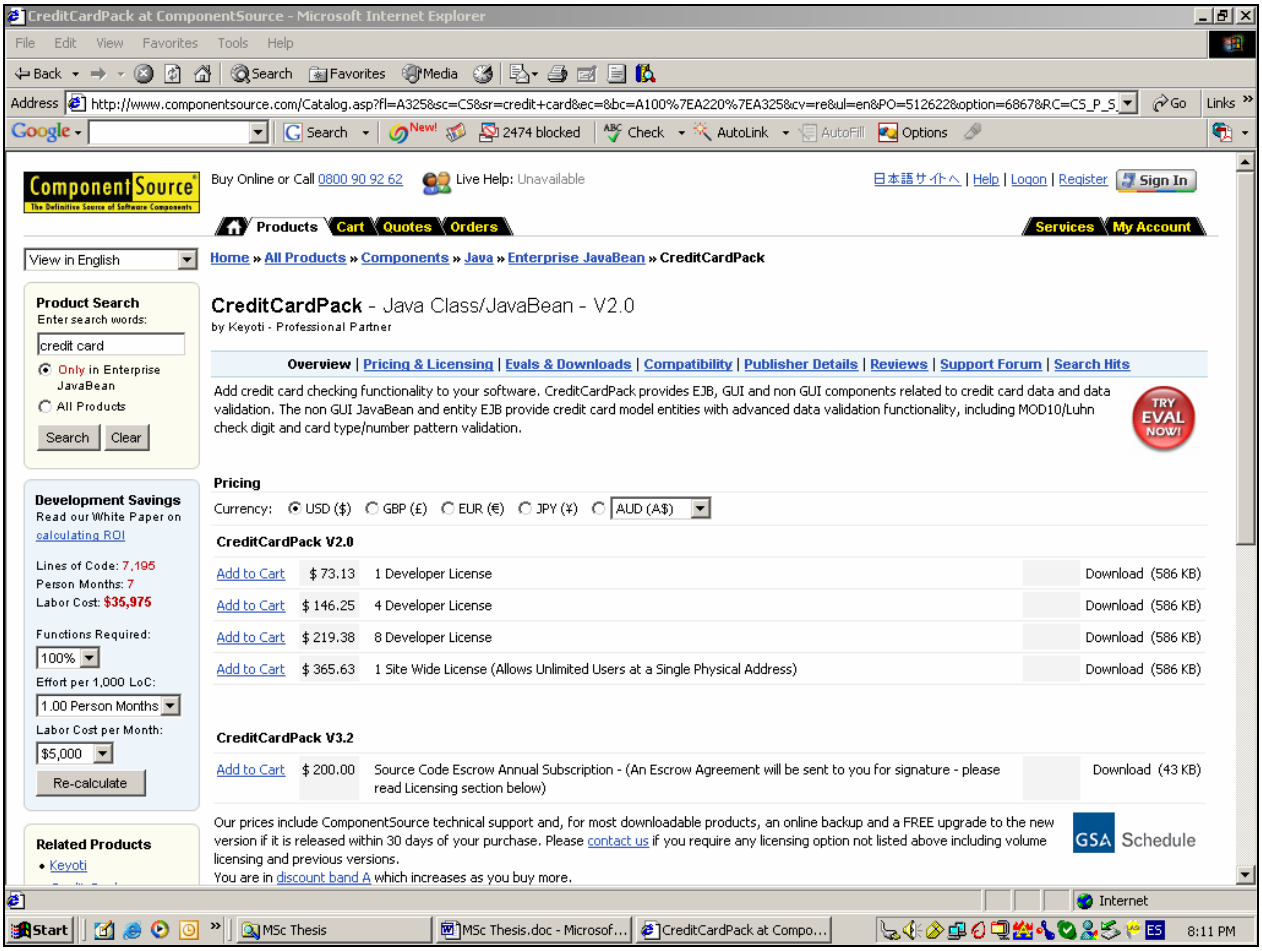


Figure 4-11 Vendor Respository {2} – Vendor Component Description {4} Example

```
      <xs:element name="componentSpecification">
            <xs:complexType>
                  <xs:sequence>
                        <xs:element ref="id"/>
                        <xs:element ref="name"/>
                        <xs:element ref="generalInfo"/>
                        <xs:element ref="features"/>
                        <xs:element ref="design"/>
                        <xs:element ref="qualityAttributes"/>
                  </xs:sequence>
            </xs:complexType>
      </xs:element>
      ...
      ...
      ...
```

Table 4-9 IPSComp XML Meta-Model - XSD Schema Example {12}

```
<componentSpecification  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:noNamespaceSchemaLocation="C:\EMOOSE\IPSComp.xsd">
      <id>111111</id>
      <name>javax.composite.SliderFieldPanel</name>
      <generalInfo>
```

```
            <version>Jaaava</version>
            <package>javax.composite </package>
            <language>Java </language>
            <model>JavaBean</model>
            <domain>Interface</domain>
            <domain>MVC</domain>
            <os>Windows</os>
            <os>Linux </os>
      ...
      ...
      ...
```

Table 4-10 IPSComp Component Description – XML Example {13}

As stated in the MDA specification web site [64]: "The MDA is a new way of developing applications and writing specifications, based on a platform-independent model (PIM) of the application or specification's business functionality and behavior. A complete MDA specification consists of a definitive platform-independent base model, plus one or more platform-specific models (PSM) and sets of interface definitions, each describing how the base model is implemented on a different middleware platform. A complete MDA application consists of a definitive PIM, plus one or more PSMs and complete implementations, one on each platform that the application developer decides to support."

The aim pursued by using the MDA view is to be able to accomplish transformation from one or more selected existing *Vendor Java Implementations* into our *IPSComp Java Implementation* which is IPSComp ontology compliant. The *Vendor Component Description Meta-Model* describes existing components from *Vendor Repositories*. The *IPSComp Component Description Meta Model* is our component description ontology presented in section 4.2.

A Walkthrough the Research Process:

First of all, in order to apply the MDA transformation it is necessary to obtain the different meta-models. The IPSComp ontology has an UML class diagram representation (section 4.2), the *IPSComp Component Description Meta-Model*. Then it is necessary to find out *Vendor Component Description Meta-Models*. I accomplished such a task by browsing the web. A couple of web sites, which sell software components (*Vendor Repositories*), were selected. **These web sites neither provide the *Vendor Component Description Meta-Model* nor the UML model**. I created a *Vendor Component Description Meta-Model* that supports the component description in different web sites (*Vendor Repositories)*. This task was performed for three *Vendor Repositories*: http://www.componentsource.com Figure 4-12, http://devcatalog.com Figure 4-13 and http://www.ecots.org Figure 4-14. Those figures depict the UML class diagram *Vendor Component Description Meta-Model* for *Vendor Repositories*. This is an inferred *Vendor Component Description Meta-Model*, but it leads to follow the idea behind the contribution.

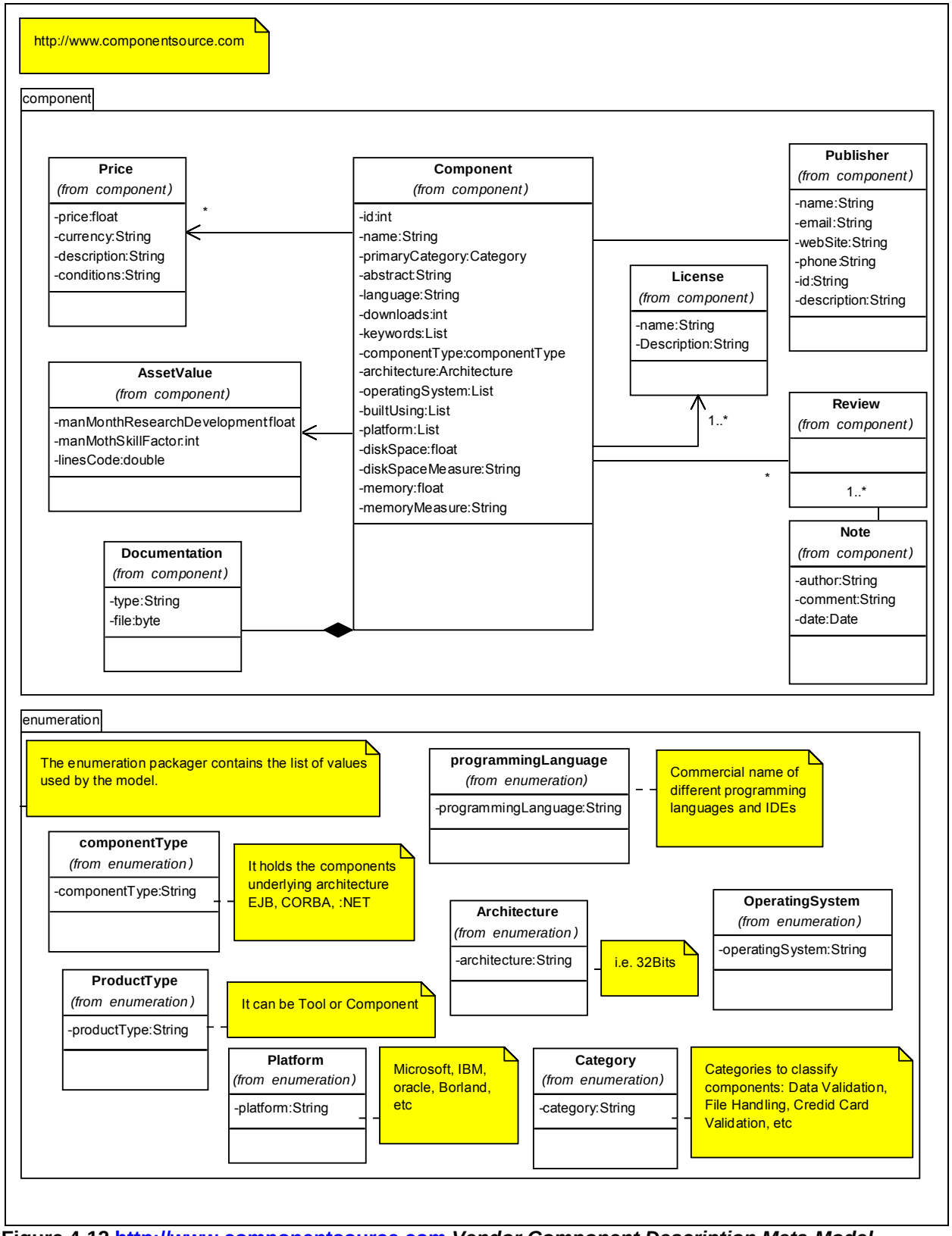


Figure 4-12 http://www.componentsource.com *Vendor Component Description Meta-Model*

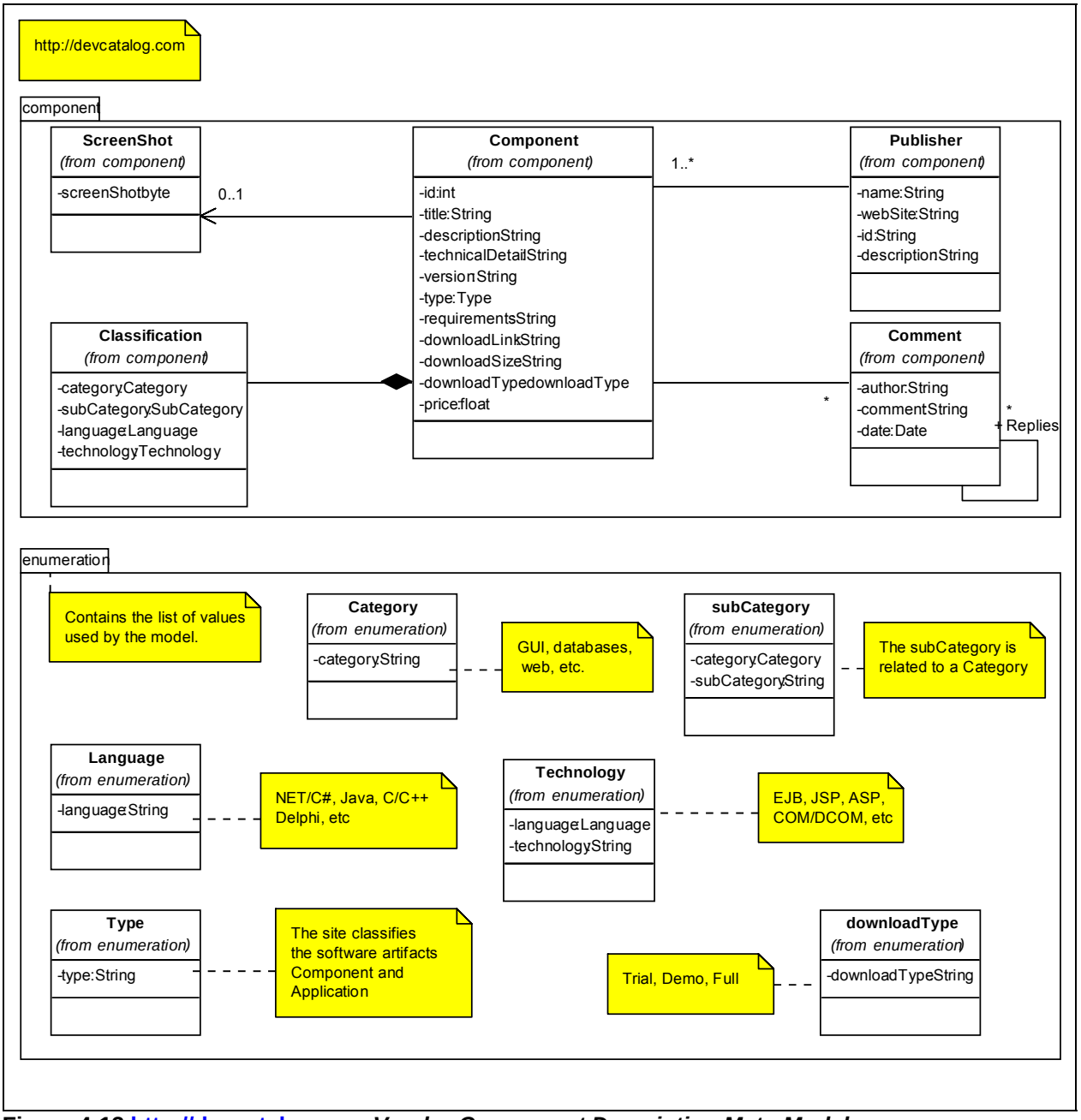


Figure 4-13 http://devcatalog.com *Vendor Component Description Meta-Model*

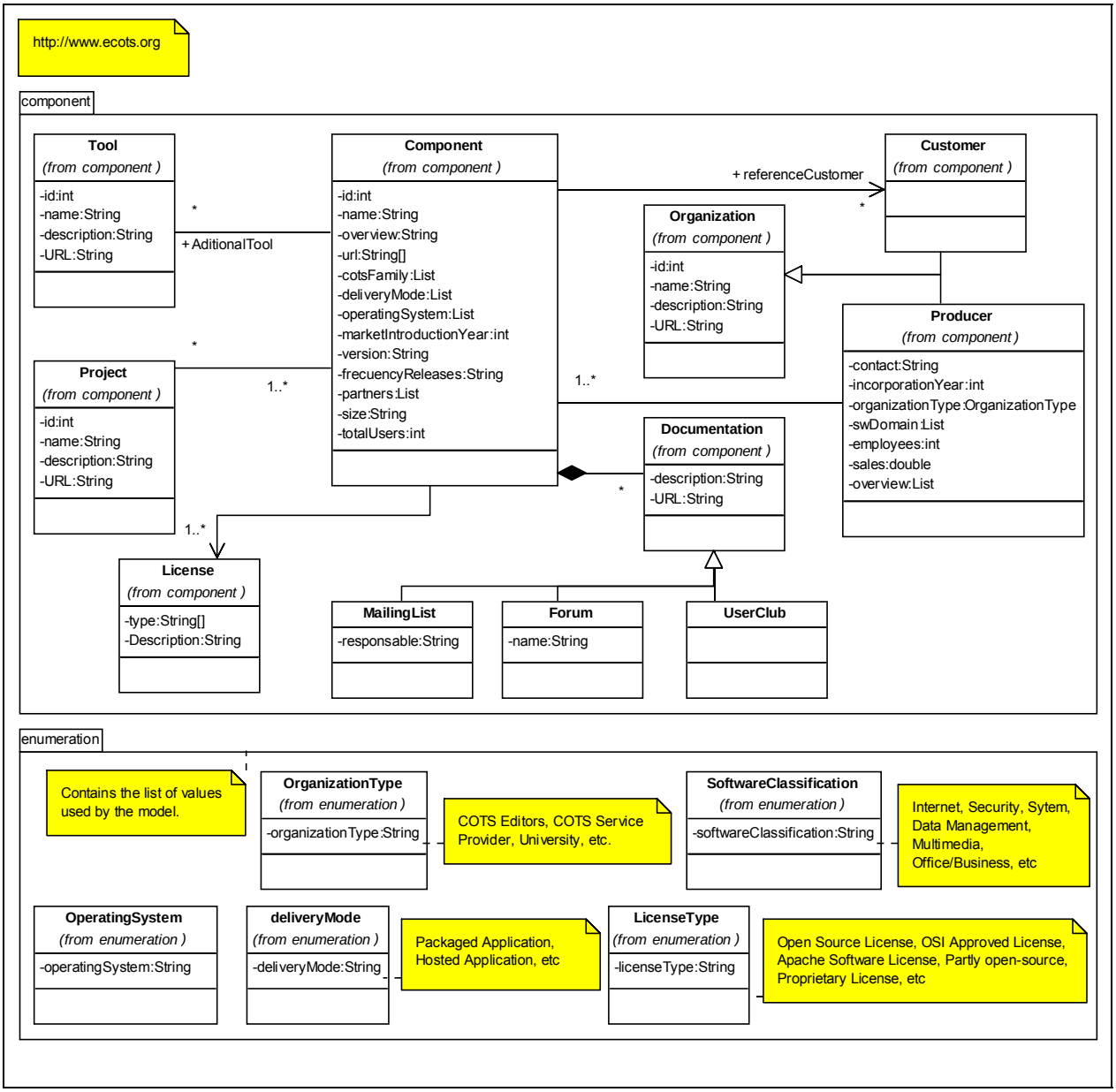


Figure 4-14 http://www.ecots.org *Vendor Component Description Meta-Model*

Secondly, once the *Vendor Repositories* were searched and the different *Vendor Component Description Meta-Model* created, it was necessary to compare those meta-models between them and with the *IPSComp Component Description Meta-Model* in order to identify commonalties and differences. Actually the aim behind this task was to come up with an intermediate model. Such a model will contain those elements that are essential for a component description (*Essence Component Description Meta-Model*), in the domain chosen, which is the commercial web sites for component repositories. These commercial web sites selling software components produce a components market.

The *Vendor Repositories* selected provide more or less the same sort of information for the components stored in them. On the first hand they have a set of software components and a set of producers. For the producers they display the name, contact information (web site, email) and a brief producer description. As far as the software component is concerned, the component has a name, a textual description, some technical specification regarding the architecture on which it is built up, the programming language, the

operating system, a classification based on a set of keywords, a producer, a price, some of them have license information. Based on that information those web sites promote the description and marketing of the goods offered by them, software components.

These *Vendor Repositories* also provide the same means for searching. The most common techniques these *Vendor Repositories* provide for searching are keyword-based and browsing (explained in section 2.2). They might have an ontology, a taxonomy or a controlled vocabulary to classify components. Another characteristic found (explicitly documented in http://www.componentsource.com) is that the search is not case sensitive, but on the other hand it will present different results if the words included in the search are in singular or plural. After the comparison between the different *Vendor Component Description Meta-Models* and the *IPSComp Component Description Meta-Model* the *Essence Component Description Meta-Model* created is also modeled as an UML class diagram, and shown in Figure 4-15.

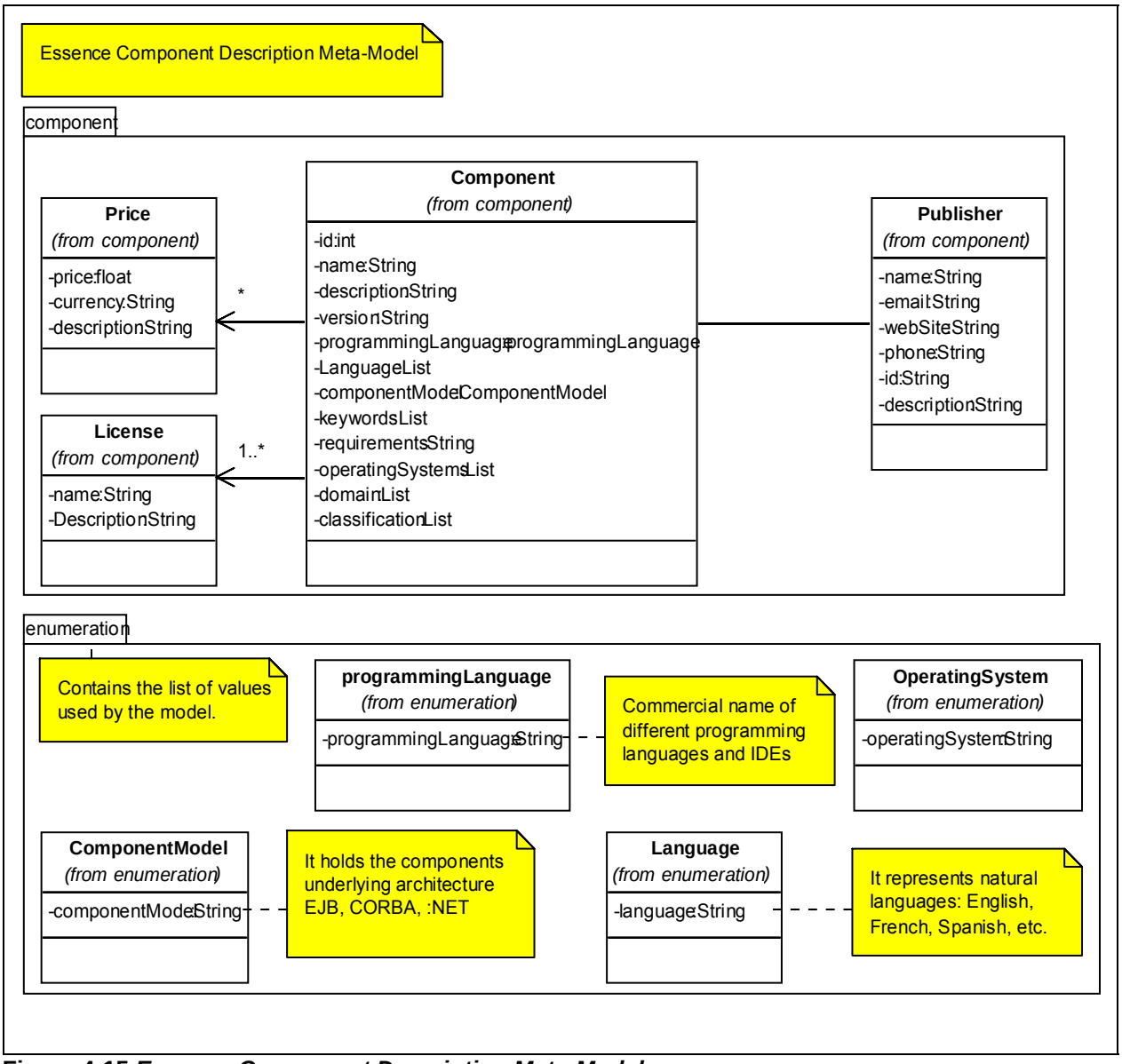


Figure 4-15 *Essence Component Description Meta-Model*

Taking into account the MDA description it is possible to state that so far there are five models. Three of

them correspond to each one of the three *Vendor Component Description Meta-Models*. The fourth one is the *IPSComp Component Description Meta-Model*. The fifth one is the *Essence Component Description Meta-Model* that was created by comparing the other four models. The idea is to be able to translate a *Vendor Java Implementation* (Item {8} Figure 4-9) into the *IPSComp Java Implementation* (Item {7} Figure 4-9). The first thing that has come out from the analysis is that there is information which should be included in the IPSComp ontology such as license, price and publisher description. Refer to section 4.1.2 to see in detail what we have added to the IPSComp Ontology.

On the other hand, having a look at the meta-models, most of the fields have been represented as string data types. Basically the idea is that the transformation will behave like a mapping, in which the attributes of one meta-model will be translated to the attributes of the Essence meta-model. This can be achieved by an API. The *IPSComp Transformation API* provides a method to create an *Essence Java Implementation*. Then by means of the methods included in the *IPSComp Transformation API* the user must populate the component description. Because not all the fields present in the *Vendor Component Description* are in the *Essence Component Description* some fields will be left out outside.

I have implemented a prototype as part of the research, the IPSComp Prototype. The IPSComp prototype includes the *IPSComp Framework* and the *Vendor Framework* for the *Vendor Repository* http://www.componentsource.com. In the IPSComp prototype the *IPSComp Framework* has a class **`GenericRepository`** that represents a *Component Repository*. In the real project implementation this most probably would be implemented in a database. The **`GenericRepository`** class implements the singleton pattern to simulate the back end storage device. In the IPSComp prototype this class provides the *IPSCopmp Transformation API* methods to create or retrieve an *Essence Java Implementation*. It will manage a component id; it will be composed from the id the component has in the *Vendor Repository* plus the *Vendor Repository* name. The *Essence Java Implementation* is implemented in the *IPSComp Transformation API* by the **`GenericComponent`** class.

Scenario Model Transformation using the *IPSComp Transformation API*.

Actor: Software component repositories integrator. This scenario allows the actor to perform a transformation from a *Vendor Java Implementation* to an *IPSCompt Java Implementation*. To accomplish a complete transformation the actor must execute five steps, explained in this section:

- Step 1: Create the *Essence Java Implementation*.
- Step 2: Populate the information in the *Essence Java Implementation*.
- Step 3: Create the *IPSComp Java Implementation*.
- Step 4: Add information from the *Vendor Java Implementation* to the *IPSComp Java Implementation*.
- Step 5: Add information to the *IPSComp Java Implementation* from a component developed in a specific platform (EJB, .NET, CORBA).

A software component repositories integrator can perform a transformation from a *Vendor Java Implementation* to an *IPSCOMP Java Implementation* using the *IPSComp Transformation API*. Table 4-11 shows an example of how the actor accomplishes steps 1, 2 and 3.

First the software component repositories integrator implements a method, in the sniped code the method **`transformToComponentEsscence`**. The method receives a *Vendor Java Implementation* (**`ComponentSource`**) as parameter and it returns an *Essence Java Implementation* (**`GenericComponent`**), which is the result of the transformation from a *Vendor Java Implementation* to an *Essence Java Implementation*.

To generate the transformation to the *Essence Java Implementation* the repository integrator in lines 1 through 5 defines some variables. The software component repositories integrator has to retrieve the repository (line 5). Then in line 6 it creates an *Essence Java Implementation* (**Step 1**). In order to perform the transformation depending on the model representation some values might require some processing. In lines 7 to 21 the actor performs **step 2**. For instance, from lines 7 to 9 the id in the *Vendor Repository* is

stored as **int**, so it has to be converted to **String** by the software component repositories integrator. From lines 14 to 19 it is shown how to handle collections, in this specific case the software component repositories integrator traverses the list by using the **Iterator** class and retrieving the value, the *IPSComp Transformation API* provides methods to include values in the different lists. Lines 20 and 21 show how to handle a **Publisher** class. This class supports the publisher IPSComp ontology concept in the *IPSComp Transformation API*. The *IPSComp Transformation API* provides setter methods to fill out the *Essence Java Implementation*. Then line 22 returns the created *Essence Java Implementation*. As it can be inferred from the example, each repository integrator must know how to handle its own repository, and the *IPSComp Transformation API* provides methods to perform the transformation.

Then to accomplish **step 3** the Software component repositories integrator can call the static method **transformModelToComponentOntology** implemented in the class **ModelTransformationGeneric** included in the *IPSComp Transformation API*, which receives an *Essence Java Implementation* (**GenericComponent**) and returns an instance of the *IPSComp Java Implementation*, as shown in lines 23 to 25.

```
public static GenericComponent transformToComponentEssence
          (ComponentSource sourceComponent){
1.   GenericRepository genericRepository;
2.   GenericComponent genericComponent;
3.   componentSource.Publisher originalPublisher;
4.   genericComponent.Publisher publisher;
     .
     .
     .
5.   genericRepository = GenericRepository.getGenericRepository
         ("http://www.componentsource.com");
6.   genericComponent = genericRepository.createGenericComponent();
7.   IntegerValue = new Integer(sourceComponent.getId());
8.   Value = integerValue.toString();
9.   genericComponent.setId(value);
10.  genericComponent.setComponentModel(sourceComponent.getComponentType());

11.  genericComponent.setName(sourceComponent.getName());
12.  genericComponent.setDescription(sourceComponent.getComponentAbstract());
13.  genericComponent.addKeyword(sourceComponent.getArchitecture());
14.  AList = sourceComponent.getBuiltUsing();
15.  Iterator = aList.iterator();
16.  While (iterator.hasNext()){
17.     value = (String)iterator.next();
18.     genericComponent.addKeyword(value);
19.  }
20.  publisher = genericRepository.createPublisher(value,
                        originalPublisher.getName(),
                        originalPublisher.getEmail(),
                        originalPublisher.getWebSite(),
                        originalPublisher.getPhone(),
                        originalPublisher.getDescription());
21.  genericComponent.setPublisher(publisher);
     .
     .
     .
22.  return genericComponent;
}
```

```
public static void main(String[] args){
23.  GenericComponent genericComponent =
        transformToComponentEssence (aExternalComponent);
24.  Component = ModelTransformationGeneric.transformModelToComponentOntology
        (genericComponent);
25.}
```

Table 4-11 Example Scenario Model Transformation Using the *IPSComp Transformation API* - Steps 1, 2 and 3.

As it can be inferred from the previous paragraphs, in the transformation from the different vendor Java implementations to the *IPSComp Java Implementation* there is a first transformation to the *Essence Java Implementation* and then from the *Essence Java Implementation* to the *IPSComp Java Implementation* (IPSComp Ontology compliant). But from step 1 to Step 3 the transformation can be seen as a dimension in the component's description domain. The fact that supports such statement is that with the *Vendor Component Description Meta-Model,* it can only be achieved a superficial component specification. It is superficial in the sense that there is not a complete technical description. The description can be seen in a Marketing level dimension. Those meta-models present concepts such as price, licenses, producers' information, component name, component description (in natural language), component architecture, etc.

As a consequence, up to here, the transformation is made in the "marketing" dimension. It will provide a first image from a *Vendor Java Implementation* into an *IPSComp Java Implementation* by traversing by the *Essence Java Implementation*.

Once the *IPSComp Java Implementation* has been obtained by performing step 1 through 3, there could be fields that have not been translated from the *Vendor Component Description*. Adding those fields could lead towards a more complete and accurate component image in the *IPSComp Repository*. The *IPSComp Transformation API* provides methods to populate the *IPSComp Component Description*.

To clarify the point exposed in the previous paragraph it is easier to take a specific example developed in the IPSComp prototype. The *Vendor Component Description* for the *Vendor Repository* http://www.componentsource.com has fields, which describe two component prerequisites: Disk space required and Memory required. These characteristics have been included in the IPSComp ontology as quality attributes. So once the transformation has been applied (see Table 4-11 Example Scenario Model Transformation Using the *IPSComp Transformation API* - Steps 1, 2 and 3.), it will be useful to add the prerequisite information. The software component repositories integrator accomplishes this using the *IPSComp Transformation API* as shown in Table 4-12 (**Step 4**).

```
/** After analyzing the information the user can use the component
qualityAttribute package included in the IPSComp Transformation API to introduce
quality attributes in the IPSComp Java Implementation.**/
1. QualityAttribute qualityAttribute;
2. String value;
3. QualityAttribute = QualityAttributeFactoryClass.getQualityAttribute
        (ComponentQualityAttribute.DISKUTILIZATION.toString());
4. value = String.valueOf(componentSource.getDiskSpace());
5. qualityAttribute.getMetric().setValue(value);
6. qualityAttribute.getMetric().setUnit(componentSource.getDiskSpaceMeasure());
7. qualityAttribute.getMetric().setFeature("Disk Space");
8. component.addQualityAttribute(qualityAttribute);
```

Table 4-12 Example Scenario Model Transformation Using the *IPSComp Transformation API* – Step 4.

In the snipped code shown in Table 4-12 line 1 defines a **`QualityAttribute`** class instance which will store quality attribute description to be added to the *IPSComp Java Implementation*. In line 3 a Factory pattern is used to instantiate the object for a specific Quality Attribute, in this case a `DISKUTILIZATION`.

In order to ensure that only defined quality attributes are assigned, an enumeration pattern is used in the IPSComp prototype. Lines 4 through 7 fill out the metric values. Only a person that knows the *Vendor Component Description Meta-Model* will know how to read values, but it will also need to become familiar with the IPSComp ontology. Finally in line 8 the new quality attribute is added to the *IPSCopm Java Implementation*.

It can be inferred from the previous example (Table 4-12) that the software component repositories integrator will be able to complete the *IPSComp Java Implementation* after the model transformation has taken place. This will allow accomplishing a more precise and complete "image" of the component description in the *IPSComp Repository*.

Before explaining step 5, it is necessary to look into more detail to the *Component Repositories* that I am modeling (Figure 4-16). Those repositories (*Components Repositories* layer) contain components implemented in a specific platform, which can be for instance EJBs, .NET, CORBA, etc. Each component has a description associated to it (A component in a Repository layer). There is a model that represents each one of the components present in the repository (Component Description Model layer). It is possible to create a meta-model for that model. This meta-model is expressed in UML and it corresponds to the UML class diagrams for the different *Vendor Component Description Meta-Model* (Component Description Meta-Model layer). As it has been stated after analyzing some of the meta-models, they provide a description in the marketing dimension.

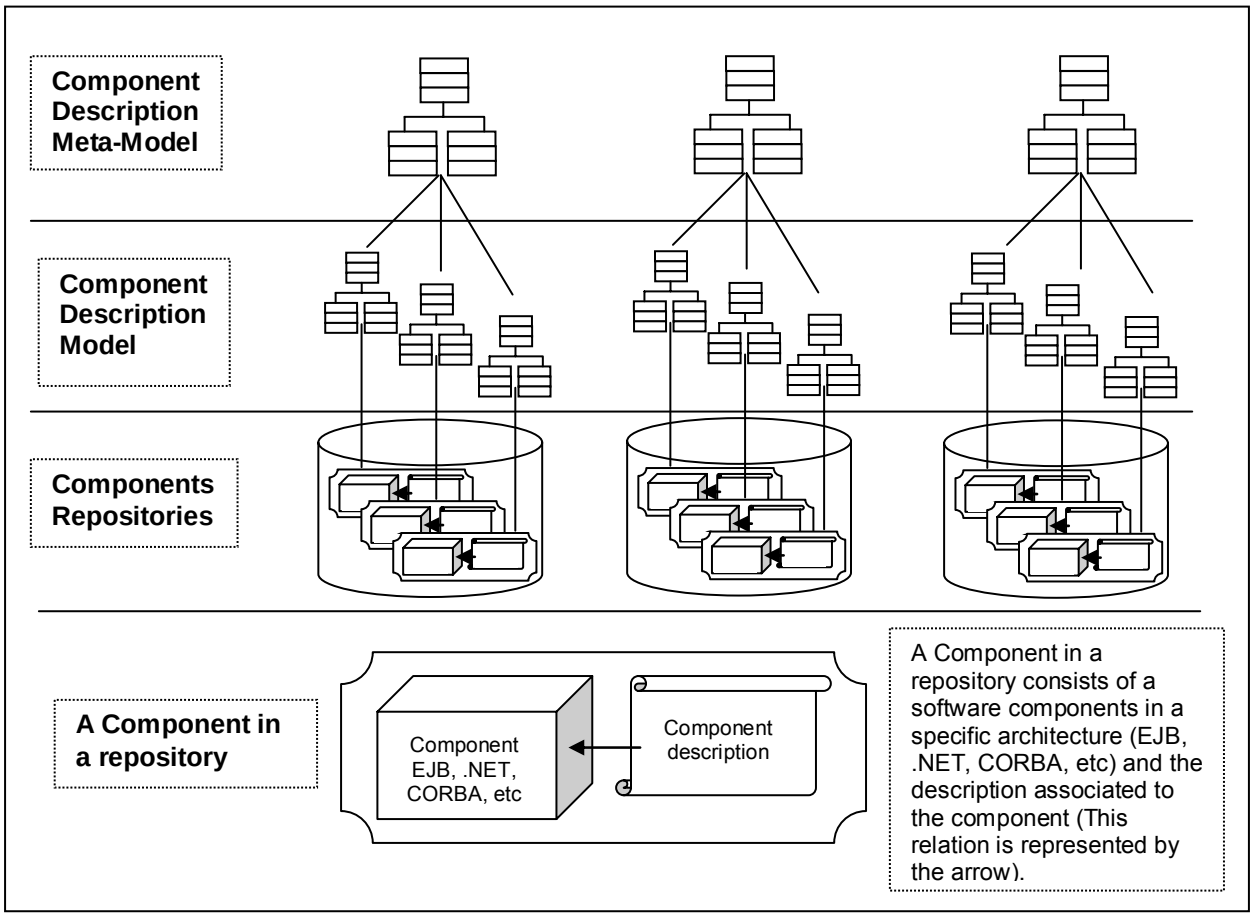


Figure 4-16 Component Repositories Domain Layers

The next step is going further down in the domain, and start taking specific component architectures.

Based on the architecture generate a more complete image of the component description in the *IPSComp Repository*. Once the marketing dimension transformation has been applied (up to step 4), the *IPSComp Transformation API* provides means to accomplish the transformation in the second identified dimension, the technical description. The idea is to take any specific component architecture (J2EE, .NET, CORBA, etc) and continue with the transformation (step 5). As shown in Figure 4-17 the outer rounded rectangle represents the set of *Vendor Repositories*. Then after applying the transformation up to step 2, a representation in the *Essence Component Description* is obtained. It is important to point out that for each *Vendor Repository* the transformation is unique, so that is why the *IPSComp Transformation API* is provided, in this way different *Vendor Repositories* may use the API to accomplish the transformation. The external repositories have a set of components developed in different specific architectures. So applying step 5, it produces a more complete image of the *Vendor Java Implementation* in the IPSComp ontology. The outcome of this process is represented in the inner rectangle.

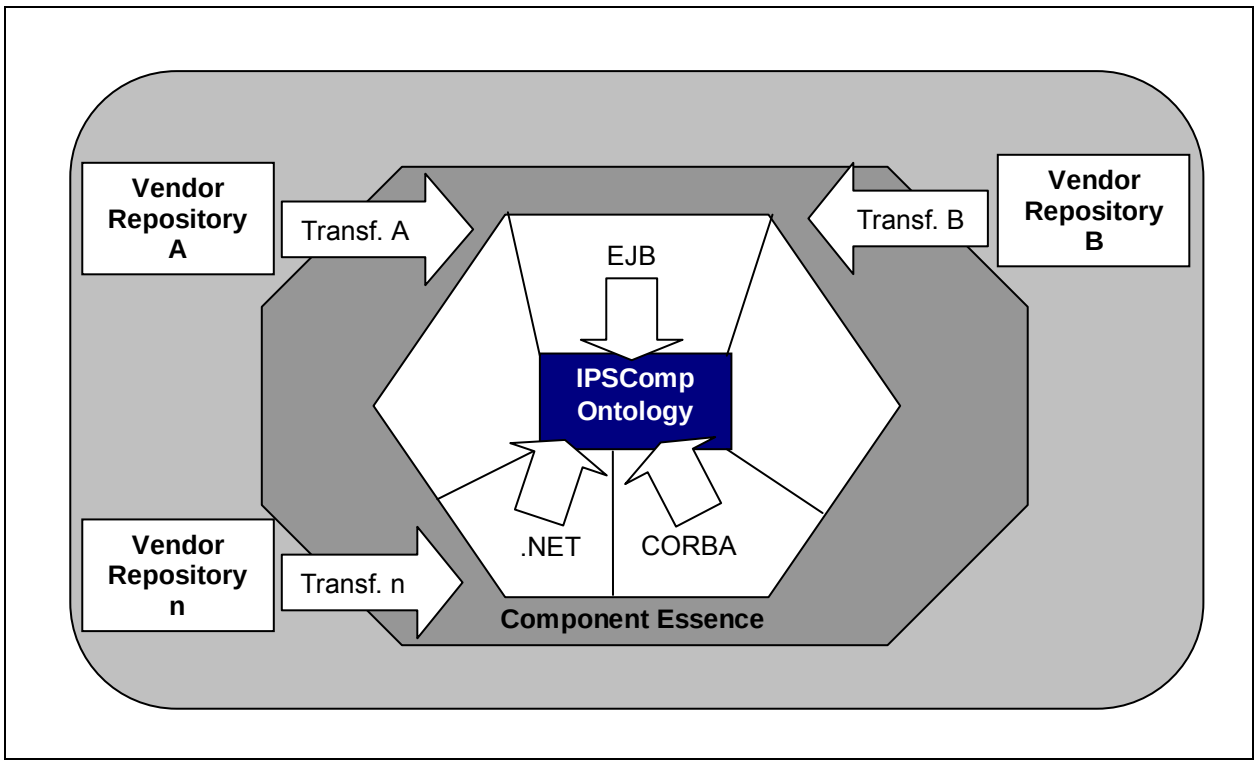


Figure 4-17 Obtaining a Component Description Image in the IPSComp Ontology

In the IPSComp prototype a specific component architecture transformation has been implemented for EJBs. The Enterprise JavaBeans specification is one of the several Java APIs in the Java 2 Platform, Enterprise Edition. The specification details how an application server provides server-side objects known as Enterprise JavaBeans, or EJBs. Enterprise JavaBeans (EJB) technology is the server-side component architecture for the Java 2 Platform. Components (JavaBeans) are reusable software programs that you can develop and assemble easily to create sophisticated applications [67].

J2EE components are packaged separately. Each component, its related files, and a deployment descriptor are assembled into a module. A deployment descriptor is an XML document that describes a component's deployment settings. For instance an enterprise bean module deployment descriptor declares transaction attributes and security authorizations for an enterprise bean. Each EJB JAR file contains a deployment descriptor, the enterprise bean files, and related files (Figure 4-18).

To sum up, in order to develop an enterprise bean, it is necessary to provide the following files:

- Deployment descriptor: An XML file that specifies information about the bean such as its persistence

type and transaction attributes. The descriptor is packed in the JAR file under the META-INF/ folder and it is called ejb-jar.xml.

- Interfaces: The remote and home interfaces are required for remote access. For local access, the local and local home interfaces are required. Message-driven beans do not use these interfaces. The remote interfaces expose the services provided by the EJB component. The home interfaces handle the EJB component life cycle.
- Enterprise bean class: Implements the methods defined in the interfaces.
- Helper classes: Other classes needed by the enterprise bean class, such as exception and utility classes.

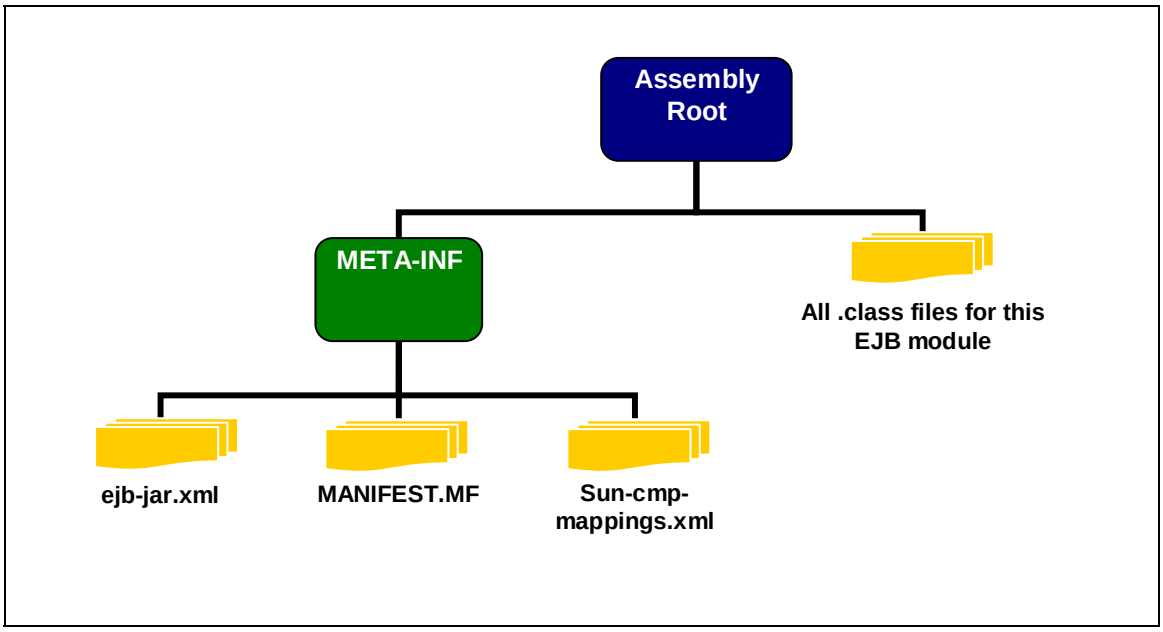


Figure 4-18 Structure of an Enterprise Bean JAR [67]

In order to apply the transformation for the J2EE architecture it is necessary to perform tasks on the JAR file that contains the EJB component. As defined in the J2EE specification a single JAR file might contain more than one EJB component. There is not a rule of thumb to define what component or set of components should be included in a single JAR file. As a matter of fact is up to the component provider and application assembler to take such decision. There exist some reasons to include several EJB components in one JAR file, for instance, if a set of EJB components share the same security parameters it is easier to handle just one descriptor to accomplish such behavior.

Because of the fact stated in the previous paragraph it is necessary to take the JAR file and extract the ejb-jar.xml file, which is the component descriptor. Then this file has to be processed to find out the tags that are necessary to load the EJB description into the *IPSComp ontology*. To implement the processing of the ejb-jar.xml file it has been taken the same approach used to load the IPSComp Ontology from the xml file. It uses a visitor pattern combined with Java reflection and the Factory pattern as explained in section 4.3.2.

The ejb-jar.xml file stores information that can be mapped to the IPSComp ontology. Even though there is a standard this descriptor file might have differences between different application servers. The standard deployment descriptor should include the following structural information for each Enterprise Bean:
The Enterprise Bean's name
The Enterprise Bean's class
The Enterprise Bean's home interface
The Enterprise Bean's remote interface

The Enterprise Bean's type
A re-entrancy indication for the Entity Bean
The Session Bean's state management type
The Session Bean's transaction demarcation type
The Entity Bean's persistence management
The Entity Bean's primary key class
Container-managed fields
Environment entries
The bean's EJB references
Resource manager connection factory references
Transaction attributes.

This information is used in the J2EE architecture in order to be able to deploy the EJB, and put them to work within the platform. On the other hand there are different J2EE specification versions, for instance, EJB applications that conform to the 2.0 specification, to the 3.0 specification, etc. The tags that provide useful information for the IPSComp ontology are:

- **<description>** It is used to provide text describing the parent element. The description element should include any information that the enterprise bean ejb-jar file producer wants to provide to the consumer of the enterprise bean ejb-jar file (i.e., to the Deployer). It is used in: cmp-field, cmr-field, container-transaction, ejb-jar, ejb-local-ref, ejb-ref, ejb-relation, ejb-relationship-role, entity, env-entry, exclude-list, message-driven, method, method-permission, query, relationship-role-source, relationships, resource-env-ref, resource-ref, run-as, security-identity, security-role, security-role-ref, session.

  In the IPSComp ontology, the description tag defined in the parents: **<ejb-jar>**, **<session>**, **<entity>** and **<message-driven>** is important. The **<description>** within the **<ejb-jar>** tag holds description for the whole descriptor. If this descriptor defines only one EJB, this might contain the EJB component description. The **<description>** within the **<session>**, **<entity>** and **<message-driven>** tags has the explanation for a particular EJB component. This will be added to the component description in the IPSComp ontology, which will be addressed to increase the behavioral component description.

- **<ejb-name>** It specifies an enterprise bean's name. This name is assigned by the ejb-jar file producer to name the enterprise bean in the ejb-jar file's deployment descriptor. The name must be unique among the names of the enterprise beans in the same ejb-jar file. It is used in: entity, session, message-driven, method, relationship-role-source.

  For the IPSComp ontology this field will be used to compare with the name attribute for each component described.

- **<remote>** It contains the fully-qualified name of the enterprise bean's remote interface. Used in: ejb-ref, entity, session.

  The interface referenced in this tag contains all the methods that form the components provided services. So the file stored in this tag must be processed to extract the methods and map them to the methods in the IPSComp ontology.

- **<local>** It contains the fully-qualified name of the enterprise bean's local interface. Used in: ejb-local-ref, entity, session.

  The interface referenced in this tag contains all the methods that form the components provided services. But this tag is used when the EJB is local, which means it will run in the same Java Virtual Machine (JVM). So the file stored in this tag must be processed to extract the methods and map them

to the methods in the IPSComp ontology.

- **<home>** It contains the fully-qualified name of the enterprise bean's home interface. It is used in: ejb-ref, entity, session.

  A bean's home interface specifies methods that allow the client to create, remove, and find objects of the same type. The home interface may also provide definitions for home business methods for entity beans. Home business methods are methods that are not specific to a particular bean instance. While the developer writes the home interface, the container creates the implementation for client interaction. In essence, the home interface provides bean management and life cycle methods.

- **<local-home>** It contains the fully-qualified name of the enterprise bean's local home interface. It is used in: ejb-local-ref, entity, session. The interface defined in this tag is analogous to the interface defined in the <home> tag but it is used when the EJB is local.

- **<ejb-class>** It contains the fully-qualified name of the enterprise bean's class. Used in: entity, message-driven, session. For the session and entity beans this class implements the components provided services.

  For the IPSComp ontology, this means that the class specified in this tag contains the EJB's properties.

- **<ejb-ref>** It is used for the declaration of a reference to an enterprise bean's home. It lists all other enterprise Java beans this bean uses. Used in: entity, message-driven, and session.

  It will represent a connection between Enterprise Java Beans. It means that the EJB where the reference is defined calls a service from the referenced component. It will be included as an underlying component of type support in the IPSComp ontology.

- **<ejb-local-ref>** element is used for the declaration of a reference to an enterprise bean's local home. Used in: entity, session, message-driven.

  It will be handled like the **<ejb-ref>** tag. It is the equivalent for local beans.

- **<security-role>** It contains the definition of a security role. The definition consists of an optional description of the security role, and the security role name. Used in: assembly-descriptor

  In the IPSComp ontology it will be mapped to the non-functional characteristic Controllability, which is a security quality attribute. This attribute indicates how the component is able to control the access to its provided services.

- **<persistence-type>** It specifies an entity bean's persistence management type. It can have as possible values Bean or Container. Used in: entity.

  The persistence will be mapped to the quality attribute Persitent which indicates whether a component can store its state in a persistent manner for later recovery. A Presence metric is used to measure this attribute. For the Session and the Message-Driven beans it has false as value.

The *IPSComp Transformation API* provides a static method **`method loadEJB (Component, String componentName, File jarFile, String tmpPath)`** which loads information from an EJB jar file (**Step 5 in scenario Model Transformation using the *IPSComp Transformation API***). It takes as parameters an instance of an *IPSComp Java Implementation*, an **`String`** with the component name as it is referenced in the ejb-jar.xml file, if it is an empty string it takes the component name stored in the

component, a **`File`** which is the jar file containing the EJB and a **`String`** which is a path where temporal files are created to be processed. The information loaded from the ejb-jar.xml file and must process to be added to the *IPSComp Java Implementation*. As explained above some tags have java files names that define either an interface or a java class. The way to process such tags is to retrieve the specific class file from the JAR containing the EJB and to gather the precise information for each case. For instance the **`<remote>`** tag contains the class name representing the EJB component provided interface. It is necessary to add the .class extension to this file name, extracted from the JAR file and from this file, which is an interface extract all the public methods. This information is stored in the Methods feature of *IPSComp Java Implementation*, this provides the method name, return type and parameters list. For the method's return type and parameters list, the data type is the data that will be stored in the *IPSComp Java Implementation*. Besides, because this is the EJB provided interface the method has status "Provided" in the *IPSComp Java Implementation*. If the interface is a subclass of any other interface, it is also necessary to perform the same process for all the hierarchical structure. The same process must be performed in the **`<home>`** tag extracted from the ejb-jar.xml file. The difference between these 2 interfaces is that the remote represents distributed EJB components meanwhile the home represents services provided by local components, those that run in the same JVM. In order to hold this information, in the *IPSComp Java Implementation*, the method feature has a class representing the method pre condition, so in this class either the value "Remote EJB service" or "Local EJB service" will be stored.

The first idea to retrieve the component properties was to take the **`<ejb-class>`** tag. Form this tag has the java file name that implements the EJB is obtained, the .class file extension added to it, and the Java class file corresponding to that file name is extracted from the JAR file and analyzed. The component's properties correspond to the class' attributes. This approach is against the idea behind EJB components, in which the only point of contact is throughout its interfaces.

Taking into account the EJB specification, the component's interfaces might specify getter and setter methods. In [73] the authors define the term virtual field. They assign the name virtual fields to the bean fields. They use this term because it is not required that there is actually a field in the defined in the bean. The getter and setter method names just imply the name of a field, similar to JavaBean properties.

Taken the virtual field definition by convention the write method name for each property is composed by the property name, capitalizing its first letter and preceding it with the word "set". It is a void method, and receives as parameter an instance of the same data type as the property that is setting. On the other hand, the method name to read the property composed by the property name, capitalizing its first letter and preceding it with the word "get". The return data type is the same as the property that is reading and it does not have parameters. If these methods are found in the interface file the information is also stored in the property description in the *IPSComp Java Implementation*. Furthermore, an EJB property can be classified as Read Only, Write Only, or Read Write. This classification is based on the set of getters and setters methods defined for each property. For instance, if a property has only a set method it is defined as Write Only, if it has only a get method it is defined as Read Only and if it has both method it is a Read Write property.

There are some other tags for which it is not necessary to extract additional files from the ejb jar file. The information is gathered directly from the ejb-jar.xml file and added to the *IPSComp Java Implementation*. They will be described in the following paragraphs.

The **`<ejb-ref>`** and the **`<ejb-local-ref>`** tags list other EJBs a bean uses. It means that the EJB where the reference is defined calls a service from the referenced component. This will be mapped to the *IPSComp Java Implementation* as an underlying component of type support. That tag has inner tags from where it is possible to obtain: **`<description>`** is an optional tag and allows the bean provider to supply some information about the referenced bean's use. **`<ejb-ref-name>`** is the environment name the bean should use to create the referenced bean using JNDI. **`<ejb-ref-type>`** should be Session or Entity, depending on the referenced bean type.

From the **<security-role>** tag the security role name **<role-name>** is going to be extracted. With that value a `Controllability` quality attribute will be added to the *IPSComp Java Implementation*. The Controllability attribute indicates how the component is able to control the access to its provided services. It has a `Presence` metric, so the feature will hold the keyword Security-role and the role name as defined in the ejb.jar.xml file and the value is set to **`true`**.

The **<persistence-type>** allows creating a `Persistent` quality attribute which indicates whether a component can store its state in a persistent manner for later recovery. A `Presence` metric is used to measure this attribute. For the Session and the Message-Driven beans it has **`false`** as value. For the entity bean it has **`true`** as value, and the feature can be either Bean or Container depending entity bean's persistence management type.

As a conclusion for section 4.4, it has been shown the path followed to accomplish a transformation from a *Vendor Java Implementation* to an *IPSComp Java Implementation*. First it was necessary to come up with the models that participate in the transformation, those models where created after analyzing some *Components Repositories* on the web. Second by comparing those models with the *IPSComp Component Description Meta-Model* the *Essence Component Description Meta-Model* was generated. The comparison between models drove me to the identification of two dimension in the component description domain, market dimension, mostly used in commercial web sites and technical dimension. Besides the *Essence Component Description Meta-Model*, and the traversing of the transformation by the *Essence Java Implementation* is addressed to allow monitoring the transformation, in order to evolve the ontology, trying to arrive to a standard in software component description. To perform the transformation the *Java Transformation API* is provided.

Finally, the scenario to accomplish a Model Transformation using the *IPSComp Transformation API* is as follows:

Actor: Software component repositories integrator.
Purpose: Perform a transformation from a *Vendor Java Implementation* to an *IPSCompt Java Implementation*.

- Step 1: Create the *Essence Java Implementation*.
- Step 2: Populate the information in the *Essence Java Implementation*.
- Step 3: Create the *IPSComp Java Implementation*.
- Step 4: Add information from the *Vendor Java Implementation* to the *IPSComp Java Implementation*.
- Step 5: Add information to the *IPSComp Java Implementation* from a component developed in a specific platform (EJB, .NET, CORBA).

5 Conclusions

The present investigation is aiming to be the starting point to achieve a scalable proper functional architecture for the IPSComp project. The task performed during the last couple of months were oriented to tackle down some issues, or come up with ideas that can be implemented in the final project. At the end of this section there are 2 figures, one shows the proposed architecture, the other has some parts of it highlighted with red color and numbered. Those red colored elements represent portion of the system on which some kind of work has been done.

This research has presented a component ontology that is proposed to describe components in a precise manner (item {2} in Figure 5-2) as well as to facilitate the component search and retrieval process. During the definition of IPSComp ontology two dimensions that different users might be interested in, have been identified: marketing and technical dimension. These dimensions are related to the type of user that interacts with the system. For instance a component-based application developer might be interested in the technical description; meanwhile a benchmark analyst would concentrate in the subject of interest (price, size, provider-in the marketing area; performance, interfaces, security-in the technical area). The objective of this research was not to find the final word as far as component description is concerned, actually this topic is still an open issue. As a consequence IPSComp ontology must be able to evolve along with the software component domain. But the importance of this approach is to try to find an ontology description that can be conceived as a component essence description, which must be validated by different actors involved in the process.

Being aware of the facts that the IPSComp ontology is not a final version, and that there are several component repositories already developed, MDA provides means to handle these two factors. As a matter of fact, I find that MDA provides a common layer of concepts that can be applied at different domains, and also to different levels of abstraction. MDA transformation allows us to handle the evolution of the component description until the field under research reaches an agreement or standard that will fulfill the needs in the sense of component description software domain (items {3} {4} in Figure 5-2).

The component description evolution can be handled by model transformations. For the present research, the first transformation, the one that has been addressed as the transformation in the marketing level has been implemented in both senses. In section 4.4 it was shown how a component integrator could create an image of its components description in the IPSComp ontology. But this transformation has also been implemented the other way around. As a matter of fact components described in the IPSComp ontology can be transformed to the *Essence Java Implementation*, and the repository integrator could populate *Vendor Java Implementation based* on *IPSComp Java Implementation and Essence Java Implementation*. This will provide the means to incorporate *Essence Java Implementation* components to the *Vendor Repositories*. Once this has been done, those components will benefit from the features implemented in *Vendor Repositories*, such as component classification and retrieval. In order to accomplish that the IPSComp prototype counts with the means to translate *IPSComp Java Implementation* into the *Essence Java Implementation*. Once this has been done the repository integrator can use the means in the *Essence Java Implementation* to read data from it. It is also an API. In the same manner, the *IPSComp Java Implementation* has an API to read data from it. The transformation from the *IPSComp Java Implementation* to *Vendor Repositories* must be understood as a mean to migrate the *IPSComp Repository* if some other approach is taken as standard. This allows sending the components described in the system to the model in the standard one.

It is important to point out that this model transformation from the *IPSComp Java Implementation* to *Vendor Java Implementation* is not included in the IPSComp project requirements. Actually, the site must provide means to keep information under certain security levels, in order to guarantee that the information will not be used in a harmful way by different competitors. Anyway the aim of this facility to allow migration towards other repository if any other standard is taken and this facility should be excluded in the deployment phase.

Furthermore, the model transformation facility is provided by means of an API, the *IPSComp Transformation API*. The reason to choose this approach is that as the component description is still an open issue, there is a high risk that the description changes with time, which means that either the *IPSComp Component Description Meta-Model* or *Vendor Component Description Meta-Model* changes. The *IPSComp Transformation API* can be changed accordingly. But there is not control over the changes performed in *Vendor Component Description Meta-Model*. With the *IPSComp Transformation API*, repository integrators can adapt the changes to the transformation or incorporate the changes carried out in the ontology *IPSComp Component Description Meta-Model*.

Even though not final, it is the aim of this research to contribute to the component description that industry and academy is trying to reach. As it can be inferred from the IPSComp ontology here proposed in order to describe a component there is a wide set of information that must be provided. As a matter of fact, the component subscription to the system is a highly time demanding process. The developer must provide all the required information to guarantee the correct component description. This process can become tedious and there is not guarantee that users will be willing to go through it. For instance, in order to register a component in the Componex web site (http://www.componex.biz) (*Vendor Repository*), it is necessary to fill out a 10-page formulary. As a matter of fact, in [15] the author states that he does not know if a component producer will be willing to fill out all that information. In fact up to date only 6 components have been registered and those are examples. That was also one of the reasons to include the repository integration analysis in this research. As such we can take advantage of the existing repositories. Furthermore, also aiming to decrease this factor the *IPSComp Transformation API* provides a method to include EJBs from a jar file. Anyway in order to accomplish a more accurate description, it is mandatory to provide that information either in the *Component Description Meta-Model* or as explained in section 4.4 in the ejb-jar.xml for the specific EJBs case.

Because the software component is an existing artifact, it is necessary to identify which components are useful when assembling systems. In order to describe existing components a standardized specification is needed. If there is little incentive or pressure to agree on open standards a set of proprietary descriptions will be created, that is a reality today. In this research IPSComp ontology has been proposed, by taking an existing one called XCM [28] as base, and adding the quality attributes to provide non-functional description to the components (item {3} in Figure 5-2). The quality attributes have been tailored to components from the ISO 9126 norm [52]. It is a good starting point to take a standard as a base.

It is important to remark that the component provider must provide most quality attribute values, but it is necessary to have a feedback from the component end user in order to validate them. This feedback will allow obtaining more reliable information. This feedback is also important because even though the quality attribute is concrete, and the mechanism to calculate its value is clear it can have a subjective point of view. For instance to measure level of complexity to parameterize a component it will depend directly of the skills of the person using the component, accordingly a wider sample will be helpful to calculate a more accurate value.

Moving on to the Ontology implementation, (item {1} in Figure 5-2) once the IPSComp ontology was defined it was included in PLIB. Some issues concerning to the tool did not allow to create instances of the ontology. Anyway, PLIB provides a Java API, which will allow handling the ontology elements from an external application. The aim of this point was to communicate the system with the ontology manager to manipulate the ontology and to perform transformation on different ontologies.

Furthermore the ontology manager system that has to be integrated to the IPSCompo project must be a robust one. The IPSComp project will have several ontologies, one of them is the one describing software components the IPSComp ontology, but software development industry is dealing, or in contact with a variety of domains. As a consequence for each domain a specific ontology must be included in the system (item Domain-specific Ontologies and Taxonomies not red highlighted in Figure 5-2). The domain ontology is a task that should be performed by a domain specialist. This ontology must evolve with the domain. As such this will be a time demanding task, which qualified people should perform.

On the other hand, identification of software components is a complex task. It has to deal with two main issues unstructured information to describe components and also with an impressive number of possible candidates. To overcome the former the IPSComp ontology is aiming to standardize the component description. The number of candidates can not be diminished, but once the standardization has been done, we provide means to integrate component repositories in such a way that they will be described with the IPSComp ontology, it is possible to create an image of the component description in our repository, such description is IPSComp ontology compatible and as such they can share the set of tools that will be implemented in the project such as the recommender system.

Different retrieval schemes have been proposed throughout the years to retrieve software components (item {4} in Figure 5-2). Those different schemes for software retrieval process are being combined and implemented in some commercial sites as well as in research projects. As result of the combination of such techniques, researchers have shown an improvement in the retrieving process. For instance, once of the drawbacks of Signature Matching-based technique is that the result set can have components which do not accomplish the desired behavior, even though the signature matches exactly, take into account the strcpy and the strcat functions in the C language, the signature is the same but the task they perform is completely different. Preceding this technique with a semantic-based approach it is possible to limit the sample on which the signature matching is going to be performed [42]. Such techniques are applied on a specific model representation. If it is possible to perform a model transformation towards such model the source model instances will benefit from the features provided in the target repository. The IPSComp ontology holds the concepts necessary to implement a search tools based on different searching techniques.

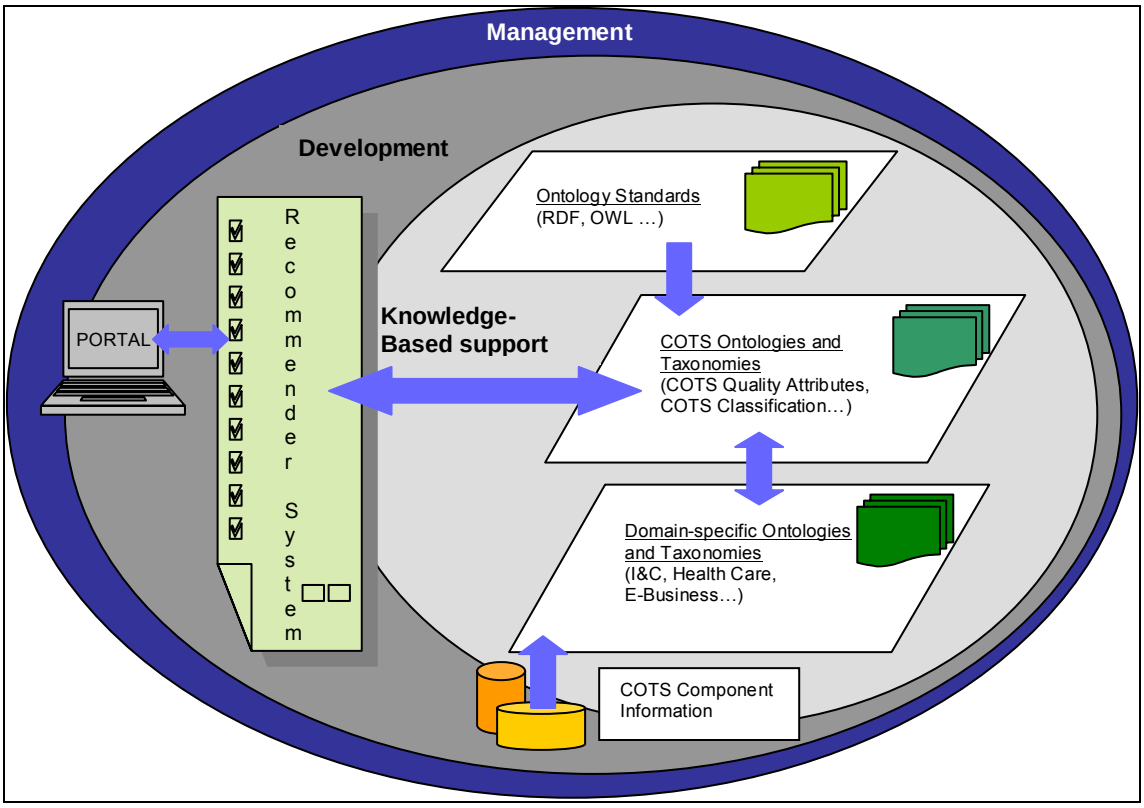


Figure 5-1 IPSComp System Architecture [48]

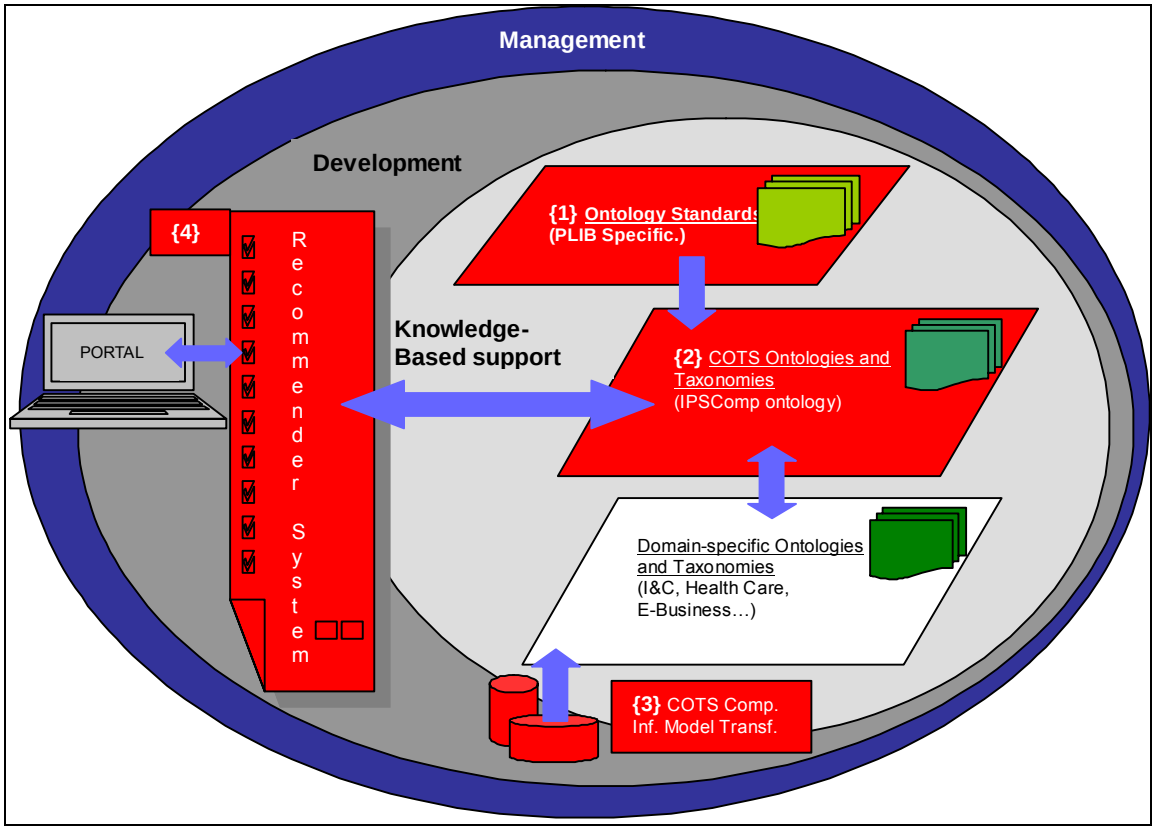


Figure 5-2 IPSComp System Architecture Analyzed

6 Future Work

The present research has shown some considerations that must be taken into account for the IPSComp project functional architecture. These considerations must be implemented and validated. Some fields will definitely need further research in order to accomplish a complete solution which will satisfy the requirements from the different users involved in the project.

It is necessary to arrive to a standardized component description. To accomplish this it is essential to start monitoring the model transformation, annotating how the ontology evolves, what concepts are being used and adjusting the *IPSComp Component Description Meta-Model* as well as the *IPSComp Transformation API* to support this information.

The IPSComp ontology has provided Non-Functional description by means of quality attributes, which have been implemented with 2 main characteristics that look for further research. The former is the **`IValue`** interface to measure different data types. It should be extended with a grammar to compare its values. This will enable the components search based on non-functional characteristics. The second characteristic is the *feature* concept, it provides context to the quality attribute. This must be complemented with an external ontology, related to the domain at which the quality attribute belongs to. As a consequence an ontology for each quality attribute should be created by an expert in the domain.

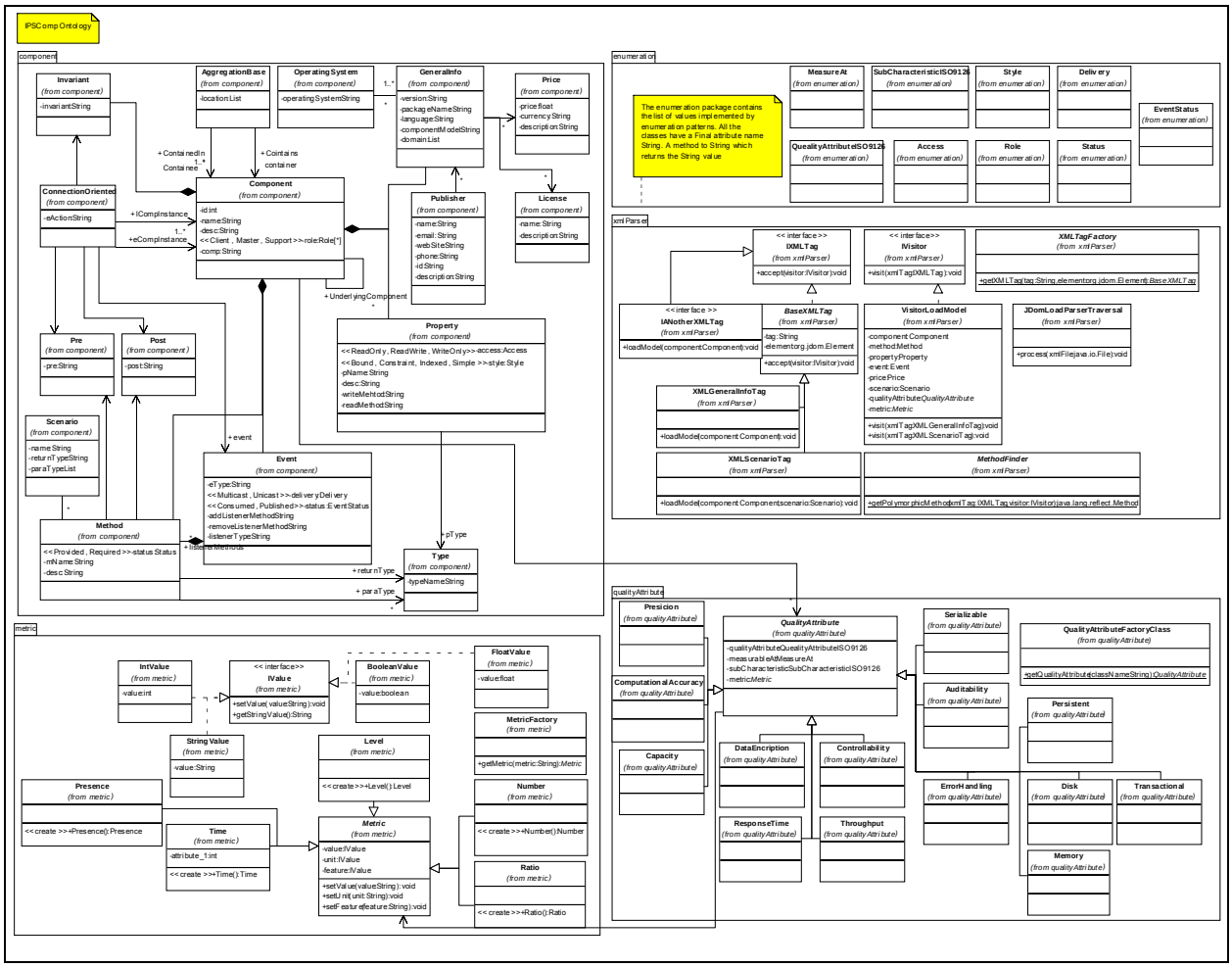
The IPSComp ontology will provide the information needed to be able to implement a component retrieval tool. As a matter of fact Tansalarak et al. [51] present a description of the components retrieval tool created on top of the XCM ontology. Either by extending this search tool including the new features the IPSComp ontology has or creating a new retrieval scheme, combining several schemes, a search tool can be implemented. It is not just a matter of providing the search tool but evaluating it. For instance, lets suppose that the work done in [51] is extended by including a grammar to retrieve components by quality attributes. It is not a matter of retrieving less total number of hits from the search but to be more accurate with respect to the user needs.

The architecture should be complemented with Users Web Mining techniques. This allows monitoring components, users and queries. The impact produced by the lack of standards can be diminished by the gathering of information. Besides, feedback from end users relating metric values and components in gender are necessary to improve the architecture, as well as the IPSComp ontology.

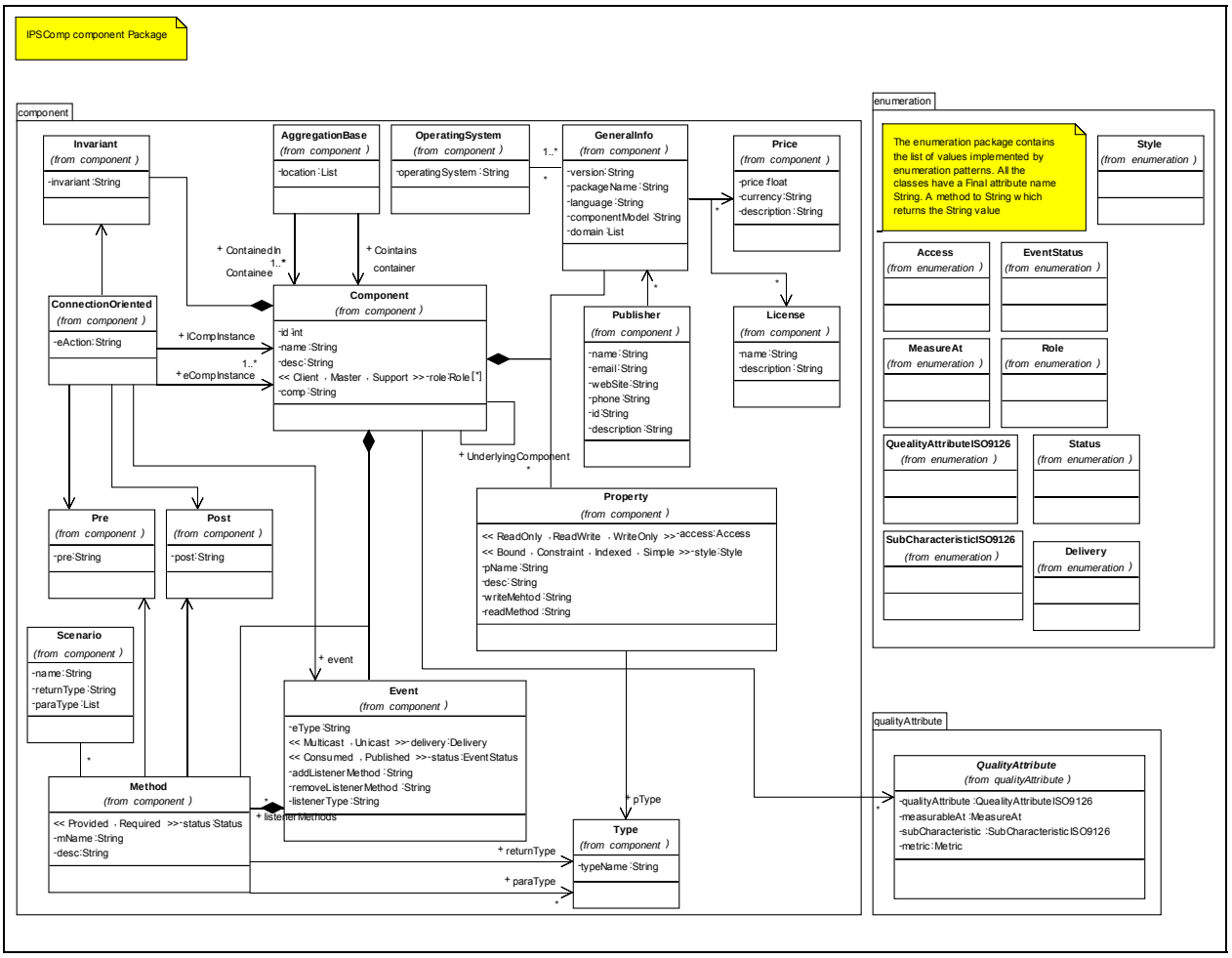
At the implementation level, it provides means to include EJBs into the IPSComp ontology; this has been implemented for EJB version 2.1, it is necessary to include different EJBs versions as well as components developed in other architectures such as .NET, CORBA, etc.

On the other hand, Web Services are intended to provide a standardized mechanism to describe, locate, and communicate with online applications. In order to offer a service description it uses Web Service Description Language (WSDL). WSDL usually describes interface information for publicly available methods, data type information for messages, binding information for transport protocols, and address information for locating services. If we take a look at the way WSDL describes a Web Service, it can be compared to the way IDL specifies a software component. As a matter of fact, both IDL and WSDL do not support any sort of semantic description. Taking advantage of this commonality some researches are being addressed to provide semantic meaning to components and web services at the same time [15, 42]. This will help in the description and discovery of web services as well as components because the same approaches can be taken in both domains.

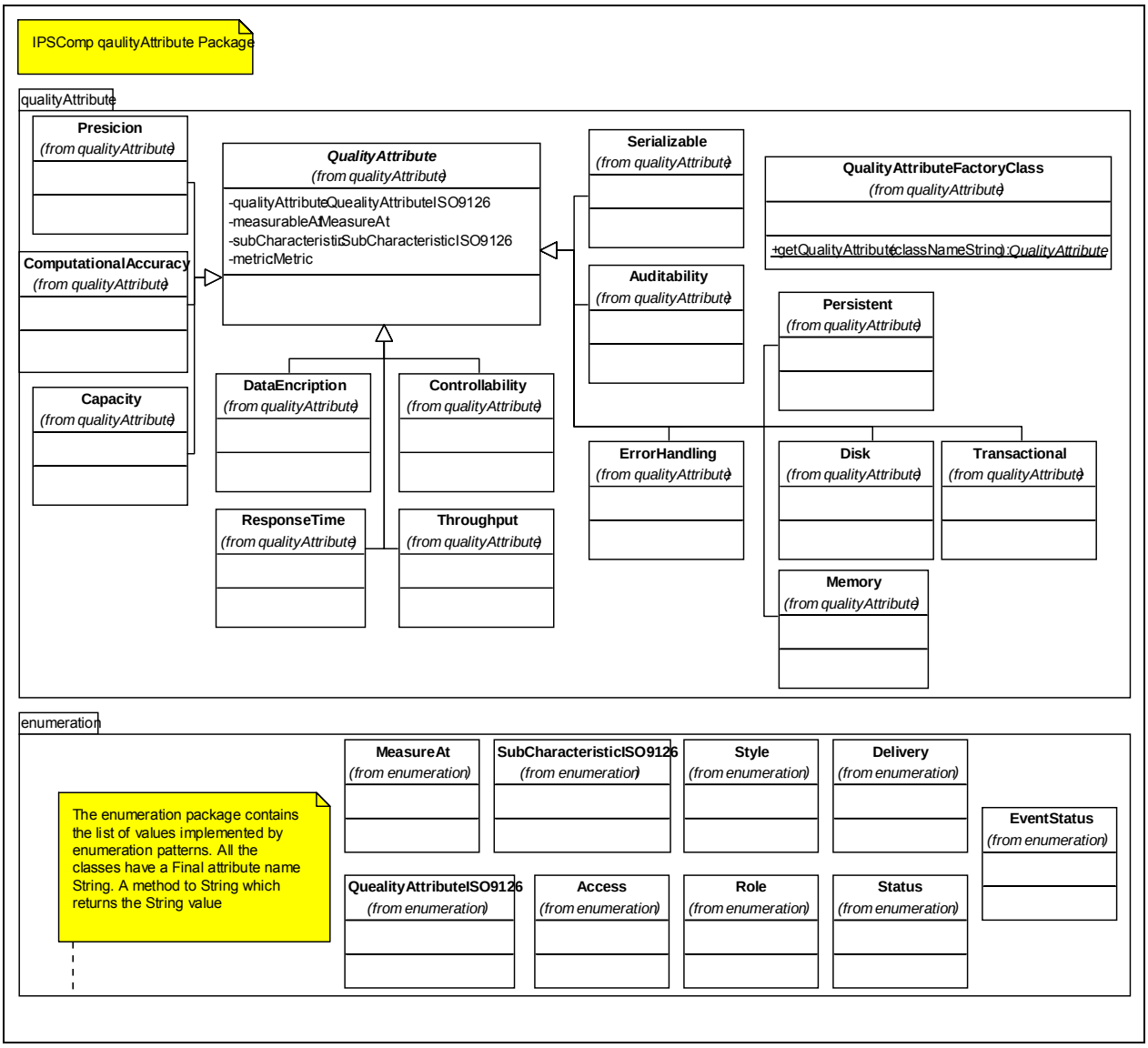
Appendix A - IPSComp Ontology UML Class Diagram



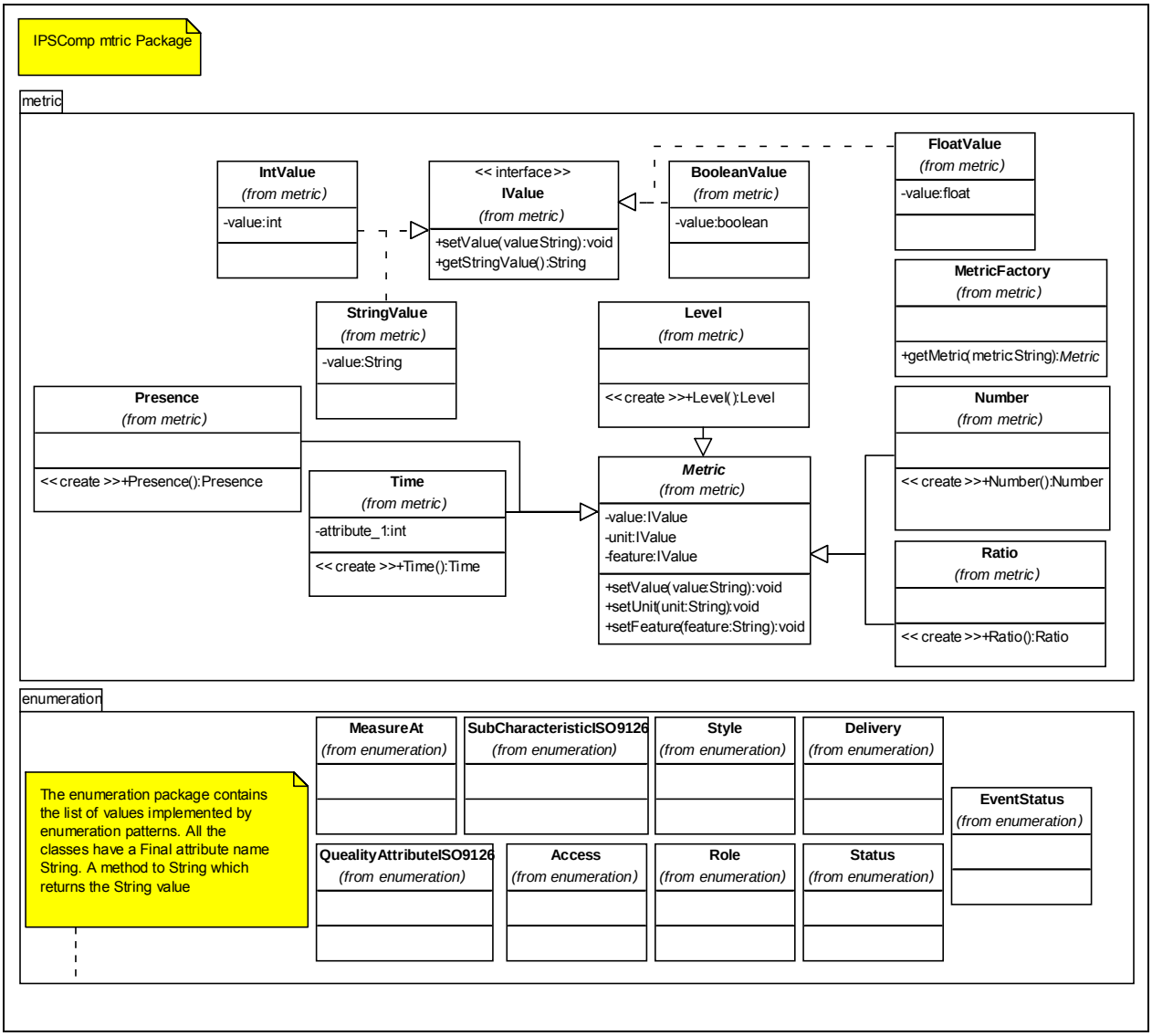
Appendix B - IPSComp Ontology component Package UML Class Diagram



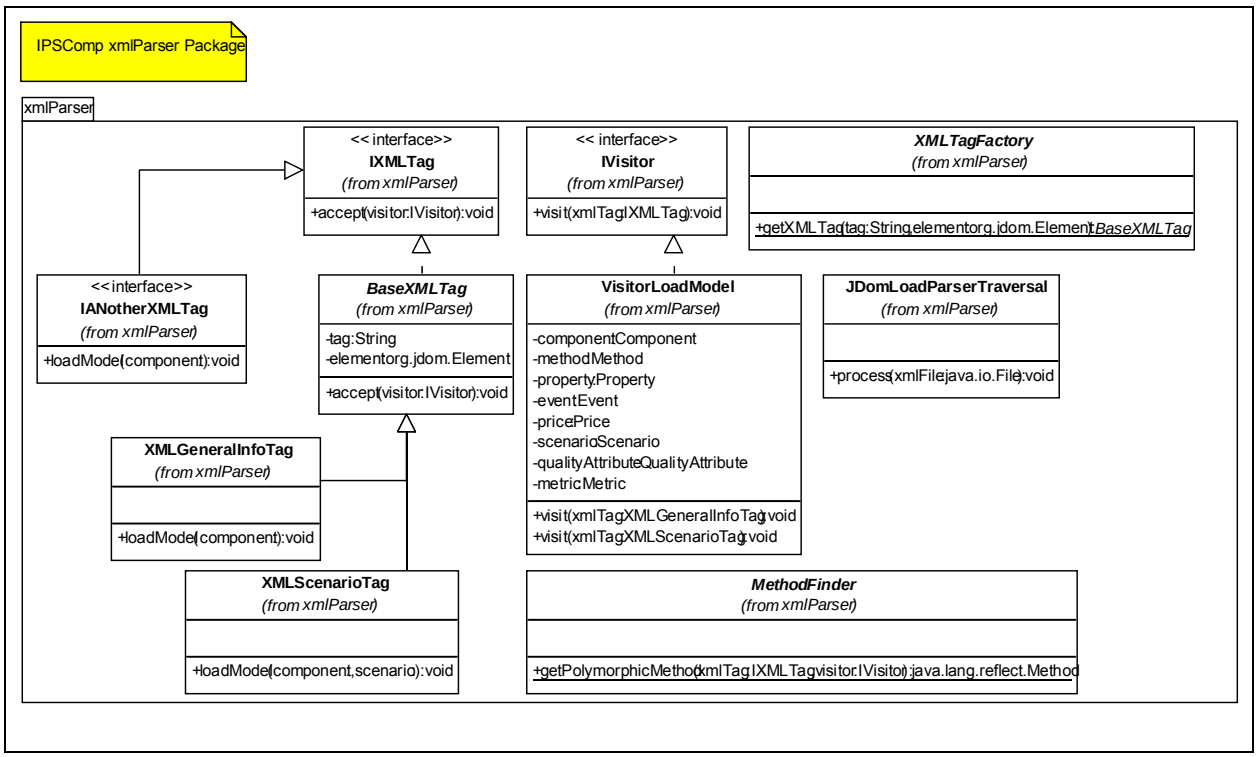
Appendix C - IPSComp Ontology qualityAttribute Package UML Class Diagram



Appendix D - IPSComp Ontology metric Package UML Class Diagram



Appendix E - IPSComp Ontology xmlParser Package UML Class Diagram



Appendix F - IPSComp XML Meta-Model - XSD Schema

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- edited with XML Spy v4.2 U (http://www.xmlspy.com) by Administrator (Administrator) -->
<!--W3C Schema generated by XML Spy v4.2 U (http://www.xmlspy.com)-->
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema" elementFormDefault="qualified">
	<xs:element name="componentSpecification">
		<xs:complexType>
			<xs:sequence>
				<xs:element ref="id"/>
				<xs:element ref="name"/>
				<xs:element ref="generalInfo"/>
				<xs:element ref="features"/>
				<xs:element ref="design"/>
				<xs:element ref="qualityAttributes"/>
			</xs:sequence>
		</xs:complexType>
	</xs:element>
	<xs:element name="generalInfo">
		<xs:complexType>
			<xs:sequence>
				<xs:element ref="version"/>
				<xs:element ref="package"/>
				<xs:element ref="language"/>
				<xs:element ref="model"/>
				<xs:element ref="domain" maxOccurs="unbounded"/>
				<xs:element ref="os" maxOccurs="unbounded"/>
				<xs:element ref="prices"/>
				<xs:element ref="publisher"/>
				<xs:element ref="license"/>
			</xs:sequence>
		</xs:complexType>
	</xs:element>
	<xs:element name="methods">
		<xs:complexType>
			<xs:sequence>
				<xs:element ref="method" maxOccurs="unbounded"/>
			</xs:sequence>
		</xs:complexType>
	</xs:element>
	<xs:element name="method">
		<xs:complexType>
			<xs:sequence>
				<xs:element ref="mName"/>
				<xs:element ref="desc"/>
				<xs:element ref="pre"/>
				<xs:element ref="post"/>
				<xs:element ref="returnType"/>
				<xs:element ref="paraType" maxOccurs="unbounded"/>
				<xs:element ref="scenarios"/>
			</xs:sequence>
			<xs:attribute name="status" use="required">
				<xs:simpleType>
					<xs:restriction base="xs:string">
						<xs:enumeration value="Provided"/>
						<xs:enumeration value="Required"/>
					</xs:restriction>
				</xs:simpleType>
			</xs:attribute>
		</xs:complexType>
	</xs:element>
	<xs:element name="properties">
		<xs:complexType>
			<xs:sequence>
				<xs:element ref="property" maxOccurs="unbounded"/>
			</xs:sequence>
		</xs:complexType>
```

```
</xs:element>
<xs:element name="property">
	<xs:complexType>
		<xs:sequence>
			<xs:element ref="pName"/>
			<xs:element ref="pType"/>
			<xs:element ref="readMethod"/>
			<xs:element ref="writeMethod" minOccurs="0"/>
		</xs:sequence>
		<xs:attribute name="access" use="required">
			<xs:simpleType>
				<xs:restriction base="xs:NMTOKEN">
					<xs:enumeration value="ReadOnly"/>
					<xs:enumeration value="ReadWrite"/>
					<xs:enumeration value="WriteOnly"/>
				</xs:restriction>
			</xs:simpleType>
		</xs:attribute>
		<xs:attribute name="style" use="required">
			<xs:simpleType>
				<xs:restriction base="xs:NMTOKEN">
					<xs:enumeration value="Bound"/>
					<xs:enumeration value="Simple"/>
				</xs:restriction>
			</xs:simpleType>
		</xs:attribute>
	</xs:complexType>
</xs:element>
<xs:element name="DataEncription">
	<xs:complexType>
		<xs:sequence>
			<xs:element ref="characteristic"/>
			<xs:element ref="lifeCycle"/>
			<xs:element ref="subCharacteristic"/>
			<xs:element ref="metric"/>
		</xs:sequence>
		<xs:attribute name="metric" type="xs:string" use="required"/>
	</xs:complexType>
</xs:element>
<xs:element name="DiskUtilization">
	<xs:complexType>
		<xs:sequence>
			<xs:element ref="characteristic"/>
			<xs:element ref="lifeCycle"/>
			<xs:element ref="subCharacteristic"/>
			<xs:element ref="metric"/>
		</xs:sequence>
		<xs:attribute name="metric" type="xs:string" use="required"/>
	</xs:complexType>
</xs:element>
<xs:element name="addListenerMethod" type="xs:string"/>
<xs:element name="amount" type="xs:decimal"/>
<xs:element name="cComposition">
	<xs:complexType>
		<xs:sequence>
			<xs:element ref="container"/>
			<xs:element ref="containees"/>
		</xs:sequence>
	</xs:complexType>
</xs:element>
<xs:element name="cCompositions">
	<xs:complexType>
		<xs:sequence>
			<xs:element ref="cComposition" maxOccurs="unbounded"/>
		</xs:sequence>
	</xs:complexType>
</xs:element>
<xs:element name="callMethod" type="xs:string"/>
<xs:element name="characteristic">
	<xs:simpleType>
```

```
                <xs:restriction base="xs:string">
                        <xs:enumeration value="Efficiency"/>
                        <xs:enumeration value="Functionality"/>
                        <xs:enumeration value="Reliability"/>
                        <xs:enumeration value="Maintainability"/>
                        <xs:enumeration value="Usability"/>
                </xs:restriction>
        </xs:simpleType>
</xs:element>
<xs:element name="cid" type="xs:string"/>
<xs:element name="comp" type="xs:string"/>
<xs:element name="compInstance">
        <xs:complexType>
                <xs:sequence>
                        <xs:element ref="comp"/>
                        <xs:element ref="cid"/>
                </xs:sequence>
                <xs:attribute name="role" use="required">
                        <xs:simpleType>
                                <xs:restriction base="xs:NMTOKEN">
                                        <xs:enumeration value="Client"/>
                                        <xs:enumeration value="Master"/>
                                        <xs:enumeration value="Support"/>
                                </xs:restriction>
                        </xs:simpleType>
                </xs:attribute>
        </xs:complexType>
</xs:element>
<xs:element name="compInstances">
        <xs:complexType>
                <xs:sequence>
                        <xs:element ref="compInstance" maxOccurs="unbounded"/>
                </xs:sequence>
        </xs:complexType>
</xs:element>
<xs:element name="compositions">
        <xs:complexType>
                <xs:sequence>
                        <xs:element ref="eCompositions"/>
                        <xs:element ref="cCompositions"/>
                </xs:sequence>
        </xs:complexType>
</xs:element>
<xs:element name="constraint">
        <xs:complexType>
                <xs:sequence>
                        <xs:element ref="inv"/>
                        <xs:element ref="pre"/>
                        <xs:element ref="post"/>
                </xs:sequence>
        </xs:complexType>
</xs:element>
<xs:element name="containee">
        <xs:complexType>
                <xs:sequence>
                        <xs:element ref="rid"/>
                        <xs:element ref="location" minOccurs="0"/>
                </xs:sequence>
        </xs:complexType>
</xs:element>
<xs:element name="containees">
        <xs:complexType>
                <xs:sequence>
                        <xs:element ref="containee" maxOccurs="unbounded"/>
                </xs:sequence>
        </xs:complexType>
</xs:element>
<xs:element name="container">
        <xs:complexType>
                <xs:sequence>
```

```
                        <xs:element ref="rid"/>
                </xs:sequence>
        </xs:complexType>
</xs:element>
<xs:element name="currency" type="xs:string"/>
<xs:element name="desc" type="xs:string"/>
<xs:element name="description" type="xs:string"/>
<xs:element name="design">
        <xs:complexType>
                <xs:sequence>
                        <xs:element ref="compInstances"/>
                        <xs:element ref="compositions"/>
                </xs:sequence>
        </xs:complexType>
</xs:element>
<xs:element name="domain">
        <xs:simpleType>
                <xs:restriction base="xs:string">
                        <xs:enumeration value="Interface"/>
                        <xs:enumeration value="MVC"/>
                </xs:restriction>
        </xs:simpleType>
</xs:element>
<xs:element name="eAction" type="xs:string"/>
<xs:element name="eCompInstance">
        <xs:complexType>
                <xs:sequence>
                        <xs:element ref="rid"/>
                        <xs:element ref="event"/>
                        <xs:element ref="eAction"/>
                </xs:sequence>
        </xs:complexType>
</xs:element>
<xs:element name="eCompInstances">
        <xs:complexType>
                <xs:sequence>
                        <xs:element ref="eCompInstance"/>
                </xs:sequence>
        </xs:complexType>
</xs:element>
<xs:element name="eComposition">
        <xs:complexType>
                <xs:sequence>
                        <xs:element ref="constraint"/>
                        <xs:element ref="eCompInstances"/>
                        <xs:element ref="lCompositions"/>
                </xs:sequence>
        </xs:complexType>
</xs:element>
<xs:element name="eCompositions">
        <xs:complexType>
                <xs:sequence>
                        <xs:element ref="eComposition" maxOccurs="unbounded"/>
                </xs:sequence>
        </xs:complexType>
</xs:element>
<xs:element name="eType" type="xs:string"/>
<xs:element name="email" type="xs:string"/>
<xs:element name="event">
        <xs:complexType mixed="true">
                <xs:choice minOccurs="0" maxOccurs="unbounded">
                        <xs:element ref="eType"/>
                        <xs:element ref="addListenerMethod"/>
                        <xs:element ref="removeListenerMethod"/>
                        <xs:element ref="listenerMethods"/>
                </xs:choice>
                <xs:attribute name="delivery">
                        <xs:simpleType>
                                <xs:restriction base="xs:NMTOKEN">
                                        <xs:enumeration value="MultiCast"/>
```

```
                                        <xs:enumeration value="UniCast"/>
                                </xs:restriction>
                        </xs:simpleType>
                </xs:attribute>
                <xs:attribute name="status">
                        <xs:simpleType>
                                <xs:restriction base="xs:NMTOKEN">
                                        <xs:enumeration value="Consumed"/>
                                        <xs:enumeration value="publish"/>
                                </xs:restriction>
                        </xs:simpleType>
                </xs:attribute>
        </xs:complexType>
</xs:element>
<xs:element name="events">
        <xs:complexType>
                <xs:sequence>
                        <xs:element ref="event" maxOccurs="unbounded"/>
                </xs:sequence>
        </xs:complexType>
</xs:element>
<xs:element name="feature" type="xs:string"/>
<xs:element name="features">
        <xs:complexType>
                <xs:sequence>
                        <xs:element ref="methods"/>
                        <xs:element ref="properties"/>
                        <xs:element ref="events"/>
                </xs:sequence>
        </xs:complexType>
</xs:element>
<xs:element name="id" type="xs:string"/>
<xs:element name="inv" type="xs:boolean"/>
<xs:element name="lCompInstance">
        <xs:complexType>
                <xs:sequence>
                        <xs:element ref="rid"/>
                        <xs:element ref="callMethod"/>
                </xs:sequence>
        </xs:complexType>
</xs:element>
<xs:element name="lCompositions">
        <xs:complexType>
                <xs:choice maxOccurs="unbounded">
                        <xs:element ref="lCompInstance"/>
                        <xs:element ref="op"/>
                </xs:choice>
        </xs:complexType>
</xs:element>
<xs:element name="language" type="xs:string"/>
<xs:element name="license">
        <xs:complexType>
                <xs:sequence>
                        <xs:element ref="name"/>
                        <xs:element ref="description"/>
                </xs:sequence>
        </xs:complexType>
</xs:element>
<xs:element name="lifeCycle">
        <xs:simpleType>
                <xs:restriction base="xs:string">
                        <xs:enumeration value="Runtime"/>
                        <xs:enumeration value="Life Cycle"/>
                </xs:restriction>
        </xs:simpleType>
</xs:element>
<xs:element name="listenerMethod">
        <xs:complexType>
                <xs:sequence>
                        <xs:element ref="mName"/>
```

```
                        <xs:element ref="returnType"/>
                        <xs:element ref="paraType" maxOccurs="unbounded"/>
                </xs:sequence>
        </xs:complexType>
</xs:element>
<xs:element name="listenerMethods">
        <xs:complexType>
                <xs:sequence>
                        <xs:element ref="listenerMethod" maxOccurs="unbounded"/>
                </xs:sequence>
        </xs:complexType>
</xs:element>
<xs:element name="location" type="xs:string"/>
<xs:element name="mName" type="xs:string"/>
<xs:element name="metric">
        <xs:complexType>
                <xs:sequence>
                        <xs:element ref="value"/>
                        <xs:element ref="feature"/>
                        <xs:element ref="unit"/>
                </xs:sequence>
        </xs:complexType>
</xs:element>
<xs:element name="model" type="xs:string"/>
<xs:element name="name" type="xs:string"/>
<xs:element name="op">
        <xs:complexType>
                <xs:attribute name="type" type="xs:string" use="required"/>
        </xs:complexType>
</xs:element>
<xs:element name="os" type="xs:string"/>
<xs:element name="pName" type="xs:string"/>
<xs:element name="pType" type="xs:string"/>
<xs:element name="package" type="xs:string"/>
<xs:element name="paraType" type="xs:string"/>
<xs:element name="phone" type="xs:string"/>
<xs:element name="post" type="xs:string"/>
<xs:element name="pre" type="xs:string"/>
<xs:element name="price">
        <xs:complexType>
                <xs:sequence>
                        <xs:element ref="amount"/>
                        <xs:element ref="currency"/>
                        <xs:element ref="description"/>
                </xs:sequence>
        </xs:complexType>
</xs:element>
<xs:element name="prices">
        <xs:complexType>
                <xs:sequence>
                        <xs:element ref="price" maxOccurs="unbounded"/>
                </xs:sequence>
        </xs:complexType>
</xs:element>
<xs:element name="publisher">
        <xs:complexType>
                <xs:sequence>
                        <xs:element ref="id"/>
                        <xs:element ref="name"/>
                        <xs:element ref="email"/>
                        <xs:element ref="webSite"/>
                        <xs:element ref="phone"/>
                </xs:sequence>
        </xs:complexType>
</xs:element>
<xs:element name="qualityAttributes">
        <xs:complexType>
                <xs:sequence>
                        <xs:element ref="DataEncription"/>
                        <xs:element ref="DiskUtilization"/>
```

```
                </xs:sequence>
            </xs:complexType>
        </xs:element>
        <xs:element name="readMethod" type="xs:string"/>
        <xs:element name="removeListenerMethod" type="xs:string"/>
        <xs:element name="returnType" type="xs:string"/>
        <xs:element name="rid" type="xs:string"/>
        <xs:element name="scenario">
            <xs:complexType>
                <xs:sequence>
                    <xs:element ref="scenarioReturnType"/>
                    <xs:element ref="scenarioParaType" maxOccurs="unbounded"/>
                </xs:sequence>
                <xs:attribute name="sName" type="xs:string" use="required"/>
            </xs:complexType>
        </xs:element>
        <xs:element name="scenarioParaType">
            <xs:simpleType>
                <xs:restriction base="xs:string">
                    <xs:enumeration value="Dynamic"/>
                    <xs:enumeration value="Kill"/>
                    <xs:enumeration value="Static"/>
                </xs:restriction>
            </xs:simpleType>
        </xs:element>
        <xs:element name="scenarioReturnType" type="xs:string"/>
        <xs:element name="scenarios">
            <xs:complexType>
                <xs:sequence>
                    <xs:element ref="scenario" maxOccurs="unbounded"/>
                </xs:sequence>
            </xs:complexType>
        </xs:element>
        <xs:element name="subCharacteristic">
            <xs:simpleType>
                <xs:restriction base="xs:string">
                    <xs:enumeration value="Accuracy"/>
                    <xs:enumeration value="Suitability"/>
                    <xs:enumeration value="Security"/>
                    <xs:enumeration value="Interoperability"/>
                    <xs:enumeration value="Compliance"/>
                    <xs:enumeration value="Compatibility"/>
                    <xs:enumeration value="Recoverability"/>
                    <xs:enumeration value="Maturity"/>
                    <xs:enumeration value="Learnability"/>
                    <xs:enumeration value="Understandability"/>
                    <xs:enumeration value="Operability"/>
                    <xs:enumeration value="Complexity"/>
                    <xs:enumeration value="Time Behavior"/>
                    <xs:enumeration value="Resource Behavior"/>
                    <xs:enumeration value="Changeability"/>
                    <xs:enumeration value="Testability"/>
                </xs:restriction>
            </xs:simpleType>
        </xs:element>
        <xs:element name="unit" type="xs:string"/>
        <xs:element name="value" type="xs:string"/>
        <xs:element name="version" type="xs:string"/>
        <xs:element name="webSite" type="xs:anyURI"/>
        <xs:element name="writeMethod" type="xs:string"/>
</xs:schema>
```

Appendix G - IPSComp Component Description – XML Example

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- edited with XML Spy v4.2 U (http://www.xmlspy.com) by Administrator (Administrator) -->
<componentSpecification xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:noNamespaceSchemaLocation="C:\Documents and Settings\Administrator\My Documents\EMOOSE\MSc
Thesis\Development\IPSComp.xsd">
	<id>543</id>
	<name>javax.composite.SliderFieldPanel</name>
	<generalInfo>
		<version>Java</version>
		<package>javax.composite </package>
		<language>Java </language>
		<model>JavaBean</model>
		<domain>Interface</domain>
		<domain>MVC</domain>
		<os>Windows</os>
		<os>Linux</os>
		<prices>
			<price>
				<amount>43.26</amount>
				<currency>Euros</currency>
				<description>This component does not have any
discounts</description>
			</price>
			<price>
				<amount>33.99</amount>
				<currency>Euros</currency>
				<description>This price is only for partners</description>
			</price>
		</prices>
		<publisher>
			<id>SunId</id>
			<name>Sun Microsystems </name>
			<email>sun@sun.com</email>
			<webSite>http://www.sun.com</webSite>
			<phone>(1) 737 883 2694</phone>
		</publisher>
		<license>
			<name>OpenSource</name>
			<description>This is an open source software ... </description>
		</license>
	</generalInfo>
	<features>
		<methods>
			<method status="Provided">
				<mName>addInt</mName>
				<desc>This is is provided interface to add two ints</desc>
				<pre>true</pre>
				<post>true</post>
				<returnType>int</returnType>
				<paraType>int</paraType>
				<paraType>int</paraType>
				<scenarios>
					<scenario sName="First scenario">
						<scenarioReturnType>Dynamic</scenarioReturnType>
						<scenarioParaType>Dynamic</scenarioParaType>
						<scenarioParaType>Dynamic</scenarioParaType>
						<scenarioParaType>Kill</scenarioParaType>
						<scenarioParaType>Kill</scenarioParaType>
					</scenario>
					<scenario sName="Second scenario">
						<scenarioReturnType>Dynamic</scenarioReturnType>
						<scenarioParaType>Dynamic</scenarioParaType>
						<scenarioParaType>Dynamic</scenarioParaType>
						<scenarioParaType>Static</scenarioParaType>
						<scenarioParaType>Kill</scenarioParaType>
					</scenario>
				</scenarios>
```

```
		</method>
		<method status="Provided">
			<mName>multiplyInt</mName>
			<desc>This is is provided interface to multiply two ints</desc>
			<pre>true</pre>
			<post>true</post>
			<returnType>int</returnType>
			<paraType>int</paraType>
			<paraType>int</paraType>
			<scenarios>
				<scenario sName="Third scenario">
					<scenarioReturnType>Dynamic</scenarioReturnType>
					<scenarioParaType>Dynamic</scenarioParaType>
					<scenarioParaType>Dynamic</scenarioParaType>
					<scenarioParaType>Static</scenarioParaType>
					<scenarioParaType>Static</scenarioParaType>
				</scenario>
				<scenario sName="Forth scenario">
					<scenarioReturnType>Dynamic</scenarioReturnType>
					<scenarioParaType>Dynamic</scenarioParaType>
					<scenarioParaType>Dynamic</scenarioParaType>
					<scenarioParaType>Kill</scenarioParaType>
					<scenarioParaType>Static</scenarioParaType>
				</scenario>
			</scenarios>
		</method>
	</methods>
	<properties>
		<property access="ReadWrite" style="Simple">
			<pName>minimumValue </pName>
			<pType>int </pType>
			<readMethod>getMinimum </readMethod>
			<writeMethod>setMinimum </writeMethod>
		</property>
		<property access="ReadWrite" style="Bound">
			<pName>currentValue </pName>
			<pType>int </pType>
			<readMethod>getCurrentValue </readMethod>
			<writeMethod>setCurrentValue </writeMethod>
		</property>
		<property access="ReadWrite" style="Simple">
			<pName>fieldWidth </pName>
			<pType>int </pType>
			<readMethod>getFieldWidth </readMethod>
			<writeMethod>setFieldWidth </writeMethod>
		</property>
		<property access="WriteOnly" style="Simple">
			<pName>minimumSize </pName>
			<pType>int </pType>
			<readMethod>getMinimumSize</readMethod>
		</property>
		<property access="ReadOnly" style="Simple">
			<pName>preferredSize </pName>
			<pType>int </pType>
			<readMethod>getPreferredSize </readMethod>
		</property>
	</properties>
	<events>
		<event delivery="MultiCast" status="publish">
			<eType>java.Beans.PropertyChangeEvent </eType>
		</event>
		<event delivery="UniCast" status="publish">
			<eType>event1</eType>
			<addListenerMethod>addListener1</addListenerMethod>
			<removeListenerMethod>removeListener1</removeListenerMethod>
			<listenerMethods>
				<listenerMethod>
					<mName>listenerMethod11</mName>
					<returnType>int</returnType>
					<paraType>int</paraType>
```

```
                                <paraType>int</paraType>
                        </listenerMethod>
                        <listenerMethod>
                                <mName>listenerMethod12</mName>
                                <returnType>float</returnType>
                                <paraType>float</paraType>
                                <paraType>float</paraType>
                        </listenerMethod>
                </listenerMethods>
        </event>
        <event delivery="MultiCast" status="Consumed">
                <eType>event2</eType>
                <addListenerMethod>addListener2</addListenerMethod>
                <removeListenerMethod>removeListener2</removeListenerMethod>
                <listenerMethods>
                        <listenerMethod>
                                <mName>listenerMethod21</mName>
                                <returnType>String</returnType>
                                <paraType>double</paraType>
                                <paraType>char</paraType>
                        </listenerMethod>
                        <listenerMethod>
                                <mName>listenerMethod22</mName>
                                <returnType>Object</returnType>
                                <paraType>List</paraType>
                                <paraType>List</paraType>
                        </listenerMethod>
                </listenerMethods>
        </event>
</events>
</features>
<design>
<compInstances>
        <compInstance role="Master">
                <comp>javax.swing.JPanel </comp>
                <cid>panel</cid>
        </compInstance>
        <compInstance role="Support">
                <comp> java.awt.BorderLayout </comp>
                <cid>border</cid>
        </compInstance>
        <compInstance role="Client">
                <comp> javax.swing.JSlider </comp>
                <cid> slider</cid>
        </compInstance>
        <compInstance role="Client">
                <comp> javax.swing.JTextField </comp>
                <cid>field</cid>
        </compInstance>
        <compInstance role="Client">
                <comp> javax.awt.BoxContainer </comp>
                <cid> boxContainer</cid>
        </compInstance>
</compInstances>
<compositions>
        <eCompositions>
                <eComposition>
                        <constraint>
                                <inv>true</inv>
                                <pre>true</pre>
                                <post>true</post>
                        </constraint>
                        <eCompInstances>
                                <eCompInstance>
                                        <rid>slider</rid>
                                        <event>change</event>
                                        <eAction> stateChanged </eAction>
                                </eCompInstance>
                        </eCompInstances>
                        <lCompositions>
```

```
                        <lCompInstance>
                                <rid>slider</rid>
                                <callMethod> getValue </callMethod>
                        </lCompInstance>
                        <op type="|"/>
                        <lCompInstance>
                                <rid> filed </rid>
                                <callMethod> setText </callMethod>
                        </lCompInstance>
                </lCompositions>
        </eComposition>
        <eComposition>
                <constraint>
                        <inv>true</inv>
                        <pre>true</pre>
                        <post>true</post>
                </constraint>
                <eCompInstances>
                        <eCompInstance>
                                <rid>field</rid>
                                <event>action</event>
                                <eAction>actionPerformed</eAction>
                        </eCompInstance>
                </eCompInstances>
                <lCompositions>
                        <lCompInstance>
                                <rid>field</rid>
                                <callMethod>getText</callMethod>
                        </lCompInstance>
                        <op type="|"/>
                        <lCompInstance>
                                <rid>slider</rid>
                                <callMethod>setValue</callMethod>
                        </lCompInstance>
                </lCompositions>
        </eComposition>
</eCompositions>
<cCompositions>
        <cComposition>
                <container>
                        <rid>panel</rid>
                </container>
                <containees>
                        <containee>
                                <rid>boxContainer </rid>
                        </containee>
                </containees>
        </cComposition>
        <cComposition>
                <container>
                        <rid>boxContainer</rid>
                </container>
                <containees>
                        <containee>
                                <rid>slider</rid>
                        </containee>
                        <containee>
                                <rid>field </rid>
                        </containee>
                </containees>
        </cComposition>
        <cComposition>
                <container>
                        <rid>frame</rid>
                </container>
                <containees>
                        <containee>
                                <rid>sliderField </rid>
                                <location>BorderLayout.SOUTH </location>
                        </containee>
```

```
                                        <containee>
                                                <rid>logo</rid>
                                                <location>BorderLayout.CENTER </location>
                                        </containee>
                                </containees>
                        </cComposition>
                </cCompositions>
        </compositions>
    </design>
    <qualityAttributes>
        <DataEncription metric="Presence">
            <characteristic>Functionality</characteristic>
            <lifeCycle>Runtime</lifeCycle>
            <subCharacteristic>Security</subCharacteristic>
            <metric>
                <value>true</value>
                <feature>SSL ceritificate</feature>
                <unit/>
            </metric>
        </DataEncription>
        <DiskUtilization metric="Number">
            <characteristic>Efficiency</characteristic>
            <lifeCycle>Runtime</lifeCycle>
            <subCharacteristic>Resource Behavior</subCharacteristic>
            <metric>
                <value>10</value>
                <feature/>
                <unit>MB</unit>
            </metric>
        </DiskUtilization>
    </qualityAttributes>
</componentSpecification>
```

References

[1] Rui S. Moreira, Gordon S. Blair, Eurico Carrapatoso. A Reflective Component-Based & Architecture Aware Framework to Manage Architecture Composition. *Third International Symposium on Distributed Objects and Applications (DOA'01)*. September 17 - 20, 2001. Rome, Italy. p. 0187.

[2] Ranieri Baraglia, Fabrizio Silvestri. An Online Recommender System for Large Web Sites. *Web Intelligence, IEEE/WIC/ACM International Conference on (WI'04)*. September 20 - 24, 2004. Beijing, China. p. 199-205.

[3] J. Kontio. A case study in applying a systematic method for COTS selection. *18th International Conference on Software Engineering (ICSE'96)*. March 25 - 29, 1996. Berlin, GERMANY. p. 201.

[4] Jie Yang, Lei Wang, Song Zhang, Xin Sui, Ning Zhang, Zhuoqun Xu. Building Domain Ontology Based on Web Data and Generic Ontology. *Web Intelligence, IEEE/WIC/ACM International Conference on (WI'04)*. September 20 - 24, 2004. Beijing, China. p. 686-689.

[5] Nead Stojanovic, Jorge Gonzalez, Ljiljana Stojanovic. ONTOLOGER: a system for usage-driven management of ontology-based information portals. *International Conference On Knowledge Capture archive. Proceedings of the international conference on Knowledge capture*. 2003. Sanibel Island, FL, USA October 23 - 25, 2003. Pages: 172 - 179. Year of Publication: 2003. ISBN:1-58113-583-1.

[6] Jean-Christophe Mielnik, Bernard Lang, Stéphane Laurier. eCots Platform: An Inter-industrial Initiative for COTS-Related Information Sharing. *Proceedings of the Second International Conference on COTS-Based Software Systems*. Pages: 157 - 167. Year of Publication: 2003. ISBN:3-540-00562-5.

[7] Fabrizio Silvestri, Diego Puppin, Domenico Laforenza, Salvatore Orlando. A Search Architecture for Grid Software Components. *Web Intelligence, IEEE/WIC/ACM International Conference on (WI'04)*. September 20 - 24, 2004. Beijing, China. p. 495-498.

[8] Seoyoung Park, Chisu Wu. Intelligent Search Agent for Software Components. *Sixth Asia Pacific Software Engineering Conference*. December 07 - 10, 1999. Takamatsu, Japan. p. 154.

[9] Sandip Debnath, Sandip Sen, Brent Blackstock. LawBot: A Multiagent Assistant for Legal Research. *Internet Computing online IEEE*. November/December 2000 (Vol. 4, No. 6). p. 32-37.

[10] Jyrki Kontio, Gianluigi Caldiera and Victor R. Basili. Defining Factors, Goals and Criteria for Reusable Component Evaluation. *Presented at the CASCON '96 conference*, Toronto, Canada, November 12-14, 1996.

[11] Robert C. Seacord, Scott A. Hissam, Kurt C. Wallnau. Agora: A Search Engine for Software Components. *Internet Computing online IEEE*. November/December 1998 (Vol. 2, No. 6). p. 62-70.

[12] L. Stojanovic, N. Stojanovic, J. Gonzalez, R. Studer. The OntoManager - a system for the usage-based ontology management. Proceeding, *ODBASE 2003*, 3-7 November 2003, Catania, Sicily (Italy).

[13] Hideki Hara, Shigeru Fujita, Kenji Sugawara, Chiba Institute of Technology. Reusable Software Components Based on an Agent Model. Seventh International Conference on Parallel and Distributed Systems: Workshops (ICPADS'00 Workshops). July 04 - 07, 2000. Iwate, Japan. p. 447.

[14] John Davies, A. Duke, and York Sure (2003). OntoShare - A Knowledge Management Environment for Virtual Communities of Practice. Proceedings of the *2nd International Conference on Knowledge Capture (K-CAP2003)*, 23-26 October 2003, Florida, USA. Edited by . ACM Press.

[15] Czarnecki, K., Dittmar, T., Franczyk, B., Hoffmann, R., Kühnhauser, W., Langhammer, F., Lenz, B., Müller-Schloer, C., Unland, R., Weber, M., Weissenbach, H., Westerhausen, J. CompoNex: A Marketplace for Trading Software Components in Immature Markets. Proceedings *Net.ObjectDays 2003*. Overhage, S., Thomas, P. (2003). Transit (ISBN 3-9808628-2-8), p. 145-163.

[16] Sven Overhage. Towards a Standardized Specification Framework for Component Development, Discovery, and Configuration. *WCOP 2003. Eighth International Workshop on Component-Oriented Programming.* Monday, July 21, 2003 At ECOOP 2003, Darmstadt, Germany (July 21-25, 2003).

[17] Johannes Maria ZAHA, Alexander KEIBLINGER, Klaus TUROWSKI. Component Market Specification Demand and Standardized Specification Of Business Components. *1st International workshop Component Based Business Information Systems Engineering* September 2nd, 2003 - Geneva, Switzerland.

[18] Vishnu Kotrajaras. Towards an Agent-Searchable Software Component Using CafeOBJ Specification and Semantic Web. *Workshop on Formal Aspects of Component Software (FACS 03)*. Pisa, Italy, 8-9 September 2003.

[19] World Wide Web Consortium Issues RDF and OWL Recommendations. Semantic Web emerges as commercial-grade infrastructure for sharing data on the Web. http://www.w3.org/2004/01/sws-pressrelease.html.en.
[20] F. McCarey, N. Kushmerick. RASCAL: A Recommender Agent for SoftwComponents in an Agile Environment. Proceedings of the *15th Artificial Intelligence and Cognitive Science Conference,* Castlebar, Ireland, Se2004.
[21] Juan P. Carvallo, Xavier Franch, Carme Quer, Marco Torchiano. Characterization of a Taxonomy for Business Applications and the Relatioships Among Them. *3rd International Conference on COTS-Based Software Systems. ICCBSS 2004*. 1-4 February 2004.
[22] C. Brewster and K. O'Hara. Knowledge Representation with Ontologies: The Present and Future. *IEEE Intelligent Systems*, 19(2):72 - 81, may 2004.
[23] R. Braga, M. Mattoso, and C. Werner. The use of mediation and ontology technologies for software component information retrieval. In Proceedings of the *2001 Symposium on Software Reusability: putting software reuse in context*, pages 19-28. ACM, 2001.
[24] M. Missikoff and F. Taglino. SymOntoX: A Web-Ontology Tool for eBusiness Domain. In Proceedings of the *Fourth International Conference on Web Information Systems Engineering (WISE'03),* pages 343-346. IEEE, 2003.
[25] A. Rector. Modularisation of Domain Ontologies Implemented in Description Logics and related formalisms including OWL. In Proc. of *Knowledge Capture (KCAP'03)*, pages 121-128. ACM, 2003.
[26] C. Pahl. Ontology-based Description and Reasoning for Component-based Development on the Web. In Proceedings of *SAVCBS'03-ESEC/FSE'03 Workshop*. ACM, 2003. Septiembre 1-2, 2003. Helsinki, Finland.
[27] M. Tallis, N. Goldman, and R. Balzer. The Briefing Associate: A Role for COTS Applications in the Semantic Web. In Proceedings of the *Semantic Web Working Symposium (SWWS)*, 2001.
[28] N. Tansalarak and K. Claypool. XCM: A Component Ontology. In *OOPSLA'04 Workshop - Ontologies as Software Engineering Artifacts*. 24-28 October 2004, Vancouver, British Columbia, Canada.
[29] Guy Pierra. The PLIB Ontology-based approach to data integration. *18th IFIP World Computer Congress (WCC\'2004)*. 2004.
[30] Guy Pierra. Context-explication in conceptual ontologies: PLIB ontologies and their use for industrial data. *Technical report : Research Report LISI/ENSMA 04-001*. 2004.
[31] Guy Pierra and Hondjack Dehainsala and Yamine Ait Ameur and Ladjel Bellatreche. Base de données à base ontologique :principe et mise en ouvre. *Journal Ingénierie des systèmes d'information*. 2005.
[32] Nicola Guarino, Claudio Masolo, Guido Vetere. OntoSeek: Content-Based Access to the Web. *IEEE Intelligent Systems.* Volume 14, Issue 3 (May 1999). Pages: 70 - 80. Year of Publication: 1999. ISSN: 1094-7167.
[33] Overhage, S., Thomas, P. WS-Specification: Specifying Web Services Using UDDI Improvements. In: Chaudri, A. B., Jeckle, M., Rahm, E., Unland, R. (eds.): Web, Web Services, and Database Systems. Lecture Notes in Computer Science (LNCS 2593), Springer, Berlin (2003): 100-118.
[34] Kokkinaki, A.I., N. Karakapilides, R. Dekker, and C. Pappis. A web-based recommender system for End-of-use ict products. In Proceedings of the *Second IFIP Conference on E-commerce, E-business, E-government*, October 2002.
[35] S. Varadarajan, A. Kumar, D. Gupta, and P. Jalote. ComponentXchange: An E-Exchange for Software Components. In Poster Proceedings of the *Tenth International World Wide Web Conference (WWW 10)*, 2001
[36] Peter FETTKE, Peter LOOS. A Proposal for Specifying Business Components. *1st International workshop "Component Based Business Information Systems Engineering".* September 2nd, 2003 - Geneva, Switzerland.
[37] Naiyana Tansalarak and Kajal T. Claypool. CoCo: Composition Model and Composition Model Implementation. *Technical Report 2004-006*, Department of Computer Science, University of Massachusetts - Lowell, June 2004. http://www.cs.uml.edu/techrpts/reports.jsp.
[38] Bobeff G. Noyè Jacques. Component Specialization. *PEP M'04* August 24-26m 2004. Verona, Italy.
[39] R. de Souza, M. Costa, R. Bragga, C. Werner, M. Mattoso. Software Components Reuse Through Web Search and Retrieval. Computer Science Department, Federal University of Rio de Janeiro, Brazil. Department of Computer Science – CTU/UFJF.

[40] O. Constant, A. Réquilé, B. Yap. Deriving Action-Based Semantics from Learning Repositories. Proceeding of the *First International Conference on Information Technology & Applications.(ICITA 2002)*. November 25-28 2002. BATHURST, AUSTRALIA. IEEE, ISBN: 1-86467-114-9 - Track 2: T in Multimedioa; Computer Networking; and Database Interface.
[41] Haining Yao, Letha Etzkorn. Towards A Semantic-based Approach for Software Reusable Component Classification and Retrieval. ACM Southeast Regional Conference Proceedings of the *42nd annual Southeast regional conference*. Huntsville, Alabama. Session: Software engineering #1. Pages: 110 - 115. Year of Publication: 2004. ISBN:1-58113-870-9. Publisher ACM Press. New York, NY, USA.
[42] Sugumaran, Vijayan; Storey, Veda C. A Semantic-Based Approach to Component Retrieval. *The DATA BASE for Advances in Information Systems* – Summer 2003, Vol. 34, No. 3, p. 8-24.
[43] Massimo Paolucci, Takahiro Kawamura, Terry R. Payne, Katia P. Sycara. Importing the Semantic Web in UDDI. Lecture Notes In Computer Science; Vol. 2512 archive. Revised Papers from the *International Workshop on Web Services, E-Business, and the Semantic Web*. Pages: 225 - 236. Year of Publication: 2002. ISBN:3-540-00198-0. Publisher Springer-Verlag London, UK.
[44] Sivashanmugam, K.; Verma, K.; Sheth, A.; Miller, J. Adding Semantics to Web Services Standards. The 2003 *International Conference on Web Services (ICWS'03),* June 2003.
[45] Stefan Decker, Prasenjit Mitra, Sergey Melnik. Framework for the Semantic Web: An RDF Tutorial. *Internet Computing online*. November/December 2000 (Vol. 4, No. 6). p 68-73.
[46] Michael Klein, Birgitta Konig-Ries. Combining Query and Preference An Approach to Fully Automatize Dynamic Service Binding. Proceedings of *the IEEE International Conference on Web Services (ICWS'04)*. June 06 - 09, 2004. San Diego, California. Publication Date: June 2004. p. 788.
[47] Gerald C. Gannod, Sushant Bhatia. Facilitating Automated Search for Web Services. *IEEE International Conference on Web Services (ICWS'04).* June 06 - 09, 2004. San Diego, California. Publication Date: June 2004. p. 761.
[48] Annya Réquilé-Romanczuk, Alejandra Cechich, Anne Dourgnon-Hanoune. Towards a Knowledge-Based Framework for COTS Component Identification*. 27th International Conference on Software Engineering ICSE'05-MPEC'05*. May 21st, 2005, St. Louis, Missouri, USA.
[49] T.R. Gruber. A Translation Approach to Portable Ontology Specifications. Knowledge Acquisition, 5:199-220, 1993.
[50] Hall, R.J. Generalized Behavior-Based Retrieval. Proceedings of the Fifteenth International Conference on Software Engineering, Baltimore, MD, May 1993. p. 371 - 380.
[51] Naiyana Tansalarak and Kajal Claypool. Finding a Needle in the Haystack: A Technique for Ranking Matches between Components. *Eighth International SIGSOFT Symposium on Component-based Software Engineering (CBSE 2005): Software Components at Work*. St. Louis, Missouri. May 14-15 2005.
[52] Bertoa, M. F., Vallecillo, A. Quality Attributes for COTS Components. In: Proceedings of the *6th ECOOP Workshop on Quan-titative Approaches in Object-Oriented Software Engineering (QAOOSE 2002)*. June 11th, 2002. Málaga Spain.
[53] Joaquina Martín-Albo, Manuel F. Bertoa, Coral Calero, Antonio Vallecillo, Alejandra Cechich, Mario Piattini. CQM: A Software Component Metric Classification Model*. 7th ECOOP Workshop on Quantitative Approaches in Object-Oriented Software Engineering (QAOOSE'2003).* Darmstadt, Germany. Tuesday, July 22nd, 2003.
[54] Luis Iribarne, Carina Alves, Jaelson Castro, and Antonio Vallecillo. A non-functional approach for cotscomponents trading. In Proc. of *WER 2001*, Buenos Aires, Argentina, 2001.
[55] A. Podgurski & L. Pierce. Behavior sampling: a technique for automated retrieval of reusable components. In Proc. *14th International Conference on Software Engineering*, 349-360. New York, N.Y.: The Association for Computing Machinery, Inc. 1992.
[56] Michael Soden, Hajo Eichler, Joachim Hoessler. Inside MDA: Mapping MOF2.0 Models to Components. *First European Workshop on Model Driven Architecture with Emphasis on Industrial Application*. March 17-18, 2004. University of Twente, Enschede, The Netherlands.
[57] Daniel Exertier, Benoit Langlois, Xavier Le Roux. PIM Definition and Description. *First European Workshop on Model Driven Architecture with Emphasis on Industrial Application*. March 17-18, 2004. University of Twente, Enschede, The Netherlands.
[58] Uche Ogbuji. XML, The Model Driven Architecture, and RDF. *XML Europe 2002. Down to Business: Getting serious about XML*. 23, 24 May 2002. Bacelona, Spain.

[59] Ivan Kurtev, Klaas van den Berg. Model driven architecture based XML processing. Proceedings of the *2003 ACM symposium on Document engineering*. Grenoble, France. SESSION: Document based architecture & applications. Pages: 246 - 248. Year of Publication: 2003. ISBN:1-58113-724-9.
[60] A Proposal for an MDA Foundation Model. An ORMSC White Paper. V00-02. ormsc/05-04-01.
[61] Tewfik Ziadi, Bruno Traverson, Jean-Marc Jézéquel. From a UML Platform Independent Component Model to Platform Specific Component Models. *Workshop in Software Model Engineering.* Tuesday October 1st 2002. Dresden, Germany.
[62] Jack Greenfield. UML Profile For EJB. Java Specification Request JSR-000026 UML/EJB(TM) Mapping Specification 1.0 Public Review Draft. Rational Software Corporation. http://jcp.org/jsr/detail/26.jsp.
[63] PIM to PSM mapping techniques. *First European Workshop on Model Driven Architecture with Emphasis on Industrial Application*. March 17-18, 2004. University of Twente, Enschede, The Netherlands. MASTER-2003-D5.1-V1.0-PUBLIC. December 2003.
[64] http://www.omg.org/mda
[65] Richard Monson-Haefel. Enterprise JavaBeans, Second Edition. March 2000. ISBN: 1-56592-869-5.
[66] Chuck McManis. Take a look inside Java classes. Learn to deduce properties of a Java class from inside a Java program. Java Indepth. http://www.javaworld.com/javaworld/jw-08-1997/jw-08-indepth.html.
[67] http://java.sun.com, http://java.sun.com/products/ejb, http://java.sun.com/j2ee/1.4/docs/tutorial/doc.
[68] Enterprise JavaBeansTM Specification, Version 2.1. Sun Microsystems. Version 2.1, Final Release. November 12, 2003.
[69] Bézivin Jean. From Object Composition to Model Transformation with the MDA. Proceedings of the 39th International Conference and Exhibition on Technology of Object-Oriented Languages and Systems (TOOLS39). Page: 350. Year of Publication: 2001. ISSN:1530-2067. Publisher IEEE Computer Society Washington, DC, USA.
[70] Joaquin Miller, Jishnu Mukerji. MDA Guide Version 1.0.1. Copyright © 2003 OMG. Document Number: omg/2003-06-01. 12th June 2003.
[71] Architecture Board MDA Drafting Team. Model Driven Architecture A Technical Perspective. Draft 21st February 2001. Document Number ab/2001-02-04.
[72] M. D. Mcllroy, Mass-produced software components. In *Software Engineering Concepts and Techniques, NATO Conference on Software Engineering*, 1969.
[73] Weaver James, Kevin Muckar, Crume James, Phillips Ron. Beginning J2EE 1.4. Wrox Press Ltd. 2003. United States. ISBN 1-86100-833-3.
[74] XML Schema Part 0: Primer Second Edition. W3C Recommendation 28 October 2004. http://www.w3.org/TR/xmlschema-0/#Intro.