# 4
# Description of the Repository

Since there are dissimilar implementation architectures, platforms and programming languages to develop software agents, to model heterogeneous agents in a generic way is a challenge. In this section, we analyze the main requirements that were used to propose an innovative methodology to allow the reuse of agent-based artifacts. We develop a semantic-based approach for building repositories of a broad variety of agent-oriented artifacts which are potentially reusable in many situations and across various application domains. Based on this approach, we implement a prototype repository capable of providing an effective identification, representation, storing and retrieval mechanism of reusable agent components modeled mainly according to their functionalities, structure and interfaces. Next, we describe our developed repository and its internal architecture, and give an explanation about the methods used for agent component modeling, recommendation, subscription and retrieval.

Our notion of the development for agent reuse process is briefly illustrated in Figure 1, in which there are four stages to be followed using the knowledge base, which starts from identifying agent components related to a given description, modeling and classifying agents, searching reusable agent components according some parameter and delivering them to the end-user.

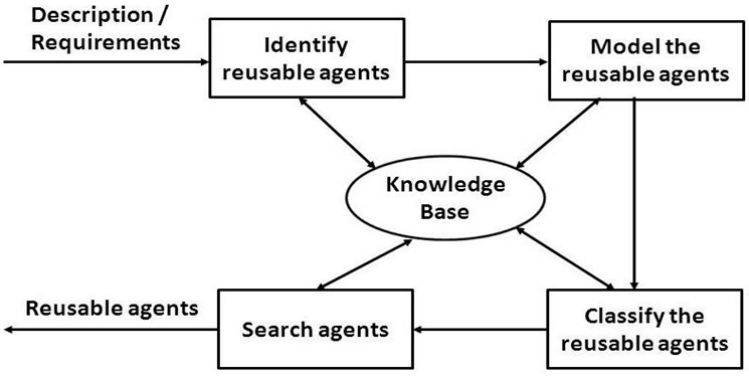


**Figure 1: General Process of Development for Agent Reuse.**

The identification of a software agent given a user's requirement is elementary based on the functionalities and interfaces of the agent. We identify some attributes of an agent which are expected to be reusable. These properties include a set of common functionalities, well-defined interfaces, general descriptions, and the ability to be reusable in several different software applications.

One way to make agents easier to find and retrieve from the repository is using metadata. In case of software reuse, metadata is a representation that describes a software asset from all aspects including how to use it and how it relates to other assets which helps locating an asset and determining if it is suitable to be used. Our metadata avoids the fact of having a purely textual specification of an agent. We use semantic web technologies to represent the agents, offering also a methodology for building an agent-oriented system, with a focus on an agent's detailed specification.

The classification process employs natural language and domain knowledge to support the potential user's selection process and contributes to the development of a standard vocabulary for software attributes. A value of a controlled and structured vocabulary relies on a predefined set of keywords used as indexing terms. These keywords are derived and defined by software engineers, and designed to best describe or represent concepts relevant to the application domain.

The system provides an interface that implements the searching and browsing mechanisms to allow the user to view and browse identifying agents that offer certain functions in a particular domain category or with particular specification, furthermore submit search queries. The user query, made in natural language, is translate into semantic representation formats in order to overcome the keyword-based barrier, and augmenting retrieval recall and precision. The search process not only discovers the requested artifacts based on a user query, but additional relevant ones that the user may not be aware of. One of the difficulties in the search process is mainly to define the criterion for measuring the relevance of an agent respect to a query. This approach enhances the retrieval by semantically matching between a user query semantic representation and agent-oriented artifact semantic descriptions against a domain ontology.

The task of searching artifacts, in our case, is supported by a recommendation system, whose facility provides assistance to finding artifacts by employing hyper-textually connected data.

In contrast to existing repositories that only retrieve a limited set of artifacts, the proposed meta-model enables the recommendation of interrelated agents that advance the search process and the subscription of agent specific application domains by users. These subscriptions will update users when new agents are inserted or another modification is realized in their domains.

Accordingly, we posit that applying a semantically enhanced approach to agent model, classification, sharing, and searching will significantly increase the efficiency of retrieving software agent-oriented artifacts.



## 4.1. Architecture of the Repository

The reuse in the agent-oriented architecture relies on a set of related specifications of agents, which defines how reusable agents should be specified and advertised to users. These specifications explain how agents can be discovered and reused, and how they interact. A critical step in the process of reusing existing agent-based artifacts for building applications is the discovery of potentially relevant artifacts.

Our repository does not only facilitate retrieving candidate agent components which match the query, but also permits browsing among the components that share some functionality or are in the same application domain. Moreover, the repository proposes a recommendation system of agent components that are related according some predefined parameters and offers a subscription service of agents grouped in categories. Figure 2 summarizes an overview of the repository with the main operations that can be executed on it.

Hence, the main goal of the repository is to model, classify and retrieve existing agent components into various groups based on their characteristics. Users can search through semantic-based retrieval mechanisms for appropriate agent components from the repository according to one or more characteristics.

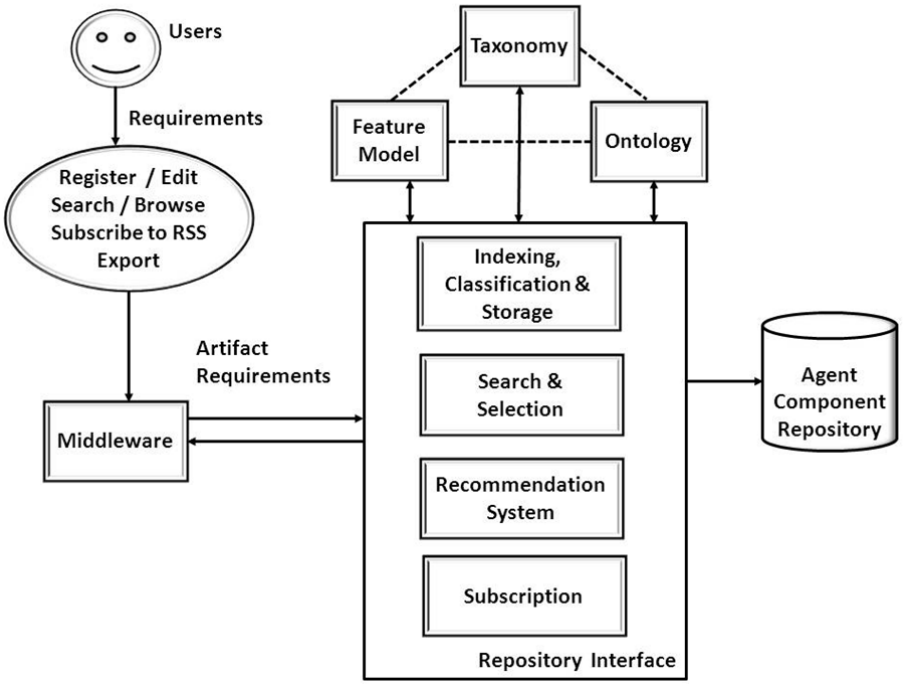


**Figure 2: Overview of the Repository.**

In the repository, agent components can be added (self-contained and encapsulated in a package JAR or ZIP files, for example), edited and replaced by newer versions over time meaning that multiple versions of the same agent component may exist simultaneously in the system. The repository has support for the following operations on agent components.

- *Register*: In order to register an agent component, a description of the agent has to be provided by the user. Once the agent is uploaded, other users of the system are notified through their preferred notification channel (email or RSS). Users can edit only agent components added by them. Users that have already made use of earlier versions of this edited agent component are informed about the change and can decide whether to update their agent instance or not. Each update is validated to avoid repeated components with the same functionalities in different shapes, which might provoke ambiguity in the search and avoidable recurrent development.
- *Indexing*: After an agent component is registered, automatically the information related to the component is indexed and its index terms are extracted as well, to help the recommendation and search processes. An

index term is a word or group of consecutive words that captures the essence topic of a document [50]. In order to properly index agent components, a semantic knowledge is used rather than only operation and attribute-level names, types, formal specifications or natural language comments.

- *List*: To list all agent components of a particular category.
- *Browse*: It reduces the searching space where users can easily browse available components in the repository together with their older versions, and choose the one needed. It also helps the end-user to be acquainted with the components already stored in the repository.
- *Recommend*: The user can establish relationships among agents already stored in the repository. The description content of the component is automatically read and indexed by the system. Afterwards, the system will recommend to the user which agent components, already stored too, are similar to the new one attending their properties. This recommendation facilitates the search.
- *Delete*: To delete an agent component. Users can delete only agent components added by them.
- *Export*: To allow the download of an originally created agent component from the repository, a newer version of existing component, or a modified existing component as a different component and save its predecessor's history.
- *Search*: Users are able to search and retrieve agents with some specific purposes. There are different types of search, all based on the semantic knowledge, for example keyword-based approach, custom search facets, tag-based queries and an advance search methods by some properties of the agents like programming language, platform, behaviors and interfaces.
- *Subscribe*: Users can subscribe to a RSS feed of agents according to a specific category. If there is any change in a subscribed channel, the subscribed users will receive a notification.

The repository has an agent-based architecture like depicted in Figure 3. Multi-agent systems are a promising technology for information retrieval [80][5]. An agent-based approach means that IR systems can be more scalable, flexible, extensible, and interoperable. Agents need a way to process and understand their information, both on the level of individual items as well as collection-wide entities. In this architecture agents are equipped with domain ontologies and use the semantic knowledge to provide efficient and accurate results.

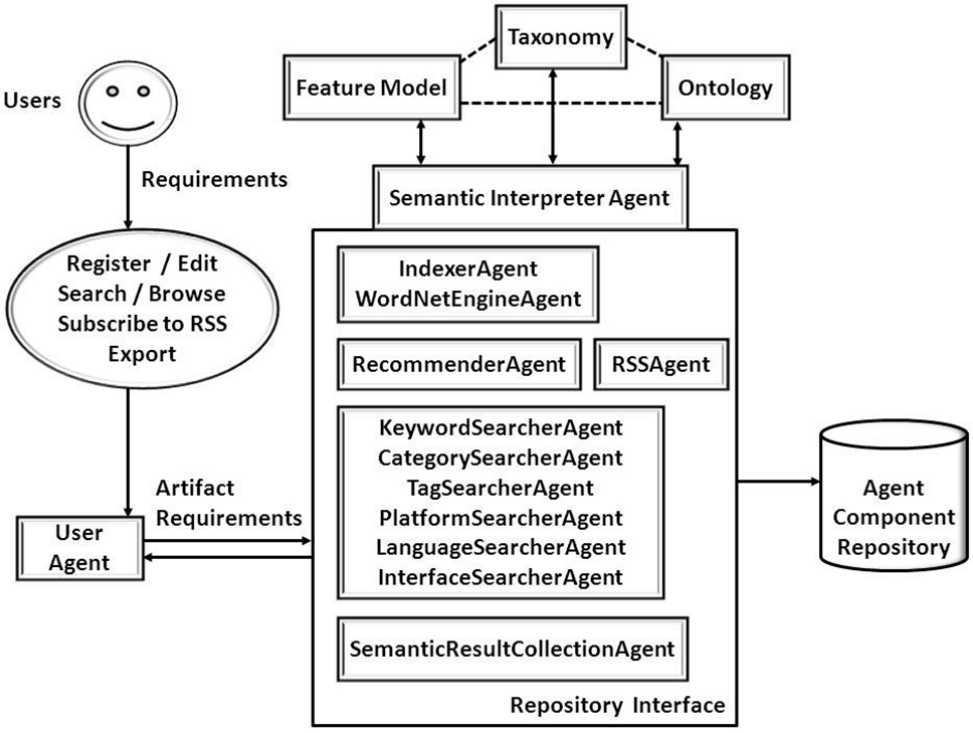


**Figure 3: Multi-Agent Architecture of the Repository.**

The architecture of the repository is formally represented by an UML class diagram depicted in Figure 4 (all operations in each class were removed to makes possible the easy visualization and understanding of the diagram).

*Agent Artifact* is the main class in the repository. It represents the agent component. *Message Interface* defines the interface of the agent component, which declares how the agent interacts in its environment. We define a *Middleware* to listening user's requests allowing the user to define his requirements, and the *User Agent* therefore replicates the requests to the subsystems responsible for the execution of such requests. Depends on the user

requirements, *User Agent* delegates the actions to execute to the *Recommender Agent* and whatever type of searching agent like *Tag Searcher Agent* to find the matched software artifacts.

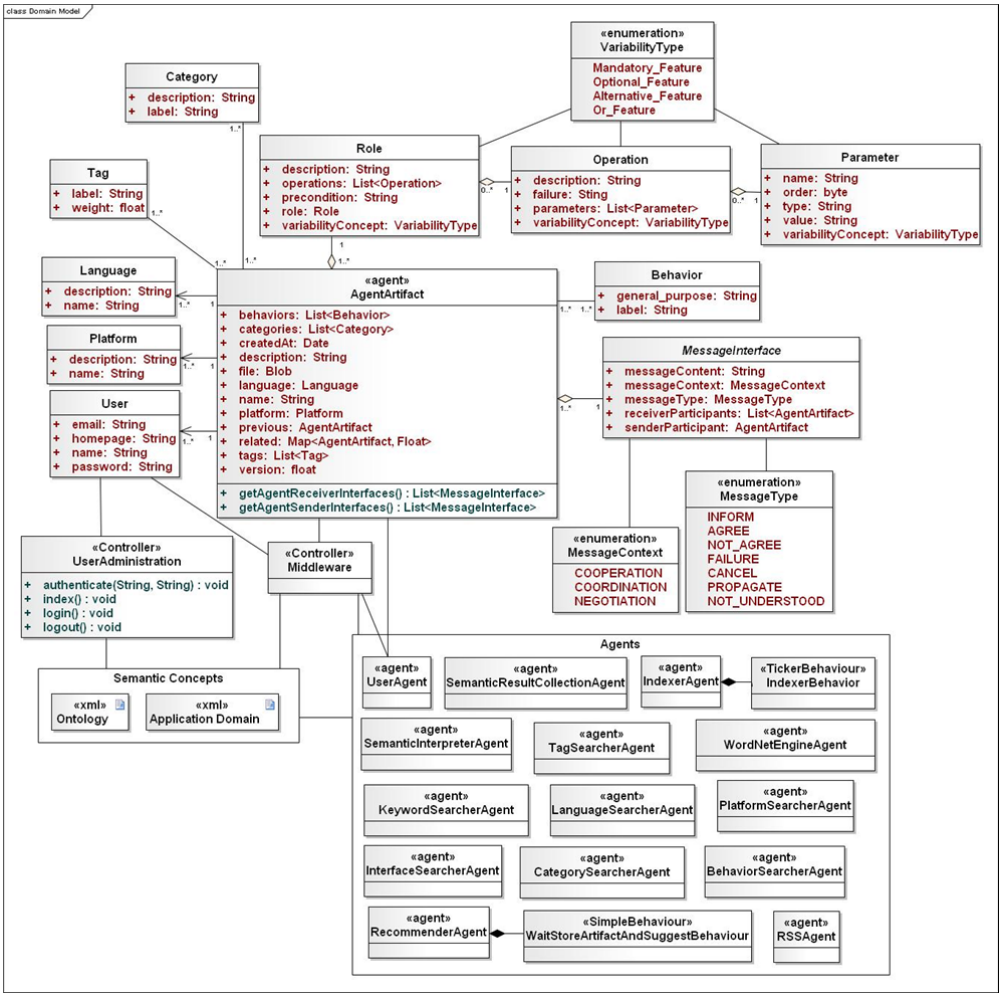


**Figure 4: UML Class Diagram of the Repository.**

### 4.1.1. Agent Modeling

The (re-)utilization of software components, especially if we are not the developer of them, is a difficult operation. To succeed at a reasonable cost, the software components must be properly described. It is fundamental not only to have metadata that give a detailed description of characteristics including what the purpose of the component is, but also how the component can be interrelated with

others, or what are its needs. Hence, modeling software components turns into a challenge.

In the agent-oriented work to date, each methodology has its own concepts and system structure, without a common factor among them. Consequentially, when trying to create a new methodology an agent meta-model becomes critical.

In this context, in order to help agent software engineers, we have defined a new type of metadata schema that details all the characteristics about agent components, and supports the storage and retrieval of software agents considering domain semantic information based on ontologies and taxonomies. In addition, we also state a model to describe and represent explicitly the agent variability making use of a feature model.

Although agents are developed under dissimilar architectures, platforms and languages and agent components are different in their interfaces, internal architecture and functionalities, we observed they could be modeled within a standard representation. Moreover, it is noted that software agents or some specific parts of them are not reused in all projects in the same way, but there are also many points of variability among these software artifacts.

Variability specification is crucial on the representation of reusable artifacts because it distinguishes how the common and variable modeling concepts differ on the applications of a family [47]. Such notion influences directly in the selection and adaptation of the concepts for future product composition. The variability is expedient to modularize changeable parts of the internal agent architecture and to make them satisfactorily generic for reuse in different contexts. The variability facilitates the design of agent component that supports various roles with alternatives operations, depending on current context and end-user preferences.

To compose our retrieval system, we propose an agent component structure modeling following the semantic of: (i) feature modeling in order to manage agent component commonality and variability and, (ii) a meta-model that provides a common and essential representation of heterogeneous agents.

The selective partitioning of the agent specification variability allows agent-oriented artifact reuse and adaptation according to the requirement for many different reuse contexts. The specification of the agents in the repository is composed of the following characteristics.

- *Roles*: Mandatory feature. Define the purposes and functionalities in an application domain of the agent or its identification within a group. There could be a precondition to perform a role. In order to a role be accomplished, it has a set of operations.
- *Operations*: Optional feature. Sequences of actions that an agent can perform to achieve one or more of its respective purposes. There could be an action to perform if the operation failures.
- *Parameters*: Optional feature. The set of conditions required to execute the operations. A parameter has a name, type, value, and an order.

Figure 5 illustrates the feature model of an agent component of the repository.



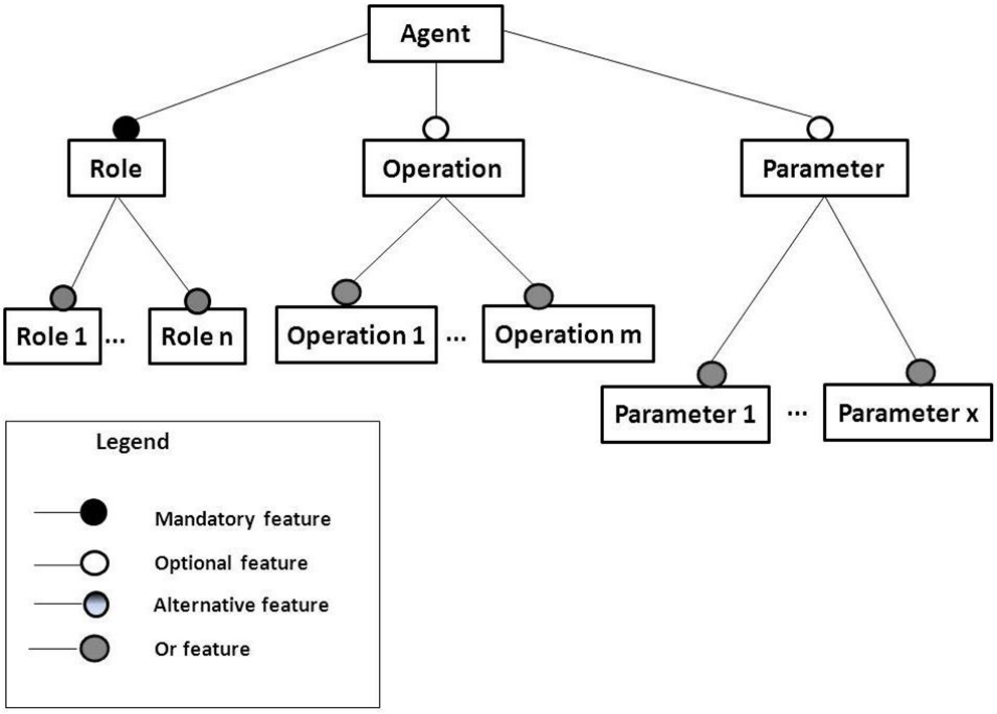


**Figure 5: Feature Model of an Agent Component.**

A concrete example of feature model of an agent is illustrated in the Figure 6. It consists on an agent that buys a specific book at the lower price at online seller libraries. The lowest price can be interpreted in three different ways: (i) the cheapest price of the book from all libraries without taking into account the

deliver price, or (ii) the cheapest price of the book plus the deliver, or (iii) no matter what the prices of the book and deliver are, the book is seller in the first library it is available.

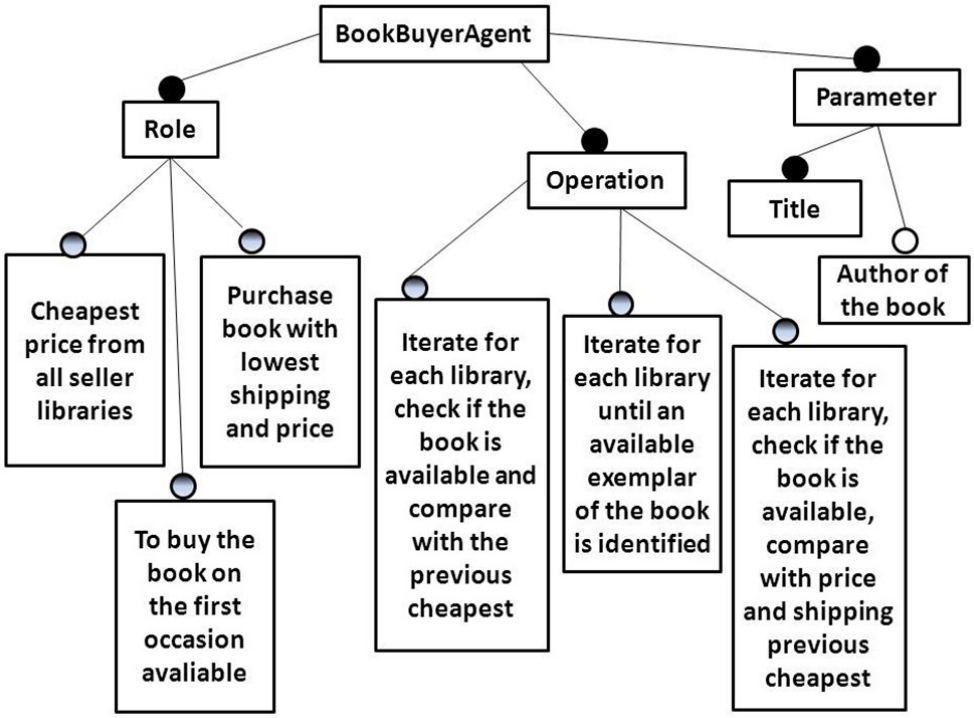


**Figure 6: Feature Model of a Book Buyer Agent.**

Ontologies for formally representing software agents would enable knowledge reuse and a standardized model for the cataloguing of agent-based artifacts. Ontologies are also relevant for describing interfaces and architecture of agent components, in addition to establishing how a software agent can be defined or classified; and the concepts, dependencies and relationships that apply to all or a subset of the totality them. An importance advantage of ontologies to maximize the reusability is also to allow the extensibility of agents, i.e., maximal possibility to extend the features of an agent without breaking the previous architecture neither the reusability of the agent in the system that contains it.

We define a meta-model to translate the dissimilar agent components into a comprehensible and logical representation. This common and essential representation establishes a precise and formal description that shares the

understanding of agent components regardless of the types of components that are in the repository, and their domain relationships. Therefore, heterogeneous agent components developed using the many existing implementation frameworks and languages can be stored and retrieved without losing any of their own characteristics.

For structuring the meta-data with the information of agents provided by the user, the system uses an ontology. After a thorough examination of available research on ontologies that describe software agents (functionalities, structure and interfaces), including the Watson Semantic Web Search[4], no appropriate result for modeling the common structure of heterogeneous agent components was found. Hence, we complement existing ontologies since some agent characteristics are not still covered, for example the structure and interfaces of heterogeneous agent components. Therefore, we propose the ontology depicted in Figure 7 to formalize the description of heterogeneous agent components. The description of the other concepts is showed in the appendix A.

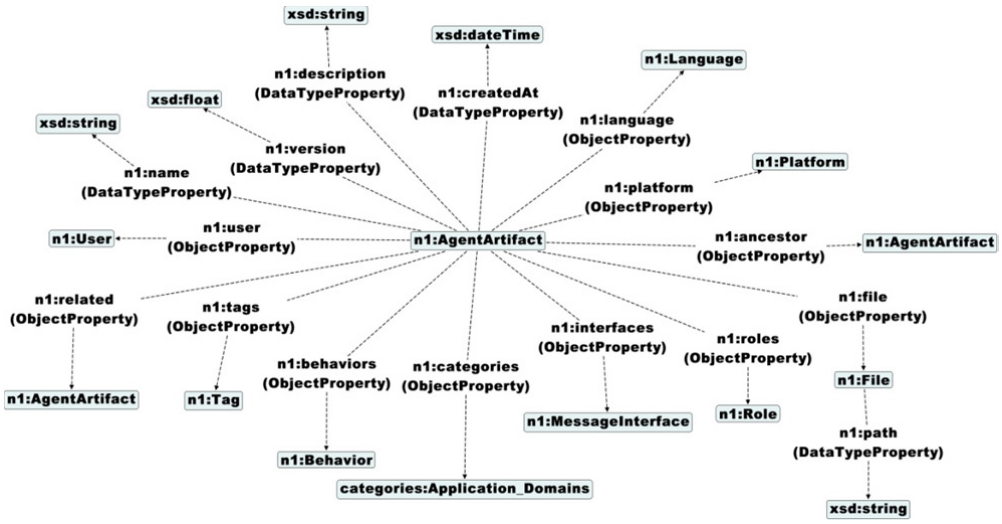


**Figure 7: Ontology to Model an Agent Component.**

A reusable agent component should have a number of attributes that are essential in determining the appropriateness of this construct in the reuse process.

[4] http://kmi-web05.open.ac.uk/WatsonWUI/

These attributes, which are considered as meta-data for the agent component specification, are the following.

- *ID*: The identifier given to the agent. It is unique in the system.
- *Name*: The name given to the agent.
- *Description*: A description of the agent in natural language, which can describe whatever information related to the agent.
- *Version:* The development version of the agent.
- *Ancestor:* If the version is not the first one, so the agent is a new one based on its ancestor.
- *Date:* The date when the agent was developed, in the format yyyy-MM-dd.
- *Language:* The programming language the agent was developed with.
- *Platform:* The platform the agent was developed with.
- *User*: The user responsible for implementing the agent.
- *Behaviors:* The characteristics of the agent according its nature, like autonomy, reactiveness, etc.
- *Roles:* The roles the agent performs.
- *Operations:* The operations each role executes.
- *Parameters:* The requirements an operation has to execute.
- *Categories:* The categories associated to kind of roles.
- *Tags:* The tags the agent can be described with a few of words.
- *File:* The file which contains the agent. The file path is the physical location of the agent in the repository.
- *Related:* There are the agents related to the current one respect to some characteristics (description, roles, ancestor, interfaces, categories and tags).
- *Interfaces:* In diverse architectures, software agents interactive with others in different way. There is not a standard format to document agent's interfaces like software components, based on the interfaces of public functions that include restrictions on the behavior of objects, such as the order in which the functions/operations should be invoked. To support a communication, compatibility or interoperability among all of these heterogeneous agents, we propose an interface model that

describes the context (cooperation, coordination, and negotiation) of relations among the agents, the types of messages the agent senders or receives (Inform, Agree, Not_Agree, Cancel, Failure, Not_Understood, Propagate, among others), the content of the message that can include ontologies, and the other agent participants.

An individual example of this ontology is depicted in Figure 8. We refer to the same agent explicated in Figure 6. We did not expand the behaviors *Reactive Agent* and *Interact Agent* neither the tags *commerce* and *Book trading*, to help to the visualization, since their representation is similar to *Autonomous Agent* and *Book Buyer* respectively.

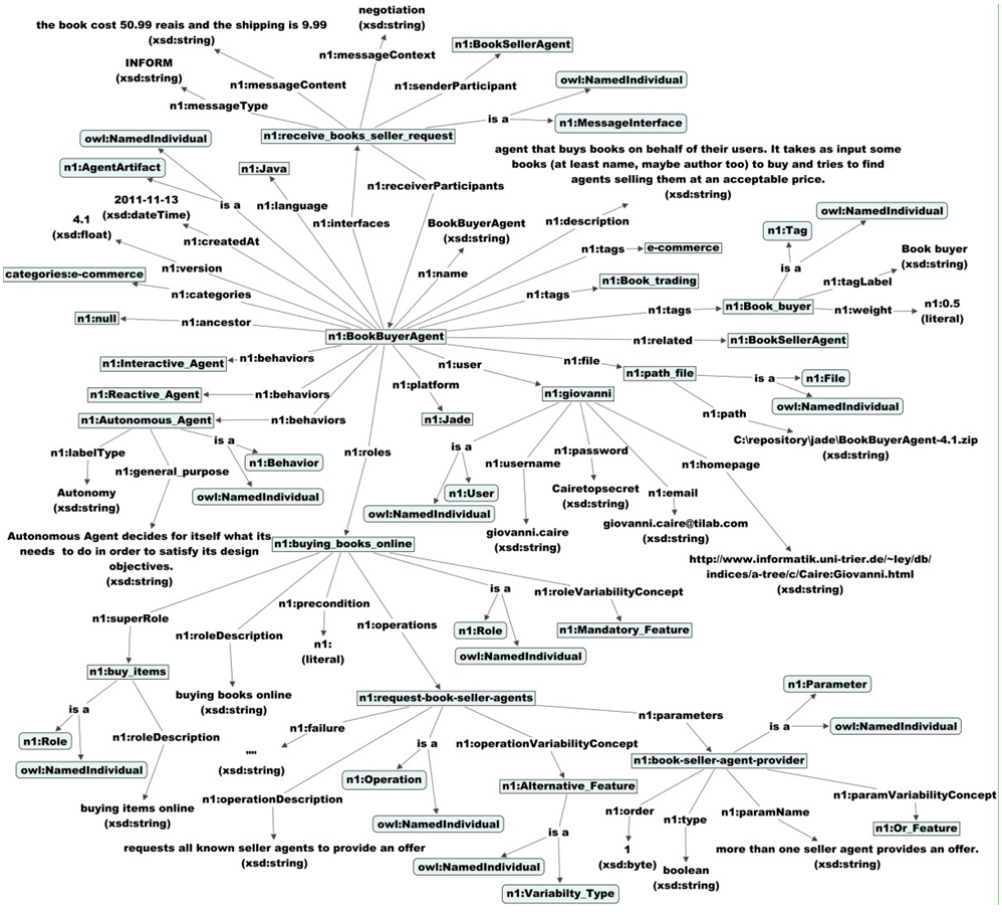


**Figure 8: An Individual of the Ontology.**

### 4.1.2. Classifying Agent Components

Our repository is organized in such a way that locating the most appropriate agents is easy for the current user. In particular, the repository assists the user in locating agents that meet some specified functionality.

Classifying agent-based artifacts allows software engineers to organize collections of them into structures that they can look easily for in future searches. Hierarchical taxonomies of application domains are been adopted, producing a manageable grouping of agent components according to established criteria, like the tasks they perform [19] and according to the agent's behaviors, but there are several perspectives to classify existing software agents, which are not unique. The proposed taxonomies are used to identify and associate general categories corresponding within domain concepts and behaviors, which map onto the common properties of agent components and to enable agents to have shared understanding of the semantics for standard terms used in the search. These concepts are represented by facets in the taxonomy. The taxonomies are built using one or several facets that compose these agent characteristics, using a common vocabulary that is familiar to the domain artifacts.

User's queries can be represented and organized more efficiently following pre-defined taxonomies, because the retrieval system can be used without any prior knowledge about the existing terms and their relationships. The taxonomies facilitate to the user during a search, to access to not only the agents related to keywords in the query, but also those agents that are interrelated semantically to them, such as those with equivalent characteristics or terms, reducing the search space and making the search results more relevant. The taxonomies target concepts and relationships that make them more understandable and usable during the search.

Several knowledge categories can be related with agents like intelligent human-computer interface agents and adaptive user modeling agents, personal knowledge retrieval agents, mobile software technologies, cooperative software agents (e.g., resource discovery and, mediators and facilitators), etc. Firstly, agents may be grouped by their mobility, i.e., by their ability to move around some environment. This yields the classes of static or mobile agents. Secondly,

they may also be classified as reactive and thirdly, along several attributes which ideally and primarily they should exhibit [55]: including autonomy, cooperation and learning.

These characteristics of agents are used to derive some behaviors of agents included in our taxonomy shown in Figure 9.

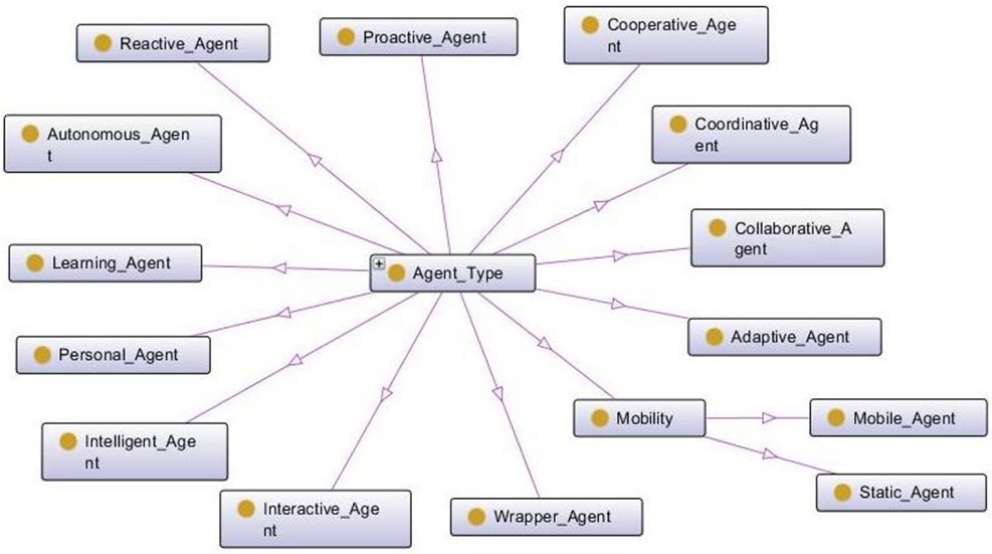


**Figure 9: Agent Behaviors Taxonomy.**

Agents may be likewise grouped by their roles in dissimilar application domains, like Figure 11 shows, e.g., WWW information gathering agents that basically help the management of the vast amount of information in wide area networks like Internet. This type of agent is named Information or Internet agents. Other example is the personal software agents that help manage the increasing amount of electronic information available. As this sort of agents are capable of initiating tasks without any explicit user prompting, they are good in undertaking tasks running in the background such as searching for information.

For instance, the facet *Ecommerce* indicates an agent is used in applications for trading goods, services and business functions such as advertising and negotiating online. The facet *Book Trading* indicates an accounting book agent that includes all securities that an organization or user regularly buys and sells on the stock market.

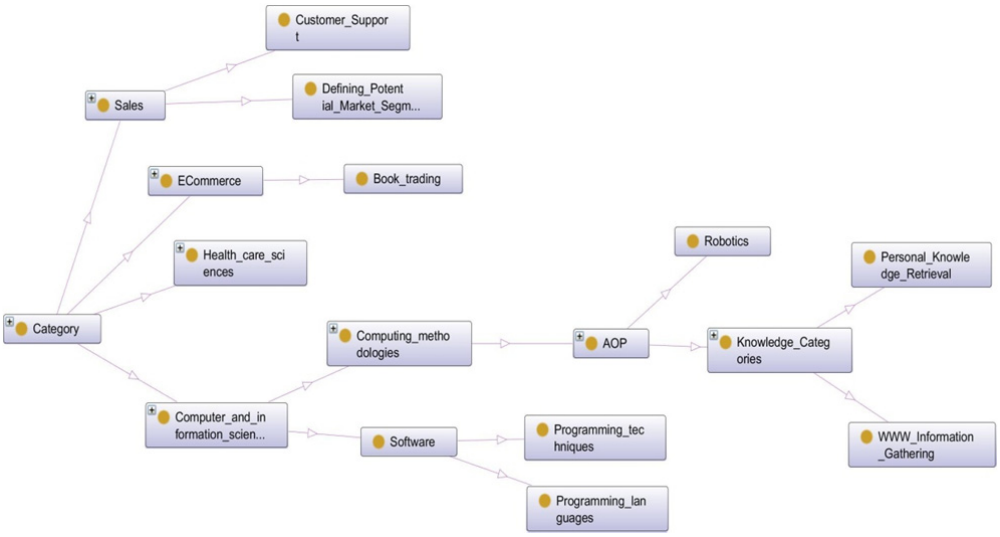


**Figure 10: Application Domains Taxonomy.**

As agents are inserted in the repository, the repository allows users to tag them with semantic references that correspond to facets from the taxonomies. The taxonomies can be evolved, increasing the diversity of the elements available to search.

To avoid an uncontrolled vocabulary for these application domains and behaviors, we propose the taxonomies illustrated in Figure 9 and Figure 10 to represent, group, and browse these items. To support browsing the taxonomies we develop an agent, *Category Searcher Agent*, to carry out this functionality.

### 4.1.3. Indexing Agent Components

We index the agent components and queries convert them into a format that the search is rapid and efficient. The indexing process is performed by means of the Lucene [48] search library. After the raw content is indexed, it is converted into units, called documents [52], used by the search engine. Lucene generates two indexes: an agent index containing all the component terms included in each agent representation and also those terms obtained through WordNet lexical ontology [82]; and a text index, containing the stems of text words that are not related to the agent entities. These component terms represent the entries in a vector for the corresponding document.

Lucene contains implementations of well-known algorithms such as the inverted file index, a stop word remover and the stemming algorithm [52]. It can index data from numerous sources, for instate, from ontologies and databases. WordNet is used to solve, during the indexing process, problems like ambiguity providing semantic knowledge i.e., synonyms. A word, in the agent representation or query, can be expanded to get its synonyms with WordNet [52] increasing the index terms of the agents, or enriching the query and probably the results as well.

A term vector is a collection of term-frequency pairs that could include information about the position for each term occurrence. Once there are more than one document vectors, associated to agent components, the similarity between them can be computed using the cosine similarity [50].

### 4.1.4. Recommendation System

Recommendation systems allow users to discover reusable artifacts, which are likely to be useful to user' requirement. Besides that objective, our RS has other one very important: to recommend agent components that a user is actively interested in but is unaware of such artifacts existence or the need for such artifacts.

The main problem of whatever recommendation system faces, it determines which artifacts can be similar or related with each other. Consequently, the first step is to determinate how measure the degree of similarity or relationship among the artifacts in the repository. Usually, this similarity can be measured in some ways: string matching/comparison, same vocabulary used, probability that documents arise from same model, same meaning of text, among others [36].

Our RS includes an alternative way (recommendations) to suggest related agents, based on a proposed methodology. The system adapts the *tf-idf* algorithm, to compute weights of terms belonging to the agent components, and advises the user which agents would be related. Later, the user can establish the relations, and a degree of relationship is assigned automatically. The RS tracks usage histories of a group of agents to recommend agents expected being needed by that user. In that way, an ontology network, that contains the discovered data and their associative relations, is constructed.

***Methodology***:

*Description*: Given the index terms those characterize a specific agent component.

(i) To compare the extracted index terms to the index terms that characterize each component in the repository, discovering which agents are related with.

(ii) To know if two index terms match, as we explained above, we create vectors for the index terms and calculate the cosine similarity between them.

*Be advised*: The comparison between two index terms is based on a semantic similarity, related to computing the similarity between concepts which are not necessarily lexically similar.

***Method***:

*Description*: Based on specific attributes that describe the underlying agent components, the degree of relationship among them is calculated. The attributes taken into account are the description, roles, interfaces, ancestor, categories and tags.

(i) To know if the description of the proposed related agent, *X*, matches with the description of the current agent, *Y*, where both descriptions are typical strings, we follow the steps of the proposed methodology, but in this case the index terms are extracted just from the descriptions.

$$\underset{d}{hit}(\mathrm{X},\mathrm{Y}) = \frac{\vec{v}(doc\_description_1) * \vec{v}(doc\_description_2)}{|\vec{v}(doc\_description_1)| * |\vec{v}(doc\_description_2)|}$$

(ii) A role matches another role if their descriptions, preconditions, description of the operations, or super roles (recursively) match. For the descriptions and preconditions, it is an analogous process to the explained above in (i). So, a role of X match a role of Y as:

$$\underset{r}{hit}(X_{r1}, Y_{r2}) = \frac{\sum_{i=1}^{4} \cos a_i}{4}$$

where 4 is the quantities of attributes taking into account, and $\cos a_i$ is the degree of similarity of the i-th attribute of the both agents.

Assuming *X* has fewer roles than *Y*, this function is calculated for each role of *X*.

(iii) An interface matches another interface if their contents and participants match. For the contents, it is an analogous process to the explained above in (i). In the case of the participants the interfaces $I_1$ and $I_2$ match, if the agent that sends messages in $I_1$ matches with the agent that send messages in $I_2$; and at least one of the agents that receives messages in $I_1$ matches with one agent that receives messages in $I_2$.

(iv) The ancestors of *X* and *Y* match if:

$$\underset{a}{hit}(\mathrm{X},\mathrm{Y}) = \begin{cases} 1, & first\_ancestor(X) \cong first\_ancestor(Y) \\ 0, & in\ other\ case \end{cases}$$

(v) The categories of *X* and *Y* match as follows:

$$\underset{c}{hit}(\mathrm{X},\mathrm{Y}) = \frac{|\,categories(\mathrm{X}) \ \cap \ categories(\mathrm{Y})|}{\min(|\,categories(\mathrm{X})|\,,|categories(\mathrm{Y})|)}$$

(vi) The tags of *X* and *Y* match as follows:

$$\underset{c}{hit}(\mathrm{X},\mathrm{Y}) = \frac{|\,tags(\mathrm{X}) \ \cap \ tags(\mathrm{Y})|}{\min(|\,tags(\mathrm{X})|\,,|tags(\mathrm{Y})|)}$$

All hits calculated are already normalize (the range is into the interval [-1…1]). Now we calculate the degree of relationship between *X* and *Y* as follows:

$$relationship(X,Y) = \frac{\sum_{i=1}^{6} hit_i}{6}$$

where 6 is the quantity of attributes we are taking into account to find the relationships between *X* and *Y*.

An example of related agents is depicted in Table 1, where the degree of relationship between both using the method is 0.889.

**Table 1: Example of Related Agents.**

| | |
|---|---|
| **ID**: ag1<br>**Name**: BookSellerAgent<br>**Description**: seller agent has a minimal GUI by means of which the user can insert new titles (and the associated price) in the local catalog of books for | **ID**: ag5<br>**Name**: BookBuyerAgent<br>**Description**: agent that buys books on behalf of their users. It takes as input some books (at least name, maybe author too) to buy and tries to find |

| | |
|---|---|
| sale.<br><br>**Version**: 1.4<br>**Ancestor**: -<br>**Date**: 2010-10-08<br>**Platform**: Jadex<br>**Language**: Java<br>**User**: jadex-admin<br>**Behaviors**: Reactive, Interactive<br><br>**Roles**:<br>R1: selling books online.<br>Mandatory.<br>Super Role: -<br>Precondition: -<br>Operation 1: to check if the requested book is in the catalogue and in this case reply with the price.<br>Mandatory.<br>Failure: -<br>Parameter 1:<br>Optional.<br>request from buyer agents (the request includes some information about the book to sell/buy).<br>Operation 2: to serve the order and remove the requested book from their catalogue.<br>Mandatory.<br>Failure: -<br>Parameter 1:<br>Mandatory.<br>purchase order.<br><br>**Categories**: e-commerce<br>**Tags**: Book trading, book seller<br>**File**: C:\repository\jadex\BookSellerAgent-1.4.jar<br>**Related**: ag5<br>**Interfaces**:<br>I1:<br>Sender:<br>Context: Negotiation<br>Types of the message: Inform<br>Content: the book costs 50.99 reais and the shipping costs 9.99 reais.<br>Receiver: ag5 | agents selling them at an acceptable price.<br>**Version**: 4.1<br>**Ancestor**: -<br>**Date**: 2011-11-13<br>**Platform**: Jade<br>**Language**: Java<br>**User**: Giovanni<br>**Behaviors**: Interactive, Autonomous, Reactive<br>**Roles**:<br>R1: buying books online.<br>Mandatory.<br>Super Role: buying items online<br>Precondition: -<br>Operation 1: requests all known seller agents to provide an offer.<br>Optional.<br>Failure: -<br>Parameter 1: more than one seller agent provides an offer.<br>Operation 2: to accept the received offer and to issue a purchase order.<br>Mandatory.<br>Failure: -<br>Parameter 1: purchase order.<br>Operation 3: to accept the best offer and to buy it.<br>Mandatory.<br>Failure: -<br>Parameter 1:<br>Alternative.<br>more than one seller agent provides an offer.<br>**Categories**: e-commerce<br>**Tags**: Book trading, book buyer<br>**File**: C:\repository\jade\BookBuyerAgent-4.1.zip<br>**Related**: ag1, ag4<br>**Interfaces**:<br>I1:<br>Receiver:<br>Context: Negotiation<br>Type of message: Inform<br>Content: the price of the book and the shipping.<br>Sender: ag1 |

The agent with ID equals to 4 is the *Buyer Agent,* that its main role is buying items online.

Our recommendation system is designed and implemented as a multi-agent system, where agents cooperate to organize and search knowledge of artifacts on behalf of their users, interacting in the same environment. Figure 11 shows the RS's architecture.

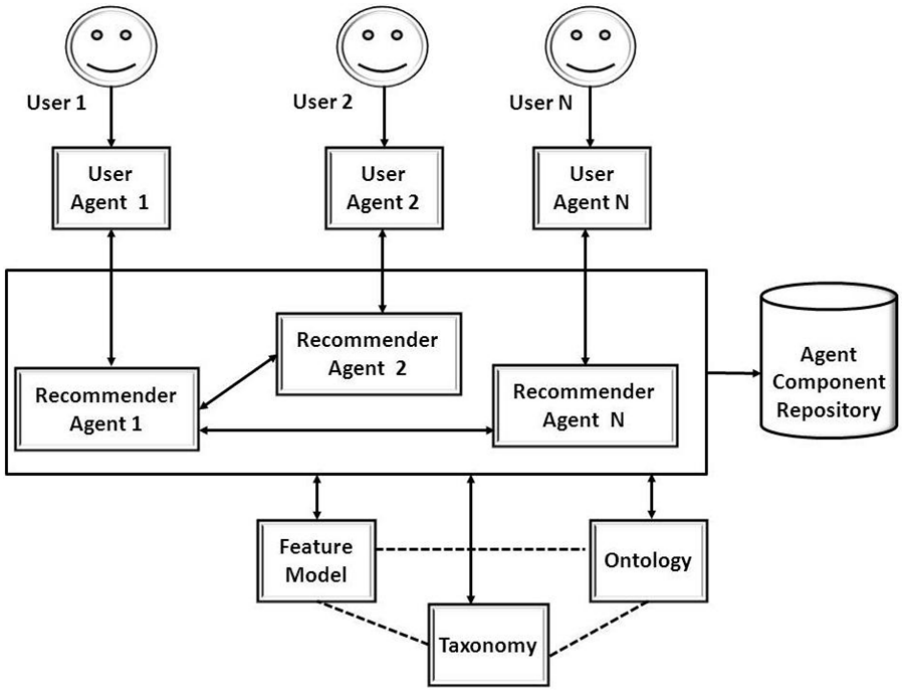


**Figure 11: Multi-Agent-based Recommendation System.**

Each user is assisted by a personal *User Agent*, which communicates with a *Recommender Agent* by means of message exchange. The main tasks of the *Recommender Agent* are: to filter and collect agents components satisfying certain requests or offering some functionalities for a particular user; to process queries of users or from other agents retrieving the agent components according to some parameters; to learn which agent components by other users' query address related characteristics to the current agent component; and to make thus suited recommendations in response. The *Recommender Agent* calculates the degree of relationship between the specific agent component and the candidate ones through combining both component specific knowledge and general domain knowledge, and the related components are returned in descending order. Within the system,

recommendation agents interact with one another, share knowledge and use similarities among users' behaviors in order to increase quality of the recommendations.

### 4.1.5. Semantic-based Search System

A fundamental goal of our repository approach is to make it easier for software engineers to formulate high-level queries for agents and have access to high-level information about the agent components retrieved. For retrieving suitable agent-based artifacts for reuse, we have developed a suite of search methods that utilizes the semantics of the agent descriptions to measure the similarity between queries and agent components. Given a textual description of the desired agent, a semantic information retrieval method can be used to identify and order the most similar agent components.

The search is driven by the taxonomies that represent the dissimilar types of the agent components and cover application domain specific knowledge, to express more appropriate queries for component retrieval. Thus, the search is performed by the semantic matching (semantic comparison) of user's requests expressed using concepts from the taxonomies with agent component descriptions found in the ontology and in the knowledge base, and also based on the syntactic structure of the specifications and the natural-language semantics of the agent's descriptions. The semantic knowledge base contains the ontological instances, the indexes, the index terms and the learned knowledge during the search process.

As we explained above, each agent component is represented by index terms. A pre-selected set of index terms can be used to summarize the contents of the agents' specification. Index terms compound a controlled vocabulary, using them as keywords to retrieve documents in an information system. An index term has some properties that are beneficial for evaluating its importance in a corpus. For example, if a word appears in the entire corpus of a collection, it is entirely useless for retrieval tasks.

There are different ways to retrieve agents in the repository, described next.

- *Keyword-based Searching*: Initially, we create a vector for the query with its terms. Afterwards, we calculate the cosine similarities between

the query vector and the vectors that represent the index terms of all agent components already stored in the repository.

- *Custom Search Facets*: The hierarchical taxonomy represents a search objective to find agent-based artifacts in a specific context or application domain, which would provide more expressive semantics than simple keyword-based searches. A tree-structure-based search representation model is conceived to allow users to browse, locate and filter their search intention by defining their own taxonomy topology. The facets are defined by the users to describe features about the agent-oriented artifacts. Features serve as descriptors, such as the agent's functionality. Nevertheless, they can be extended for other agent's implementation details, as it is possible that the artifacts could be browsed by these attributes too.
- *Tag-based Searching (Tag Cloud)*: It is a weighted list that exemplifies the density of keywords present according to its relevance on the agent characteristics using a variety of fonts in the visual design. This allows a user to quickly identify what archetypes of agents are more common in the repository. Selecting a tag, the tag cloud will work as a browser leading to a collection of agent components that are associated with that tag.
- *Platform-based Searching*: It is looked up on the ontology all agent components already stored in the repository that were developed in the platform passed as parameter. It is an analogous process respect to the *programming language*.
- *Interface-based Searching*: It is looked up all the agents that interact in a certain context, e.g. cooperation, coordination and negotiation, to achieve a specific role. In addition, it can be looked up the specification of message exchange or the description of the participants to accomplish this interaction.

The whole search process combines the following phases: (i) pre-processing the agent's descriptions and queries i.e., conflating related words to a common word stem; (ii) interpreting user queries to choose the specific domains and

establish the search method and parameters in terms of the characteristics of agents; (iii) splitting these domains into sub-queries and executes them; (iv) retrieving the agent components, rating and filtering them according to their semantic relevance with user parameters; (v) analysis and classification of the results; and (vi) presenting the results to the user in the way further exploration of each component is enabled.

A multi-agent system outlines the core of the search system of the repository. These agents collect information from the repository by taking advantage of the semantic annotations of the agent components, accessing information on behalf of the users, and also through some mechanisms they may be able to find relevant agent components and if it is necessary, to compose them to generate the results expected by the user. A multi-agent based semantic-search not only guarantees efficiency and reliability of search, but also enables automatic and effective cooperation for semantic integration [24]. The users of the repository, mostly software engineers, can compose their applications by choosing the features for the desired product. Figure 12 gives a general idea of the architecture of the proposed mechanism of searches based on a multi-agent system.

Users provide an initial query, which takes the form of free-text-based terms. The *Indexer Agent* preprocesses the query, being supported by the *WordNet Engine Agent*. The latter, for indexing, uses WordNet. During the indexing process, it is removed apostrophes, other intra-word punctuation and stop words; letters with diacritics are normalized; it is reduced words to a root form (stemming) or changed words into the basic form (lemmatization). The new terms go to become a new query. In this step, the domain application taxonomy is used by the *Semantic Query Agent* to transform the free-text-based query into standard terms used in the internal system. The *Semantic Query Agent* determines the actions in the actuator according to the types of the query. The respective type *Search Agent* looks for this information and gives the results, which are analyzed by the *Semantic Result Collection Agent*. This agent learns, taking into account the query and the results, and inferences if there is new relevant information that can enrich the current semantic knowledge base; and passes the top matched results to the *Semantic Query Agent.* The last agent provides the results to the user. Besides data from the taxonomies and ontology, the semantic knowledge

base also contains learned and inferred knowledge about semantic similarity provided by the feedback after a search.

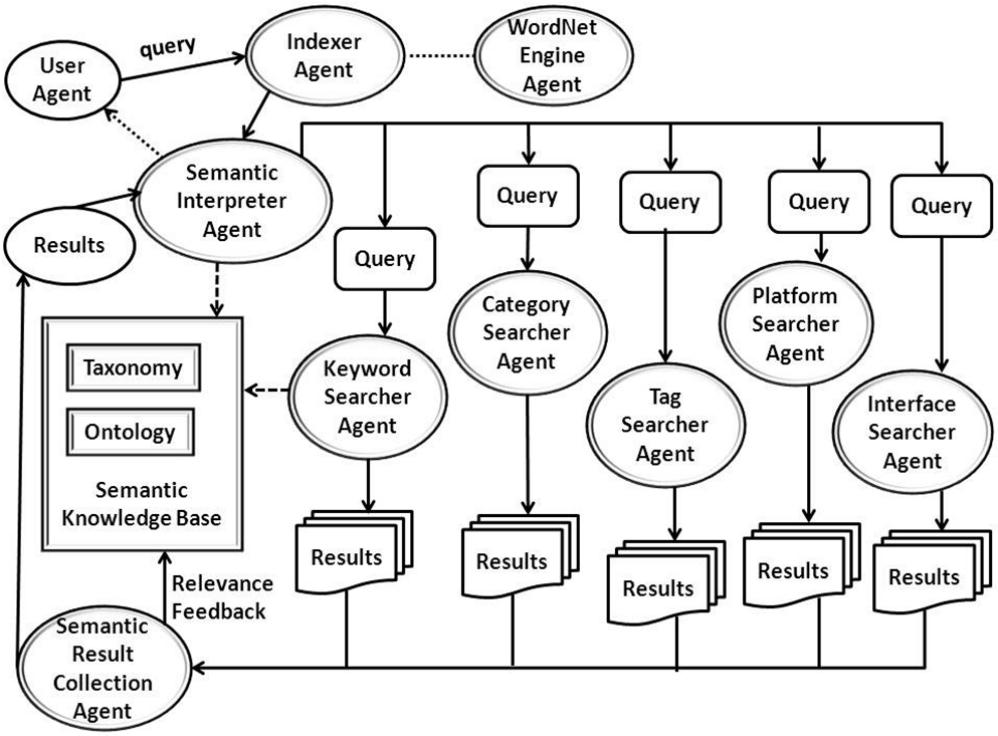


**Figure 12: Multi-Agent-based Semantic Search.**

### 4.1.6. Subscription Service

We create a subscription service of agents, or *RSS Agent*, in the repository that allows users being update of the existence of agent-oriented artifacts already stored. The user can subscribe in some application domains and will receive all the information about them. The RSS feed is easily rendered like any other format (json [32], html). Because it is delivered as XML, content from various agent application domains can be linked into novel presentations without difficulty.

## 4.2. A Usage Scenario

The main purpose of this thesis was to propose a new methodology that would relief the large effort required for the development and set-up of agent-based solutions by extending the scope of the applications.

Our representation and usage of agents are part of a repository for managing reusable agent-based artifacts. Agents are but one kind of reusable components that developers can search for, browse, and integrate into their own applications. A typical agent reuse cycle looks like [15]:

1. Formulate a description of the problem to be solved.
2. Based on this description, perform searches in the agent component repository (not necessarily the description itself).
3. Evaluate the retrieval agent components for their reusability. If one of these components is close enough to the stated requirement, proceed to the next step.
4. (not in all the cases) Adapt or extend the retrieved component to the current problem.
5. Integrate the component in the system, and test it.

A developer can make use of the abovementioned search methods to support the *Step 2* and the design process of the agent-based system that is being developed, and get support in the early implementation phase. The keyword-based search could be preferentially used at this point even it would be imprecise, since the developer has no clear ideas of the agents which has to be developed (this is not a critical problem but rather a logical consequence), or does not always know the form of the solution to his problem restricting himself to formulate effectively queries.

The repository helps users, through search methods or a subscription service, to know which agent components exist or can be extended, promoting the agent reuse, An agent component can be set up to be reused on multiple systems at design phase. Sometimes, some aspect details are mandatory if the component is to be reused, others are optional in some situations. For this, the proposed feature model is a good option to provide a basis for the reuse process of agents since it is capable to represent in a structured way the dissimilar characteristics

(mandatory, optional, or, alternative) of the agents in a given domain. The feature model assists the software engineers in the integration of the component produce agent-based applications without being needed to know agent's specification in detail.

An agent component can be set up to be reused at runtime as well. It is just to be aware of how the interfaces of the agent are. The interface description of an agent can be obtained by requesting it to the agent itself. Nevertheless, the repository brings a type of search based on the agent interfaces. So, knowing the interfaces of a specific agent, other agents can interact with this one.