PROYECTO FIN DE CARRERA

Título:	Diseño e Implementación de una Arquitectura de Agente Personal Basada en Tecnologías de Bot y Recuperación de Información.
Título (inglés):	Design and Implementation of a Personal Agent Architecture Based on Bot Technologies and Information Retrieval.
Autor:	Javier Herrera Planells
Tutor:	Miguel Coronado Barrios
Departamento:	Ingeniería de Sistemas Telemáticos

MIEMBROS DEL TRIBUNAL CALIFICADOR

Presidente:	Gregorio Fernández Fernández
Vocal:	Mercedes Garijo Ayestarán
Secretario:	Carlos Ángel Iglesias Fernández
Suplente:	Marifeli Sedano Ruiz

FECHA DE LECTURA:

CALIFICACIÓN:

UNIVERSIDAD POLITÉCNICA DE MADRID

ESCUELA TÉCNICA SUPERIOR DE
INGENIEROS DE TELECOMUNICACIÓN

Departamento de Ingeniería de Sistemas Telemáticos
Grupo de Sistemas Inteligentes

PROYECTO FIN DE CARRERA

DESIGN AND IMPLEMENTATION OF A PERSONAL AGENT ARCHITECTURE BASED ON BOT TECHNOLOGIES AND INFORMATION RETRIEVAL

Alumno: Javier Herrera Planells

Tutor: Miguel Coronado Barrios

Ponente: Dr. Carlos Ángel Iglesias Fernández

2013

Vires acquirit eundo

– Virgilio

Resumen

Este proyecto se centra en integrar las ventajas que ofrecen los sistemas de agentes inteligentes, las tecnologías de recuperación de información y los procesadores de lenguaje natural, todo ello ofrecido como un sistema de asistencia personal.

Gracias a ello, hemos podido dotar al asistente personal la capacidad de resolver dudas y apoyar el proceso de aprendizaje en una plataforma de e-learning. Esto abre la oportunidad de que el alumno sea capaz de obtener ayuda en cualquier momento, sin necesidad de recurrir al profesor. Al mismo tiempo, gracias al sistema de agentes se comprueba y mantiene un proceso de aprendizaje correcto. Por último, para la resolución de dudas del alumno, se ha conseguido que nuestro asistente personal acceda a fuentes externas de datos utilizando técnicas de recuperación de información, de manera que también amplíe su base de conocimientos existente.

Durante el desarrollo, hemos lidiado con una de las mayores limitaciones que los procesadores de lenguaje natural sufren hoy en día: al ser tan grande su campo de aplicaciones, normalmente no son capaces de ejecutar órdenes del usuario que requieran uso de módulos externos. Para abordar esta limitación, implementamos sobre una red basada en eventos varios módulos de ejecución externos, tales como el sistema de agentes capaz de apoyar el proceso de aprendizaje, o el módulo de recuperación de información.

Palabras clave: Agentes, recuperación de información, Unitex, ChatScript, NLP, WebSocket, Jason, SIREn, asistente personal, red de eventos.

Abstract

This project aims to integrate the advantages of intelligent agent systems, information retrieval and Natural Language Processing in a personal assistance system.

Thanks to this integration, we have been able to provide the personal assistant the ability to solve questions and support the learning process in an e-learning platform. This opens the opportunity for the student to be able to find help in any moment, not having to resort to the teacher. At the same time, thanks to the agents system the learning process is checked and supported to be the correct one. Furthermore, for solving the student questions, our personal assistant has been given the capability to access external sources of data using information retrieval techniques, so he can also improve his existing knowledge base.

During the development, we dealt with one of the main limitations that the natural language processors nowadays have: as there is a wide range of applications for them, they usually are unable to execute orders from the user that require using external modules. We overcome this limitation by implementing over an event-based network several execution modules, as the agent system that is able to support the learning process, or the information retrieval module.

Keywords: Agents, information retrieval, Unitex, ChatScript, NLP, WebSocket, Jason, SIREn, personal assistant, evented web.

Contents

Listings

List of Figures

Chapter 1

Introduction

"Ut sementem feceris, ita metes"

— Cicero

1.1 Motivation

Personal assistants nowadays have presence in many fields: educative platforms [1], shopping assistance [2], database querying [3] or virtual city tours [4] are just some examples. In this document we describe our project, which is applied to the educative platforms field, thanks to its increase over the last years and the evolving technologies that add new features to them every day. These new features increase the complexity of the system, and still they should be offered easily to a usually low experienced user.

For this reason, it becomes necessary a way the user can easily interact with the system. Apart of using a well-designed graphical interface, there is also the alternative of using other technologies, such as the conversational bots. We will study the available technologies for understanding the conversation language (the natural language) and creating conversation: the corpus processor Unitex, the AIML implementations, and its evolution to the recently released ChatScript. This way, we can select the most appropriated technologies for our personal assistant.

In [2], it has been demonstrated that personal assistance increases notably the user experience when he needs to do tasks. Thanks to the work of researchers in the area of artificial intelligence, and especially intelligent agents and natural language technologies, it is possible to develop personal assistants as agents that are able to understand users' requests just by understanding their words.

In order to implement an architecture that mixes natural language understanding with other technologies for further processing the user query, we apply our work to a use case in the e-learning field, achieving several previously set requirements such as an event network for connecting external services that provide their own new features.

Personal assistance can feature the benefits of information retrieval and, furthermore, making use of the Semantic Web . According to the definition of W3C, *"The Semantic Web provides a common framework that allows data to be shared and reused across application, enterprise, and community boundaries"*. The main purpose of the Semantic Web is evolving the current web by enabling users to find, share, and integrate information conveniently [5]. It is also regarded as an integrator of the different content and information applications and systems. Semantic Web has been applied in publishing, blogging, and many other areas [6].

The amount of structured data in the semantic format form recently grows dramatically fast on the web. Linked data is available online with hundreds of millions of entity descriptions in data sets such as DBpedia, Uniprot and Geonames. This technology of Semantic Web introduces a new concept and method of the digital associative library: accessing data provided in open, standardized formats that able to be connected and accessed seamlessly and facilitating the reuse by others. Our personal assistant will be able to retrieve information stored in semantic format by making use of the the SIREn entity retrieval technology. He will store the results in his own knowledge base, and offer it to the user when demanded.

In order to improve the intelligence of a personal assistant, it is a good idea to benefit from the intelligent agent technologies [7]. For example, the behaviour of our bot will be supervised by a Jason agent system. Using the assertions extracted from the conversation with the user, it will help improving the learning process by offering examples to the questions or sending tests to the user asking about previously explained concepts.

Finally, the application is provided as a web service, so the user can access it using a client that can be implemented in any platform, such as desktop computers, smartphones, tablets or even embedded systems.

1.2 Rationale

In this document, we describe the development stages of our project: a personal assistance system that integrates the advantages of intelligent agent systems, information retrieval and Natural Language Processing.

Our project makes use of this integration to create a personal assistant that is able to offer help in a educative platform. It is able to understand the natural language in a conversation with the student and reply him using this way of communication.

For the student's questions, our assistant is able to find an answer in his own knowledge base or external sources. Our assistant has been given the capability to access these external sources of data using information retrieval techniques, so he can also improve his existing knowledge base.

Furthermore, the assistant is provided with the intelligence that agent systems offer, in this case to support the learning process. For example, using this technology the assistant is able to keep track of the explained concepts and make tests after to ensure the student understood well the explanations.

Our assistant opens the opportunity for the student to be able to find help in any moment, not having to resort to the teacher. At the same time, thanks to the agent system, the learning process is checked and supported to be the correct one.

1.3 Structure of the document

We will next present the structure of this document, in order to make easier for the reader to go through the different chapters:

Chapter 1 presents the context that surrounds this project and the motivation that made us work in the different areas it is composed by.

Chapter 2 describes an actual sight of the State of the Art of personal assistants; also, other ways of assistance that the user can achieve by communication via natural language, and the actual technologies in the information retrieval and intelligent agents fields.

Chapter 3 compares the different enabling technologies that can be used for the implementation and achievement of the different features of this project.

Chapter 4 shows several use cases that have been crucial to deduce the design requirements of the architecture of our system.

Chapter 5 explains in detail the architecture designed for this project. Both the different modules to be implemented, and the communication between.

Chapter 6 makes a demonstration of the implemented architecture, making use of the technologies that have been selected from the enabling technologies chapter and applying them to the e-learning field.

Chapter 7 includes several conclusions from the design and implementation of our system and remark some future work that could continue this project.

At the end of this document, useful related information is provided in order to help the developer to use the system described.

Chapter 2

State of the Art

" Historia est vitae magistra."

2.1 Introduction

In this chapter, an actual sight of the State of the Art of conversational agents is presented. Also, we present other ways of assistance that the user can achieve by communication via natural language.

Conversational agents are human-like agents with the ability to create linguistic interaction by using engines that analyze the user input (i.e. what the user writes) and find fetch the best possible answer consulting their knowledge base and sometimes fetching more according information from the environment.

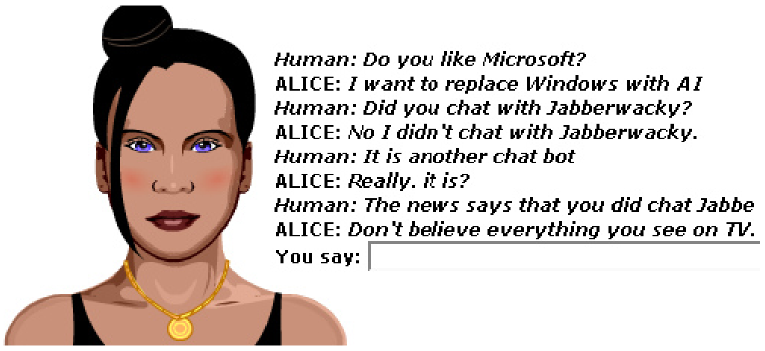
The areas of application of conversational agents will be also discussed. They are not only used for process assistance (like information retrieval or online shopping) but also for studying the user behaviour or e-learning. For further execution, some intelligence afterwards is needed, so many of these alternatives use intelligent agent systems. We will describe them next.

This will give us an idea of the personal assistants used nowadays and their abilities, as well as the different alternatives to achieve them.

2.2 Evolution of conversational agents

In this section, we will discuss one of the lines of evolution dedicated to implement agent-based behaviours. We will show the steps made to improve how these bots think and act: starting from simple pattern-matching technologies as Artificial Intelligence Markup Language (AIML), to discourse act mapping and fact databases that increase their performance.

Our starting point is A.L.I.C.E.: an artificial intelligence natural language chat that utilizes **AIML**[1] an XML language designed for creating stimulus-response chat robots. It was created by Richard S. Wallace.



Human: Do you like Microsoft?
ALICE: *I want to replace Windows with AI*
Human: Did you chat with Jabberwacky?
ALICE: *No I didn't chat with Jabberwacky.*
Human: It is another chat bot
ALICE: *Really. it is?*
Human: The news says that you did chat Jabbe
ALICE: *Don't believe everything you see on TV.*
You say:

Figure 2.1: Client interface for the A.L.I.C.E. chatbot.

The reference character, ALICE, won the 2000, 2001 and 2004 Loebner prizes [8], and is used as a starting point for new developers. The core feature of the language is the easy ability to create conversational agents through the process of pattern matching. ALICE has 120,000 patterns that either trigger a direct response or cause a reinterpretation to a simpler form. One of the most famous early pattern based natural language programs is ELIZA, which utilized several hundred patterns, and tried to mirror the personality of the conversational partner.

Between 2003 and 2005, **Façade** [9, 10] was presented as a story game for Windows/Mac using natural language to interact inside a drama that takes place in a small simulated virtual world, the apartment of the married couple Grace and Trip, in which the player, Grace and Trip can continuously move anywhere, use objects, speak dialog and gesture at any time.

Bruce Wilcox presented **ChatScript** [11, 12] in 2010, a chatbot engine written in C++, originally for Avatar Reality (a start-up company that wanted to create a virtual world called Blue Mars). It was after revised to the open-source ChatScript.

As its creator describes, AIML was a simple word pattern-matcher. Façade pattern-

[1]`http://www.alicebot.org/aiml.html`

Figure 2.2: Screenshot of the Façade game

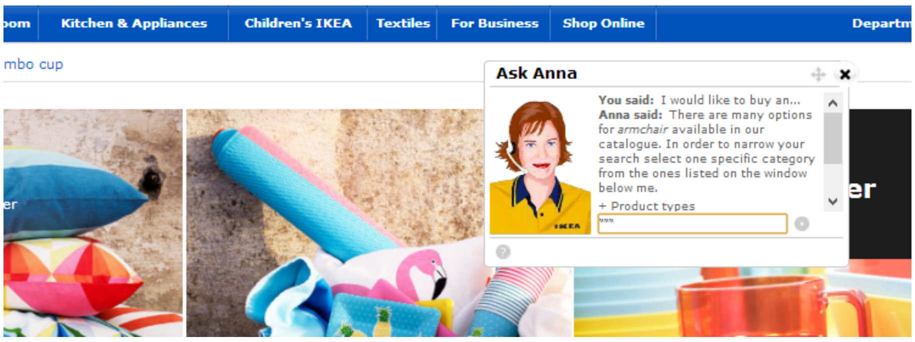
matched into discourse acts, a tightly restricted form of meaning. ChatScript aims to pattern-match on general meaning. It therefore focuses on detecting equivalence, paying heavy attention to sets of words and canonical representation. It also makes data available in a machine searchable form (fact triples).

Suzette and Rosette are the bots written in ChatScript that won 2010 and 2011 Loebner Competitions. Meanwhile ALICE had 120,000 rules, Suzette won having only 15,000, what explains its versatility and its bigger language simplicity.

To start describing the **application fields** of conversational agents, and because our project has been applied to e-learning, we will first take a look to the field of solving questions that appear repeatedly (i.e. **FAQ**'s) . This is a task that a chatbot can do well, not being necessary the human assistance. Teaching them the response once, will make them able to answer any user, saving them the time spent on finding the solution by themselves. **Educative platforms** also need this virtual assistance for the possible questions that could arise to the students [1]. With the increasing importance of e-learning over the last years, it clearly becomes necessary new features that help the students to get them quickly solved, not needing to wait for the teacher's answer.

Besides of the previous application fields, two other important related applications are **database querying** and **information retrieval**. Requesting data and querying a database using ontologies has actually been an attractive feature for many lines of investigation [3, 13, 14]. Giving to the bot the ability translate the natural language of user's input text to SPARQL queries, it is possible to access resources as DBPedia or Wordnet and retrieve the information for him without any knowledge about query languages.

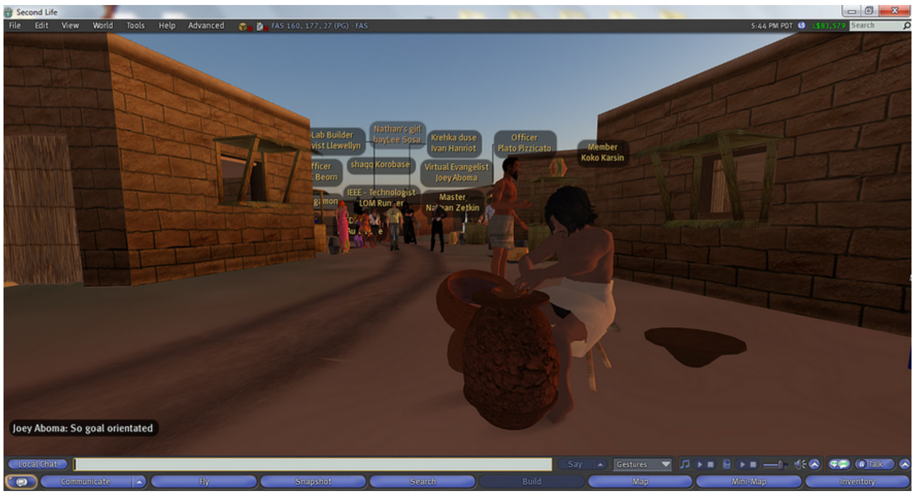
Another interesting application field of personal assistance is **online shopping**. In this field, the personal assistants are able to offer help to clients going through the shopping steps (for example, selecting the product or the shipping method and payment steps). It has been shown [2] that having an assistant that can suggest products to the clients improves drastically the sales and the easiness of use. One example of personal assistant in online shopping is shown in Fig. 2.3).



You said: I would like to buy an...
Anna said: There are many options for *armchair* available in our catalogue. In order to narrow your search select one specific category from the ones listed on the window below me.

Figure 2.3: IKEA shopping assistant

When it comes to **virtual city tours**, conversational agents have also been used to present information and interact with the users visiting a virtual city, as in [4] where a virtual city is presented in Second Life. Here, agents play the role of tourist guides, showing the users the most important places of the recreated city. Users can ask them for further information about any of the points of interest of the city. This way, users take part in a more realistic experience.



Joey Aboma: So goal orientated

Figure 2.4: The City of Uruk virtual tour in Second Life

Finally, it has been demonstrated that the user inputs can show patterns of **user behaviors** that can be studied. For example, some ideas propose the use for videogames market study [15] or the politeness of the students [16]. In this last one example, agents combined with chatbot technologies use a number of techniques to detect various norm violations, including insulting and following. By using rewards and punishments, they try to reinforce the desired behavior of the users.

2.3 Agent systems for personal assistance

Intelligent agents are programs that act on behalf of their human users to perform laborious information-gathering tasks [17]. The overall goal is to reduce the amount of effort required by the human to complete the tasks he intends. This involves any task where the user can be assisted: for example, the last two decades have seen a marked interest in agent-oriented technology, spanning applications as diverse as information retrieval, user interface design and network management. Also virtual learning, e-commerce, or e-health.

The behaviour of personal agents is based on user's profile: both the one the agent may guess from user's interactions, and the profile the agent may read from the user's social data and activity in social networks [18]. For this case, connectivity is essential: personal agents connect to the user's agenda, his social network account, his email inbox, even his current presence and location, for after behaving according to the information fetched.

According to the task they perform, personal agents can be classified into the five main categories described next. Some authors suggest an additional category named Decision-making Agents that help user to take a decision; however, this category is too general since some other categories could be classified as decision-making.

Personal resources and information management agents acquire, organize, retrieve, and process information in their personal spaces. They follow the items that the user share, such as books or CD's, to know what is borrowed, and also organize files, web bookmarks or emails, keep track of the commercial transactions notifying the user about his budget.

The second category of personal agents are the **purchasing and trading agents**, that guide the user in the shopping process to help him to choose the best items to purchase, according to some defined criteria.

Task and time management agents offer help to the user involving planning, execution and oversight of tasks, and also managing actual and potential commitments. This way, user can be relieved of routine tasks.

Another category is **reminder agents**. They check the user's calendar in order to remind him about important events; as example, Google Calendar, which tracks all the meetings, task deadlines and any event in general with a configured alert. Reminder agents are not classified as time management agents since they are not responsible for structuring the work and meetings in the user's calendar, but just monitoring them and notifying about important events.

In the field of information retrieval, as seen in [19], **recommender and filtering agents** have demonstrated enough capability to earn user interests incrementally and with continuous update and automatically providing documents (e.g. a personalized newspaper) that match the user interests, and helping the user refine search so as to increase retrieval of relevant documents.

More recent investigations show even a wider range of possibilities for personal agents, such as [20], where a multi-agent personal memory assistance system is developed. In this project, ageing and the decrease of the capacity of remembering or storing new memories, which affects life quality, is supported by developing a Personal Memory Assistant. It helps the user to remember things and occurrences in a proactive manner, addressing also socialization and relaxation events that should be part of the user's life.

However, one of the most used personal assistance is using software for Smartphones such as Siri or Sherpa. Siri[2] is an intelligent personal assistant and knowledge navigator which works as an application for Apple Inc.'s iOS. The application uses a natural language user interface to answer questions, make recommendations, and perform actions by delegating requests to a set of Web services. Apple claims that the software adapts to the user's individual preferences over time and personalizes results. Available in other platforms such as Android or Blackberry OS, Sherpa[3] offers a deep and comprehensive knowledge of the languages which it analyzes, its emphasis on language-independent conceptual representations of the results of the analysis, for finally making and analysis process which integrates five levels of linguistically motivated processing to arrive at an unambiguous, language-independent representation of the user's intent.

[2]`http://www.apple.com/es/ios/siri`
[3]`http://www.sher.pa`

Chapter 3

Enabling Technologies

" De nihilo nihilum"

3.1 Introduction

Today, we can find many existing technologies that let us move inside a wide range of design choices for our system. Also, new technologies still arise every day, broadening even more this range. The following sections include the past, present and future of each technology we focus in.

In the technologies that are composed by multiple processing steps, such as the natural language ones, we will also show how investigation has replaced single parts of this process in order to improve their performance. For example, when understanding or executing a requested action. As it can be seen not all the investigations look only for an application, but also for an improvement of comprehension and performance of their systems.

3.2 Natural language understanding technologies

Natural language understanding (NLU) technologies consist on extracting structured information from natural language inputs. There is a wide range of possible techniques to achieve this task, and we will see that their efficiency usually depends on three factors: the application field, the corpus extension and the language used.

3.2.1 AIML

AIML is not itself a NLU technology, because its mode of operation approaches more to pattern matching techniques than the extraction of structured information using grammars. However, it is good enough when specifically designed, and its when knowledge base is large enough. It is usually preferred also for its performance in computation costs terms.

3.2.1.1 Technology description

In AIML, the basic knowledge unit [21] consists of an input question, an output answer, and an optional context. This basic knowledge item is called ***category***. The question, or stimulus, is called ***pattern***. The answer, or response, is called ***template***. The optional context can be of two types, named ***that*** and ***topic*** respectively. The conversation is carried on trough algorithms for automatic detection of *patterns* in the dialog data.

The simplest *category* is the atomic one, where both *pattern* and *template* contain only text. A simple example of atomic *category* is:

```
<category>
  <pattern>WHO ARE YOU</pattern>
  <template>I am a bot</template>
</category>
```

Listing 3.1: The simplest AIML category

The pattern can contain also the "_" and "*" wildcards, to match one or more words; in the template the <star/> tag is used to access wildcard bindings. The <template> tag can contain also other tags, as an example <setname = ”> ...</set> and <getname = ”/> tags allow to set and get the values of variables, such as information about chatbot or user, performing some sort of basic user modeling. The <think> tag allows evaluating an expression hiding the visualization of the intermediate processing answer. These tags are exemplified in the available corpus that can be found in the AIML homepage[1]. Also, they are referenced in a complete AIML code that can be found in the end of this AIML section.

For the **pattern matching process**, all the data is precompiled into a tree structure and input is matched in a specific order against nodes of the tree, effectively walking in a de-

[1]http://www.alicebot.org/aiml/aaa/

cision tree. The first matching node executes its template, which may start a new match or do something else. The stimulus-response categories are stored in this tree structure, managed by an object called Graphmaster [22], implementing a pattern storage and matching algorithm. Graphmaster forms the core of the engine and implements the search algorithm that allows the bot to match the user input and construct an appropriate response.

Matching behavior can be described in terms of the previously described object, which is also a common implementation of the AIML pattern expression matching behavior. Given an input starting with word X, and a Nodemapper of the graph:

(1) Does the Nodemapper contain the wildcard "_"? If so, search the subgraph rooted at the child node linked by "_". Try all remaining suffixes of the input following X to see if one matches. If no match was found, try (2).

(2) Does the Nodemapper contain the wildcard "X"? If so, search the subgraph rooted at the child node linked by X, using the tail of the input (the suffix of the input with X removed). If no match was found, try (3).

(3) Does the Nodemapper contain the wildcard "*"? If so, search the subgraph rooted at the child node linked by *. Try all remaining suffixes of the input following X to see if one matches. If no match was found, go back up the graph to the parent of this node, and put X back on the head of the input.

(4) If the input is null (no more words) and the Nodemapper contains the <template> wildcard, then a match was found. Halt the search and return the matching node.

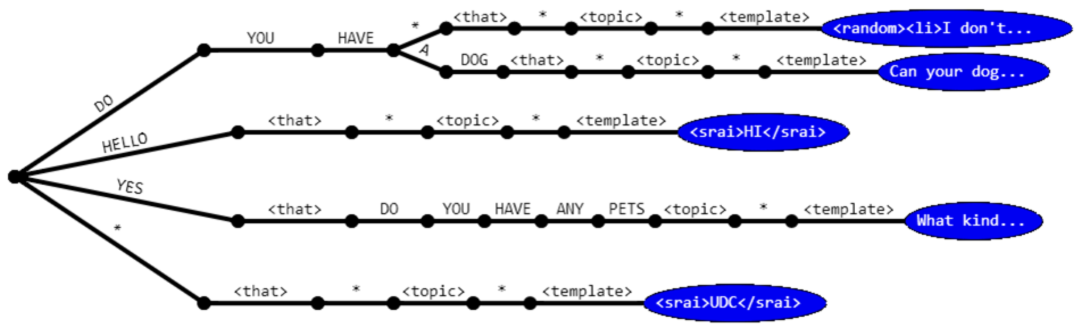


Figure 3.1: AIML search in the knowledge base to find a category that contains the pattern that best matches the question.

3.2.1.2 Web-oriented implementations of AIML language

Actually, independent developers have implemented the AIML engine in different programming languages, such as PHP, Java, Ruby, C++, Perl and others. In our project, we will focus on web-oriented ones. Focusing then in this field, we can find several solutions that are presented next.

Program-O

Program-O is an AIML engine written in PHP. It was under active development by Elisabeth Perreau and a group of dedicated open source developers (mainly Dave Morton) with a last known update in 2013. Its last version provides not only an AIML file upload module as remarkable features: also a complete administration panel that includes a teaching interface, a bot configuration panel and a bot testing module.

Figure 3.2: Program-O logotype

This solution includes also an SQL database that optionally keeps the entries of the new AIML files uploaded, in order to improve its speed and easiness of modification. Finally, other minor functionalities that remark the high level of development of this solution are the personalities that can be assigned to the bots, a spell check option and a word censor panel.

Program-E

This solution is also written in PHP and uses SQL as backend database. Otherwise, this project became orphan in May 2011, with some known bugs still in it. It is composed by a converter which loads the AIML files into the database, an AIML engine, and several chat interfaces in HTML, Flash and XML.

Program-D and Chatterbean

These two engines are both implemented in Java, and also both have its latest known release in 2006. The standard release of Program D provides a J2EE web application implementation that can be deployed as a WAR file. On the other side, Chatterbean can be integrated in an AJAX application. An interesting built-in functionality of Program D is the support of multiple bots in a single server instance. Otherwise, none of bot provide enough stability and the SQL update of the AIML files as the previously described solutions.

A comparison table of the web-oriented implementations is shown below. It is ranked by the latest release date available that they offer:

Solution	Implementation	Multiple bots	SQL copy	Latest Release
Program O	PHP, MySQL	Yes	Yes	2013
Program E	PHP, MySQL	No	Yes	May ’11
Labsmedia A.P.B.	PHP	No	No	July ’11
Program D	Java, J2EE	Yes	No	Mar ’06
Chatterbean	Java, AJAX	No	No	May ’06

Table 3.1: Comparison table of the web-oriented implementations

3.2.1.3 Limitations

AIML is only designed for conversation, and each application field presents extra needs that can not be covered with simple conversation. This limitation has made programmers to be constantly implementing new AIML tags that achieve the needed features. One of the most requested features is a parallel channel of communication that is hidden to the user, usually for sending commands to execute actions in the user’s device. This is the case of CallMom [23], an application for phones that allows voice control of the device. One example of use of the <oob> (out-of-band) tag they implemented is presented next:

```
<category>
  <pattern>SENDTEXTMESSAGE</pattern>
  <template>
    <oob>
      <sms>
         <recipient><get name="numbertodial"/></recipient>
         <message><get name="textmessage"/></message></sms>
    </oob>
    Now sending "<get name="textmessage"/>".
  </template>
</category>
```

Listing 3.2: Sample of use of the out-of-band <oob> tag

This <oob> tag allows inside commands as <sms> to send an SMS, <dial> to make phone calls, or <battery> to check the battery charge level. Other examples of new tag implementations to the core of AIML have been the following ones:

- Using new tags to trigger external plugins that allow better understanding the user's query. For example, using Bayesian networks, Cyc, or ontologies [21, 24]
- Making SPARQL queries to ontologies, such as DBpedia, [25] to retrieve information in external sources.

Also, other modifications in the core have been built to link AIML and BDI agents together [16]. This allows further user request processing, such as analyzing the user behaviour and implementing a reward system. Nowadays, community that still develops new features for AIML can be found in Pandorabots [26] and the Foundation ALICE AIML set on Google Code [27], as the out of band tag for hidden communication between server and client.

One of the most recent bots created by the previous communities is Jeannie [28], able to answer questions, send emails, voice dial the phone contacts, set alarms/reminders, listen to music automatically and many other features. It is powered by Google, Bing, WolframAlpha, Yelp and Trueknowledge.

Also, in 2013, AIML 2.0 draft has been released [29], including the Pandorabots extensions to the language, the out-of-band <oob> tag, and new wildcards, conditional loops, local variables, and the ability to specify attribute values in tags.

3.2.2 Unitex

Unitex[2] is a corpus processing system, based on automata-oriented technology. It comes from the Laboratoire d'Automatique Documentaire et Linguistique of the Paris-Est Marne-la-Vallée University, under the direction of Maurice Gross.

The main features that it offers and are interesting for our project are:

[2]`http://www-igm.univ-mlv.fr/~unitex/`

- Building, checking and applying electronic dictionaries.
- Pattern matching with regular expressions and recursive transition networks.
- Handling ambiguity via the text automaton.
- Building an automaton from a certified corpus.

With these features, it offers us the possibility to implement a module that extracts structured information, using grammars, from the user's input written in natural language. For this, we can easily design graphs that will make matches in the user sentence and generate a structured output for the other modules.

A very illustrative example is this graph, that handles the student queries demanding examples about a Java concept:

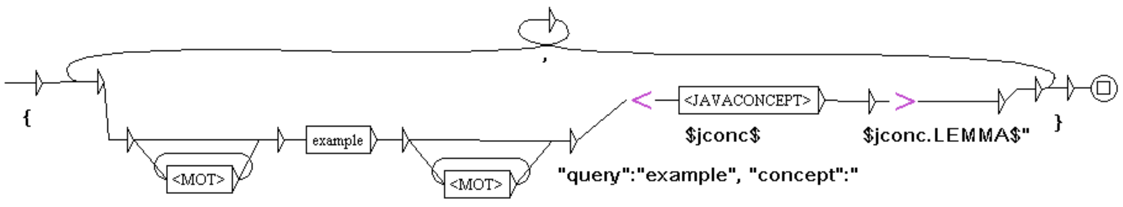


Figure 3.3: Sample graph in Unitex

This graph matches all the phrases containing some arbitrary words (*mot* means *word* in French), followed by the word "example", followed by some other arbitrary words, and then followed by a Java concept of a list that we also specify:

```
for,for.JAVACONCEPT
for loop,for.JAVACONCEPT
while,while.JAVACONCEPT
while loop,while.JAVACONCEPT
while structure,while.JAVACONCEPT
do-while,do-while.JAVACONCEPT
do-while loop,do-while.JAVACONCEPT
```

Listing 3.3: Java concept list used by the example graph

This list (called dictionary), used with the graph, will map the different ways to name a concept to a unified name. For example, "while", "while loop" or "while structure" will be all understood as "while". If the user query is "Please, could you give me an example of the while loop?", or "Do you know any example using the while structure?", the output produced by our graph used with the dictionary would be a JSON output with the extracted information:

```
{
"query":"example",
"concept":"while"
}
```

Listing 3.4: JSON output for a question about

In fact, dictionaries can be divided in two types. We will observe the differences between them with another example that detects ordinary and cardinal numbers in a phrase.

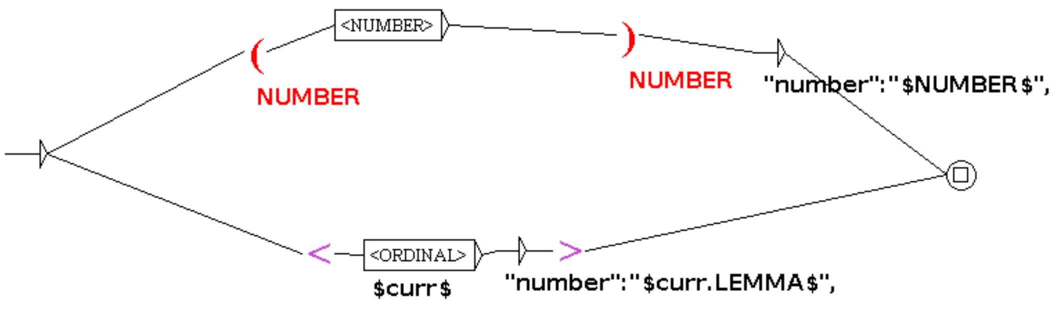


Figure 3.4: Unitex subgraph that maps ordinal and cardinal numbers to their integer value

For this graph, the two kind of dictionaries that we have used are:

- **DELA dictionaries**, for the <NUMBER> box: this way, we will recognize user inputs directly written in integer form, such as "1", "2" or "3", without needing to alter them.

```
1,.NUMBER
2,.NUMBER
```

```
3,.NUMBER
```

Listing 3.5: Sample of DELA number dictionary for Unitex

- **Morphological dictionaries**, as for the <ORDINAL> box, handle the cardinal numbers (first, 1st, second, 2nd, ...) and map them to ordinal numbers (1, 2, ...). The advantage of these morphological dictionaries is that it will allow us to map the different forms to name a concept (second, 2nd) to a unified one (2). Applied to the flight booking field, we can use one of these dictionaries to recognize the different ways to talk about a flight (fly, flight, plane, airplane or aeroplane) and map them to a unified name (flight) that will be easier to handle by the other modules. This way, user inputs as "I want to book a flight" or "I would like to take a plane" would be easily understood as queries to book a flight.

```
3rd,3.ORDINAL
third,3.ORDINAL
three,3.ORDINAL
```

Listing 3.6: Sample of a morphological dictionary for Unitex

Finally, to be able to integrate the Unitex features in a project, there are various existing ways for it. Four our project, our choice has been to use one of its wrappers, called pyUnitex[3] that allows calling the Unitex functions from a Python application.

3.2.3 Façade

Facade's major NLP improvements were matching to intermediate facts and then to discourse acts (such as "greetings" when introducing or "sports" when talking about sports), the OR operator to match sets of words, and the NOT operator to exclude words. Façade's NLP is built on top of Jess, a Java extension of the CLIPS expert-system language. Jess allows you to declare and retract facts and have rules trigger when all of their preconditions match. Using Jess, it can access Java as its scripting language on the pattern side and the output side. Façade built on top of this a template script compiler that let them write NLP rules and compile them into Jess. They wrote 800 template rules, which translated into

[3]http://github.com/moliware/pyunitex

6800 Jess rules, a small number for dealing with NLP. One improvement Façade made over AIML was: We don't map text to a reaction. We map to discourse acts. Façade didn't try to fully understand the input. Instead, it pattern-matched it into 50 discourse acts. These included agree, disagree, criticize, and flirt.

Rules in Façade could be defined as follows:

```
(defrule positional_Is
(template (tor am are is seem seems sound sounds look looks))
=> (assert (iIs ?startpos ?endpos)))
```

Listing 3.7: Sample rule in Façade

The above rule says if you see any words that approximate the verb to be, assert into working memory a new fact of iIs at the position found.

From the point of view of the programmer, the NLP of Façade has too many words, making too hard to write new rules. Last update of Façade 1.1b was provided by the authors in November 2006, with no supporting community afterwards.

3.2.4 ChatScript

As commented in the previous section, AIML was a simple word pattern-matcher and Façade pattern-matched words into discourse acts, a tightly restricted form of meaning. To overcome the limitations that they present, ChatScript aims to pattern-match on general meaning: it focuses on detecting equivalence, paying heavy attention to sets of words and canonical representation. It also makes data available in a machine searchable form (fact triples).

3.2.4.1 Topics, rules and basic syntax of ChatScript

ChatScript makes the writer of the corpus to group all the rules in topics in order to offer the ability to map the point of the conversation to a specific topic and making a better response, or even asking questions to the user to maintain the conversation.

```
topic: ~book_flight[fly flight]
```

Listing 3.8: Sample rule in ChatScript using concepts

In this example, the topic book_flight would be triggered when the user wrote a question with the words "fly" or "flight" inside, such as "I would like to book a flight". The rules written after this line would be the ones referred when book_flight topic is active.

The first advantage of ChatScript over the two previous is that its language it's much cleaner; this line does the same as the simple first AIML sample that needed 3 different tags:

```
?: (who are you) I am a bot
```

Listing 3.9: Simplest rule in ChatScript

There are different kinds of rules:

Responders their syntax starts with "s:", "?:" and "u:". They react to statements (e.g.: I live in Madrid), to questions (e.g.: Where do you live?) or to both, statements or questions.

Rejoinders their syntax starts with "a:", "b:", ... "q:". They are possible pattern-responses that can join after a question.

Gambits their syntax starts with "t:" They offer a topic gambit when chatbot has control (e.g.: Ask me questions, please).

Another advantage that ChatScript takes over AIML is the use of this last kind of rule, that offers an approach to proactivity making the bot throw its own questions or statements being him who creates further conversation.

Inside the parenthesis "( )" of the matching sentence, the syntax is composed by the following parts:

End of the sentence match: using ">" we can specify to match the end of the sentence.

Start of the sentence match : using "<", we specify the current position of matching to the start of the sentence.

Wildcards: using "*" we match 0 or more words in sequence. *1, *2... * followed by a number names how many words it absorbs. * 3. means from 0 up through that number, or approximately that number.

Variables : using _x ... the chatbot memorizes local variables for the user and stores them in the user log for remembering after.

```
s: ( My name is _*1 _*1 >)
$firstname = '_0
$lastname = '_1
Nice to meet you.
```

Listing 3.10: Saving variables in ChatScript

Chat is also **self-extinguishing**: the user doesn't want to get the same information twice so, by default, ChatScript both marks rules when they get used, to avoid using them again, and looks up its current output to see if it has already said it recently. Nevertheless, this feature can be disabled for a pattern using the `^noerase()` function.

Special comments, introduced by `#!`, exemplify the immediately following rule. They give an input expected from a user that should be matched and handled by this rule. It can be said that this documents which input is expected to match the rule below. In addition, it allows the engine to automatically test the corpus.

```
#! tell me your story
u: (your story) This is my story...
```

Listing 3.11: Comments in ChatScript

ChatScript also allows to declare **concepts**, which can group synonyms or affiliated words. This way, we can use only one word (the concept) in the pattern to refer to a set of words. In the example below, we define the concept "meat" with the words bacon, ham, beef and chicken, so any user input such as "do you like beef" will match the rule.

```
concept: ~meat [bacon ham beef chicken]

?:(do you like ~meat) I love meat.
```

Listing 3.12: Concepts in ChatScript

ChatScript comes with a thirteen-hundred predefined concepts in English, including the parts of speech and extensions like numbers, and proper-names. Finally, recall that topics are also concepts.

3.2.4.2 ChatScript relevant features

To start with, ChatScript can run **multiple bots** at once, each one with a unique persona. So one user can connect and talk with a specific personality while another user connects and talks with a different one (or the same one).

Also, it can offer **assistance to several different users**: the user can return to the chat with some days later, and the system knows what happened in previous conversations so this can be the start of a new conversation. The system keeps a log file per user recording their conversations for the author, and a topic file for each user-chatbot pairing, where it stores the current state of conversation for the engine. If the script records facts about the user during conversation, these are stored in a facts file, one per user, to be read in again by the engine. This log files can be accessed from the USERS folder.

ChatScript uses **WordNet** , a computerized dictionary, together with the WordNet ontology (that includes asserts as "a puppy is a dog, is a canine, is a mammal, is a being"). You can refer to a WordNet meaning in your patterns, which automatically stands for any refining word below it in the WordNet hierarchy. This provides a bunch of instant sets of words.

This can be used the same way as the concepts that we mentioned in the previous section: in the topic definition, baseball 1 is a reference to the WordNet definition one of baseball, which also covers lower level words hardball and softball. That is a small use, but a reference like plant 2 refers to thousands of plants (plant 1 refers to industrial plants). The carpus writer can also create his custom ones by editing the files in the folder RAWDATA/WORLDDATA/, which will be loaded when launching ChatScript. ChatScript includes also itself a dictionary with parts of speech and word attribute knowledge (it knows common male first names etc), for doing spell-correction, part-of-speech-tagging and parsing.

Flow control of the bot can be custom-defined by the writer in the corpus of the bot, making available predefined discourse acts and the custom behavior of the bot. The normal flow of control defined is to try to invoke a pending rejoinder. This allows the system to directly test rules related to its last output, rules that anticipate how the user will respond. Unlike responders and gambits (seen in the previous section), the engine will keep trying rejoinders below a rule until the pattern of one matches and the output does not fail. Not failing does not require that it generate user output: merely that it doesn't return a fail code. Responders and gambits are tried until user output is generated (or you run out of them in a topic).

If no output is generated from the rejoinders, the system would test responders. First in the current topic, to see if it can be continued directly. If that fails to generate output, the system would check other topics whose keywords match the input to see if they have responders that match. If that fails again, the system would call topics explicitly named which do not involve keywords. These are generic topics the writer might have set up.

If finding a responder fails, the system would try to issue a gambit. First, from a topic with matching keywords. If that fails, the system would try to issue a gambit from the current topic. If that fails, the system would generate a random gambit.

Once an output is found, the work of the system is nominally done. It records what rule generated the output, so it can see rejoinders attached to it on next input. And it records the current topic, so that will be biased for responding to the next input. And then the system is done. The next input starts the process of trying to find appropriate rules anew.

3.2.4.3 Deployment of ChatScript

By default, ChatScript offers a TCP socket interface (server and client) for communication. It can be run under Linux, Windows or MacOS.

Below, we present an example of communication using the TCP interface of ChatScript. It is written in Python, and it allows to implement a personalized client:

```
#ChatScript Server TCP interface parameters
BUFFER_SIZE = 1024
TCP_IP = '127.0.0.1'
TCP_PORT = 1024
```

```
s = socket()
s.connect((TCP_IP, TCP_PORT))

#Message to server: username, botname, null (start conversation) or
    message
socketmsg = username+"\0"+botname+"\0"+message
s.send(socketmsg)
data = s.recv(BUFFER_SIZE)

s.close()
```

Listing 3.13: Sample code for communicating with ChatScript server

Using a personalized client enables the ability to write a HTTP interface to offer this service in the Web. It is done in the case study of this project: by using a Python web framework named Bottle [30], we have been able to implement the ChatScript service on it, giving our personal assistant the benefits of ChatScript. This is further explained in the Case Study chapter of this document.

3.2.4.4 Limitations of ChatScript

As AIML, ChatScript is mainly designed for conversation, so extending its functionality to create customized actuators requires modifying its C++ source code. For implementing new features, new control functions (the ones called with f̂unction, e.g.: n̂oerase()) can be added editing the source file src/functionExecute.cpp. The code will need to be recompiled once these changes are applied:

```
g++ -lpthread -lrt -funsigned-char src/*.cpp -O2 -oLinuxChatScript32 2>
    err.txt
```

Listing 3.14: Recompiling ChatsScript source code

The creator also recommends, as alternative to editing the source code, to use out-of-band commands. Normally, this would be done by making ChatScript a subsystem of some larger program, as we do in our project. ChatScript exchanges then out-of-band information with this larger program. The syntax for out-of-band information is to put it within brackets [ ], as the heading text of the input or output [31].

ChatScript is nowadays under new updates: last version 3.0 was released March 10th, 2013. Its community also uses a forum [32] for development and question solving.

3.2.5 Bot technology evaluation

3.2.5.1 Use case and architecture

In this section, our intention is to test the functionality of the chatbot languages already presented and compare, from the programmer's point of view, which one would be the most appropriate to our needs. Also, some other alternatives using other components (for example, to better understand the user input) will be presented.

Our use case will be focused on flights booking. Online flight booking has increased notably during the last years [33], enabling the opportunity to buy tickets through the website of each airline, or their alliances (One World, Sky Team, etc.). Also, many new online travel agencies such as eDreams, or Travelgenio have demonstrated that online selling is nowadays a vital distribution channel for the airlines.

This is why many competitors struggle to offer the best service to the clients: to have the best flight finder engine, the best business model, and also to think on the user by offering the easiest interface for buying. In this struggle to offer a new selling channel that can be used by any kind of user, conversational agents start playing an important role. They offer precious help by assisting the users through the steps of booking a flight.

A suitable architecture for a system that would enable this possibility is presented in the Fig. 3.5. In this architecture, when the user input arrives via a client interface (1), it is sent to a preprocessing stage (2) that uses a NLP to translate the natural language of this input. After this conversion, the output is sent to the Flow Controller (3) that will apply Rules and Grammars (4) from the corpus of the chatbot and decide to give a direct response to the user, or to proceed with an Execution engine (5) for further actuations (for example, information retrieval using or Local/Device Services). Finally, the response is sent to be formatted (6) and shown to the user.

The main focus of our online flight booking assistant will be directed to preprocessing well the user input, the flow control and the rule and grammar engines. These three steps are made inside the core of our chatbots' engines, offering functionalities such as tokenization,

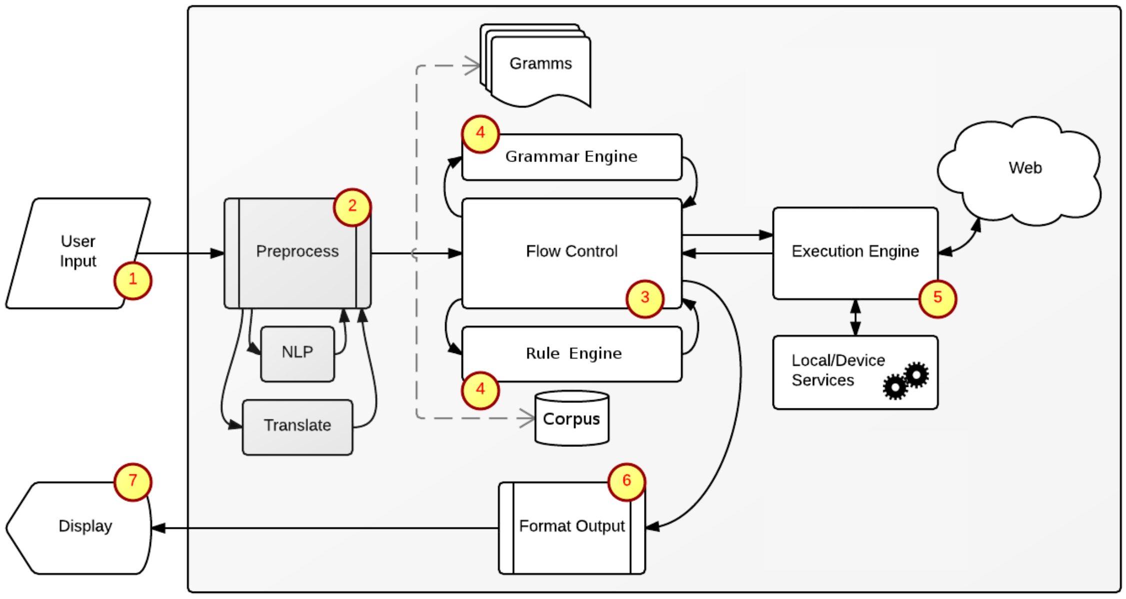


Figure 3.5: Architecture of the described use case

wildcards, or even using ontologies and concept matching. For AIML, we will present other alternatives for these steps in order to improve the performance of each stage. Once the user input is understood, the execution engine would look for flights in order to offer options for the user. This stage would be hand-made, as there are no actual technologies that match our intentions. Finally, the results would be formatted for sending back and present to the user.

We present, in the next sections, the suggestion of written corpus code for the bot, in order to remark the differences between AIML and ChatScript and how robust they can be with a simple well-written pattern.

3.2.5.2 AIML implementation

In this example we show how to make use AIML to create a bot that offers assistance in an online travel agency when booking flights. For finding flights, suppose we had an actuator that obtains information from a booking website, such as pricing, times or destinations. This command to be executed by the actuator is supposedly made by implementing a new tag <find_flight> in the core of our AIML engine. This tag that calls the actuator providing the parameters source, destination, currency and people.

```
<category>
   <pattern>* FROM *</pattern>
   <template>
      <think>
         <set name="source"><star index="2" /></set>
      </think>
      Ok, you want to go from <star index="2" />.
   </template>
</category>


<category>
   <pattern>* GO TO *</pattern>
   <template>
      <think>
         <set name="destination"><star index="2" /></set>
      </think>
      Ok, you want to go to <star index="2" />.
   </template>
</category>

<category>
   <pattern>* PEOPLE *</pattern>
   <template>
      <think>
         <set name="people"><star index="2"/></set>
      </think>
      Ok, <star index="2" /> people are travelling.
   </template>
</category>

<category>
   <pattern>* FINISHED *</pattern>
   <template>
      <find_flight>
        <source><get name="source" /></source>
```

```
          <destination><get name="destination" /></destination>
          <people><get name="people" /></people>
        </find_flight>
        Looking for flights from <get name="source" /> to <get name="
           destination" /> for <get name="people" /> people...
    </template>
  </category>
```

Listing 3.15: Example use case written in AIML

Our bot should also contain patterns for basic conversation such as greetings and goodbyes, which we have omitted in order to focus only in the desired flight booking patterns. One of the limitations of this example is that we always suppose that the user enters the data, one by one, separately in each submitted chat line. In case that we wanted to accept the data combined in a single chat line, the number of patterns would increase. Also, this example could be more robust if we added more patterns using the <srai> tag to match other ways of expression (for example, writing "person" instead of "people").

3.2.5.3 Alternatives to AIML for each processing stage

So far, many lines of investigation have demonstrated a large variety of options to achieve well comprehending and executing the users' requests, as it can be seen in these following sections. Alternatives for understanding the user input can be grouped in two ways:

a) **Pattern matching**, when the aim is creating simple conversation. Other alternatives are pointing to simple actions, as well as fetching information from a database [3, 13] by using additional self-made features. These features would need to be implemented in the core of the matching engine.

b) **Morphological analysis**, in case we need of extracting variables from the user text. Alternative methods of analysis that have been used are the Standford Parser combined with Morphadorner [34], or FreeLing [35]. Also for further comprehension, Bayesian networks and other different methods have been applied also [2].

Apart from these alternatives improving understanding, some research lines have implemented agent technologies to their bots. One example is the case of [16], where the agents of the intelligent agent system are able to analyze the user input. This is done for detecting the user behaviours, especially norm violations as insulting, or others as compliments. In this system, by using rewards and punishments, they try to reinforce the desired behaviour of the users.

3.2.5.4 Implementation in ChatScript

Here we describe an example of ChatScript corpus with the same functionality as the previous case that was written in AIML. In this case, the mentioned external actuator could be called by implementing a [find_flight] command. This command would be sent when all the variables are matched or when the user express it by saying "finished".

```
concept: ~location [ madrid sevilla barcelona ]
concept: ~number [ 1 2 3 4 5 6 7 8 9 10 ]

topic: ~BOOK_FLIGHT (fly flight plane book )
t: [Hello] [Hi] [Hey], how can I help you finding your flight?

#! I want to fly from Madrid
u: ([go fly travel] from _~location)
OK, you want to go from _0.
$source = '_0

#! I want to go to Barcelona
u: ([go fly travel] to _~location)
$destination = '_0
OK, you want to go to $destination .

#! The booking is for 3 people
u: (_~number [person people traveller])
$people = _0
OK, $people people are travelling.

#! I have finished telling you
u: ([finished])
[find_flight($source, $destination, $people)]
```

```
Looking for flights from $source to $destination for $people people...
```

Listing 3.16: Example use case written in ChatScript

As we can see, the concepts of ChatScript (such as *location* or *number*) give more possibilities in a single pattern than AIML, which needed separate patterns for each one. Not only the samples written with #! would match, also all the other combinations of the desired words. Also, concepts will restrict the value of wildcards ('*' in AIML) to values like names of cities or numbers.

Another advantage over AIML is that the ChatScript user log is able to store these $source, $destination and $people variables for future chats or calling bot functions. Some AIML implementations do include it, such as Program-D, but the AIML specification does not. We can also operate with these variables. For example, assignments extended across arithmetic operations:

```
$price = $price + 20 * 5 / 59
u: (For less than _0<26 euros) It's difficult to find flights for less
    than 26 euros!.
```

Listing 3.17: Arithmetic operations in ChatScript

Also, in ChatScript it is possible to test variables in patterns:

```
?: ($source=$destination [finished]) You specified the same cities in
    source and destination. Please choose another one.
```

Listing 3.18: Test patterns in ChatScript

ChatScript supports several levels of memorization. The ultimate variable is the fact, which is treated in a different way:

```
u: (harry has a dog) ^createfact(Harry own dog) Really? I didn't know it.
?: (does harry have a dog) ^query( direct_svo Harry own dog) Yes.
```

Listing 3.19: Memorization in ChatScript

This way, ChatScript engine has the ability to run bots that have a better human-like behavior.

3.2.5.5 Conclusions

Using both exposed solutions it is possible to implement a bot able to offer help in an online travel agency. However, when looking for robust functionality, wildcards and concepts in ChatScript offer a very distinctive advantage overcoming AIML. This, and the ChatScript general syntax, makes the corpus code much simpler than in the AIML case. In addition, it is shown how the ability of ChatScript to handle user variables and facts improves the human lookalike behavior.

AIML

- Follows an XML structure - not as human-friendly as ChatScript.
- Rules are only designed for pattern-matching the input user.
- Use of <srai> in one additional pattern for each predictable alternative to the input user text.
- Corpus is loaded by the administrator, and dynamic loading while chatting is usually impossible (excepting some implementations, such as Program-O).
- Corpus for rule definition. The process of chatting and understanding is defined in the core.
- For implementing new functions it is necessary to implement new tags by modifying the core.
- User log and variables not usually remembered (depends on AIML implementation).
- No existing debugging module by default.
- New updates nowadays, such as AIML 2.0 with the <oob> tag.

ChatScript

- Follows its own syntax, much more concise and versatile.
- Rules designed for pattern-matching and proactivity, using gambit t: questions.
- Use of concepts, or brackets in the same pattern. No additional patterns needed. Also ontologies and WordNet.
- Corpus is loaded with administration commands. They allow to reload dynamically the corpus, and other administration tasks.
- Corpus for rule definition, and also script specifying the process that the bot follows.
- New functions by implementing new script functions ($\hat{f}$unction)
- User log and fact base is always remembered and loaded in next conversations.
- Advanced debugging module including the #! lines and other features.
- New updates nowadays, such as the ChatScript 3.0.

3.3 Agent technologies: Jason

Jason[4] is an open-source interpreter for an extended version of AgentSpeak. AgentSpeak is a programming language for BDI agents that has an elegant notation based on logic programming. Jason has been developed by Jomi F. Hubner and Rafael H. Bordini.

Jason provides a platform for the development of multi-agent systems. As one of the most relevant feature four our project, it allows to implement environments and user-defined internal actions written in Java, as it is itself written in this language. We will summarize in the next sections how it works internally, as explained in [36] and [18].

Reasoning cycle

The reasoning cycle of Jason is shown in Fig. 3.6. It starts by (1) perceiving the environment. The environment can be implemented by the developer using the Environment class in Java. This opens a wide range of possibilities for implementing communication protocols or other functionalities. After this, the agents will proceed by (2) updating the belief base and (3) receiving communication from other agents, always (4) filtering the 'socially

[4]`http://jason.sourceforge.net/`

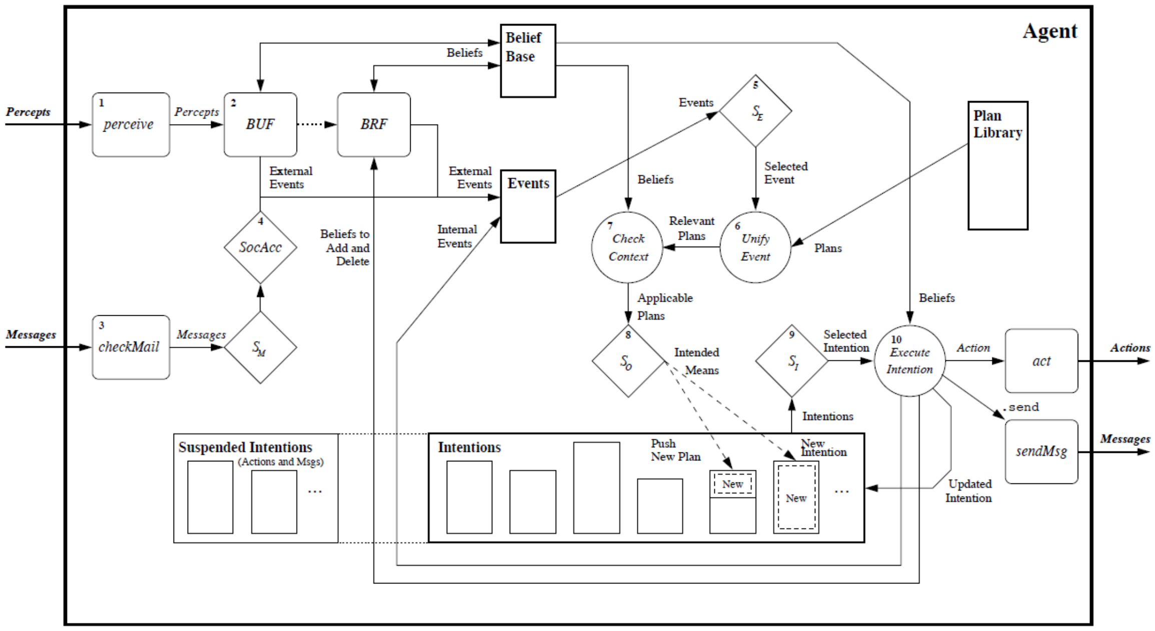


Figure 3.6: Reasoning cycle of Jason

acceptable' messages based on their knowledge about reliability of the other agents. Next, they will continue by (5) selecting an event and (6) retrieving all relevant plans in order to (7) determine the applicable plans and (8) selecting one of them. For achieving this plan, they will (9) select an intention for further execution and iterate (10) executing each step of the intention. Once all the intentions are committed successfully, the plan is finished.

Multiagent architecture

The Jason architecture represents a simulated environment (the *Environment* class), which stores all the percepts of the agent and triggers custom methods to access and process that information. For truly representation of the environment, an auxiliary class called *Model* will be involved, storing the values that the sensors provide and offering methods to access the actuators. This *Model* class is not included in the Jason architecture, so the Environment class must provide the methods to capture the information kept in the Model (see Fig. 3.7).

Perception model

For understanding the perception model in Jason, it is crucial to study in detail the operation of the main processes of the *Environment* class: percepts, agPercepts and updateAgs.

percepts (List<Literal >) stores the general percepts common to all agents, so they will

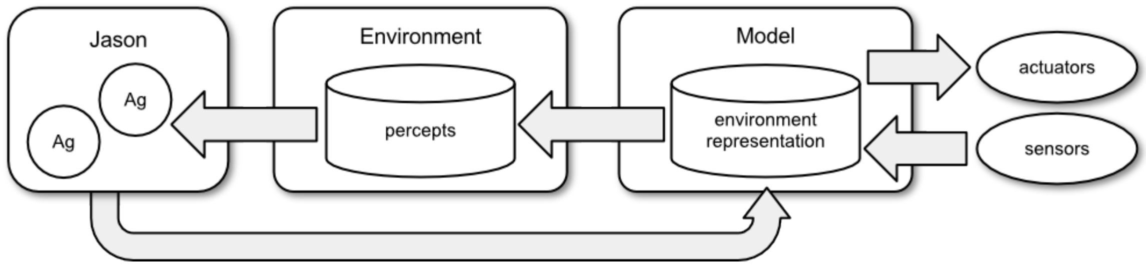


Figure 3.7: Environment and model functionality in Jason

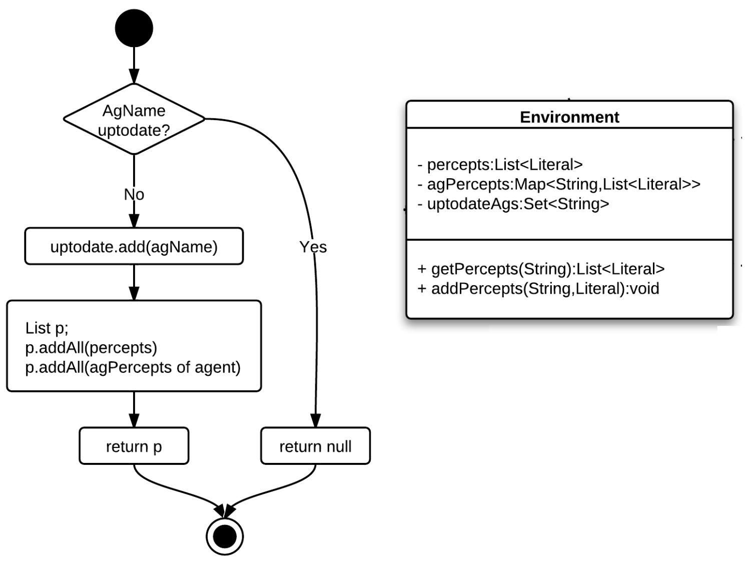

AgName uptodate?
No
Yes
uptodate.add(agName)
List p;
p.addAll(percepts)
p.addAll(agPercepts of agent)
return p
return null
Environment
- percepts:List<Literal>
- agPercepts:Map<String,List<Literal>>
- uptodateAgs:Set<String>
+ getPercepts(String):List<Literal>
+ addPercepts(String,Literal):void

Figure 3.8: Environment getPercepts flow chart and the UML Environment class

be transferred the belief base of every single agent defined. They will only be deleted if it is explicitly done. .

agPercepts (Map<String,List<Literal> >) stores the private percepts that are transferred to the belief base of the specific agent.

uptodateAgs (Set<String>) for performance optimization purposes, keeps the names of the agents whose percepts up to date, so when the systems request the percept list for any of the agents in the list, it will not be provided due to it is not necessary to change it.

User-defined internal actions

Jason offers by default some functions (called standard internal actions). One of example is the "send" internal action, used for communicating agents each other. Apart from the default internal actions, developers (who normally have some specific requirements when developing their systems) usually need to implement new user-defined internal actions. For example, legacy code, database connections, graphical user interfaces, custom agent perception/capabilities or special intention handling.

User-defined internal actions are accessed calling the name of the library, followed by '.', and followed by the name of the action. For example, a new internal action called *send* in a library called *maia*, can be used (both in the context and in the body of the agent) as it is done in the following script:

```
+affirm(R)
        <- maia.send(R," has been affirmed").
```

Listing 3.20: Calling a user-defined internal action in Jason

Internal actions should override the *execute* method. This method is called by the AgentSpeak interpreter to execute the internal action. The third argument of this function contains the arguments passed to the body of the code of the internal action. In addition, it is important to remark that the *suspendIntention* method returns true when the intention is meant to be suspended state, as .at and also .send with askOne as performative.

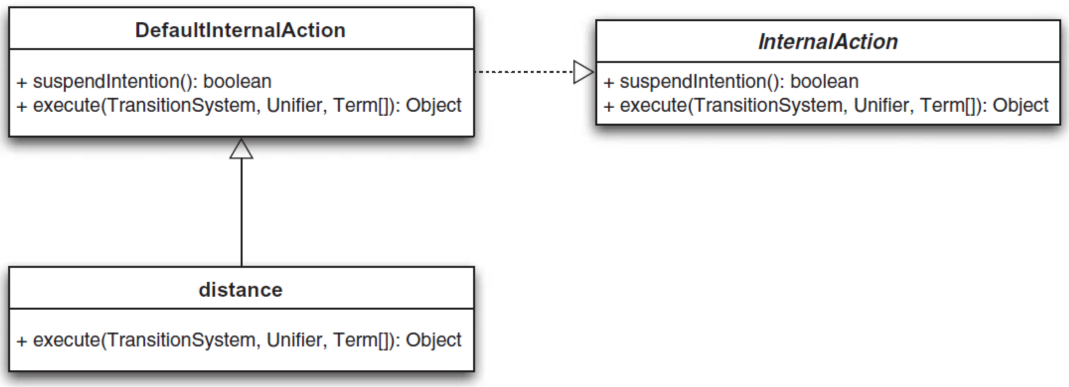

DefaultInternalAction
+ suspendIntention(): boolean
+ execute(TransitionSystem, Unifier, Term[]): Object
InternalAction
+ suspendIntention(): boolean
+ execute(TransitionSystem, Unifier, Term[]): Object
distance
+ execute(TransitionSystem, Unifier, Term[]): Object

Figure 3.9: Internal action class diagram.

Communication between agents

While Jason interprets automatically any received messages according to the formal semantics, it is the programmer's responsibility to write plans which make the agent take appropriate communicative action.

The general form of the pre-defined internal action for communication is:

```
.send(receiver,illocutionary_force,propositional_content)
```

Listing 3.21: Communication in Jason

And the meaning of each of its parameters is as follows:

receiver is the destination agent or agents.

illocutionary_force is a term often representing a literal but sometimes representing a triggering event, a plan or a list of either literals or plans. The list of all available performatives are as follows:

- **tell** intends the receiver to believe a fact.
- **untell** intends the receiver not to believe a fact.
- **achieve** request the receiver to achieve a state of affairs, by creating a goal.
- **unachieve** request the receiver to drop the goal of achieving a state of affairs.
- **askOne** asks the receiver if the content of the message is true for him.
- **askAll** asks all the receivers for an answer to a questions.
- **tellHow** informs the receiver of the know-how of a plan.
- **untellHow** requests the receiver to forget a plan.
- **askHow** asks the receiver the relevant plan for a triggering event.

propositional_content refers to the content of the message.

Agents can ignore messages: for example, when a message delegating a goal is received from an agent that does not have the power to do so, or from an agent towards which the receiver has no reason to be generous. Whenever a message is received by an agent, if the agent accepts the message, an event is generated.

3.4 Evented web technologies

Evented Web [5] is a term that groups the technologies coupled with a vision of the traditional web APIs complemented by other APIs that produce events. Adding a callback mechanism to web APIs, it makes the web more like a giant evented framework.

Node.js is probably the most popular evented framework. It has been the base for building WebSocket implementations such as Maia. Both Node.js and Maia are presented in the next sections.

3.4.1 Node.js

Node.js [6] [18] is an environment for executing server-side JavaScript, based on Google Chrome's runtime (V8 engine). It has a boiling community behind it, despite being a very new technology: in addition to the uncountable source codes that we can find for it, there are existing libraries for implementing an incredible large set of features. This demonstrates the big number of possibilities of node.js, as well as the easiness of development using this platform.

Furthermore, as Node.js is community driven, most of these available libraries are open source. Through the Node Package Manager [7] there are hundreds of packages that can be installed easily.

3.4.2 Maia

Maia[8] is an evented and agent-oriented architecture built on Node.js. Its cornerstone are event messages [18]. A Maia event can be either informative or a request to trigger an action in a remote entity, depending on the context or a special namespace that can be used to specify it.

The basic components of any Maia event are the following:

Sender contains a unique identifier of the sending entity. It is recommended to append the identifier of the bus it is connected to, if the scenario contains more than one.

ID contains a unique identifier of the event for the specified entity (sender).

[5]`http://progrium.com/blog/2012/11/19/from-webhooks-to-the-evented-web/`

[6]`http://nodejs.org/`

[7]`http://npmjs.org]`

[8]`https://github.com/gsi-upm/Maia`

Timestamp points the time of the original emission. This makes time reasoning possible and prevents side effects of asynchronous communications.

Name describes the event, and is the only required field. It consists on a complete namespace that allows complex processing of events and advanced filtering for the triggers.

Payload contains either more information about the entities involved in the event, or the parameters in case the event is a request.

Callback specifies a function to execute whenever there is data to return, or just to acknowledge the reception of the event.

In Maia, events are defined as nodes of a taxonomy. This brings up several advantages. The main one, is that we can define an ontology of intents, so every item in the ontology has a set of properties that define the action to be taken and the response given. This way, we can filter events not only by class, but also by their properties, and even use reasoning with them.

Nodes in Maia are loosely coupled, which allows higher flexibility than other networks. All the information is inside the message, including emitter and sender, included as strings contained in the Name of the event. This makes routing events as easy as parsing event names and applying certain rules. The routing complexity in Maia can be as simple or complex as the designer desires: the only changes will be in the routers. Connected nodes are only influenced by this in the way that they have access to a higher or smaller range of events from more or less sources.

3.5 Information retrieval technologies for the Semantic Web

3.5.1 The RDF framework and the Web of Data

RDF (Resource Description Framework) is a language for describing things or entities on the World Wide Web [37]. It allows websites to expose semi-structured information for machine reuse, implementing something referred to as the *Web of Data* or Semantic Web . The idea behind this Web of Data is to *create a universal medium for the exchange of data, where data can be shared and processed by automated tools as well as by people* [9]. A typical use of Web of Data are automatic interactions with advanced clients: for example, integrating the user's calendar and contact list by encoding entities using formats such as HCard, FOAF,

[9]`http://www.w3.org/2001/sw/Activity.html`

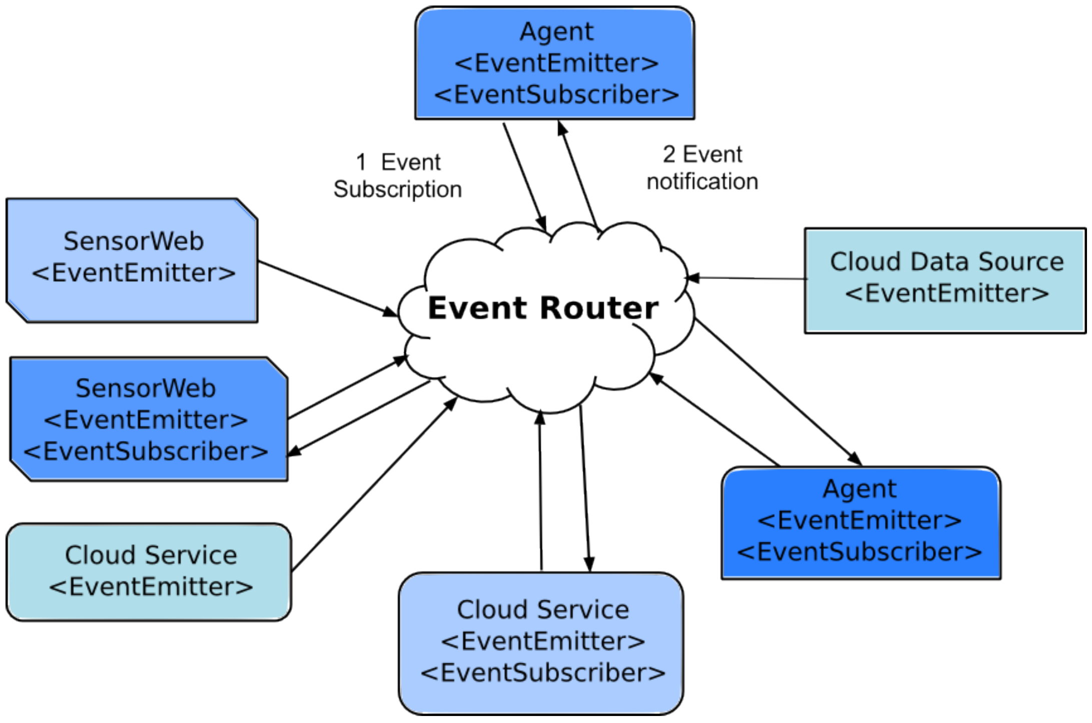


Figure 3.10: Flow of events: entities can simply send event notifications to the Event Router, or subscribe to be forwarded a certain subset of events.

or VCard. Other applications are, for example, search engine result customization and advanced data mashups.

A good explanation of the RDF structure can be found in [38]: the RDF data is structured as connected graphs, and is composed of triples. A triple is a statement consisting of three components: a subject, a predicate and an object. Such a statement can be anything, for instance "Peter has a friend named John". This could be formally structured as a triple in an RDF graph as this: Peter hasFriend John, where Peter would be the subject, hasFriend would be the predicate, and John would be the object. This example is an abstraction of how triples should be structured, as the structure of triples is built around Uniform Resource Identifiers (URIs), literals and blank nodes.

Moreover, this means that the subject and predicate, and in many cases the object, are represented by a URI, meaning that they have a unique identifier to represent them. The object of the triple can either be a literal (such as a textual description), a date or an integer. Subjects and objects can also consist of blank nodes –anonymous nodes representing resources where a URI or literal has not been given.

RDF data is built on the idea of utilizing unique namespaces or vocabularies for describing data. This means that every data resource represented by a URI is a part of a unique namespace that identifies what that resource is a part of. For example, if we wanted to specify the latitude of a geo location, we could the predicate "http://www.w3.org/2003/01/geo/wgs84_pos#lat" where the namespace would be "http://www.w3.org/2003/01/geo/wgs84_pos#" (the knowledge domain) and "lat" would be the local name of the latitude description within that namespace. Furthermore, this means that common knowledge domains and vocabularies can be reused by external data sets, making the data more interoperable in terms of sharing, implementing, and interchanging data between systems.

Opposite to the traditional *Web of documents*, RDF data makes possible for computers to understand the information that they are displaying to the users. This means that they can help the users to put the information into context, inferring and exploring new data relationships, and making searching more accurate and efficient.

Linked Data technology

Linked Data refers to interlinking data, usually stored in RDF. . The idea behind Linked Data is not focusing only in linking *documents* together, but linking *data* together [39]. Its purpose is giving data meaning to both humans and machines by defining unique resources to describe concepts.

For example, when saying the word "apple", one could specify either the fruit *apple* or the company *Apple*. Humans can usually make sense of which "apple" refers to by the given context, but the machines cannot. By linking the concept "apple" to a Unique Resource Identifier (URI) pointing to a resource describing the specific concept, even machines can understand which concepts the text refers to [38]. Tim Berners-Lee outlines four principles of linked data:

- Use URIs to denote things.
- Use HTTP URIs so that these things can be referred to and looked up ("dereferenced") by people and user agents.
- Provide useful information about the thing when its URI is dereferenced, leveraging standards such as RDF and SPARQL.
- Include links to other related things (using their URIs) when publishing data on the Web.

The goal of the W3C Semantic Web Education and the Outreach group's *Linking Open Data community project* is to extend the Web with a data commons by publishing various open datasets as RDF on the Web and by setting RDF links between data items from different data sources. By September 2011 (see Fig. 3.11), this had grown to 31 billion RDF triples, interlinked by around 504 million RDF links. There is also an interactive visualization of the linked data sets to browse through the cloud.

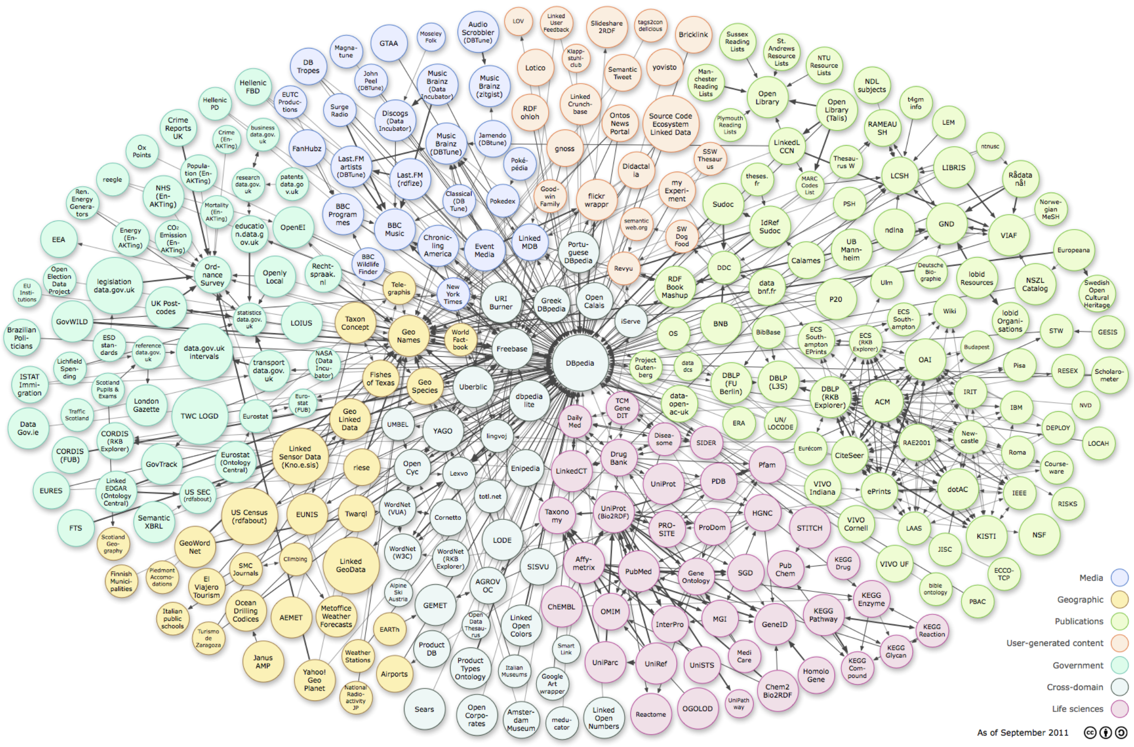


Figure 3.11: The Linking Open Data cloud (September 2011)

3.5.2 Scrappy

Scrappy allows collecting data from the Internet and storing it in structured data formats[10] [40]. It can be described as an intermediary that provides access to the web for semantics, abstracting tasks such as information extraction or recursive crawl. Data can be collected in a wide variety of formats such as RDF, JSON, or PNG nTriples.

It uses a web resource ontology to define mappings between ontology terms and not listed web resources. This enables obtaining RDF data from the web using multiple interfaces, such as command line console, web services or web proxies. Scrappy was developed in Ruby language, and it uses the Webkit library for visual processing of web resources.

The main reason for its development was the recent initiatives, such as the emerging

[10]`https://github.com/josei/scrappy`

Linked Data Web, which were providing semantic access to available data by porting existing resources to the semantic web using different technologies. These technologies were, for instance, database-semantic mapping and scraping. Nevertheless, these scraping solutions were based on ad-hoc solutions complemented with graphical interfaces only looking to speed up the scraper development.

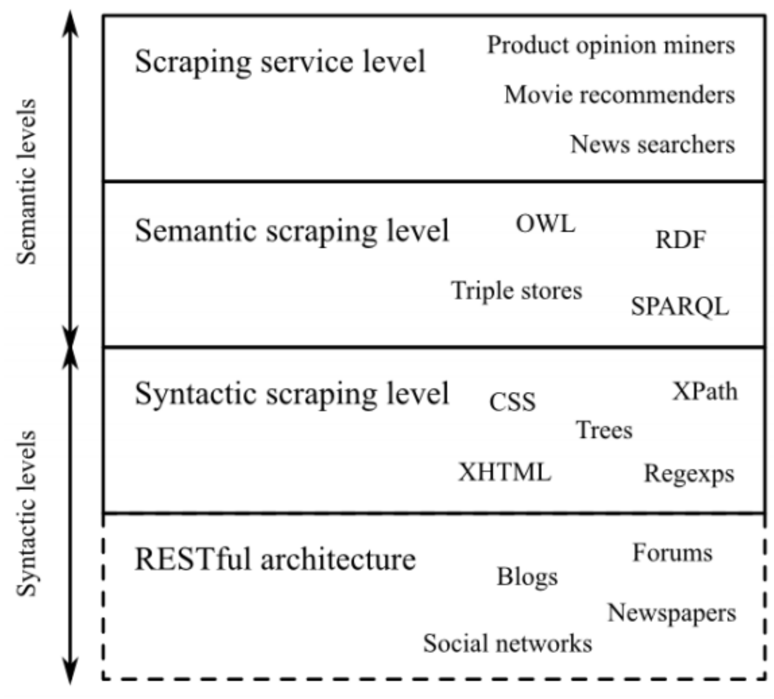


Figure 3.12: Semantic scraping framework proposed in Scrappy

The framework for web scraping in Scrappy is based on semantic technologies. It is structured in three levels stacked on top of the REST architectural style:

Scraping services provides an interface to generic applications or intelligent agents for gathering information from the web at a high level. This level comprises services that make use of semantic data extracted from unannotated web resources. Possible services that benefit from using this kind of data can be opinion miners, recommenders or mashups that index and filter pieces of news.

Semantic scraping defines a semantic RDF model of the scraping process, in order to provide a declarative approach to the scraping task. This level defines a model that maps HTML fragments to semantic web resources. By using this model to define the mapping of a set of web resources, the data from the web is made available as knowledge base to scraping services.

Syntactic scraping provides an implementation of the RDF scraping model for specific technologies. This level gives support to the interpretation to the semantic scraping model. Wrapping and Extraction techniques such as DOM selectors are defined at this level for their use by the semantic scraping level.

This framework allows defining services based on screen scraping by linking data from RDF graphs with contents defined in HTML documents. This way, it is possible to build a semantic scraper that uses RDF-based extractors to select fragments and data from web documents and build RDF graphs out of unstructured information.

3.5.3 SIREn

Although most attention in the Semantic Web community has focused on building triple stores with expressive query languages and using database technology, there is still a small number of Semantic Web search engines that implement large-scale search using inverted indices [41]. The best known example is Sindice[11] and its open source information retrieval engine SIREn[12] [42], which builds on Lucene, the popular Java information retrieval package.

Apache Lucene[13] is a free open-source high-performance, scalable information retrieval engine written in Java. It offers full-featured text search, based on indexing mechanisms. SIREn is implemented on top of Apache Lucene in order to achieve efficient semi-structured information retrieval.

SIREn is an entity retrieval system designed to provide entity search capabilities over datasets as large as the entire Web of Data. It supports efficient full text search with semi-structural queries and exhibits a concise index, constant time updates and inherits Information Retrieval features such as top-k queries, efficient caching and scalability via distribution over shards [43]. It is demonstrated how this system can effectively answer queries over 10 billion triples on a single commodity machine.

With SIREn, it becomes possible to search across millions of highly and diversely structured documents having millions of different attributes. According to the benchmark reports, SIREn exhibits an index size ratio of 13-15%, whereas this value is approximately 50% for Sesame[14]. Also, the indexing time of SIREs is 50-100 times faster than that of Sesame.

Even if it offers several features as supporting the multiple formats which are used on

[11]http://sindice.com

[12]http://siren.sindice.com

[13]http://lucene.apache.org

[14]https://dev.deri.ie/confluence/display/SIREn/Benchmarks

the Web of Data, the main use case for which SIREn is developed is still entity search . Given a description of an entity (a query such as the one in the right side of Fig. 3.13), it is able to locate the most relevant entities and datasets. In these query graphs, oval nodes represent resources and rectangular ones represent literals. For space consideration, URIs have been replaced by their local names.

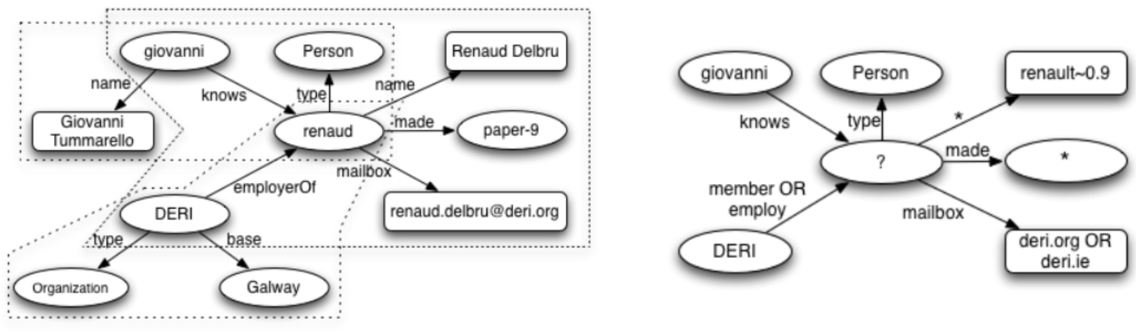


Figure 3.13: Visual representation of an RDF graph (left) and Star-shaped query matching (right)

The figure on the left shows an RDF graph and how it can be split into three entities *renaud*, *giovanni* and *DERI*. Each entity description forms a sub-graph containing the incoming and outgoing relations of the entity node which is indexed by the system.

Finally, including the features described above, the overall design requirements of SIREn comes to four points:

1. Support for the multiple formats which are used on the Web of Data.

2. Support for searching an entity description given its characteristics (entity centric search).

3. Support for context (provenance) of information: entity descriptions are given in the context of a website or a dataset.

4. Support for semi-structural queries with full-text search, top-k query results, scalability over shard clusters of commodity machines, efficient caching strategy and incremental index maintenance.

3.6 Conclusions

In this chapter, we have studied the different enabling technologies that enable the opportunity to implement an efficient personal assistance system. As seen in these sections, there are multiple choices and still today new continue arising. This involves an exhaustive study of all of them, so we are able to select the most appropriated alternatives.

In the field of **natural language understanding**, it can be seen that AIML and its implementations provide a robust technology, and it is still updated with new features today. It has been widely used over the last years, and its high number of implementations are proof of it. Nevertheless, ChatScript overcomes its overall performance. It reinvents the language for writing corpus, presenting a much more simple syntax that also offers several features that AIML does not. We have studied these features relevant for our project, such as concepts, proactivity or fact handling.

As **intelligent agent system**, we have demonstrated that Jason offers a really good performance level for our system. Having already an advanced agent technology, with features as a powerful reasoning cycle, it also implements an open perception model and user-defined internal actions.

For our project, we will also need a communication network that allows us to connect actuators and execution modules. Our choice has been to implement the **event web technology** Maia, built on Node.js, for offering an event based network that allows the exchange of event messages between modules.

Finally, for better assisting the user, we enable the access to large amounts of information making use of **information retrieval** technologies. We will especially use SIREn, built on top of Lucene. This allows us to index semi-structured information and offer an incredibly quick entity retrieval service. This information can be optionally collected from unstructured sources in the Web, and then be converted to the desired semi-structured format only using Scrappy.

Chapter 4

Requirement Analysis

"Crede quod habes, et habes."

4.1 Overview

This chapter describes one of the most important stages in software development: the requirement analysis using different scenarios. For this, a detailed analysis of the possible use cases is made using the Unified Modeling Language (UML). This language allows us to specify, build and document a system using graphic language.

The result of this evaluation will be a complete specification of the requirements, which will be matched by each module in the design stage. This helps us also to focus on key aspects and take apart other less important functionalities that could be implemented in future works.

As commented before, our project aims to develop a personal assistance system that integrates the advantages of agent systems, information retrieval and Natural Language Processing. Our personal assistant should have the capability to solve user's questions using all of these features.

4.2 Use cases

These sections identify the use cases of the system. This helps us to obtain a complete specification of the uses of the system, and therefore define the complete list of requisites to match. First, we will present a list of the actors in the system and a UML diagram representing all the actors participating in the different use cases. This representation allows, apart from specifying the actors that interact in the system, the relationships between them.

Actors list

The list of primary and secondary actors is presented in table 4.1. These actors participate in the different use cases, which are presented later.

Actor identifier	**Role**	**Description**
ACT-1	User	System user.
ACT-2	Admin	System administrator.
ACT-3	Agent System	Intelligent agent system.
ACT-4	QA System	Question-Answer System.
ACT-5	Chatbot	Chatbot loaded into the system.
ACT-6	External service	External services for executing actions.
ACT-7	Linked Open Data Server	Server storing Linked Open Data.
ACT-8	Web Knowledge Base	Knowledge base available on the Web.

Table 4.1: Actors list

UML diagram of use cases

The UML diagram presented in Fig. 4.1 represents all the described actors and how they participate in the different use cases. These use cases will be described the next sections, including each one a table with their complete specification. Using these tables, we will be able to define the requirements to be established.

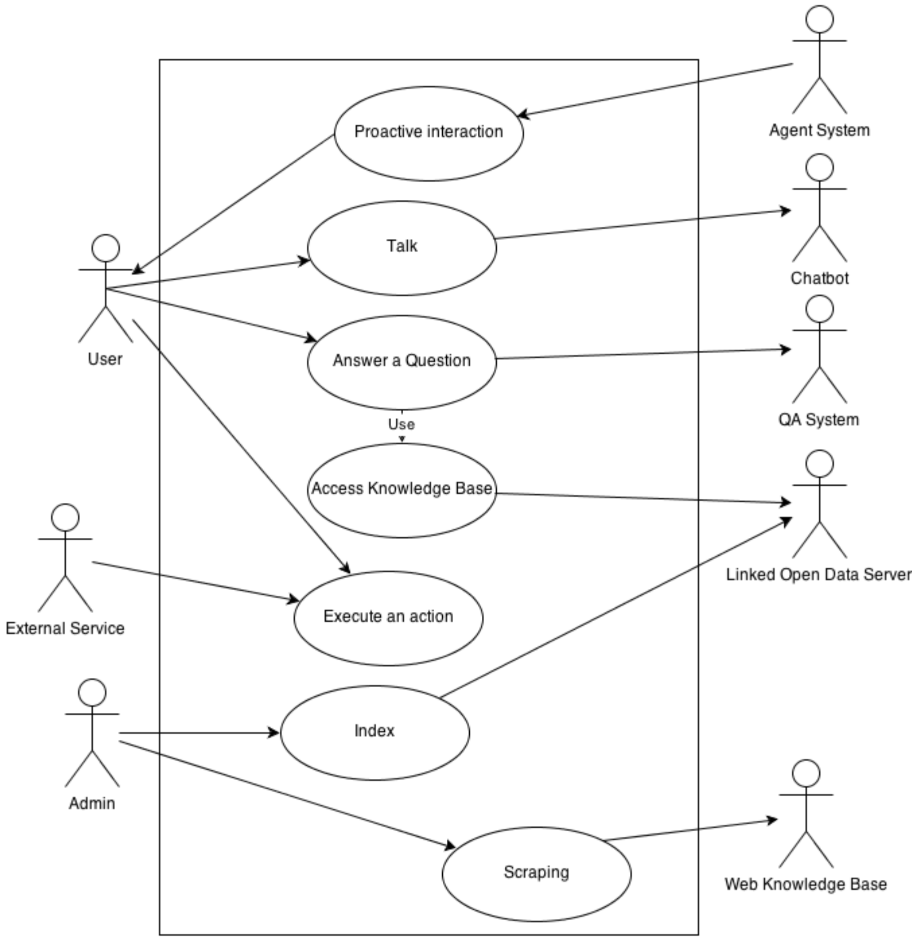


Figure 4.1: Representation of the natural language use cases

4.2.1 Proactive interaction (CU-1)

Our personal assistant should be proactive. This means that, collecting the assertions extracted from the conversations and the user profile, it should create its own propositions or questions and send them to the user. For example, in the e-learning field, if a lesson has been already explained to the student, the personal assistant should propose him to revise this lesson the next time he accesses to the platform.

Additional intelligence needs be implemented in order to support proactivity, so using intelligent agents becomes a requirement at this point. The summary of this use case in presented in table 4.2.

Identifier	CU-1
Name	Proactive interaction.
Description	Support proactivity using intelligent agent technologies.
Actors	Agent System, User.
Preconditions	-
Usual flow	1. The Agent System collects assertions about the user and the conversation. 2. Agent's plans according to these assertions are triggered, producing proactively an output that will be sent to the user.
Alternative flow	-
Postconditions	-

Table 4.2: Use Case 1

4.2.2 Talk (CU-2)

The user talks to the personal assistance system, and the Chatbot is able to maintain a conversation with the user. For this, natural language needs to be both understood and produced in the reply for the user. The summary of this use case in presented in Table 4.3.

4.2.3 Answer a question (CU-3)

The user sends his questions to the personal assistance system, and the QA System handles the task to answer them. The summary of this use case in presented in table 4.4.

4.2.4 Access knowledge base (CU-4)

In order to find information, the knowledge base is accessed. Information can be fetched from the Linked Open Data Server. 4.5.

Identifier	CU-2
Name	Talk.
Description	Maintain a natural language conversation with the user.
Actors	User, Chatbot.
Preconditions	The conversation corpus is loaded into the chatbot.
Usual flow	1. The user sends a query to the system written in natural language. 2. The chatbot understands the user's query. 3. The chatbot provides a natural language response for the user.
Alternative flow	-
Postconditions	-

Table 4.3: Use Case 2

Identifier	CU-3
Name	Answer a question.
Description	Handle questions sent by the user and find a response for them.
Actors	User, QA system.
Preconditions	-
Usual flow	1. The user sends a question to the system. 2. The QA system identifies the question. 3. The QA system retrieves the needed information accessing to the knowledge base. 4. The QA system generates the answer for the user.
Alternative flow	-
Postconditions	-

Table 4.4: Use Case 3

Identifier	CU-4
Name	Access knowledge base.
Description	Access knowledge base and fetch information from the Linked Open Data Server.
Actors	Linked Open Data Server.
Preconditions	The Linked Open Data Server contains information ready to be fetched.
Usual flow	1. Information is fetched from the Linked Open Data Server and added to the knowledge base.
Alternative flow	-
Postconditions	-

Table 4.5: Use Case 4

4.2.5 Execute an action (CU-5)

The system should be able to trigger actions to be executed in external services, usually on user's demand. These external services usually run in a remote system, so this communication should be over a network. The summary of this use case in presented in Table 4.6.

4.2.6 Index (CU-6)

Semi-structured information is indexed and then loaded into the Linked Open Data Service. The summary of this use case in presented in Table 4.7.

4.2.7 Scraping (CU-7)

The information on the Internet is normally presented in unstructured formats. Using scraping techniques, the information can be retrieved from a web knowledge base and then converted to semi-structured format. The summary of this use case in presented in Table 4.8.

Identifier	CU-5
Name	Execute an action.
Description	Execute actions by external services.
Actors	User, External service.
Preconditions	-
Usual flow	1. The user sends a query that requires executing an action. 2. The request of execution is sent to the external services. 3. The action is executed, and the possible output for the user is sent back.
Alternative flow	-
Postconditions	-

Table 4.6: Use Case 5

Identifier	CU-6
Name	Index.
Description	Index semi-structured information for the Linked Open Data Service.
Actors	Admin, Linked Open Data Service.
Preconditions	-
Usual flow	1. The administrator provides semi-structured information to the indexer. 2. The information is indexed and loaded in the Linked Open Data Service.
Alternative flow	-
Postconditions	-

Table 4.7: Use Case 6

Identifier	CU-7
Name	Scraping.
Description	Retrieve unstructured information from a web knowledge base on the Internet and convert it to semi-structured format.
Actors	User, Web Knowledge Base.
Preconditions	Web scrapper able to identify each content.
Usual flow	1. The administrator runs the scrapper over the website containing the desired web knowledge base. 2. The scrapper fetches the pieces information in the web knowledge base and stores them with a semi-structured format.
Alternative flow	-
Postconditions	-

Table 4.8: Use Case 7

4.3 Conclusion: summary of requirements

In this chapter, we have studied the use cases that are applied to our project. Now, we can focus our study in some clear requirements presented next:

1. Ability to understand natural language, as well as maintaining conversations in natural language with the user.

2. Proactive interaction with the user by using intelligent agents. Proactivity means not only giving response to his queries, but also creating self-made propositions.

3. Apart from simple conversation, offering help by answering the user's questions about a subject depending on the application field.

4. Storing data in a knowledge base that holds information to answer the user's questions. Possibility of quickly access this knowledge base to fetch an answer for the user.

5. Executing actions in external services, in order to improve the reply for the user or to execute actuators on user's demand. Heterogeneous but interchangeable end-points, providing modularity and a common interface for connecting new services.

6. Possibility of scraping contents from unstructured knowledge bases on the Internet, converting them to a semi-structured format and then indexing and offering them in a Linked Open Data Service.

7. Multiple user support, in order to offer the service to any number of users at the same time.

Chapter 5

Architecture and design of the solution

"Good fortune is what happens when opportunity meets with planning."

—Thomas Alva Edison

5.1 Introduction

This chapter describes in detail the design stage of the project, which is the most important and determining stage for achieving the desired system or product. Design means applying a set of skills and methodologies to obtain a detailed result, in such a way that any software developer can be able to implement it.

Sometimes, the term "design" is associated to the creation of a graphical interface. Otherwise, in the software development field, "design" refers to the process of meeting all the requirements specified in a previous development stage. For the design of our personal assistance system, we will describe both its global architecture and the details of each one of its components.

5.2 Global description of the system

In this section, we will provide a general sight of our personal assistance system, by briefly describing its global architecture. After this, in the following sections, we will describe in detail each one of its modules.

The requirements set in the previous development stage (section 4.3) are met in the designed architecture. For this, it is divided in different modules as shown in Fig. 5.1.

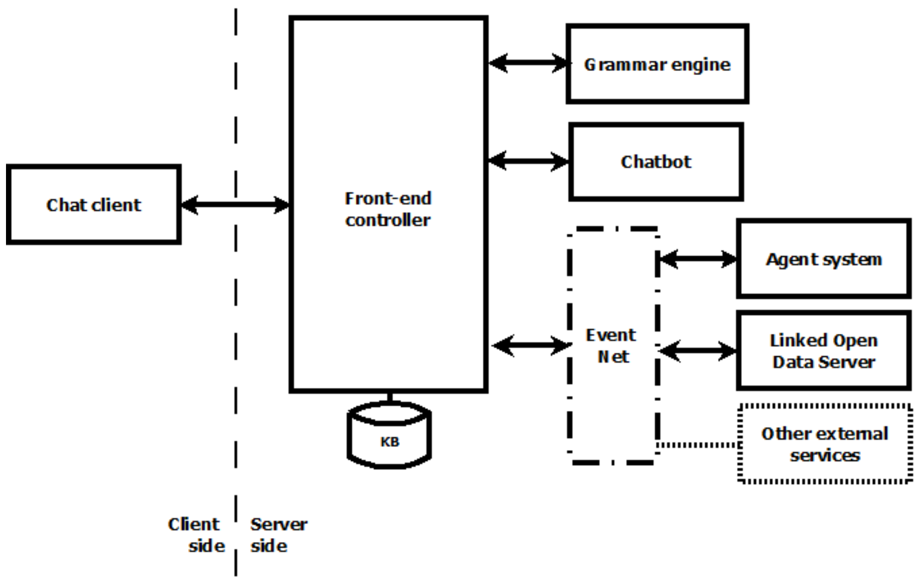


Figure 5.1: Global architecture of the system

These modules are:

Chat client, in the client side. It is a user interface that allows him to communicate with the system, by talking to the personal assistant in natural language. As well as answering, the system asks for feedback to the user in some specific cases.

Front-end controller. It controls the process followed to generate a reply for the user's query. It triggers the other modules to understand the user query, execute external actions if needed, and finally formulate the reply that is sent back to the user.

Grammar engine, used for detecting specific patterns by applying grammars to the user query. These grammars are usually applied to match phrases containing a set of concepts specified in a list (for example, 'do-while', a concept of the Java programming language). This list of concepts can also be updated dynamically during the conversation.

Chatbot. The chatbot is able to maintain a conversation in natural language with the user. It applies the patterns specified in its corpus, which usually produce both a natural language response and out of band commands (hidden for the user) that are handled by the front-end controller.

KB, or Knowledge Base. Contains information ready to be provided by the Chatbot when the user sends a question about a subject. It is dynamically updated when new information is fetched from the Linked Open Data Server.

Event network. It allows the communication between the front-end controller and the external services, such as the agent system or the Linked Open Data Server. This network is based on event messages that contain the information exchanged by these modules.

Agent system. An intelligent agent system that receives assertions collected during the conversation and provides them as beliefs to the agent(s) running on it. It provides our personal assistance system an extra interaction with the user, by triggering plans that automatically produce replies that are sent to the user.

Linked Open Data Server. Contains indexed information, which has been previously added by the system administrator. When the user sends a question and no information is found in the KB, this service is ready to provide the needed information.

Other external services (optional). Other possible external services that could execute different actions. Depending on the application field, these external services can offer the extra features that may be needed.

The sequence of steps needed to formulate a reply for the user may vary, depending on the modules that the system needs to trigger. We show next two examples of these sequences.

In the first case, the simplest one, we will suppose that the user is only making simple conversation. For example, when asking to the personal assistant "Are you the teacher?" the first time he uses the system. In this case, the Chatbot is able to formulate a reply by himself: "No. I am a virtual assistant". Here, it does not deduce any assertion for the agent system neither. The UML sequence diagram of this case is shown in Fig. 5.2.

The steps shown in Fig. 5.2 are described next:

1. The user submits a line of simple conversation in the chat client. It is received by the front-end controller.

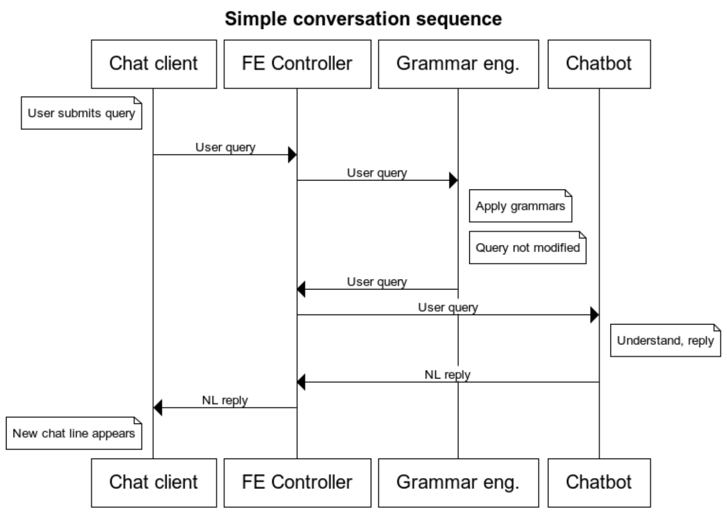


Figure 5.2: UML sequence diagram of simple conversation.

2. The grammars are applied to the query. No changes are applied to the query, because any specific pattern is detected in the query.

3. The Chatbot receives the unmodified user's query. As its corpus contains a pattern matching this query, it formulates the reply by himself.

4. The reply is sent back to the user. It will appear as a new conversation line in the chat interface.

The second example is shown in Fig. 5.3. We show the processing steps of a more complex case: the user makes a question about a concept, and the system needs to make use of all the other modules for formulating the answer.

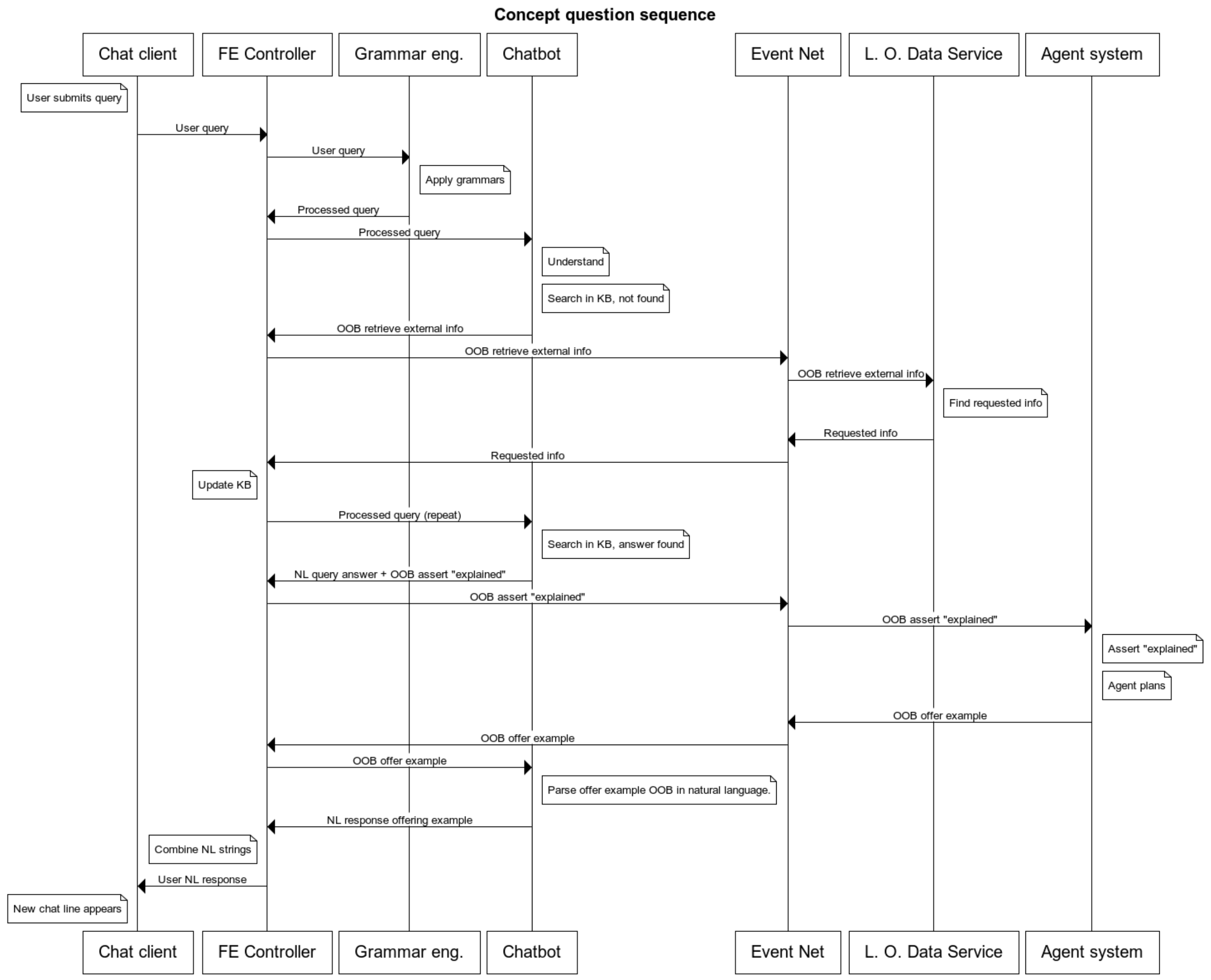


Figure 5.3: UML sequence diagram of handling an user's question

The steps shown in Fig. 5.3 are described next:

1. The user submits a question about a concept using the chat client. This query is received by the front-end controller.

2. The grammars are applied to the query, detecting that the query is a question about a concept. This produces an out-of-band command telling about this detection.

3. The Chatbot receives the processed query, requesting to answer a question about the specified concept. He looks for the requested information in the KB, but it does not contain any. He sends then and out-of-band command requesting to retrieve information about this concept in external sources.

4. The request is put into the Maia network where all the modules are connected, and the Linked Open Data Service is the one that accepts the request. After finding the requested information, it sends a response with it.

5. The front-end controller receives the information and updates the KB with these new contents. The chatbot is now able to answer the question with this information. Also, he informs of an assertion: this specific concept has been explained to the user.

6. The assertion is put in the Maia network and received by the agent system. The corresponding agent receives it, and this triggers one plan that generates an output to be sent to the user (e.g. offering an example, or making a test). This output is parsed in natural language by the chatbot.

7. Both the answer providing the requested information and the agent output are replied to the user. This reply will appear as a new conversation line in the chat interface.

5.3 Front-end controller module

The front-end controller is the main control module of the system. Since the moment when the user sends his query, this module triggers all the processes needed for formulating an answer for the user. In the next subsections, about its functional and structural model, we will see all these processes that it triggers, both locally and externally. The front-end controller needs to be provided as web service, so we have chosen the Python web framework Bottle [30] for implementing this module.

5.3.1 Functional model

As we commented before, the function of this module is formulating the answer for the user's query. This process is composed by the activities shown in Fig. 5.4.

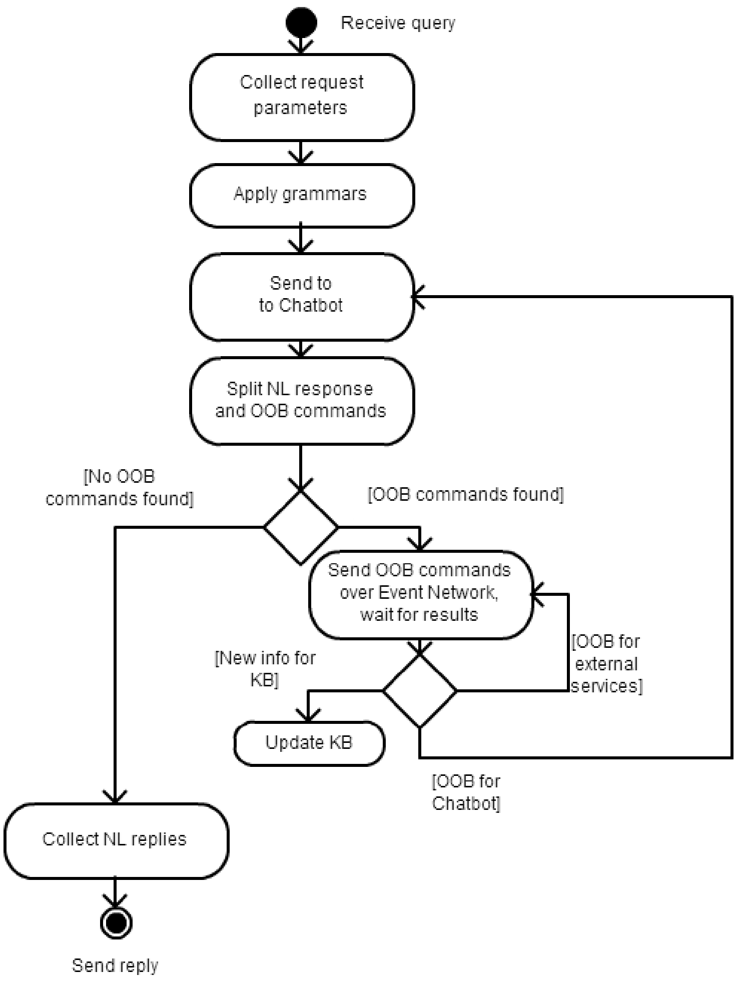


Figure 5.4: UML activity diagram of the front-end controller.

The interaction of these activities with the other modules can be easily understood seeing the Fig. 5.5. The description of each of these interactions is commented below.

Request parsing. The user's query is received in a HTTP request sent to the front-end API. All the parameters of this request are collected and processed.

Grammar application. The first processing made to the user's query will be applying the provided grammars. They will usually detect queries mentioning a concept that exist in a list they have (called dictionary). In case of detection, it produces an out-of-band

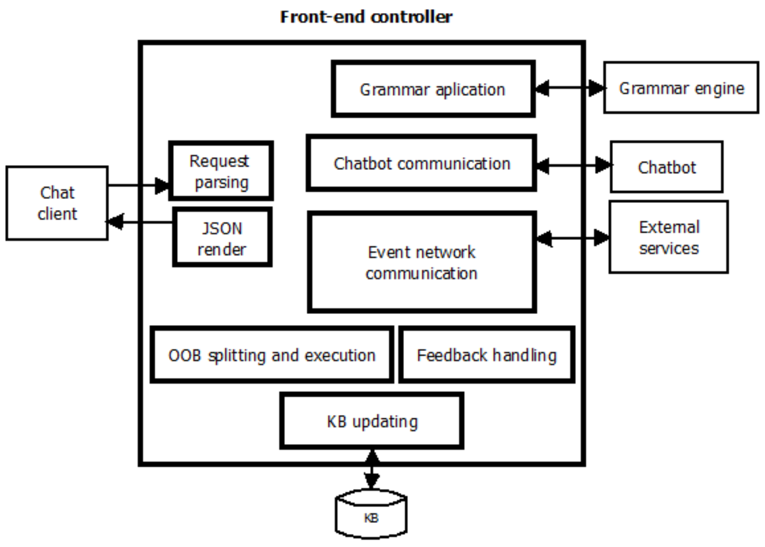


Figure 5.5: Front-end controller diagram.

command telling about it.

Chatbot communication. After the grammars are applied, the already processed query will be handled to the Chatbot. It can both provide a natural language response or out-of-band commands for further execution.

OOB splitting and execution, and Maia communication. The controller splits a string containing both natural language response and out-of-band commands inside brackets. Then, if any out-of-band commands are found, proceeds triggering their execution. The out-of-band commands requesting execution of external services will be sent through the event network.

KB updating. In case the external services (specially the information retrieval ones) fetch information for the knowledge base, it is updated with the new data.

JSON rendering. The HTTP request made by the client will be responded with a JSON entry. It contains the parameters provided in the original query, together with the formulated answer and other data (for example, if collecting feedback with the user). This response, being in JSON format, can be easily represented by the chat client.

Feedback handling. Using the chat client, the user can provide feedback for a given answer. This feedback is sent to the front-end controller the same way as the queries, and then it is collected and logged for further system improvements.

5.3.2 Structural model

The structure of the front-end controller is shown in the UML class diagram shown in Fig. 5.6.

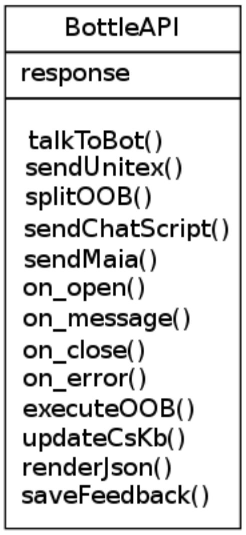


Figure 5.6: Front-end controller.

These functions that compose the front-end controller are described next:

- `talkToBot()` will be triggered each time that the chat client sends a request to the controller API. This function collects request parameters, shown in table 5.1. Then, it controls the process of understanding the query and formulating the reply for the user. A detailed description of the client chat module can be found in section 5.4.

- `sendUnitex()` insures that the grammars are applied to the user's query. Usually, this means detecting queries containing concepts listed in a dictionary and, if case of detection, produce an out-of-band command informing about it. A detailed description of this module can be found in section 5.5.

- `sendChatScript()` creates a TCP connection with ChatScript and sends a message containing the already processed query. ChatScript can both provide a natural language response or out-of-band commands for further execution. A detailed description of this module can be found in section 5.6.

- `sendMaia()` is triggered when an out-of-band command requests to execute an action in a external service. It sends a network event containing the requested action, and waits for a reply. When a reply arrives, the `on_message()` function is executed.

Parameter	Name	Description
`q`	User's query	Textual query submitted by the user.
`user`	User	User requesting help of the system.
`bot`	Bot	Selected bot for the platform.
`type`	Response format	Format of the response (usually JSON).
`feedback`	Response feedback	Flag indicating that this message contains only positive (1) or negative (2) feedback about the provided response.
`response`	Response given to the query	Response under feedback evaluation.

Table 5.1: Input parameters for the API of the front-end controller

Other functions for this connection are `on_open()` and `on_close()`, triggered when the channel is opened or closed, and `on_error()`, that logs possible errors in the communication. A detailed description of this module can be found in section 5.7.

- `splitOOB()` is called with a string as parameter, and it splits it into an array. This array contains the natural language in the string, separated from out-of-band commands that the query may contain. After this split is made, the controller usually calls `executeOOB()`, that proceeds with the execution of these out-of-band commands.
- `updateCsKb()` updates the Knowledge Base with new information, usually coming from the information retrieval when it was requested.
- `saveFeedback()` is called when feedback is submitted by the user, saving it into the system log. This feedback may be collected by the administrator for improving the system after.
- The `parseJson()` function finally renders the reply in JSON for providing it as response for the HTTP request. This response contains the parameters shown in table 5.2

Finally, we will illustrate the control of the behaviour of the Front-end controller by describing the different **out-of-band commands** that it supports, presented in table 5.3. To

Parameter	Name	Description
`q`	User's query	Textual query submitted by the user.
`user`	User	User requesting help of the system.
`bot`	Bot	Selected bot for the platform.
`response`	Response	Response given to the query.
`full_response`	Full response flag	Flag meaning that this is a final answer. Usually, it means that feedback for this answer should be collected.
`mood`	Bot mood	Indicates the emotional state (happy, sad, etc.) caused in the bot.
`sessionid`	Session ID	Session ID that is hold in this conversation.

Table 5.2: Output parameters for the API of the front-end controller

differentiate them from the natural language answer, they are always expressed between brackets together with their parameters: `[command parameters]`.

Command	Description	Example
`sendcs`	Send to chatbot	`[sendcs Goodbye]`
`sendmaia`	Send to ext. service	`[sendmaia assert explained(concept for)]`
`updatekb`	Update KB	See listing 6.1.

Table 5.3: Out-of-band commands supported by the front-end controller

Depending on the connected external services over the event-network, we can send external commands inside the `sendmaia` tag for triggering them. In our case, as our external modules are the Agent system and the Linked Open Data Service, we will use two commands that they can accept: `assert` and `retrieve`. Their description can be consulted in sections 6.3.5 and 6.3.1 respectively.

5.4 Chat client module

The chat client module consists on a JavaScript chat interface that communicates with the front-end controller by HTTP requests. This client module is designed to be included as a dropdown tab in a website, so the user can easily click on it and start the conversation with the personal assistant. This client, implemented in JavaScript, is the one used in our project. Nevertheless, other implementations for Android and the Google Chrome browser are also available[1].

The activity diagram of this module is shown in Fig. 5.7. When the user's query submitted, it is collected together with contextual information (such as the user's name, or the bot assisting in this website). These parameters are submitted in a request to the API of the front-end controller, in the server side. When the response arrives, the parameters are parsed and the answer is shown to the user. The parameters contained in the request and response are described in the previous section, in tables 5.1 and 5.2.

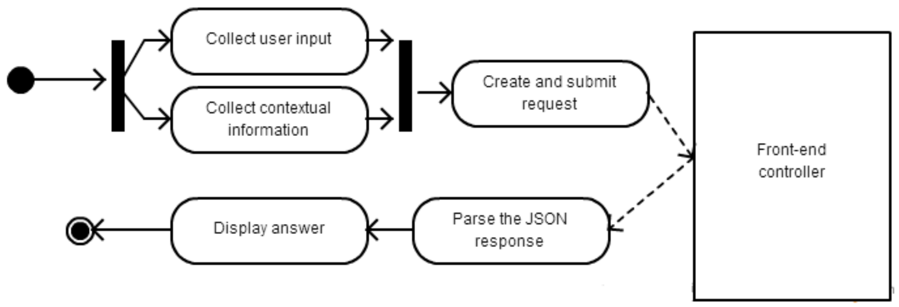


Figure 5.7: UML activity diagram of the client module.

5.5 Grammar engine module

This module is used for detecting specific patterns by applying grammars to the user query. These grammars are usually applied to match phrases containing specific concepts described a list (for example, 'do-while' as a concept of the Java programming language). This list of concepts may be updated dynamically during the conversation. Also, in case of detection, it produces an out-of-band command telling about it.

The technology used in this module is Unitex, presented in section 3.2.2. For specifying our list of concepts to detect, we will make use of its morphological dictionaries, described in the mentioned section 3.2.2. Once the grammars and dictionaries are ready, they can be

[1]`http://www.gsi.dit.upm.es/index.php/en/technology/software/224-gsibot.html`

applied to the user query as the activity diagram in Fig. 5.8 shows.

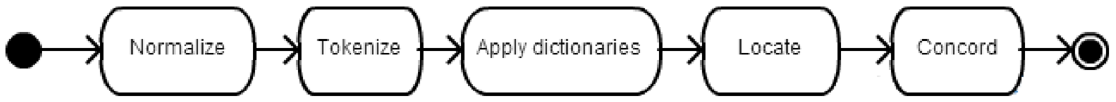


Figure 5.8: UML activity diagram of the Grammar engine module.

The activities mentioned in Fig. 5.8 are described next:

Normalize the user's query, identifying the text separators: *space*, *tab*, and *newline*. Each sequence that contains a *newline* separator is identified as an unique newline. The other two separators are replaced by a single space.

Tokenize chops the query into lexical units. For example, "what does iteration mean" is chopped into "what", "does", "iteration", "mean".

Apply dictionaries to the query. Here, the morphological dictionaries containing the lists of desired concepts to detect are applied. For example, the word "iteration" is detected as concept of our list.

Locate applies the grammars (designed as graphs) to the query. It saves the references to the found occurrences into an index file called *concord.ind*. This is an intermediate file that will be used by the next activity to generate the final output.

Concord using the *concord.ind* file, produces a concordance. The result of the application of this activity is a file containing the desired output, specified when designing the grammars. In our case, this output is an out-of-band tag that tells that the user is asking about the concept "iteration".

5.6 Chatbot module

ChatScript, presented in section 3.2.4, will be the technology used in this module. It offers us the key features that we need for our Chatbot module. A brief summary of these key features is:

- Understanding natural language and maintaining a conversation mapped into different topics.
- Simple writing of corpus, featuring responders, rejoinders, gambits, etc. (described in section 3.2.4.2).

- Using knowledge bases and the facts defined on them. In our case, this knowledge base will include information about concepts.
- Multiple user support, providing service to any number of users at the same time.
- Ability of running multiple bots running at the same time.

As interface for external communication, ChatScript only offers a TCP socket communication. The implementation of its TCP socket only allows to exchange messages containing three basic parameters, described in table 5.4. These parameters are the same for both incoming and outgoing messages.

Parameter	**Name**	**Description**
username	User's name	Name of the user conversating with the chatbot.
botname	Bot	Selected bot for the conversation.
message	Message	Message contained: either user's query or the chatbot's response.

Table 5.4: Message parameters in the TCP socket communication of ChatScript

It could be also possible to implement the protocols of our event-based network inside ChatScript. This could be done by writing a Maia client in C++ language for ChatScript, and after be integrated inside its source code. Otherwise, the high frequency of updates (bi-weekly) of ChatScript would imply a big amount of work implementing this client in each new version, so we have preferred to use its default TCP communication.

5.7 Event Network module

The Maia event-based network, described in section 3.4.2 will be used for this module. As seen there, the cornerstone of Maia are event messages. The basic parameters of any of these events are shown in table 5.5.

As nodes in Maia are loosely coupled, it allows higher flexibility than other networks. All the information is inside the message, including emitter and sender, all of it as strings contained in the Name of the event. This makes routing events as easy as parsing event names and applying a set of rules.

Parameter	Name	Description
name	Name	Describes the event. It consists on a complete namespace that allows complex processing of events and advanced filtering for the triggers.
sender	Sender ID	Contains a unique identifier of the sending entity. It is recommended to append the identifier of the bus it is connected to, if the scenario contains more than one.
id	Event ID	Contains a unique identifier of the event for the specified entity (sender).
timestamp	Timestamp	Points the time of the original emission. This makes time reasoning possible and prevents side effects of asynchronous communications.
payload	Payload	Contains either more information about the entities involved in the event, or the parameters in case the event is a request.
callback	Callback	Specifies a function to execute whenever there is data to return, or just to acknowledge the reception of the event.

Table 5.5: Message parameters in the TCP socket communication of ChatScript

5.8 Agent system module

Jason, the intelligent agent system presented in section 3.3, will be the technology used in this module. Using it, we can implement as many agents as we need. This way we can benefit from the advantages of having an intelligent multi-agent system, where each agent can offer a different kind of support. This also helps us building a proactive personal assistance system that is able to make its own propositions to the user. For example, in a learning platform, when the user accesses for the second time to the platform, the agent starts recommending to revise a concept. We will explain in detail the proactivity of our agent in chapter 6.

The common activity flow, regardless of the application field and the number of agents, is shown in the UML activity diagram of Fig. 5.9. The description of each of these activities is commented next.

Listen incoming assertions. Once our agent is started, it starts listening to incoming assertions received from the event network. When an assertion is received, it is auto-

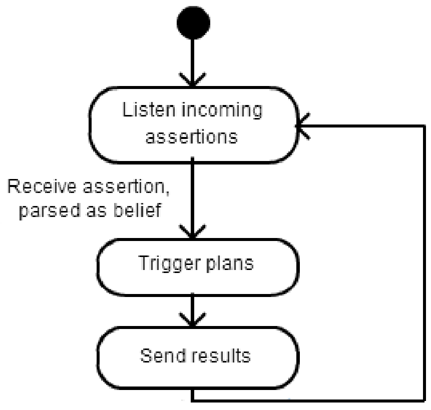

Listen incoming assertions
Receive assertion, parsed as belief
Trigger plans
Send results

Figure 5.9: UML activity diagram of the intelligent agent module

matically parsed and introduced as belief to the agent.

Trigger plans. Our agent implements several plans that are triggered when specific beliefs are added. This way, when a belief is added, the agent usually triggers a plan that produces a set of desired actions.

Send results. Once the actions of the plan are executed, the result is sent back over the event network.

Apart from these activities, it is important to recall a requirement established in section 4.3: the multiple user support, in order to offer the service to any number of users at the same time. To satisfy this requirement and not mixing different actions for different users, we will need to include the user's name both in beliefs and goals of our agents.

5.9 Linked Open Data Server module

SIREn, the entity retrieval engine presented in section 3.5.3, will be the technology used in this module. SIREn works only with semi-structured information, so the system administrator will need to have it first.

If the information is actually in a non-structured format (such as the information from many websites), we can fetch and convert it into a structured format by using scraping techniques. In section 3.5.2 we presented Scrappy, an application that is able to perform this task. As this is our case, due to the non-structured format of the data in the website with our desired information, we will show these steps in the Case Study chapter, in section 6.3.1.

Once we have our information in a semi-structured format, we can provide it to SIREn, which will index it and start offering its retrieval service. This process is shown in the activity diagram presented in Fig. 5.10.

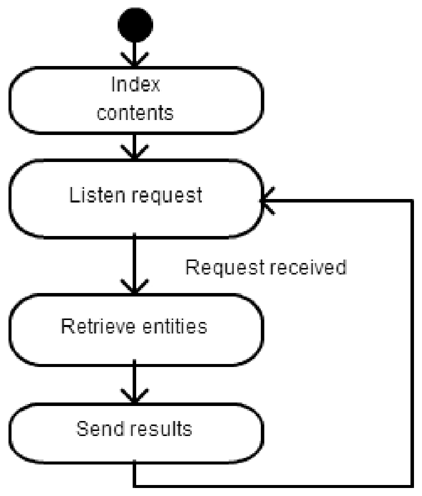


Figure 5.10: UML activity diagram of the Linked Open Data Server module

The description of each one of these activities is commented next:

Index contents When the module is launched, the contents (called documents) provided as semi-structured information are indexed as entities. This is done by the internal function `index()` and the class `SimpleIndexer` that iterates over each document adding them as entities to the index. As already commented in section 3.5.3, SIREn is built on top of Lucene, and that is why the `SimpleIndexer` class uses its indexing libraries for this task.

Listen request The module starts listening to incoming requests received from the event network.

Retrieve entities When a request is received, it performs an entity retrieval for the specified search parameters. These parameters can be either a single keyword to search over the whole documents, or a node:keyword pair to find only these documents where the specified node contains the specified keyword.

Send results The results found are sent back over the event network.

5.10 Conclusion

In this chapter, we have presented the architecture that has been designed for our system. We have used the UML developing language to formalize these descriptions, as well as providing an easy way to understand the different modules.

We have started presenting the global architecture, designed for meeting the requirements set in the previous development stage, gathered in section 4.3). Next, we have taken into account the technologies available nowadays, studied in chapter 3, to implement each one of its modules. This does not only allow to satisfy the requirements, but also to offer extra features that help improving the performance of our system.

It is still important to remark that we want to make our system as much modular as possible, so we need to be able to replace any of these previous modules. However, due to the very frequent release updates of ChatScript and the big amount of work implementing the Maia event network on it, we have decided to use its built-in TCP socket for communications with the Front-end controller. Nevertheless, this is a restriction that only ChatScript presents, and we also focus our desire of modularity on the modules for external actions. This is why the Maia event-based communication have been chosen, as it allows the connection and disconnection of new modules, offering the possibility to implement new actuators in our system.

Thanks to this architecture, we overcome one of the biggest limitations of the actual chatbot developments nowadays: they are built for a specific application field, and they do not offer an easy way to implement new features and actuators needed in other fields. Otherwise, our personal assistance system can be applied to a wide range of fields: its architecture includes modules that are useful in any application field, together with an easy way to connect new external services and actuators that allow to implement new features on it.

Chapter 6

Case Study

"Action is the foundational key to all success."

— Pablo Picasso

6.1 Introduction

In this chapter, we present a Case study that uses the architecture designed in Chapter 5 to integrate a personal assistant in an e-learning platform. First, we give a brief introduction to scenario and the proposed solution. After, we describe the implementation and configuration of each module of the architecture for being used in the learning platform. Finally, we will discuss the conclusions derived from the implementation of this Case study.

6.2 Scenario

The scenario for our Case study is based on a website that provides a description of all the concepts in the Java language. It is named Java Vademécum[1], and it is written by José A. Mañas, professor in the department of Telematics Engineering (DIT) in the Technical University of Madrid (UPM).

Our intention is to build a personal assistant that can be integrated in this website, as

[1] `https://www.dit.upm.es/~pepe/libros/vademecum/index.html`

shown in Figs. 6.1 and 6.2. This way, the students can quickly click on the personal assistant tab and get answers for their questions. This interaction is made by simply chatting, as the personal assistant is able to maintain a complete conversation with the student.

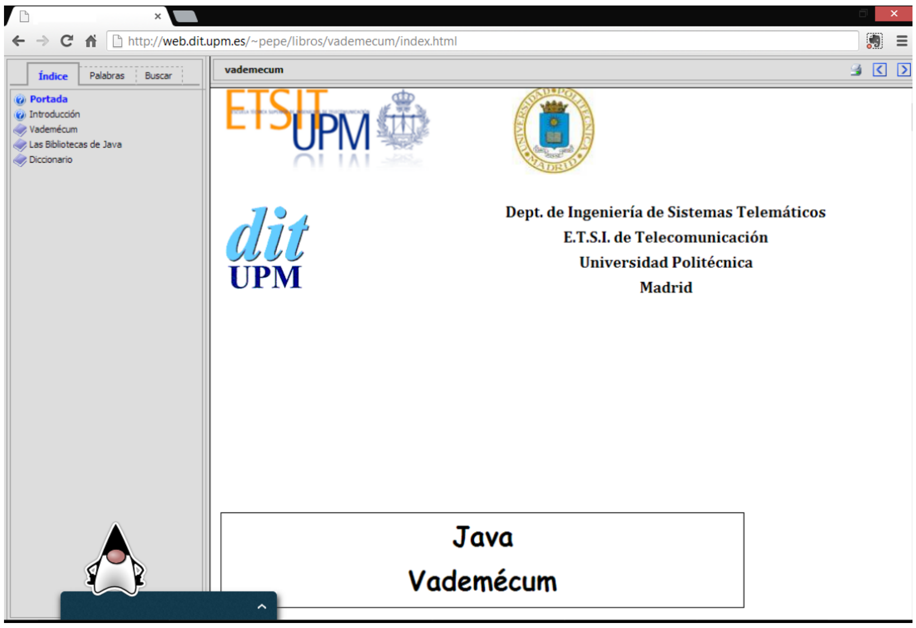
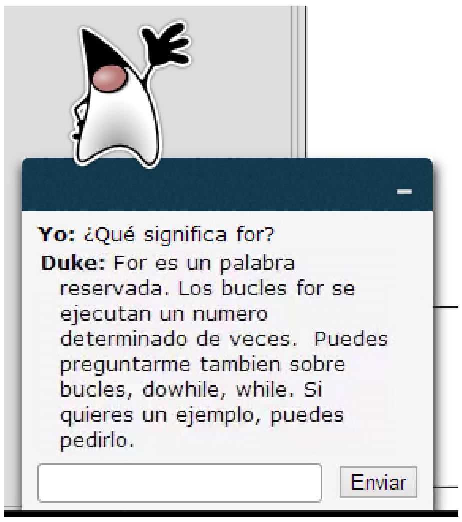


Figure 6.1: Integration of the personal assistance option in the Java learning website.



Yo: ¿Qué significa for?
Duke: For es un palabra reservada. Los bucles for se ejecutan un numero determinado de veces. Puedes preguntarme tambien sobre bucles, dowhile, while. Si quieres un ejemplo, puedes pedirlo.

Figure 6.2: Conversation between the student and the personal assistant.

In the next section, we will describe how we have achieved to successfully implement this personal assistance system. We will describe in detail the configuration of each module of its architecture, which was designed and presented in Chapter 5.

6.3 Proposed solution

Our project will start from the architecture designed in Chapter 5. In this chapter, we decided also the technology that we will use for each module. The next step will be implementing our Case Study using this set of technologies, as shown in Fig. 6.3.

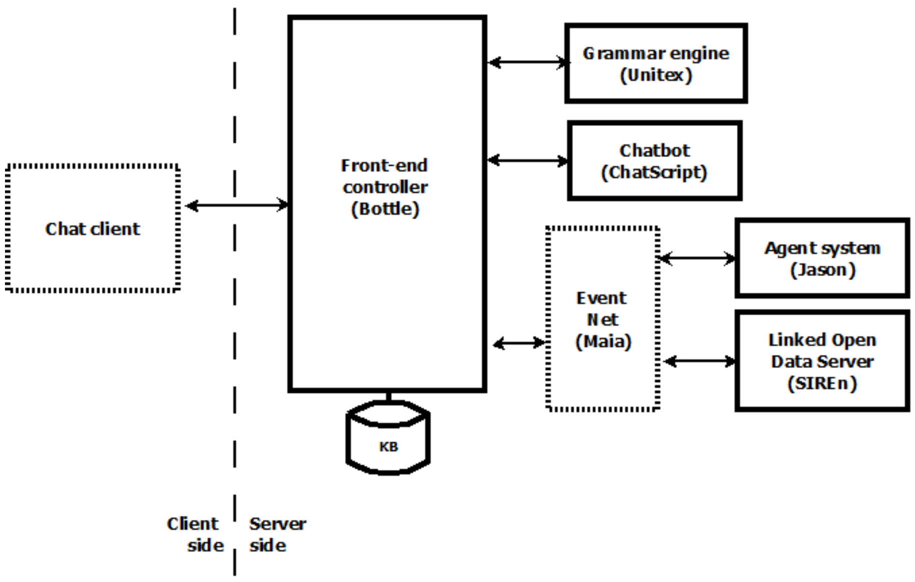


Figure 6.3: Architecture of the system for the e-learning Case Study

In the next sections, we will describe how we have used in our Case Study the modules marked in solid line in Fig. 6.3. This detailed description starts with the retrieval of the desired information, and after with the implementation of the modules in order to be able to work using this information:

- Collecting all the contents in the website by performing a scraping process that stores them into a structured-format. After, indexing them to be provided by the **Linked Open Data Service**, implemented using SIREn. This is described in section 6.3.1.
- Describing how the behavior of the **Front-end controller**, implemented using the Bottle framework, has been applied to this Case study. Specially, the Knowledge Base containing Java concepts. This is described in section 6.3.2.
- Writing a corpus for our **Chatbot**, implemented using ChatScript. It includes both patterns for maintaining a simple conversation with the student and out-of-band patterns for providing other desired features. This is described in section 6.3.3.
- Designing grammars to be used by the **Grammar engine**, implemented using Unitex.

For example, the grammars that detect if the student is asking about a concept of the Java language. This is described in section 6.3.4.

- Creating a *teacher* agent for our Jason **Agent system**, able to support the conversation and motivate the student to continue learning about the Java subject. This is described in section 6.3.5.

6.3.1 Information retrieval features implementation

SIREn, presented in section 3.5.3, is the technology that we will use to implement this module. As we commented in section 5.9, it works only with semi-structured information, so the system administrator will need to obtain it first. This is why we will present the whole information retrieval process next: scraping the non-structured contents in the website into the desired semi-structured format, and then indexing them for finally being offered by our Linked Open Data Service.

Information scraping

In our Case study, the information in the Java Vademécum website is presented with non-structured format. This is the case of most websites, where we will need to fetch all its contents and store them in a structured format. Using the scraping features of Scrappy, presented in section 3.5.2, we are able to perform this task, as we describe next.

First, we will identify the structure of the information in the website. The Java Vademécum is a collection of *documents*, each one describing a concept of the Java language. We can deduce a common structure for these documents:

- The name or **label** of the document. For example, *do-while* would be the label of the document about the do-while loops.
- The **type** in which the documents can be classified in:
  - **Concepts.** For example, *for* or *while.*
  - **Reserved words.** For example, *boolean.*
  - **Methods.** For example, *arraycopy.*
  - **Acronyms and vocabulary.** For example, *API* or *Console.*
- The **resource** URI that identifies each document.

- The **content** of the document, usually containing a description.
- An optional **example** of use.
- A set of documents that are **related** to this document.

Using Scrappy, we can collect all the documents of the Java Vademécum and store them in the RDF format (presented in section 3.5), producing the output that is shown in Appendix B. As we can see, these documents present relationships between them, and they are identified by an URI, so we can benefit of them by treating them as Linked Data, as presented also in section 3.5. For this, we will need to choose an ontology that allows us to define the relationships between documents.

For our project, we have chosen the SIOC[2] ontology. This ontology is mainly designed to link online community sites between them. Mainly, this is done by describing the documents published in communities, and finding connections between them. This ontology provides the concepts and properties required to describe information from not only message boards, but also wikis, weblogs, etc. In our case, we will use the four properties presented in table 6.1 from its `sioc:Post` class to establish the relationships between our documents.

Property	**Description**
`sioc:topic`	Type in which the documents can be classified in (concept, reserved word, method, acronym or vocabulary).
`sioc:uri`	Unique URI for identifying the document.
`sioc:content`	Content of the document.
`sioc:links_to`	Related documents.

Table 6.1: Parameters from sioc:Post used

Once we have identified the structure of the documents, and chosen the SIOC ontology for storing them preserving their relationships, we will proceed to start the scraping process over them.

To understand this process, we will first focus in the **Semantic scraping level** of Scrappy [40]. This level defines the model that maps HTML fragments to semantic web

[2]`http://sioc-project.org/ontology`

resources in RDF. Recall that HTML is a markup language for documents with a tree-structured data model, and the RDF data model is a collection of node triples for structured information.

Also, the **Syntactic scraping level** of Scrappy is used. It provides the required technologies to extract data from web resources. The scraping techniques make use of different Selectors that allow to identify HTML nodes. Their semantics are defined in the previously described Semantic scraping level, allowing to map data in HTML fragments to RDF resources.

We will specify the desired mapping between HTML fragments to RDF resources by writing a **scrapping script** for Scrappy. This script is serialized using the YARF format, and launching it will proceed fetching all the contents from the desired website and storing them in the desired semi-structured format.

Content indexing and Linked Open Data Service

Once we have our information in a semi-structured format, we can now provide it to our Linked Open Data Service module. This module is implemented using SIREn, presented in section 3.5.3. We describe next how SIREn indexes the information and then starts offering its retrieval service.

For the process of indexing our documents, SIREn already provides built-in functions. This indexing is done by the internal function `index()` and the `SimpleIndexer` class. This `SimpleIndexer` class is the one that iterates over each document, adding them as entities to the index. We also commented in section 3.5.3 that SIREn is built on top of Lucene, so the `SimpleIndexer` class makes use of the indexing libraries of Lucene for this task. It is also important to recall that the document indexing is triggered each time that SIREn is launched.

Once the documents are indexed, SIREn starts its service for the event-network, listening to incoming requests. To be accepted, a message has to be be an out-of-band tag with the `retrieve` command. For example, for retrieving entities about the *do-while* Java concept, an out-of-band tag `[retrieve do-while]` is sent to the Linked Open Data Service.

When a message like this one is received and accepted, SIREn performs the entity retrieval for the specified search parameters. These parameters can be either a single `keyword` to search over any field of the documents, or a `node:keyword` pair to find only these documents where the specified node contains the specified keyword.

Once the requested information is found, it is sent back over the event network together with an `updatekb` out-of-band command, as shown in the next listing. This information should be updated in the KB by the front-end controller, so it can be offered as answer to the user (described in section 5.2).

```
[updatekb
{
"label":"do-while",
"content":"Bucles que se ejecutan una o mas veces. La condicion de
    terminacion se chequea al final.",
"resource":"http://web.dit.upm.es/~pepe/libros/vademecum/topics/83.html",
"links_to":[{"label":"Bucles"},{"label":"while"},{"label":"for"}]
"type":"palabra reservada",
"example":"do { sentencia; } while (condicion); do { sentencia 1;
    sentencia 2; ... sentencia ...; } while (condicion);",
} ]
```

Listing 6.1: Answer for the do-while retrieval request

6.3.2 Front-end controller module implementation

In this section, we will describe the application of our front-end controller to the learning platform. Specially, the knowledge base containing Java concepts.

Knowledge Base

As we describe in section 6.3.1, the request of information retrieval sent to the Linked Open Data Service is replied with an `updatekb` tag containing the information found, as shown in listing 6.1.

The front-end controller will insert this information in a knowledge base. As our learning platform is about the Java language, it has been named *kb-java*. This knowledge base contains a set of triples describing each concept of the Java language. For example, when the information about the *do-while* concept is fetched, the front-end controller updates the knowledge base adding the following content to it:

```
( dowhile content Bucles_que_se_ejecutan_una_o_mas_veces.
   _La_condicion_de_terminacion_se_chequea_al_final. )
( dowhile resource http://web.dit.upm.es/~pepe/libros/vademecum/topics
   /83.html )
( dowhile links_to bucles )
( dowhile links_to while )
( dowhile links_to for )
( dowhile type palabra_reservada )
( dowhile example do_{_sentencia;_}_while_(condicion);_do_{_sentencia_1;
   _sentencia_2;_..._sentencia_...;_}_while_(condicion); )
```

Listing 6.2: Description of the do-while concept as triples for ChatScript

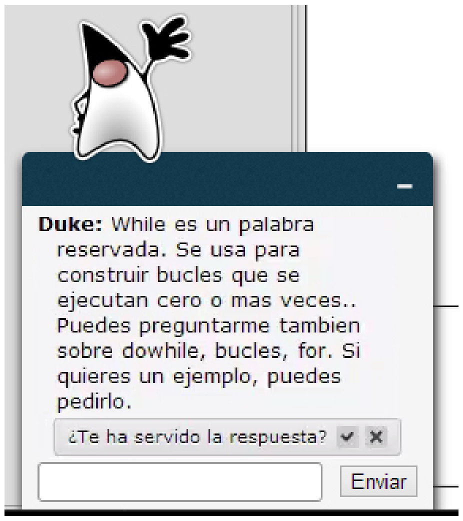
As it can be seen, the contents about the *do-while* concept are expressed in triples. As this knowledge base is imported by ChatScript, they are written in the format `(subject verb object)` that ChatScript uses [11]. In our case, the *subject* is the name of the concept, and the *verb* and *object* express different contents about this concept.

Following the activity diagram presented in section 5.2, we can see that the next step after updating the knowledge base is triggering an action in the Chatbot, which will create an answer for the user using this recently obtained information.

Handling user feedback

Improving our personal assistance system is also important. By collecting the user's feedback, as shown in Fig. 6.4, we make this possible. In our case, we will collect feedback from the students in order to know the quality of the answers given to their questions.

This request can be triggered by appending a `fullresponse` flag to an output. This means that the reply contains a fully responded answer for a user's question. This flag is detected by the Front-end controller, producing the request for the user's feedback that is shown in 6.4. The flags to show this request in the user interface, and to inform of the given feedback back to the Front-end controller, were both presented in section 5.3.



Duke: While es un palabra reservada. Se usa para construir bucles que se ejecutan cero o mas veces.. Puedes preguntarme tambien sobre dowhile, bucles, for. Si quieres un ejemplo, puedes pedirlo.

Figure 6.4: Asking the user for feedback

6.3.3 Chatbot module implementation

In this section, we will describe the implementation of our chatbot using ChatScript, presented in section 3.2.4. As we commented, thanks to ChatScript we are able to create a conversational bot that is able to maintain a natural language conversation with the user. For describing its behavior, we need to write the corpus of the bot, which is mainly composed by a set of patterns that will be matched depending on the input that it receives. We will present next the corpus that our chatbot works with, both for maintaining simple conversation with the student and giving answers for his questions about the Java subject.

Simple conversation patterns

The main conversational corpus of a bot in ChatScript is usually found in the *simpletopic.top* file. In section 3.2.4 we presented the wide range of features that ChatScript gives to us, such as wildcards or concepts, so recommend to revise this section (or the ChatScript documentation) to understand its corpus scripts. As we already described in section 3.2.5.4 the development of an example corpus that includes greetings and goodbyes, we will focus now in the patterns that are specifically related to the e-learning field.

The corpus of our bot will include two simple conversation patterns applied to the e-learning field. The source code of this corpus can be downloaded from the project repository[3], and it is presented next:

[3]`https://github.com/gsi-upm/calista-bot`

```
concept: ~teach [learn help teach]
topic: ~elearning (professor ~teach)

#! Are you our teacher?
u: (are you * teacher)
    ^noerase() ^repeat()
  No; I am a virtual assistant to help you finding answers for your
      questions.

#! What can you help me with?
u: (what * ~teach)
    ^noerase() ^repeat()
  I can help you with your questions about the Java programming language.
```

Listing 6.3: Excerpt of simple conversation corpus patterns in ChatScript

Here, we have defined:

- One concept `~teach`, which encapsulates the words *learn*, *help* and *teach*.
- One topic `elearning`, that is accessed when the input contains *professor* or any of the words encapsulated by the concept concept `~teach`.
- A pattern `(are you * teacher)` which matches any input asking if the chatbot is the teacher of the subject. For example "Are your our teacher?".
- A pattern `(what * ~teach)` which matches any input containing the words *learn*, *help* and *teach*. For example, "What can you help me with?".

In addition, we implement an additional set of ChatScript out-of-band patterns in the *simplecontrol.top* control script. They are used to access the knowledge base of concepts, which was described in section 6.3.2, and other interactions with the user, such as giving answer to his questions.

Recall that the user questions about concepts (for example, "What does *do-while* mean?") are detected in the Grammar engine module (see Fig. 5.3 in section 5.2). An out-of-band command `[sendcs (java question dowhile)]` is produced. This command, when received back to the Front-end controller, makes it send a message to ChatScript containing the input `[java question dowhile]`.

This input is matched in ChatScript with one of the out-of-band patterns in the control script. This script can also be downloaded from the project repository[4]. An excerpt of it is presented in the next listing:

```
#Answer to user question, if concept found in the KB
#! [java question dowhile]
u: (\[ java question _* \] ) #E.g. "dowhile" is saved in _0
   ^noerase() ^repeat()
   ^query(direct_sv _0 type ?) #Get the "type" field of the concept
   _0 is an @0object  #E.g. Print "Do-while is a reserved word."
   ^query(direct_sv _0 content ?) #Get the "content" field with the
       description
   . @0object. #Print the description of the concept
   ^query(direct_sv _0 links_to ?) #Get the "links_to" fields with the
       related concepts
   . You can also ask me about
   loop() { ^pick(@0object), } #Loop printing all the related concepts
    . \[sendmaia assert explained(concept _0) \] #OOB command to make an
        assertion. E.g., explained(concept dowhile)

#Retrieve information about concept, if not found in the KB
#! [java question dowhile]
u: (\[ java question _* \] )
    ^noerase() ^repeat()
  \[sendmaia retrieve _0\] #OOB command to retrieve information about the
       concept
```

Listing 6.4: Out-of-band patterns in ChatScript for answering user's questions

As we can see, there are two possible patterns for the input `[java question dowhile]`. Both of them are used in the activity diagram in section 5.2 and described next:

- If the chatbot does not find any information about *do-while* in the knowledge base, the

[4] `https://github.com/gsi-upm/calista-bot`

first pattern fails and the second one is matched. It produces an out-of-band command `[sendmaia retrieve dowhile]` that, once received back in the Front-end controller, will be sent over the event network and trigger the external module Linked Open Data Service.

- If the chatbot finds existing information about *do-while* in the knowledge base (for example, if the knowledge base was recently updated with information about it), the first pattern is matched. It will present the answer in natural language for the user, as seen in Fig. 6.2. Also, it will produce an out-of-band command telling to assert that *do-while* was explained: `[sendmaia assert explained(concept dowhile)]`. The triggered actions when sending this assertion are described in section 6.3.5.

All the out-of-band ChatScript patterns that we have implemented to our chatbot make use of the same functions that in the previous case, so they have been summarized in table 6.2. For making them easy to read, they are presented as examples using *kb-java* as KB and *do-while* as Java concept.

OOB pattern	**Action/output**
`[import kb-java dowhile]`	Refresh *kb-java* in the chatbot, for explaining *do-while* after.
`[java question dowhile]`	NL answer with information about *do-while*, or OOB command to retrieve it.
`[java offer example]`	NL answer "If you want an example, you can ask me for it." .
`[java sample dowhile]`	NL answer with an example about *do-while*, if existing.
`[java recommend random]`	NL answer recommending to revise a random concept of the KB.
`[java recommend dowhile]`	NL answer recommending to revise *do-while*.
`[java test dowhile]`	NL test about *do-while*.
`[java affirm dowhile]`	OOB command asserting *do-while* as user's answer for the test.
`[java testok dowhile]`	NL answer: "Correct. The answer is *do-while*".
`[java testfail dowhile]`	NL answer: "Incorrect. The answer was *do-while*".

Table 6.2: Out-of-band patterns implemented to the ChatScript bot

6.3.4 Grammar engine module implementation

In this section, we will describe the configuration of our Grammar engine, which is implemented using Unitex, presented in section 3.2.2. We will present the designed grammars and write a dictionary of Java concepts for detecting, for example, if the student is asking about a concept of the Java language.

As we already described in section 3.2.2 how grammars in Unitex are designed, we will focus on the main one of our system, used to detect questions about Java concepts. It is presented in Fig. 6.5.

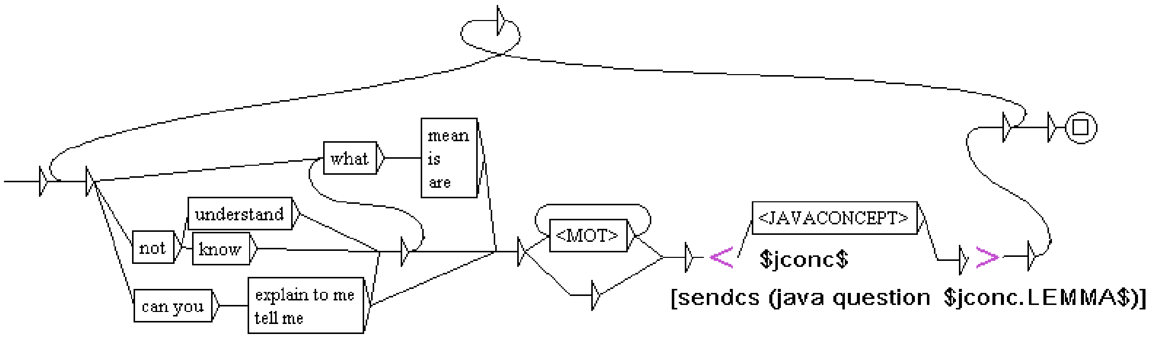


Figure 6.5: Unitex grammar used to detect user questions

This graph matches all the phrases containing the possible combinations of words specified in its branches, and ending by a word expressing a Java concept (as specified in the JAVACONCEPT morphological dictionary). The main benefit from these dictionaries is that they can be dynamically updated, so the system is able to detect new concepts on the go. The dictionary of Java concepts that we use follows the same structure as the one presented in section 3.2.2:

```
for,for.JAVACONCEPT
for loop,for.JAVACONCEPT
while,while.JAVACONCEPT
while loop,while.JAVACONCEPT
while structure,while.JAVACONCEPT
do-while,do-while.JAVACONCEPT
do-while loop,do-while.JAVACONCEPT
```

Listing 6.5: Java concepts dictionary in Unitex

For example, using the grammar of Fig. 6.5 and the morphological dictionary described in listing 6.5, the user's input "Can you explain to me what is *do-while*?" would be detected as a question about the Java concept. This will produce the output out-of-band command `[sendcs (java question do-while)]` that will be sent to ChatScript, whose behavior can be consulted in section 6.3.3.

Apart from this case, the grammars designed for our Case study are able to detect patterns affirming a concept as answer for a test (for example, "The answer is *do-while*" produces `[sendcs (java affirm dowhile)]`), or asking for examples about a concept (for example, "Can you give me an example of *do-while*?" produces `[sendcs (java sample dowhile)]`). Both of them follow the same structure as the grammar in Fig. 3.3, but producing `sendcs` out-of-band commands that are sent to ChatScript.

6.3.5 Agent system module implementation

In this section, we will describe the implementation for our Agent system module. This module is implemented using Jason, presented in section 3.3. As we commented, thanks to this module, we can benefit from the advantages of intelligent agents for supporting the conversation with the students and support their learning process. It also helps us building a proactive personal assistance system that, for example, recommends to revise a concept automatically each time that the student returns to the platform.

In our Case Study, the Agent System contains one *teacher* agent that is able to provide this set of basic functionalities to support the learning process (offer examples, perform tests, and recommend to revise random concepts each time the student comes back to the platform). These features could be expanded in future works as mentioned in Chapter 7, both by improving the number of plans of our teacher agent or by implementing a higher number of agents that communicate with each other.

The overview of our *teacher* agent is presented in Fig. 6.6. It is important to remark first that our agent needs to support **multiple users** using the system at the same time, so we will make use of *annotations* to provide this functionality. For example, in our case we make assertions like "the concept *do-while* has been explained", by creating a belief `explained(do-while)`. When adding multi-user support, this assertion may come from different users, so it is necessary to detail from which user it came. For this, in the assertion `explained(do-while)`, we can specify the user's name by appending it as an annotation to the belief. The complete belief, if it the *do-while* concept was explained for a the user *John99* would be `explained(do-while)[user(John99)]`. Thanks to these annotations containing the user's name, it is also possible triggering different plans for different users at the same time.

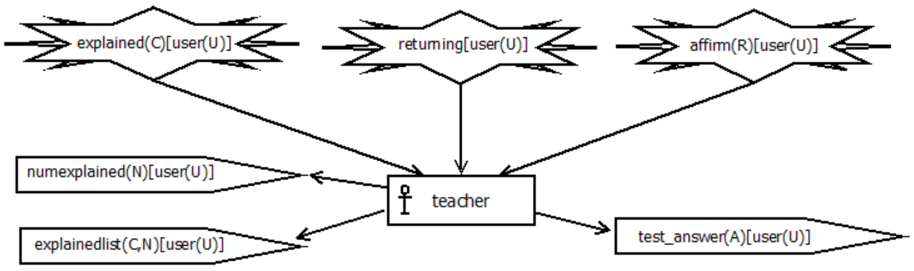


Figure 6.6: Overview of the teacher agent.

The possible perceptions of the agent are described next. These perceptions will trigger the plans of the agent, producing the desired actions:

- **explained(*concept*)[user(*username*)]** is received when a concept has been explained to the student. If it one of the first explanations given, the agent will offer him further help if asking for examples. For the third and next explanations, we suppose that the student already knows this, so the agent will now send tests about the previously explained concepts.
- **returning[user(*username*)]** is received when the user is using again the system. Each time the user returns to use the system, the agent offers him to revise a random concept.
- **affirm(*concept*)[user(*username*)]** is triggered when the user affirms an answer for the test that was sent to him before. The answer will be checked and the student will be informed if it is correct or not.

To make this possible, the beliefs presented next are used:

- **numexplained(*N*)[user(*username*)]** holds the number of concepts that have been explained to the user. Starting from zero, it is increased each time that a concept is explained to the user.
- **explainedlist(*concept*,*N*)[user(*username*)]** holds the history of the concepts explained to the user. For example, if the first concept to be explained was *while*, and the second one was *do-while*, two beliefs `explainedlist(while,0)[user(U)]` and `explainedlist(do-while,1)[user(U)]` will be created.

- **test_answer(*concept*)[user(*username*)]** holds the correct answer for a test that has been sent to the user. This way, when the user provides a response, it will be checked with this correct answer.

In Listing 6.6, we present an excerpt of code with the initial goals and the main plans that our *teacher* agent uses. Recall that our *teacher* agent has been designed as a proof of concept, so its behavior is a very simplified version of the features that an Agent system could actually offer in future works. The complete script of our *teacher* agent can be downloaded from the project repository[5].

```
/* Initial goals */
!launch.

/* Plans */

+!launch
  <- maia.start. //Start the Maia connection for listening incoming
      messages

//When a first or second concept are explained, reminds the student that
    it is possible to ask for examples
+explained(C)[user(U)]
  : numexplained(N)[user(U)] & N<=1
  <- -+numexplained(N+1)[user(U)]; //Update number of explained concepts
    +explainedlist(C,N)[user(U)]; //Note explained (C)oncept
    maia.send("[sendcs (java offer example)] [user ",U,"]"). //Offer
        example to the user

//When the concept is explained, note it and send to the user a test for
    previously explained concepts
+explained(C)[user(U)]
  : numexplained(N)[user(U)] & N>1
  <- -+numexplained(N+1)[user(U)]; //Update number of explained concepts
    +explainedlist(C,N)[user(U)]; //Note explained (C)oncept
    ?explainedlist(A,N-2)[user(U)]; //Get another (A)lready explained
        concept
```

[5]https://github.com/gsi-upm/calista-bot

```
    -+test_answer(A)[user(U)]; //Correct answer for the next test is the
        Already explained concept
    maia.send("[sendcs (java test ",A,")] [user ",U,"]"). //Send test to
        the user, asking about the already explained concept

//Check if the user (R)esponse is the correct (A)nswer for the test
    previously sent (case of correct response), and inform
+affirm(R)[user(U)]
  : test_answer(A)[user(U)] & R=A
  <-  maia.send("[sendcs (java testok ",A,")] [user ",U,"]"). //Tell to
      user that the given response is correct.

//Check if the user (R)esponse is the correct (A)nswer for the test
    previously sent (case of incorrect response), and inform
+affirm(R)[user(U)]
  : test_answer(A)[user(U)] & R\==A
  <-  maia.send("[sendcs (java testfail ",A,")] [user ",U,"]"). //Tell to
       user that the given response is incorrect.

//Recommends a random concept if the user is returning (not the first
    time using the system)
+returning[user(U)]
  <-  maia.send("[sendcs (java recommend random)] [user ",U,"]"). //
      Recommend random concept to the user
```

Listing 6.6: Main goals and plans of the teacher agent in Jason

Finally, as it can be seen, the plans are only triggered by adding one of the previously mentioned perceptions to the agent's knowledge base. These perceptions are automatically added by the `maia.start` internal action when an assertion is received from the event network.

6.4 System evaluation

In this section, we make an evaluation of the system implemented in our Case Study. We will use this evaluation to show the benefits from including a Grammar Engine, a Linked Open Data Server and an Agent system. We will remark all the features obtained from them and the possibilities that they offer, opposite to only using only ChatScript or other Chatbot technology alone.

Grammar Engine module evaluation

The Grammar engine of our system has successfully demonstrated to be able to detect specific patterns by the application of grammars to the user query. These grammars match phrases containing specific concepts described a list (for example, *do-while* is detected as a concept of the Java programming language). This list of concepts can be updated dynamically during the conversation. Unitex is the technology used in this module, and it was presented in section 3.2.2. In this section we also explained how to write a list of concepts to detect, making use of the morphological dictionaries of Unitex.

In case of detection of a special pattern as commented above, the Grammar engine module produces an out-of-band command telling about it. Otherwise, if any detection is produced, it does not modify the user's query. In the next listing, we present an actual log containing a successful detection about the *do-while* concept and the corresponding output produced.

```
22:33:36,823 INFO User query: ¿Eres el profesor?

22:33:36,828 INFO Unitex input: ¿Eres el profesor?

22:33:36,888 INFO Unitex output: ¿Eres el profesor?

22:33:36,888 INFO ChatScript input: ¿Eres el profesor?

22:33:36,900 INFO ChatScript output: No; soy un bot asistente para ayudarte
 a encontrar respuestas.

22:33:36,900 INFO Response for the user: No; soy un bot asistente para
ayudarte a encontrar respuestas.
```

```
22:33:58,503 INFO User query: ¿Con qué me puedes ayudar?

22:33:58,503 INFO Unitex input: ¿Con qué me puedes ayudar?

22:33:58,572 INFO Unitex output: ¿Con qué me puedes ayudar?

22:33:58,573 INFO ChatScript input: ¿Con qué me puedes ayudar?

22:33:58,585 INFO ChatScript output: Puedo ayudarte con tus preguntas sobre
 Java.

22:33:58,585 INFO Response for the user: Puedo ayudarte con tus preguntas
sobre Java.

22:34:24,823 INFO User query: No entiendo qué significa do-while.

22:34:24,823 INFO Unitex input: No entiendo qué significa do-while.

22:34:24,890 INFO Unitex output: [sendcs (java question do-while)]

22:34:25,001 INFO ChatScript output: Dowhile es un palabra reservada.
 Bucles que se ejecutan una o mas veces. La condicion de terminacion se
 chequea al final. Puedes preguntarme también sobre bucles, for, while.

22:34:25,003 INFO Response for the user: Dowhile es un palabra reservada.
 Bucles que se ejecutan una o mas veces . La condicion de terminacion se
 chequea al final. Puedes preguntarme también sobre bucles, for, while.
```

The benefit from implementing a Grammar engine in our system is clear: as the dictionary of concepts can be updated dynamically, our system can add new words to this dictionary if new information from external sources is retrieved.

In the opposite side, if we think of using ChatScript alone without a grammar engine, it actually allows also writing dictionaries for pattern detection. These dictionaries, which can be found in the WORLDDATA folder, group words inside ChatScript concepts (for example, *for*, *while* or *do-while* can be grouped into a concept *~ javaconcept*). However, these dictionaries are read only once: when ChatScript is launched. This means that, each time we need to update them with new words, we would need to restart ChatScript to reload them.

If this is not a problem because our dictionaries do not need to be updated dynamically, we may deactivate and stop using the Unitex module. We would then need to write a dictionary of Java concepts to be loaded in ChatScript. This way, ChatScript can make the different pattern detections about Java concepts by himself:

```
12:53:15,325 INFO User query: No entiendo qué significa do-while.

12:53:15,328 INFO ChatScript input: No entiendo qué significa do-while.

12:53:15,921 INFO ChatScript output: Dowhile es un palabra reservada.
 Bucles que se ejecutan una o mas veces. La condicion de terminacion se
 chequea al final. Puedes preguntarme también sobre bucles, for, while.

12:53:15,923 INFO Response for the user: Dowhile es un palabra reservada.
 Bucles que se ejecutan una o mas veces . La condicion de terminacion se
 chequea al final. Puedes preguntarme también sobre bucles, for, while.
```

Linked Open Data Server module evaluation

The Linked Open Data Server module of our system contains a large knowledge base of information, ready to be offered over its service. The engine that handles it, SIREn, was presented in section 3.5.3. Our large set of data is not a problem for the performance of the system: the contents are indexed by SIREn, and can be consulted with a very high speed rate [42] when the user sends a query.

Next, we present a successful query to the Linked Open Data Service for retrieving information about the *do-while* concept. When the information is retrieved by this external service, it is received over the Maia network and updated in the knowledge base. A response for the user is then generated by the Chatbot:

```
 22:44:42,613 INFO User query: ¿Qué son los bucles do-while?
```

```
22:44:42,618 INFO Unitex input: ¿Qué son los bucles do-while?

22:44:42,671 INFO Unitex output: [sendcs (java question do-while)]

22:44:42,671 INFO ChatScript input: [sendcs (java question do-while)]

22:44:42,683 INFO ChatScript output: [sendmaia retrieve do-while ]

22:44:42,683 INFO Sending message to the Maia network: [retrieve do-while]

22:44:45,687 INFO Sending to Maia {"name":"message","data": {"name" : "[re
trieve do-while ] [user anonymous]"}}

22:44:46,691 INFO Received and accepted from Maia: [updatekb]{"label":"do-
while","topic":"palabra reservada","resource":"http://web.dit.upm.es/~pepe
libros/vademecum/topics/  83.html","content":"Bucles que se ejecutan una o
mas veces. La condicion de terminacion se chequea al final.","example":"do
 { sentencia; } while (condicion); do { sentencia 1; sentencia 2; ... sent
 encia ...; } while (condicion);","links_to":[{"label":"Bucles"},{"label":
 "while"},{"label":"for"}]}

22:44:46,693 INFO Updating ChatScript knowledge base kb-java

22:44:46,697 INFO ChatScript input: [sendcs (import carpetafacts/kb-java
dowhile)]

22:44:46,709 INFO ChatScript output: [sendcs (java question dowhile ) ]

22:44:46,710 INFO ChatScript input: [sendcs (java question dowhile ) ]

22:44:46,726 INFO ChatScript output: Dowhile es un palabra reservada.
Bucles que se ejecutan una o mas veces. La condicion de terminacion se
chequea al final. Puedes preguntarme también sobre bucles, for, while.

22:44:46,726 INFO Response for the user: Dowhile es un palabra reservada.
Bucles que se ejecutan una o mas veces . La condicion de terminacion se
chequea al final. Puedes preguntarme también sobre bucles, for, while.
```

On the other hand, we could think of using ChatScript alone without any external data service. ChatScript allows importing information from a knowledge base stored into fact triples, as the one shown in Listing 6.2. The main drawback of not using an external service is that every fact in the knowledge base needs to be loaded each time the Chatbot is launched. This, together with the need of exploring the whole large knowledge base of facts for each user's query, may drastically decrease the speed our system.

In case this is not a problem, and we prefer using ChatScript alone without an external data service, the actions produced would be the next ones:

```
22:34:24,823 INFO User query: No entiendo qué significa for.

22:34:24,823 INFO Unitex input: No entiendo qué significa for.

22:34:24,890 INFO Unitex output: [sendcs (java question for)]

22:34:24,891 INFO ChatScript input: [sendcs (java question for)]

22:34:24,904 INFO ChatScript output: For es un palabra reservada. Los bucle
for se ejecutan un numero determinado de veces. Puedes  preguntarme también
 sobre bucles, while, dowhile.

22:34:24,904 INFO Response for the user: For es un palabra reservada. Los
bucle for se ejecutan un numero determinado de veces. Puedes  preguntarme
también sobre bucles, while, dowhile.
```

Agent system module evaluation

The Agent system module receives assertions collected during the conversation and provides them to the different agents running on it. This provides our personal assistance system an extra interaction with the user, by triggering plans that produce replies that are sent to the user.

Jason, the intelligent agent system presented in section 3.3, was the technology used in this module. Using it, we can implement as many agents as we need. This way, we can benefit from the advantages of having an intelligent multi-agent system, where each agent can offer a different kind of support. This also helps us building a proactive personal assistance system that is able to make its own propositions to the user.

For example, in our learning platform, if a concept was explained previously to the user, the Agent system will send tests to revise this concept afterwards:

```
17:53:58,548 INFO User query: ¿Qué es iteracion?

17:53:58,598 INFO Unitex input: ¿Qué es iteracion?

17:53:58,654 INFO Unitex output: [sendcs (java question iteracion)]

17:53:58,654 INFO ChatScript input: [sendcs (java question iteracion)]

17:53:58,667 INFO ChatScript output: Iteracion es un concepto. Aplicar una
funcion repetidamente. Puedes preguntarme también sobre for, while, bucles.
[sendmaia assert explained (concept iteracion ) ]

17:53:58,667 INFO Sending message to the Maia network: [assert explained
 (concept iteracion ) ]

17:54:01,671 INFO Sending to Maia {"name":"message","data": {"name" :
 "[assert explained (concept iteracion ) ] [user anonymous]"}}

17:54:02,674 INFO Received and accepted from Maia: [sendcs (java test
 bucles)]

17:54:02,674 INFO ChatScript input: [sendcs (java test bucles)]

17:54:02,690 INFO ChatScript output: Para repasar, seguro que puedes
 decirme el concepto al que se refiere este ejemplo: "while (condicion)
  sentencia; while (condicion) { sentencia 1; sentencia 2; ... sentencia
   ...; }"

17:54:02,692 INFO Response for the user: Para repasar, seguro que puedes
  decirme el concepto al que se refiere este ejemplo: "while (condicion)
   sentencia; while (condicion) { sentencia 1; sentencia 2; ... sentencia
```

```
    ...; }"

17:54:17,538 INFO User query: Se trata de bucles while.

17:54:17,578 INFO Unitex input: Se trata de bucles while.

17:54:17,660 INFO Unitex output: [sendcs (java affirm while)]

17:54:17,661 INFO ChatScript input: [sendcs (java affirm while)]

17:54:17,671 INFO ChatScript output: [sendmaia assert affirm (concept
 while ) ]

17:54:17,673 INFO Sending message to the Maia network: [assert affirm
 (concept while ) ]

17:54:20,677 INFO Sending to Maia {"name":"message","data": {"name" :
 "[assert affirm (concept while ) ] [user anonymous]"}}

17:54:21,688 INFO Received and accepted from Maia: [sendcs (java testfail
 bucles)]

17:54:21,690 INFO ChatScript input: [sendcs (java testfail bucles)]

17:54:21,707 INFO ChatScript output: Incorrecto. Se trata de bucles.

17:54:21,709 INFO Response for the user: Incorrecto. Se trata de bucles.
```

As it can be seen, features like performing tests to the user to revise previously explained concepts may be impossible to implement using ChatScript alone. The further intelligence provided by the Agent system is crucial for task like this one, as it is necessary to maintain a list of the previously explained concepts and generate a pair question-answer that is checked afterwards.

6.5 Conclusion

In this Case Study, we have successfully implemented a personal assistance system in an e-learning platform. We have described the implementation and configuration of each module of the designed architecture, so our personal assistant has the ability to help students with their questions about the subject and support the learning process.

First, we have presented the scenario where we integrate our personal assistant: a Java learning website which contains a large set of documents describing the different concepts, methods, reserved words, acronyms or vocabulary of the Java programming language. Our personal assistant should be able to use this information to formulate answers for the user's question.

The first step for this has been scraping the information provided in the website into a semi-structured format for our Linked Open Data Service technology, SIREn. The scraping task can be easily done using Scrappy, and then it will be SIREn who indexes these contents and offer them.

Next, we have described how the behavior of the Front-end controller has been implemented. Specially, the knowledge base that holds information about the concepts, and finally how the user's feedback is requested and accepted.

The corpus of our Chatbot will implement a set of simple conversation patterns for greetings, goodbyes and questions about the system. Also, a set of out-of-band patterns for offering the desired features: answering the user's questions, offering examples, performing tests, etc. The detection of user's inputs mentioning concepts, otherwise, will be made by the Grammar engine module, as it can hold a dictionary of concepts that can be dynamically updated.

A simple intelligent agent has been designed for our Case Study. Once it is launched, it starts listening to incoming assertions from the event network. These assertions are added as beliefs to the agent's knowledge base, triggering a plan that usually produces an output to be sent to the user: offer an example, perform a test, or recommending to revise a random concept.

In conclusion, we can see that, nevertheless, many improvements can be done to this Case study implementation if it was integrated in a learning platform. From writing larger corpus for the chatbot, to making use of the Jason multi-agent system to benefit from many more advantages of this technology. We will present all these possible future works in the next chapter.

Chapter 7

Conclusion and future work

“Learn from yesterday, live for today, hope for tomorrow. The important thing is not to stop questioning.“

— Albert Einstein

7.1 Conclusions

In this chapter, we comment the achieved goals and we expose the conclusions deduced after implementing this project. Finally, we present some different possible lines of investigation oriented to improve the system developed within this project.

In this project, we have developed a personal assistance system. We started specifying several requirements, then designing an architecture to match them all, and finally presenting the implementation of a case study based on online learning. It has been successfully demonstrated that there are enough tools nowadays to build a personal assistance system. We can make good use of these technologies if we understand all the components that compose them, and how they are communicated between them. For example, as the Jason agent system allows the inclusion of Java libraries, we have been able to include an event network client on it for its communication with the system. This just one example in the wide range of features that we can implement to our system, such as the information retrieval modules that we also included to it.

Thanks to the natural language modules that we have included in our system, we can offer a simple user interface that meets the needs of low experienced users. This way, our project can be applied to any subject in the e-learning field and any kind of user. Nevertheless, we can observe that, even if the maturation process of natural language understanding has reached a good level, it still needs hard work. Personal assistance systems based on natural language still have a long way to go to be hundred-percent functional, being able to handle any kind of conversation topic and executing external actuators on user's demand.

Finally, it has been successfully demonstrated that the integration of our personal assistant in an online learning platform provides an added value to it, saving both time to the student waiting for an answer, and effort to the teacher writing an answer and motivating the student. As our personal assistance system is offered as a Web service, it is also possible to access it from any device, such as desktop computers, smartphones, tablets or even embedded systems, opening even more its range of possible applications.

7.2 Achieved goals

After defining a set of goals to achieve, the designed architecture and all of its modules have been correctly implemented. Natural language understanding, communication between modules, use of intelligent agents and information retrieval features work successfully together.

In was in the Requirement analysis chapter where we set several goals to achieve within this project, and then, in the next chapter, we designed the Architecture of solution, according to them. The final outcomes once the Case study has been implemented demonstrate that these goals have been successfully achieved:

- To develop a system that is able to understand natural language, using a cutting-edge technology as ChatScript, which offers a simple way to write a bot corpus and handling a knowledge base. Also, apart from simple conversation, to offer help by answering the user's questions about a subject depending on the application field.
- Achieving a heterogeneous architecture with interchangeable end-points, especially when providing a common network for external services that allows modularity. Being able to execute actions in these external services, in order to improve the reply for the user or to execute actuators on user's demand.
- Possibility of scraping contents from unstructured knowledge bases on the Internet, storing them in a semi-structured format. Then, indexing and offering them using a Linked Open Data Service.

- Maintaining a knowledge base that holds information to answer the user's questions. Possibility of quickly access this knowledge base to fetch an answer for the user.
- Proactive interaction with the user, by using intelligent agents. Not only giving response to his queries, but also creating self-made propositions.
- Multiple user support, in order to offer the service to any number of users at the same time.
- To enable the possibility of getting assistance from different devices, by using a client-server architecture that offers service from the Internet thanks to the compliance with HTTP protocols of its REST interface.

7.3 Future work

Once our system has been implemented, we can see that continuing our labor and adding more features can be very promising. In any research line, there is always enough room to improve and continue the work already done. Here we present both possible improvements to the implemented modules, and possible research lines that this project offers:

- Natural language understanding: our project includes a small number of patterns of conversation. As it can be seen, the effectiveness of the Chatbot in the conversation depends on the precision of its corpus. One starting point for improving our personal assistant would be adding more patterns for conversation, which would provide it a more human lookalike personality. In this line, we can also focus future works on using the pos-parsing and self-reflection features that ChatScript offers for improving the reply for the user.
- Using SPARQL queries by the agent system to explore an ontology describing, for example, the contents of a subject in a learning platform. In the case of learning about the Java language, navigating inside the tree of the Java Learning Object Ontology (JLOO) would allow us to propose related concepts much more precisely. Also, using a higher number of agents to provide a larger set of functionalities.
- Increasing the efficiency of the entity retrieval module by scraping a larger quantity of contents from the Web. Having more contents, it could become also necessary to improve the result filtering, in order to get a unique best answer for the user.
- Security. The explained architecture is only a practical first approach to the solution, but it is still important to apply security policies and authentication. This is crucial

if the information shared between nodes is confidential or not all the parties can be trusted. The concept of our event-based network, Maia, supports adding simple solutions for authentication and limiting scopes.

- Enhancing the graphical interface of the chat used in the client side. For example, improving its displaying capabilities, such as showing formatted HTML, graphics or other smart objects.
- Finally, it could be also useful the development of an administration panel that allows easily to add new contents to the Linked Data Open Service, new patterns to the Chatbot, or making any possible configuration.

Appendix A

Installation manual

In this manual, we describe the installation of the different components of the system the same way that is has been done in this project. We will show the installation steps of the different modules that system in this project uses, as well as their dependencies.

A.1 Project repository

It is possible to get to the latest version of the project in `https://github.com/gsi-upm/calista-bot`. In this repository, all the different components of the project can be found. In Listing A.1 it is shown the folder tree of the project.

```
calista-bot
  Chat client
  ChatScript
  FE Controller
  Agent system
  SIREn
  Unitex
```

Listing A.1: Folder tree of the project

A.2 Java Development Kit (JDK) installation

In our project, we have used Java JDK 7u45. It includes all the libraries and dependencies that Jason and SIREn need to compile a project once modified. We can previously check if JDK is already installed by using the following command, either on Windows or Linux:

```
java -version
```

On Windows, in case we need to install JDK for the first time, we can download it from the next URL and follow the installation instructions.

`http://www.oracle.com/technetwork/es/java/javase/downloads/index.html`

On Linux, we can install JDK with the following command:

```
sudo apt-get install sun-java6-bin, sun-java6-jre, sun- java6-jdk
```

It is recommended to have installed the latest version of Java. In case we need to update it, we can run the last command but using the option `--reinstall`.

A.3 Python installation

In our project, we have used Python 2.7.5 for running out Front-end Controller, which is implemented with the Bottle Python framework. We can check our version of Python or if it is already installed in our system, both on Windows on Linux, running this command:

```
python --version
```

In order to install Python on Windows, we can download it from `http://www.python.org/download` and follow the installation instructions.

On Linux, it can be installed using a package manager such as APT:

```
apt-get install python
```

A.4 Node.js installation

Node.js v0.10.21 is used by the Maia event-based network of this project. To install Node.js, it is possible either to compile the sources or use the binaries available in the PPA repository.

We will install it using the PPA repository as it's easier and will keep our installation updated.

First of all, we add the PPA repository:

```
sudo apt-get install python-software-properties
sudo add-apt-repository ppa:chris-lea/node.js
sudo apt-get update
```

Now, we install the Node.js package, and NPM (nodejs package manager) so we can install node modules easily later:

```
sudo apt-get install nodejs npm
```

To test the node.js installation, it is possible to open a new file called `"helloworld.js"` with the following code:

```
#!/bin/env node
var http = require('http');

http.createServer(function (req, res) {
  res.writeHead(200, {'Content-Type': 'text/plain'});
  res.end('Hello World\n');
}).listen(1337, "127.0.0.1");

console.log('Server running at http://127.0.0.1:1337/');
```

And run it with:

```
node helloworld.js
```

Now, it is possible to access clicking the link in the terminal, or opening the web browser and going to `http://127.0.0.1:1337/`.

A.5 Jason installation

Jason 1.3 has been used for the Agent system module of our project. Before installing it, the Java JDK has to be previously installed, as shown in section A.2. This version of Jason can be downloaded from `http://sourceforge.net/projects/jason/files/`. It can be run on Windows by opening the executable file that this folder contains.

On Linux, this is done running the ".sh" file inside the /bin/ folder:

```
sh jason.sh
```

A.6 Front-end controller deployment

For implementing our Front-end controller module, we have used Bottle 0.11.2. It can be found together with the script for our Front-end controller in the `/calista-bot/FE Controller/` folder of the project repository.

Once we have Python installed, as shown in section A.3, we can launch the Front-end controller module both on Windows or Linux by running the following command inside the module folder:

```
python talkbot.py
```

A.7 Chatbot module deployment

For our Chatbot module, we made use of ChatScript 2.0. It can be found together with the corpus of the chatbot of our project in the `/calista-bot/ChatScript/` folder of the project repository.

To launch the service ChatScript on Windows, the following command should be run inside the ChatScript folder:

```
chatscript
```

On Linux, the following command should be run inside the ChatScript folder:

```
./LinuxChatScript32
```

We may also locally run ChatScript by adding "local" as option of the running command. This will show a chat interface in the console instead of launching the ChatScript service.

A.8 Agent system module deployment

The scripts of the agent system in our project can be found in the `/calista-bot/Agent system/` folder of the project repository.

Our Agent system is implemented using Jason 1.3. The steps of installation of Jason were described in section A.2. After installing and opening the the Jason environment, we can launch the Agent system module by opening the `/Agent system/jason_elearning.mas2j` file of our project.

A.9 Linked Open Data Server deployment

Our Linked Open Data Server makes use of the 1.0 commercial version of SIREn. This commercial version can be requested to Sindice[1]. Also, a development version of SIREn can be found in `https://github.com/rdelbru/SIREn/`.

The first using SIREn, we need to compile the source code. We can do it by running:

```
mvn clean package assembly:single
```

Once the project is compiled, we should be able run any of its sample classes:

```
java -cp ./target/siren-jar-with-dependencies.jar org.sindice.siren.demo.
bnb.BNBDemo
```

[1] `http://siren.sindice.com/`

Appendix B

Results of the scraping process

In this appendix we will present the results of using Scrappy, the scraping technology presented in section 3.5.2, in our Case Study. We were able to collect the non-structured documents presented in the Java Vademécum and store them as structured information. More specifically, we have stored the data into the RDF format, which was presented in section 3.5.

The scraping process that we followed for obtaining this information was described in the section 6.3.1. We present an excerpt of these results in the next Listing B.1, which contains the RDF structured format of three documents of the Java Vademécum: *Bucles*, *do-while* and *iteracion*.

```
<sioc:Post rdf:about="http://web.dit.upm.es/~pepe/libros/vademecum/index.
    html">
    <dc:title>Bucles</dc:title>
    <sioc:content>Fragmentos de codigo que se ejecutan repetidamente.</
        sioc:content>
    <sioc:uri rdf:resource="http://web.dit.upm.es/~pepe/libros/vademecum/
        topics/34.html" />
    <sioc:topic rdfs:label="concepto" rdf:resource="http://web.dit.upm.es
        /~pepe/libros/vademecum/topics/concepto" />
    <sioc:links_to dc:title="while" rdf:resource="http://web.dit.upm.es/~
```

```
        pepe/libros/vademecum/topics/35.html"/>
    <sioc:links_to dc:title="for" rdf:resource="http://web.dit.upm.es/~
        pepe/libros/vademecum/topics/37.html"/>
    <sioc:links_to dc:title="do-while" rdf:resource="http://web.dit.upm.es
        /~pepe/libros/vademecum/topics/83.html"/>
</sioc>

<sioc:Post rdf:about="http://web.dit.upm.es/~pepe/libros/vademecum/index.
    html">
    <dc:title>do-while</dc:title>
    <sioc:content>Bucles que se ejecutan una o mas veces. La condicion de
        terminacion se chequea al final.</sioc:content>
    <sioc:uri rdf:resource="http://web.dit.upm.es/~pepe/libros/vademecum/
        topics/83.html" />
    <sioc:topic rdfs:label="palabra reservada" rdf:resource="http://web.
        dit.upm.es/~pepe/libros/vademecum/topics/palabra_reservada" />
    <sioc:links_to dc:title="Bucles" rdf:resource="http://web.dit.upm.es/~
        pepe/libros/vademecum/topics/34.html"/>
    <sioc:links_to dc:title="while" rdf:resource="http://web.dit.upm.es/~
        pepe/libros/vademecum/topics/35.html"/>
    <sioc:links_to dc:title="for" rdf:resource="http://web.dit.upm.es/~
        pepe/libros/vademecum/topics/37.html"/>
</sioc>

<sioc:Post rdf:about="http://web.dit.upm.es/~pepe/libros/vademecum/index.
    html">
    <dc:title>iteracion</dc:title>
    <sioc:content>Aplicar una funcion repetidamente.</sioc:content>
    <sioc:uri rdf:resource="http://web.dit.upm.es/~pepe/libros/vademecum/
        topics/141.html" />
    <sioc:topic rdfs:label="concepto" rdf:resource="http://web.dit.upm.es
        /~pepe/libros/vademecum/topics/concepto" />
    <sioc:links_to dc:title="while" rdf:resource="http://web.dit.upm.es/~
        pepe/libros/vademecum/topics/35.html"/>
    <sioc:links_to dc:title="Bucles" rdf:resource="http://web.dit.upm.es/~
        pepe/libros/vademecum/topics/34.html"/>
    <sioc:links_to dc:title="do-while" rdf:resource="http://web.dit.upm.es
       /~pepe/libros/vademecum/topics/83.html"/>
</sioc>
```

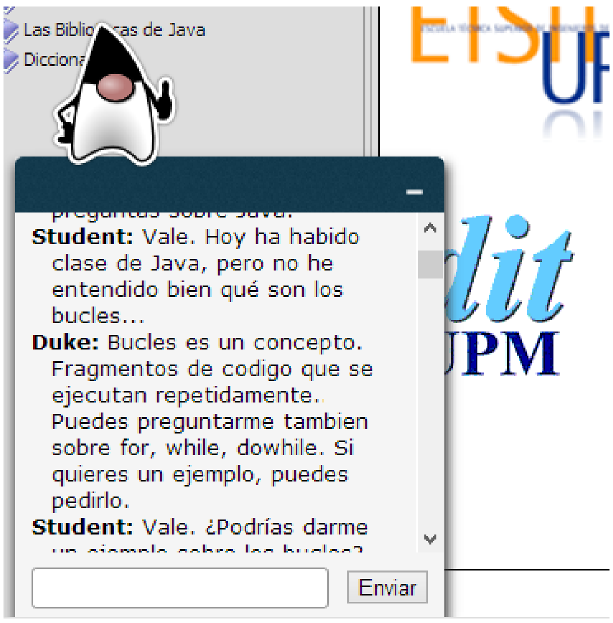
Listing B.1: RDF data obtained in the scraping process

As we can see, these documents present relationships between them, and they are identified by an URI, so we benefit from them by treating them as Linked Data, as presented also in section 3.5. For this, they have been stored using the SIOC ontology, as described in section 6.3.1.

Appendix C

Corpus for testing

In this appendix, we will show the results of testing the personal assistance system implemented for our Case Study. Here, the set of features implemented using the different modules of the system are tested, producing successfully the expected responses.



Student: Vale. Hoy ha habido clase de Java, pero no he entendido bien qué son los bucles...
Duke: Bucles es un concepto. Fragmentos de codigo que se ejecutan repetidamente. Puedes preguntarme tambien sobre for, while, dowhile. Si quieres un ejemplo, puedes pedirlo.
Student: Vale. ¿Podrías darme

Figure C.1: Screenshot of the conversation for testing the system.

The corpus for testing is composed by a set of inputs that the student will send to the personal assistant, including both simple conversation and questions about the subject. These inputs (the *Student* queries) and the corresponding produced outputs (the replies from

Duke, the personal assistant) are presented next. As it follows the flow of a natural language conversation, we will present it the same way as it has been produced. This conversation has been made by using the client chat interface implemented in the platform, as shown in Fig. C.1.

Student: Hola.
Duke: Buenas.
Student: ¿Eres el profesor?
Duke: No; soy un bot asistente para ayudarte a encontrar respuestas.
Student: Ah, perfecto. ¿De qué forma me puedes ayudar?
Duke: Puedo ayudarte con tus preguntas sobre Java.
Student: Vale. Hoy ha habido clase de Java, pero no he entendido bien qué son los bucles...
Duke: Bucles es un concepto. Fragmentos de codigo que se ejecutan repetidamente. Puedes preguntarme tambien sobre for, while, dowhile. Si quieres un ejemplo, puedes pedirlo.
Student: Vale. ¿Podrías darme un ejemplo sobre los bucles?
Duke: "while (condicion) sentencia; while (condicion) { sentencia 1; sentencia 2; ... sentencia ...; }"
Student: Y en este ejemplo, ¿qué significa while?
Duke: While es un palabra reservada. Se usa para construir bucles que se ejecutan cero o mas veces.. Puedes preguntarme tambien sobre iteracion, for, dowhile. Te propongo repasar antes sobre iteracion.
Student: Vale, ¿qué significa iteración?
Duke: Iteracion es un concepto. Aplicar una funcion repetidamente. Puedes preguntarme tambien sobre for, bucles, while. Para repasar, seguro que puedes decirme el concepto al que se refiere este ejemplo: "while (condicion) sentencia; while (condicion) { sentencia 1; sentencia 2; ... sentencia ...; }"
Student: Gracias. Esa pregunta trata de bucles, ¿no?.
Duke: Exacto. Se trata de bucles.
Student: ¿Puedes darme un ejemplo sobre while en concreto?
Duke: "while (n > 0) { fact+=n; n–; }"
Student: ¡Gracias! ¿Puedes explicarme ahora qué son las estructuras for que has mencionado antes?
Duke: For es un palabra reservada. Los bucles for se ejecutan un numero determinado de veces. Puedes preguntarme tambien sobre bucles, while, dowhile. Si quieres un ejemplo, puedes pedirlo.
Student: ¿Puedes darme un ejemplo de uso de for?
Duke: "for (inicializacion; condicion; actualizacion) accion; for (inicializacion; condicion;

actualizacion) { accion 1; accion 2; ... accion ...; }"
Student: Genial. ¿Y qué quiere decir el do-while que comentabas?
Duke: Dowhile es un palabra reservada. Bucles que se ejecutan una o mas veces . La condicion de terminacion se chequea al final. Puedes preguntarme tambien sobre bucles, while, for. Para repasar, seguro que puedes decirme el concepto al que se refiere este ejemplo: "for (inicializacion; condicion; actualizacion) accion; for (inicializacion; condicion; actualizacion) { accion 1; accion 2; ... accion ...; }"
Student: La respuesta a tu pregunta es la estructura for, ¿no?
Duke: Exacto. Se trata de for.
Student: ¡Bien! ¿Puedes darme un ejemplo de do-while para terminar?
Duke: "do { sentencia; } while (condicion); do { sentencia 1; sentencia 2; ... sentencia ...; } while (condicion);"
Student: Estupendo, creo que con eso ya sé suficiente para la tarea de hoy. Gracias, nos vemos.
Duke: Nos vemos.

Supposing that the student accesses the platform again a different day, the assistant will remember it and will recommend to revise a random concept to start the conversation, as shown next:

Student: Buenas.
Duke: Hola de nuevo. Hoy te propongo repasar while.
Student: Perfecto; precisamente tengo algunas preguntas relacionadas con eso. ¿Puedes explicarme de nuevo para empezar los bucles while?
Duke: While es un palabra reservada. Se usa para construir bucles que se ejecutan cero o mas veces. Puedes preguntarme tambien sobre dowhile, bucles, for. Si quieres un ejemplo, puedes pedirlo.
Student: ¿Qué ejemplo sabías sobre el uso de while?
Duke: "while (n > 0) { fact+=n; n–; }"
Student: Ah sí, ahora recuerdo todo. Hoy quería preguntarte qué significa break, ya que he visto que lo usan en algunos bucles while.
Duke: Break es un palabra reservada. Se emplea para forzar la terminacion de un bucle. Es util en bucles cuya condicion de terminacion no se puede chequear comodamente ni al principio (bucles while) ni al final (bucles do-while).". Puedes preguntarme tambien sobre continue. Si quieres un ejemplo, puedes pedirlo.
Student: ¿Puedes darme un ejemplo donde se use break?

Duke: "for (i=0;i=10;i++) { String linea = entrada.readLine(); if (linea == null) break; // se hace algo con la linea leida}"
Student: Vale, así que se puede usar también con las estructuras for... ¿Y qué significa el continue que me has propuesto antes?
Duke: Continue es un palabra reservada. Se emplea para forzar la terminacion de una pasada en un bucle . Es util cuando queremos abortar limpiamente una ejecucion del bucle ; pero que este siga ejecutandose.. Puedes preguntarme tambien sobre break. Para repasar, seguro que puedes decirme el concepto al que se refiere este ejemplo: "while (n > 0) { fact+=n; n–; }"
Student: Por supuesto. Se trata de la estructura while.
Duke: Exacto. Se trata de while.
Student: Vale. Esas eran todas mis dudas. Gracias y adiós.
Duke: Adiós.

Index

Bibliography

[1] Burguillo J. C. Mikic Fonte, F. A. and M. L. Nistal. An intelligent tutoring module controlled by bdi agents for an e-learning platform. *Expert Systems with Applications.*, 2012.

[2] H. Shimazu. Expertclerk: A conversational case-based reasoning tool for developing salesclerk agents in e-commerce webshops. *Artificial Intelligence Review, 18(3), 223-244*, 2002.

[3] Vassallo G. Gaglio S. Augello, A. and G. Pilato. A semantic layer on semi-structured data sources for intuitive chatbots. *Complex, Intelligent and Software Intensive Systems, 2009. CISIS'09. International Conference*, 2009.

[4] Ijaz K. Bogdanovych, A. and S. Simoff. The city of uruk: Teaching ancient history in a virtual world. *Intelligent Virtual Agents*, 2012.

[5] Yongjuan Zhanga, Tao Chenb, Wenjuan Youc, Jianrong Yud, Jilin Sune, and Heng Chenf. A new efficient semantic web platform based on the solr, siren and rdf.

[6] Eric Miller. The wbc's semantic web activity: an update. *Intelligent Systems, IEEE*, 19(3):95–97, 2004.

[7] Pattie Maes. Modeling adaptive autonomous agents. In *Artificial Life*. Citeseer, 1994.

[8] Loebner Prize. http://en.wikipedia.org/wiki/loebnerprize. *Wikipedia*, 2012.

[9] M. Matheas. Façade: An experiment in building a fully-realized interactive drama. *http://classes.soe.ucsc.edu/cmps148/Spring08/MateasSternGDC03.pdf*, 2012.

[10] B. Wilcox. Beyond façade: Pattern matching for natural language applications. *Own journal*, 2012.

[11] Community. Chatscript documentation. *http://chatscript.sourceforge.net/*, 2013.

[12] S. Wilcox. Suzette, the most human computer. *Own journal*, 2012.

[13] M. Kimura and Y. Kitamura. Embodied conversational agent based on semantic web. *Agent Computing and Multi-Agent Systems*, 2006.

[14] Community. Program w website. *http://programw.sourceforge.net*, 2012.

[15] Ruskov M. Sasse M. A. Seager, W. and M. Oliveira. Eliciting and modelling expertise for serious games in project management. *Entertainment Computing*, 2011.

[16] T. Bosse and S. Stam. A normative agent system to prevent cyberbullying. *Web Intelligence and Intelligent Agent Technology*, 2011.

[17] Katia Sycara, Anandeep Pannu, M Willamson, Dajun Zeng, and Keith Decker. Distributed intelligent agents. *IEEE expert*, 11(6):36–46, 1996.

[18] J. Fernando Sanchez-Rada and Miguel Coronado. Design and Implementation of an Agent Architecture Based on Web Hooks. Master's thesis, Universidad Politecnica de Madrid, Septiembre 2012.

[19] Liren Chen and Katia Sycara. Webmate: a personal agent for browsing and searching. In *Proceedings of the second international conference on Autonomous agents*, pages 132–139. ACM, 1998.

[20] Ângelo Costa, Paulo Novais, Ricardo Costa, Juan M Corchado, and José Neves. Multi-agent personal memory assistant. In *Trends in practical applications of agents and multiagent systems*, pages 97–104. Springer, 2010.

[21] Augello A. Pilato, G. and S. Gaglio. Modular knowledge representation in advisor agents for situation awareness. *Journal of Semantic Computing*, 2011.

[22] Community. Aiml explanation via implementation description: Graphmaster. *http://www.alicebot.org/TR/2011/section-explanation-via-implementation-description-graphmaster*, 2012.

[23] Community. Callmom. *http://callmom.pandorabots.com*, 2012.

[24] K. Coursey. Living in cyn: mating aiml and cyc together with program n. *http://www. daxtron. com*, 2004.

[25] Vassallo G. Gaglio S.] Augello, A. and G. Pilato. A semantic layer on semi-structured data sources for intuitive chatbots. *Complex, Intelligent and Software Intensive Systems, 2009. CISIS'09*, 2009.

[26] Community. Pandorabots. *http://pandorabots.com/botmaster/en/home*, 2012.

[27] Community. The foundation aiml alice. *http://code.google.com/p/aiml-en-us-foundation-alice/*, 2012.

[28] Community. Jeannie bot in google play store. *https://market.android.com/*, 2012.

[29] Community. Aiml 2.0 draft news. *http://alicebot.blogspot.com/2013/01/aiml-20-draft-specification-released.html*, 2012.

[30] Community. Bottle: Python web framework. *http://bottlepy.org*, 2012.

[31] ChatScript forum thread. Putting commands in output text. *http://www.chatbots.org/ai_ zone/viewthread/1182/*, 2012.

[32] ChatScript community forum. Forum 44. *http://www.chatbots.org/ai_zone/viewforum/44/*, 2012.

[33] Community. Thay airways grows online bookings by 200e-merchandise. *http://www.amadeus.com/sg/x136578.html*, 2012.

[34] W. Wong. Flexible conversation management using a bdi agent approach. *Internet Journal*, 2012.

[35] Burguillo J. C. Llamas M. Rodríguez D. A. Mikic, F. A. and E. Rodríguez. Charlie: An aiml-based chatterbot which works as an interface among ines and humans. *EAEEIE Annual Conference, 2009*, 2009.

[36] Hübner J. F. Bordini, R. H. and Wooldridge. Programming multi-agent systems in. agentspeak using jason. *John Wiley and Sons Ltd.*, 2007.

[37] Frank Manola, Eric Miller, and Brian McBride. Rdf primer. *W3C recommendation*, 10:1–107, 2004.

[38] Magnus Stuhr. Filt-filtering indexed lucene triples. 2012.

[39] Christian Bizer, Tom Heath, and Tim Berners-Lee. Linked data-the story so far. *International Journal on Semantic Web and Information Systems (IJSWIS)*, 5(3):1–22, 2009.

[40] 3rd International Conference on Agents and Artificial Intelligence (ICAART 2011). *A Semantic Scraping Model for Web Resources - Applying Linked Data to Web Page Screen Scraping*, Roma, January 2011. SciTePress.

[41] Peter Mika. Distributed indexing for semantic search. In *Proceedings of the 3rd International Semantic Search Workshop*, page 3. ACM, 2010.

[42] Eyal Oren, Renaud Delbru, Michele Catasta, Richard Cyganiak, Holger Stenzhorn, and Giovanni Tummarello. Sindice. com: a document-oriented lookup index for open linked data. *International Journal of Metadata, Semantics and Ontologies*, 3(1):37–52, 2008.

[43] Renaud Delbru, Nickolai Toupikov, Michele Catasta, and Giovanni Tummarello. A node indexing scheme for web entity retrieval. In *The Semantic Web: Research and Applications*, pages 240–256. Springer, 2010.