Development of a Framework for AIML Chatbots in HTML5 and Javascript

by

Filippo Malvisi

LIU-IDAILITH-EX-A-14/021-SE

2014-04-24
Development of a Framework for AIML Chatbots in HTML5 and Javascript
Final thesis

Development of a Framework for AIML Chatbots in HTML5 and Javascript

by

Filippo Malvisi

LIU-IDA/LITH-EX-A--14/021--SE

2014-04-24

Supervisor: Annika Silvervarg
Examiner: Erik Berglund
Abstract

Chatbots are software agents that interact with the user in a conversation. The main goal of their creation was to resemble a human being in the way they perform said interaction, trying to make the user think he/she is writing to another human being. This has been implemented with varying degrees of success. One of the most popular languages for the definition of a chatbot knowledge base is AIML.

This thesis focuses on the implementation of an AIML interpreter written in Javascript to allow for a web-based client-side specific usage of AIML chatbots. The interpreter must guarantee the compliance of properly formed AIML documents, perform all the necessary pre-processing duties for the correct usage of the chatbot and ensure the correctness of both pattern matching of user input and chatbot response.

The interpreter fully exploits the DOM tree manipulation functions of the jQuery library to achieve said goals, treating AIML files as if they were normal XML files. The result is a well performing, fully functional AIML interpreter tailored around AIML 1.0 specification.
# Table of Contents

Chapter 1 Introduction ........................................................................................................ 11

Chapter 2 Chatbots and AIML .......................................................................................... 13
  2.1 Basics of Chatbots .................................................................................................... 13
  2.2 Artificial Intelligence Markup Language (AIML) .................................................... 14
    2.2.1 Basic Structure ................................................................................................. 15
    2.2.2 AIML Chatbots for pedagogical usage ............................................................ 17

Chapter 3 Design ............................................................................................................... 19
  3.1 Requirements Analysis ............................................................................................ 19
  3.2 Main Design Choice: Interpreter Architecture ....................................................... 20
    3.2.1 Possibility to use external XML parser/framework ............................................ 21
    3.2.2 Javascript, its frameworks and to-Javascript converters at-a-glance .................. 21
    3.2.3 Final Choice: jQuery ....................................................................................... 23
  3.3 Other Design Choices ............................................................................................. 24

Chapter 4 Implementation of AIML.js and results ......................................................... 25
  4.1 Use Cases ................................................................................................................ 25
  4.2 AIML.js Overview .................................................................................................. 25
    4.2.1 Variable Initialization ....................................................................................... 26
    4.2.2 Configuration Files and Loading a Chatbot .................................................... 27
    4.2.3 AIML File Parser ............................................................................................ 29
    4.2.4 Input Processing and Pattern Matching .......................................................... 30
    4.2.5 Output Processing ........................................................................................... 34
  4.3 Challenges Faced during Implementation ................................................................ 34
  4.4 A Look at the Interface ........................................................................................... 35

Chapter 5 Testing and Evaluation .................................................................................... 37
  5.1 Correctness .............................................................................................................. 37
  5.2 Perceived Performance ......................................................................................... 37
  5.3 Notes on Compatibility ......................................................................................... 41

Chapter 6 Conclusions ...................................................................................................... 43
  6.1 Future Improvements ............................................................................................. 43

References .......................................................................................................................... 45

Appendix A: Interaction Example ..................................................................................... 47
# Table of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>AIML.js Use Case 1</td>
<td>26</td>
</tr>
<tr>
<td>2</td>
<td>AIML.js Use Case 2</td>
<td>26</td>
</tr>
<tr>
<td>3</td>
<td>AIML.js Demo</td>
<td>36</td>
</tr>
<tr>
<td>4</td>
<td>DevTools overall Flamechart</td>
<td>38</td>
</tr>
<tr>
<td>5</td>
<td>DevTools AJAX Flamechart</td>
<td>39</td>
</tr>
<tr>
<td>6</td>
<td>DevTools i/o Flamechart</td>
<td>40</td>
</tr>
</tbody>
</table>
Preface

This dissertation is an original intellectual product of the author, F. Malvisi. It is part of the final thesis project at Linköping’s University for the Master’s programme of Computer Systems. The thesis project was offered and supervised by Annika Silvervarg, of the Department of Computer and Information Science (IDA).

The main reason that sparked my interest into this project was its web-based nature combined with a context of artificial intelligence for an educational purpose. Web-based development is something I have honed my skill and experience about during the entirety of my academic studies, including my bachelor thesis which was also centered on the usage of HTML5 and Javascript ([1]). On the other hand, artificial intelligence is something that curriculum-wise was completely foreign to me, albeit fascinating; that is why I decided that this would be an excellent thesis opportunity: improving skills in my favorite field of studies, while learning and getting to understand a completely new subject which can still be a very useful addition to my curriculum.

Linköping, April 2014

Filippo Malvisi
Acknowledgements

There are many people that deserve to be thanked for the fulfillment of this thesis. I would like to start with my examiner, Erik Berglund, and my supervisor, Annika Silvervarg, who have always been available and keen to explain and guide me through this process. Bureaucracy, project requirements and scope can seem intimidating and overwhelming at the beginning of a project; however, such burdens can feel a lot lighter when working with competent, kind and dependable people.

I would also like to express gratitude to my fellow course colleagues, especially Matic and Andrei, with whom I started this journey two years ago and shared various laboratories and projects. Two minds working on the same problem usually deliver very interesting and varied solutions, and you guys have surely been the most proactive lab-partners I have had in my experience at Linköping’s University.

Gratitude is also very much due to my roommates, current and former. I would especially like to thank Michael, who decided to start an international collective in a Rydsvägen apartment. His decision made my search for accommodation a lot easier (in an especially busy year), and his help with getting information about everything Swedish has been invaluable ever since.

Appreciation and gratitude also go to my parents, Carla and Roberto, whose support has been invaluable. From the very basic economic type of support, which was essential for a non-working student like me, to the moral support that only loving parents can give. It is very hard for Italian parents with an only child to let go of their “baby”, especially if they will only be able to see him once or twice a year. I am thankful that regardless of the hardship, I was still supported in my decision to move to Sweden.

Last but certainly not least, I have one more person to whom I will always be grateful. Without whom I would have never moved to Linköping, nor would I have known all the wonders of Sweden and the Swedish people, I would likely not have continued my studies, let alone venture so far outside my comfort zone and live on my own in a foreign country (and much more...). If you ever happen to read this, there is so much I would want to tell you, but here I will just say: “Thank you for everything”.

Linköping, April 2014
Chapter 1

Introduction

The Educational Technology Group at the universities of Linköping and Lund develops educational software with the purposes of exploiting them as instruments to explore learning processes and coming up with pedagogical software with a real-world value as pedagogical tools\(^1\). In their project hWorld their intent is implementing educational software for the subject history for children aged 10-12 years old. The educational and learning experience in this system is partly achieved with the interaction of the end-users (said children) with chatbots, which are software agents with an artificial intelligence that allows them to understand the user input and provide a meaningful response according to pre-compiled knowledge. The chatbots can be pedagogical agents or personifications of historical figures who will be able to talk about their life and work. The chatbots are developed with the Artificial Intelligence Markup Language (AIML\(^2\)) since the group has previous experience with that.

In previous applications the developers opted for existing AIML interpreters in languages such as Java and Python (respectively ProgramD\(^3\) and PyAIML\(^4\)). The AIML interpreter is a critical part of the framework as it needs to process operations at real-time based on user input and its output speed and correctness is what will affect the user’s learning (in this case) experience mostly. It is therefore crucial that this component performs according to the developers’ expectations and within limits of acceptable response times as perceived by the user.

A problem that the research group had to face in their previous projects was the difference in availability, stability and performance of network connections in schools and structures where they implemented their solutions. The fact that Java- and Python-based interpreters for AIML are mainly server-side is hugely impacted by inefficient or limited connectivity and can easily lead to performance degradation or even partial or total disruption of service for the application.

In order to avoid this and related problems, the new framework is expected to be web-based with a completely client-side AIML interpreter, which should allow excellent performance even with limited or unreliable network connections.

Although the framework to be developed will be used to support the educational software, it should be general enough that it can be used for other similar applications and different types of chatbots and it should also feature the possibility to use several chatbots in the same application simultaneously (albeit with separate chat windows or input fields).

\(^1\) [http://www.lucs.lu.se/educational-technology/](http://www.lucs.lu.se/educational-technology/)
\(^2\) [http://www.alicebot.org/aiml.html](http://www.alicebot.org/aiml.html)
\(^3\) [http://aitools.org/Program_D](http://aitools.org/Program_D)
This thesis’s main purpose is the development of a web-based framework that integrates with an AIML interpreter completely built in client-side Javascript that is compliant with the AIML specification wherever possible and implements a set of features similar to the ones in PyAIML. It also includes a simple and basic HTML5 chat interface for user interaction for demonstration purposes.

The result should be a perfectly working interpreter that allows users to select the chatbot and options (such as optional files for the chatbot) that he/she wants to interact with and be able to interact with the chatbot in a comfortable way, similar to chatting with other human beings.

The focus of the thesis will therefore be centred on the following factors:

- The development of an AIML-compliant interpreter in client-side Javascript that can run even on a simple HTML5 page.
- The analysis of which parts of the AIML specification may be unable to be fully operational in the interpreter due to the nature of Javascript (especially for security reasons).
- The evaluation of the performance of the interpreter according to correctness and perceived performance (see Chapter 5).

Completing this thesis will provide an open source Javascript-based AIML interpreter, making it possible to use AIML chatbots in web-based application without any need for server-side coding as well as fulfilling the need of the customer.

The remaining chapters’ content is divided as follows:

- Chapter 2 will deal with an introduction to chatbots and AIML and will explore their usage in the modern world, including examples theorized or built for educational purposes.
- Chapter 3 will explore design options considering the project’s requirements and the final choices that have been made in their regards, including the motivations behind such choices.
- Chapter 4 will explore the implementation of the interpreter, which will be called AIML.js, the challenges faced during the implementation phase, an overview of the resulting application as well as any requirements or features that could not be implemented or could only be partially implemented.
- Chapter 5 will analyse the testing and evaluation procedures used to ensure the software meets requirements, along with performance tests of a more formal nature.
- Conclusions will end the dissertation along with possibilities for future improvements.

As it will be explained later, some features are not implementable due to security restrictions of Javascript and its interaction with the Operating System.
Chapter 2

Chatbots and AIML

Basic principles of chatbots and their usage and purposes within history will be briefly explained in Section 2.1. Section 2.2 will focus on AIML with a brief overview about its success, a description of its main structure and its employment in the development of chatbots for educational purposes.

2.1 Basics of Chatbots

Chatbots (also known as Chatterbots or chatter robots) are software agents that simulate an entity, usually a human counterpart of vague or specifically defined characteristics, with whom the user can interact in a conversation (either written, oral, or mixed). One of the first and main goals of chatbots has always been to resemble an intelligent human person and make it hard or impossible for the other party of the conversation to understand their real nature (as in artificial). With the development of more and more chatbots of various architecture and capabilities the purposes for their usage has widely broadened.

The first chatbot, ELIZA, was developed by its creator, Joseph Weizenbaum in the middle of the ’60s to pass the test for software intelligence devised by Alan Turing in [2] (more recently known as the Turing test). The basic principle of ELIZA and other early chatbots is using pattern matching algorithms on the input provided by the user to identify key words or sentences, match them against a pre-compiled, hard-coded database of patterns and provide the best suitable answer based on the pattern matching alone.

These chatbots can still prove sufficient to fool the user into believing they are “talking” to a human being, but are very limited in improving their knowledge base at runtime, and have usually little to no means of keeping track of all the conversation data (lack of powerful history caches).

The most used pattern-matching chatbot in current days is A.L.I.C.E. (Artificial Linguistic Internet Computer Entity), three times winner of the Loebner Prize. A.L.I.C.E. is developed in AIML, which is a well-formed XML dialect with its own specification. The usage of AIML with interpreters that follow its specification allows it for a much more powerful set of features including conditional matching and recursion in matched results (see also Section 2.2.1). A.L.I.C.E. has spawned a lot of related bots throughout the years, some of which implemented their own additional AIML tags to further broaden the features of the language.

http://www.loebner.net/Prizef/loebner-prize.html
More recently, bots with real-time reasoning and knowledge improvement have been developed using complex natural language processing (NLP) systems and evolutionary algorithms such as Jabberwacky\(^7\) and Kyle\(^8\), both award-winning\(^9\) chatbots, although they are more complex to setup and might require more knowledge and training to maintain or troubleshoot.

There have been attempts to increase the knowledge of chatbots with other means as well, such as allowing them to use internet search engines and parse their results for answers, for example as specified in [3] (extending AIML tags to use RDF and SPARQL), translating information contained in a corpus (a text of arbitrary length in machine readable format) into AIML as obtained by Abu Shawar in [4], or simply from a “frequently asked questions” (FAQ) and a glossary as demonstrated by Giovanni de Gasperis in [5]. The level of interactivity with the chatbot is also a field of research that is seeing improvements, with researchers trying to make chatbots be able to express mood changes and even graphical animation, such as in [6].

Finally, the scope and purpose of chatbots have also varied greatly throughout history: apart from simply trying to fool a human user into believing they are speaking with another human, they have also been used to extract and retrieve information from the user, even for malicious purposes. Other malicious applications of chatbots involve spamming in public chats and instant messaging applications. More recently they have also been used in Massive Multiplayer Online games for spamming, disruptive behavior in public chats or masking other violations of the Terms of Conduct.

Obviously there are also many positive purposes for which chatbots have been developed and used. They are in fact used for offline troubleshooting, automated customer service and educational/pedagogical purposes (mostly using AIML as it is simple yet effective, see Section 2.2.2). It has also been shown by Crutzen et al. in [7] that chatbots might also be effective in helping people discussing topics that they might feel otherwise embarrassed or unwilling to talk about, such as sexuality and addictive substances.

### 2.2 Artificial Intelligence Markup Language (AIML)

The Artificial Intelligence Markup Language (AIML) is an XML dialect invented and developed by Richard Wallace for his bot A.L.I.C.E. which turned out to be one of the most famous and effective chatbots in history, winning a series of important prizes in the artificial intelligence sector as previously stated. Its notoriety and the fact that all its components are released under a GNU-like license allowed for the spreading of usage of A.L.I.C.E. and newly created bots which exploited AIML.

AIML’s main purpose is the definition of some knowledge that a software agent (usually a chatbot) has. This is done through the enumeration of AIML categories that define a reaction by the chatbot (defined in an element called template) upon the matching of a pattern from a given string (usually what the user writes in the chat window). The categories, reactions and every single feature of AIML are embedded into the appropriate AIML tags (which are by all accounts XML tags).

---

\(^7\) [http://www.jabberwacky.com/](http://www.jabberwacky.com/)


\(^9\) Jabberwacky won the Loebner Prize 2 years in a row, Kyle won the Leodis AI award in 2009
The main features of the AIML pattern matching are a simple and easy-to-understand syntax (it does not even reach the complexity of regular expressions, it just allows for single or multiple jolly characters between words or suffixes and prefixes), and the ability to create hierarchies of pattern matching through recursion. The Pattern-Template chain can also be used to increase the knowledge of the software agent (by adding categories directly to the AIML file from user-inputted text).

Since AIML is an XML dialect, it is not a programming language, and needs to be interpreted or parsed to be of any computational use.

Officially recognized interpreters\(^\text{10}\) have been so far developed in the following programming languages:

- Lisp
- Java
- PHP
- Ruby
- Perl
- Pascal
- .NET
- C++
- SETL

The interpreter must implement a series of task to successfully implement the AIML specification correctly:

- Pre-processing: load optional files (substitution file, predicates file, sentence splitting tokens) so that all incoming inputs get translated according to it before pattern matching is initialized (for the substitution file: correct typos, convert synonyms, or switch person);
- AIML file parsing: the chatbot AIML file (sometimes called brain) is loaded, categories might be translated in a data structure more easily readable by the interpreter;
- Pattern matching: the interpreter needs to match incoming input to the loaded AIML categories, and provide the result (from the template element).

### 2.2.1 Basic Structure

The AIML specification is formally defined in \(^\text{8}\)\(^\text{11}\) and the interpreter developed in this thesis is based upon that, although slightly less formal tutorials for the usage of AIML can be very useful when dealing with the practical aspects of the language. One such tutorial is the one by Marietto et al. in \(^\text{9}\).

```xml
<aiml>
  <category>
    <pattern>HEJ</pattern>
  </category>
</aiml>
```

\(^\text{10}\) As per the list on: [http://www.alicebot.org/downloads/programs.html](http://www.alicebot.org/downloads/programs.html)

\(^\text{11}\) The reference claims: “NB: Contents of this document are subject to change! This document should not be used as reference material or cited as a normative reference from another document.” However it is still used as there have been no changes since August 2006, and there is no mention in the author’s website that this version of the standard is still being worked on.
This snippet of code illustrates various tags of AIML and their function. It needs to be noted that, in order to avoid problems with case-sensitivity, all input text from the user has to be fully capitalized before accessing the pattern matching features of AIML.

The `<aiml>` tag usually encloses the whole AIML file, with the exception of the DOCTYPE tag that could precede it to define the document as XML compliant (not needed). If DOCTYPE detailing the XML version and the charset used is not present, the `<aiml>` tag takes as key-value pairs the AIML version used in the file (1.0 or 1.01) and the charset used.

The `<category>` tag is the building block of the AIML file. Each unit of knowledge and Pattern-Template response chain are contained within a category, as well as every other optional tag. The more categories an AIML file has, usually the more knowledgeable it is, although with substitutions during pre-processing, conditions and recursion (through a wise usage of the `<srai>` tag), redundant or similar patterns can be “compressed” into an overall more manageable file size.

The `<pattern>` tag contains the input sentence that has to be matched to activate that category. The input string can contain jolly characters such as * or _ as placeholders for any single string, allowing to match prefixes, suffixes or infixes in a given sentence, with the difference that information contained in the substring discarded by _ is usually of no interest, while strings contained in *-marked areas can be accessed in the `<template>` section with the `<star>` tag.

The `<template>` tag follows the `<pattern>` tag and completes each category. It contains the whole reaction of the chatbot to the matched pattern, which can vary from a simple textual response, to a more complex form of processing including setting and matching of variables and conditions, recursion, etc.

The `<think>` tag is located in the template section and allows the chatbot to process and compute information about the matched pattern without giving any information to the user. This can be useful to set the topic (with the `<topic>` tag) or variables (with the `<set>` tag) to better keep track of the conversation or even retrieve information to make sure that the matched pattern makes sense in its context (<that> and <get> tags).

The `<set>` tag, as already stated is used to set a variable with a key-value pair. To retrieve a variable the `<get>` tag can be used in a similar fashion.
The `<condition>` tag is used to determine the next reaction to take (either direct output or more complex processing) based on the matching. If the `<condition>` tag has key-value pairs itself the condition is only activated if those are matched (if the value associated to the variable in the key is correct), otherwise it is skipped. If the `<condition>` tag does not have a key-value pair it needs to have list elements (`<li>` tags). List elements with key-value pairs are only accessed if those are matched; otherwise the default one is accessed if it exists (a list item without a key-value pair). In the case above mentioned, if the variable `hej-count` is 1 the first item will be executed, for every other value the second item will be. The inner text (not tags) of the executed list item will be displayed, in this case, as the response text.

The `<srai>` tag is probably the most powerful tag in AIML as it can link multiple categories together. It can be used for recursion, giving one single template section for different patterns that have the same meaning (are synonyms or close), it can be used to map very similar specific patterns to a more generic one (when the bot needs to provide a generic answer), and it can be used as a run-time synonym replacer for words not matched in the invoked substitution file (which cannot be changed at run-time).

The `<that>` and `<topic>` tag allow to specify patterns to be matched in a specific context. More specifically `<that>` refers to the last chatbot output, while `<topic>` is set as a variable by the bot when specific categories are triggered. Since they both have priority over more generic categories, already used patterns can be reused in these specific situations with different templates, therefore conveying different, usually more specific, answers.

For all tags not described or not even mentioned in this section, their usage is described in [8] and [9], while they are certainly useful, this section does not want to provide full insight on the AIML feature set, but only highlight its potential and define the most common scenarios.

### 2.2.2 AIML Chatbots for pedagogical usage

Because of the many advantages of AIML (openness, ease of use, notoriety, etc.), it has been widely experimented upon by researchers and AI enthusiasts to enhance its potential and the features of its specification. Several custom tags have been created that work with specific interpreters for the newly created bots and one has a wide variety of options to choose from if he/she decides to develop an AIML-based chatbot.

One peculiar field of research of AIML chatbots is using the chatbot as a way of teaching the bot’s knowledge to the user. It has been, in fact, observed, that in a world where the Internet and technology keep progressing, the new generations of students and youngsters are more and more familiar and even attached to forms of digital interaction and communications such as chats and instant messaging applications.

It is natural, then, to pursue the usage of tools that exploit these familiar technologies to aid in the children’s learning process, as they generally perceive these applications as positive and fun, so they might be more good-willed to learn through these. To make these experiences even more complete and entertaining it is good to also try to implement more complex way of interactions such as animations, such as in the already mentioned article in [6]. Additionally, this technology can also help in areas with different type of problems, such as the scarcity of well-prepared teachers, as it may be the case in some developing
countries. Here chatbots can be adapted to take into account constraints and limitations of the particular situation in which they are introduced and provide an effective educational tool as suggested by Patrick Bii in [10].

Chatbots created for such purpose can also be greatly aided by AIML specification enhancement. For example, adding the ability to research the Internet through search engines for questions that the chatbot cannot match to its knowledge base can greatly increase the overall knowledge of the bot as illustrated in [3], however it is important in these cases to filter the results and make sure that the formulated response is the best possible. A solution to this problem is suggested by Orlando De Pietro and Giovanni Frontera in [11].
Chapter 3

Design

This Chapter covers all the design choices made during this project. It starts with an analysis of the given functional requirements in Section 3.1 and identifies the main design choice as the identification of the best interpreter architecture, compares the available options based on several aspects and comes to a definitive choice in Section 3.2 and its subsections. Section 3.3 deals with minor design choices such as details of the HTML5 interface and the hierarchy of chatbot-supporting configuration files.

3.1 Requirements Analysis

At the beginning of the design phase (ideally even before) it is always important to double-check the given requirements for the project (especially the functional ones), to make sure they are generally understood, they do not have contradictions between them and the ideas explored in this phase of the project are within the constraints of the requirements and allow for all of them to be fulfilled in this or a later phase (usually implementation).

It is therefore important to be reminded of the following points:
- What needs to be implemented?
- Why does it need to be implemented?
- Who will be the end-users of the project?

These points are crucial to be fully and correctly answered at this stage as they should affect design choices and will eventually be affected by them once said choices are made.

What needs to be implemented is a Javascript interpreter for AIML, the previously used interpreter (PyAIML) can be analysed and used as a starting point (specific implementation of functions) where applicable.

Why is this particular project needed? To ensure the possibility to develop and use chatbots in AIML in the new game that is being created by the customer as well as in future applications that might be based on this framework. This is HTML5 based and already has Javascript components, so it is natural to also build the interpreter in Javascript. Leaving everything client-side also allows for greater performance and eliminates a series of problems related to server connectivity of the previous version used for the chatbots framework (with PyAIML run on a server), especially in cases where connectivity is poor or limited.
Who will be the end user of the interpreter? Other people with an Interactive and Cognitive Sciences background on the chatbot side, other people with a Computer Science background who might want to extend the project. It is also important to remember that children aged around 11 years old will use the final product, so the user interface will have to be easy to understand and use.

These questions and their answers promptly spawn a series of observations on the design choices for the project, quickly easing and narrowing the fields of research to a limited number of issues:

- Javascript best coding practices (for the Implementation phase).
- AIML is a XML dialect, consider the possibility of using pre-existing XML parsers/frameworks written in Javascript (most notably the ones browsers have built-in, which are faster but can lead to portability issues, or complex XML manipulation frameworks).
- Consider the differences in size, complexity and guessed performance of using raw Javascript against known Javascript frameworks.
- Since PyAIML can be used as a starting point, consider the option of converting the existing PyAIML into Javascript with converters. Consider advantages (mainly time and complexity if no major issues arise) and disadvantages (what if there’s a major issue, how easy would it be to expand the project, does automatic conversion maintain/improve performance, does it obscure debug capabilities).
- Compare the performance of the considered frameworks/converters (where applicable) for a single page web application where the implemented interpreter will likely be the main script. Do the frameworks/converters already offer testing plugins/suites for evaluation/optimization (would be useful in the Testing and Evaluation phase)?
- The end users will be children who use pedagogical chatbots developed by the customer to interact with them in a formative experience. Even if other kind of chatbots will be implemented, it is assumed the end-user will always be someone who is merely interested in the input-response feature of the chatbot, with no additional concerns for web-development or implementation details, the main goal is the display of the correct chatbot response in a timely manner.
- Other people working with the implementation might be chatbot developers or also other people that want to use the interpreter in their solution (as the interpreter will be made available as open source), it is therefore important to make sure the code is not obscure, that it is fairly easily extendable and easy-to-debug.

### 3.2 Main Design Choice: Interpreter Architecture

In this section different choices for the interpreter architecture will be discussed, their pros and cons evaluated, and a final choice will be made based on merits of the architecture, requirements, and the author’s previous experience with said architecture.
3.2.1 Possibility to use external XML parser/framework

XML parsers are already present and built-in features of modern browsers, however they differ somewhat between each other so cross-browser compatibility of the same parsed code is not an option natively. These parsers, being already pre-loaded in browsers are the most efficient and best performing that are currently available, correctness should also be guaranteed by the browser manufacturer.

External XML Frameworks, such as Google AJAXSLT\(^\text{12}\) and Sarissa\(^\text{13}\) offer incredible XML manipulation tools (such as XSL Transformations and XPath support), but their feature-set feels like it reaches far beyond what is required of an AIML interpreter, which just needs to check for well-formedness of a file and browse its content with very little need for addition (only if the chatbot is in “learning” mode) and virtually no need for editing/manipulation of existing text. So while external XML solutions have been researched, they have been discarded even before being tried as they are clearly meant for a very different purpose of the one that is of interest.

3.2.2 Javascript, its frameworks and to-Javascript converters at-a-glance

Raw Javascript tends to be less professional-looking, yet it can be more versatile than frameworks when used by capable hands, as it might be possible to implement the same functions in a much smaller footprint. Less powerful functions are readily available. Performance can be very good (if best practices are followed), but it is also very easy to end up in common pitfalls. Since it is not a framework, there is no built-in testing/performance feature either.

Frameworks offer a standardized (more or less modular depending on the framework) set of features and code “style” that allows for both readily available powerful functions and a professional (and in most of the cases tidier) look-and-feel of the source code. There are many varieties of them, some more focused on graphical aspects (not even considered), some that focus on single-page web applications, and some that fully exploit the potential of architectures such as Model-View-Controller or Model-View-ViewModel. Most of the frameworks still offer a lot more than is needed (for example some offer declarative binding and templates, which are very powerful features, but not so useful in this project), but also some functions that are very relevant and might turn out to be very useful (XML documents as a hierarchy of DOM Nodes for easier selection/navigation or jQuery\(^\text{14}\)’s parseXML() function that delegates XML parsing to the parser of the browser that the webpage is being run on).

Converters: if the conversion is successful they lead to a working result in very little time (also true for minor conversion problems/bugs) compared to any other implementation, this allows for a shift of focus towards other possible concerns: mainly the possibility and ease of debugging, the possibility and ease to extend the project with new features directly in Javascript, without having to add it to PyAIML or as a PyAIML plugin (since that has not been developed by the author or the customer), the transparency (as in ease of reading and understanding) of the converted code so that other people can implement/fix/extend features of the interpreter, etc.

\(^{12}\) http://goog-ajaxslt.sourceforge.net/
\(^{13}\) http://dev.abiss.gr/sarissa/
\(^{14}\) http://jquery.com/
It is also relevant to understand how the Python behaviour of functions/variables/structures is changed into Javascript. For both Emscripten\textsuperscript{15} and Pyjamas\textsuperscript{16} (the main Python-to-Javascript converters) there are varying degrees of how the converted code behaves and performs. These converters, since they include Python compilers and Javascript interpreters, usually offer a varying degree of performance optimization and testing which might be useful and is surely a positive built-in feature.

### 3.2.2.1 Javascript Frameworks

**AngularJS\textsuperscript{17}**: Model-View-Controller framework backed by Google that is mainly architecture driven, which is not necessarily bad for the project. The full declarative programming and two-way data binding is interesting but not useful at all for an interpreter, as are all the filters (formatting, filtering and sorting) for the HTML declarations. While not sporting a bigger than needed footprint, the main features offered miss the requirements of this project and fall short where it is needed (DOM Manipulation seems to be frowned upon in controllers, which would shift most of the AIML parsing to the model).

It is also fairly prescriptive on how to build applications with it, which might be a limitation. It doesn’t seem to offer any XML specific functions or special parsing ability, so it would probably be better for data-driven application like CRUD\textsuperscript{18} admin panels or applications.

**BackboneJS\textsuperscript{19}**: Light MVC framework, defines very little and gives most of the control to the developer, has interesting but not very useful (from this project’s perspective) features like a routing controller. While it is more interesting than Angular because it’s not very prescriptive or declarative in nature, and also fairly minimalistic and elegant which are always a plus, in order to really shine it needs to couple its models with data from RESTful services (JSON-based), so it would mean translating the AIML data structure from XML-based to JSON in order to fully exploit BackboneJS’s potential.

**jQuery**: More a library than a real framework, it differentiates itself from the rest of the list by being more DOM driven than providing actual architectural structuring. Its main features are surely the ease of access it provides, the use of AJAX calls and responses and DOM navigation and manipulation. This alone makes jQuery stand out as XML files can easily be fed to jQuery’s DOM navigator, just by pretending they are actual HTML files (although some errors may occur if there are HTML tags in the XML file), but everything becomes even better when jQuery allows the developer to use a \texttt{parseXML()} function that delegates the parsing of XML files to the browser currently in use. Finally, there is also an external jQuery plugin called jParser which is an XML parser with some additional features, although it might be slower than the \texttt{parseXML()} function, but it is good to have options to choose from. The only main downside of jQuery could be the messiness of the code.

---

\textsuperscript{15} https://github.com/kripken/emscripten/wiki

\textsuperscript{16} http://pyjs.org/

\textsuperscript{17} http://angularjs.org/

\textsuperscript{18} Create Update Retrieve Delete – acronym that explains the main basic functions admin panels for common websites achieve.

\textsuperscript{19} http://backbonejs.org/
KnockoutJS is a Model-View-ViewModel framework that is slightly lighter than AngularJS but offers similar features, although it does not directly support jQuery templating (which can be used in Angular). Knockout also fully focuses on data-binding, so it is more useful for other types of application (data-driven), just like Angular.

Other Frameworks more similar to jQuery that could have been considered were: Dojo, Prototype and MooTools, which are long-standing adversaries of jQuery but have been discarded because jQuery is the library that focuses best on DOM manipulation of the lot (while MooTools for example drifts away into trying to implement OOP and inheritance in Javascript), and has generally the better performance among these as seen in the online tests and analyses in [12], [13] and [14] and in the paper by Gizas et al. in [15].

3.2.2.2 Converters

Pyjamas (or Pyjs) is actually a GWT port to Python, so all the good and bad of the standard GWT distribution should be ported into the lighter programming language Python. While it could be manageable to translate a small application and surely has interesting features like different level of optimizations during compiling and conversion, most of the GWT plugins are not available in a ported form in Pyjs. Additionally personal (negative) experience with GWT suggests the author to avoid it unless necessary which decreases considerably the relative value of this option for this particular project. Especially in this area where there is a certain degree of freedom of choice, unless one choice has a clear functional advantage to all the others (which is not the case for Pyjs, as it is being asserted), personal experience should be seriously taken into account as well as academic papers and other informed opinions of specialists in the sector.

Emscripten: LLVM to Javascript converter, can convert C/C++ (and by extension CPython) to LLVM through Clang (provided with Emscripten). Very big projects have already been converted (most notably Unreal Engine 3) and it is surely an alternative if one would like to look at a different approach to the project (less towards implementation more towards extendibility and debugging of something already implemented).

There are however a number of problems: there are obviously limitations on what can be translated (coding limitations) mainly related, for the scope of this project, to memory setting and reusing (especially if the reuse happens for a different data-type than the original usage). More notably there is a performance issue, where the translation is slower than native code (by a factor of 2x to 5x depending on the optimization used according to [16]) and the performance is also heavily browser-dependant which might be unwanted ([17] shows that a general conversion result runs twice slower on Chrome than on Firefox in most cases, 4 times slower if the best optimization is used).

3.2.3 Final Choice: jQuery

The chosen architecture for the interpreter is the jQuery library. Past experience with very few issues and great results combined with the before-mentioned features that are very helpful for this project (DOM Manipulation, XML parsing functions, etc.) make it the best overall choice.

---

20 [http://knockoutjs.com](http://knockoutjs.com/)
21 [http://www.unrealengine.com](http://www.unrealengine.com/)
3.3 Other Design Choices

Secondary design choices include the HTML5 interface on which the interpreter will be tested and the hierarchy of the optional files to load with a given chatbot.

The HTML5 interface will be a simple HTML webpage in which the user will be able to choose the optional files and the bot to load in the chat, a chat window in which messages will be displayed and a user input field in which the user will be able to write messages. Since it is for demonstration purposes only, it does not need to be more complex than needed. Obviously the application developers will probably want to load bots and optional files themselves and not let the user choose them from a menu, so it will have to be possible to do so with a direct function call in Javascript in a real world application.

The support files will be organized as such:

- Substitutions, predicates, and sentence splitting tokens will all be in separate XML files (similar to ProgramD)
- There will be a main XML file (bot configuration file) for each bot setup that will hold one combination of paths to valid AIML file(s) and a path of each type of support files (one substitutions file, one predicates file, one file containing sentence splitting tokens). The paths will be relative to the file structure and need to be valid. The bot configuration file (or its path) is passed to the interpreter which will check and retrieve all the linked files and start pre-processing on them and eventually load the requested chatbot.
Chapter 4

Implementation of AIMLjs and results

This Chapter focuses on the implementation of the interpreter previously discussed and designed. Basic use cases will be first presented, and then an overview of the main and required files will be shown, with a focus on the two main algorithms used to get an appropriate chatbot response from a user-inputted message. Finally there will be a brief discussion on challenges faced during the implementation phase, whether or not they could be overcome and how that was accomplished.

4.1 Use Cases

There are two main use-cases for this application:

1: The user selects a configuration file to load the chatbot from -> the system loads the selected and referenced resources (this is necessary in the Demonstration version, it can be skipped/omitted if the page was to load pre-selected resources);

2: The user sends an input to the system -> the system tries to match the input with its knowledgebase and returns either a response (if the input was matched) or a default message;

4.2 AIMLjs Overview

AIML.js is the main file of the application and fulfills all the duties required by an AIML interpreter.

The following flowchart displays the main functions and features of the interpreter and how they interact together:
Use Case 1 - Initialization:

![Diagram of Use Case 1](image1)

Use Case 2 – I/O Iteration:

![Diagram of Use Case 2](image2)

### 4.2.1 Variable Initialization

In the variable initialization section the end-user is given all sorts of control over the details of the interpreter. While some options are merely editable to have some freedom structuring own applications (such as the location of configuration files), or for displaying purposes only (for formatting decisions), others have a double...
purpose: it is the case with selectors for display and input retrieval, these offer both a default value (that can be overridden with different types of calls to the interpreter) but also to allow for multiple instances of the interpreter to be run (one can run on the pre-existing chat with default selectors, others can be created, for example, dynamically with different selectors, and then another instance of the interpreter (with the same or different bots) can be run by specifying the new selector values during call time.

Additionally it was decided to have configuration values right in the beginning of the file to make it easier to find them as a function to initialize them would have kept the code cleaner, but would have reduced overall usability.

### 4.2.2 Configuration Files and Loading a Chatbot

These are the different types of configuration files used by the interpreter:
- `Configlist.xml`: an XML configuration file used in the demonstration version to list all available botfiles to allow user selection of one;
- `Botfile`: an XML file that identifies a specific set of files for a given bot, it includes the bot name, the bot predicates file, the bot substitutions file, a splitting file and the bot brain file(s);
- `Predicates file`: an XML file containing pre-set predicates and bot values;
- `Substitutions file`: an XML file containing a list of input patterns and the values they need to be replaced with during runtime;
- `Splitting file`: an XML file containing a set of characters to split the input on at runtime;
- `Brain file(s)`: one or more AIML files containing the knowledge base for the bot.

The XML files all start with `<?xml version="1.0" encoding="UTF-8"?>` to allow for UTF-8 values and they use the following namespaces and schema locations:

```
xmns="http://ida.liu.se/aimljs/bot-configuration"
xmns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://ida.liu.se/aimljs/bot-configuration http://ida.liu.se/aimljs/bot-configuration.xsd"
```

Each XML file has a different syntax, examples follow:

**Predicates file:**

```xml
<properties>
  <property name="name" value="value"/>
</properties>
```

**Substitutions file** (note that the “find” value needs to be a Regular Expression):

```xml
<substitutions>
  <input>
    <substitute find="\bhey\b" replace="hello"/>
  </input>
</substitutions>
```

AIML files need to be compliant with the AIML 1.0 specification\(^\text{22}\) (which is the only one supported at this stage).

Different ways have been tried to allow xml/aiml files accessibility and listing (so that very little prior knowledge would be required when running the application), since Javascript is a client-side language, access to local files/directories is forbidden due to security reasons, but there are workarounds, especially for XML/text-based files. One can for example recreate a sandboxed file system with HTML 5 File API. With it, it would be fairly easy to allocate temporary or persistent memory for an app-dedicated file system with a lot of features (command-line functions, full CRUD options for files and directories and so on). This is a very powerful tool, that works especially well in Google Chrome (since Chrome has a specific API for it) and would expose server-side language functions (like automatic discovery and listing in HTML sections of the available configuration file and directory hierarchy of files), but requires a lot of setting up (all files have to be manually put in the sandboxed file system at least once), it uses additional storage space and ultimately seems to be unnecessary for the task at hand.

\(^{22}\) [http://www.alicebot.org/documentation/aiml101.html](http://www.alicebot.org/documentation/aiml101.html)
Thankfully, being the configuration files mostly XML files or derivatives (AIML is an XML dialect), they can be accessed fairly easy with XMLHttpRequests, or AJAX calls. This is made even easier with jQuery, but it still has some issues or details to be careful about. First and foremost, the main difference with Server-side functions or the HTML5 File API is that the exact path and name of each needed file needs to be known beforehand, this is true for both webserver-hosted applications (through the HTTP protocol) or local ones (through file://). Additionally local resources cannot be accessed by local scripts in Chrome (it needs to be run with special flags to allow for that and these flags might expose the browser to security threats if it is also used for normal web browsing). Also, since they are AJAX calls, there is no easy way to add information (writing or appending) to the requested XML/AIML files. Finally, XML is a very picky dataType when it comes to AJAX calls, as it requires the webserver (or the agent serving the XML pages) to specify an appropriate content type for the response (usually "text/xml"), otherwise a parse error occurs even if the page is successfully accessed.

With no server-side programming possible with Javascript, there is no way to make sure the server sets the appropriate Content Type, the best effort is to make sure the xml files are well formed (or at the very least properly formatted). This requires more workarounds, such as the definition of fictional (or even real, if there is reason for it) XML name spaces, Schema locations and Schema file(s). As of now they have been set to http://ida.liu.se/aimljs/bot-configuration but this address is purely a placeholder for the sole purpose of having a correct content-type assigned to the document by the webserver.

A chatbot can be loaded in two different ways with AIML.js: through the function loadFromConfig(file, options) or through the function loadChatbot(options), depending whether a Botfile is used or not. Another difference between the two functions is that the first one needs to be called within the document.ready() section of the Javascript code, as it uses default variables in case not all of them are overwritten from the file parameter, whereas loadChatbot(options) can be called at any moment, but the options array needs to contain initialized values for all the variables needed by the interpreter.

4.2.3 AIML File Parser

The AIML File Parser function (load_aiml(file, debugLevel, id, returnValue)) checks the compliance of an AIML file to the AIML Specification. If it finds a malformed <aiml> element a fatal error occurs, if it finds malformed <category> or <pattern> elements, the relative category is removed from the local memory (not from the AIML file). Finally, if malformed template elements or unknown elements in a template are found, a warning might be given (if the debugLevel parameter is set accordingly) and those elements might be ignored. If the parsing is successful, the AIML file is stored in the interpreter’s memory.

The AIML Parser in this project acts in a different way from parsers in the other interpreters used by the research group (PyAIML, Program D) as it does not parse the AIML document by character position (keeping track on where it is in the AIML structure by recognizing elements and setting a state variable), but it handles the document as an XML file with its own DOM tree (made of AIML nodes), and parses it from root (<aiml> element) to leaves instead, using jQuery functions such as .find(), .children() and .each() to easily traverse and explore
it. In this way state variables are only useful for debugging and can be ignored and it easier to find malformed elements are jQuery provides functions to check for attributes and content of elements regardless of their order. Finally, it would likely be easier to check for more complex element structures with this method, if any such structure would become available in future versions of AIML.

### 4.2.4 Input Processing and Pattern Matching

Once the user enters an input in the appropriate field and submits it through either clicking the submit button or the enter key, this is split into sentences according to the Splitting File options, and then each sentence is run through the input substitutions and an appropriate AIML pattern is matched from the brain files. Also histories are kept of input messages, input sentences (including `<srai>` values that called the pattern matching function), output messages and topic values.

The Pattern matching algorithm finds all the matching patterns for a given input and it is split in 4 sections according to their priority:

- The section where both a Topic and a `<that>` value exist;
- The section where Topic exists but only categories without `<that>` are considered;
- The section where only categories without a Topic, but with a `<that>` are considered;
- Finally, the section with general categories that do not belong to a Topic or have a `<that>` element.

The algorithm checks if one or more patterns match the categories in the first relevant section (if a topic is not set the first 2 sections are completely ignored, etc.), and proceeds to the following section if a result was not found. If more than 1 result was found, this can happen due to the possibility of wildcards ("_" and "*"") in the pattern, the `<that>` element and/or the topic value, all the valid results are added to an array and the decision on the correct match is postponed.

This is a code snippet from the algorithm when a topic value and a `<that>` are set (other sections are similar):

```javascript
... //if there is a topic
if(topic && topic != ""){
  dummy = $.extend(true, [], brainCollection);
  for (var i = 0; i < dummy.length; i++){
    //look first for that topic in the brain files
    $(dummy[i]['xml']).find("topic").each(function (){
      //save the topic name for later use
      topicName = $(this).attr("name");
      //convert the topic name to a Regular Expression
      var regstr = $(this).attr("name").replace(/_/g, ".+");
      regstr = $(this).attr("name").replace(/\+/g, ".+");
      regstr = new RegExp("^" + regstr + "$");
      //filter for matches
      if($(this).attr("name").match(regstr))
        return $(this).attr("name").match(regstr);}
    }).length > 0){
```
// if there are matching topics save them
subset[i] = $(this).filter(function (){
    return $(this).attr("name").match(regstr);
});
topicFound += 1;
// if that is defined
if(that && that != ""){
    // look for categories with matching <that> in
    // the current topic categories
    $(subset[i]).find("category").each(function (){
        // only interested in categories with <that>, ...
        if($(this).find("that").text() &&
            $(this).find("that").text() != ""){
            // only interested in <that> matched by the
            // stored value
            var strmatch = $(this).find("that").text()
                .replace(/\+g, ".+");
            strmatch = strmatch.replace(/_/g, ".+");
            strmatch = new RegExp("^" + strmatch + "$");
            if(that.match(strmatch)){
                // if it is found, it is the wanted section
                subsubset[i] = this;
                thatFound += 1;
                $(subsubset[i]).each(function (){
                    // convert the pattern into a RegExp and
                    // check it against the input
                    var inpmatch = $(this).find("pattern").text()
                        .replace(/\+g, ".+");
                    inpmatch = inpmatch.replace(/_/g, ".+");
                    inpmatch = new RegExp("^" + inpmatch + "$");
                    if(input.match(inpmatch)){
                        // save the topic name as a category element
                        // for sorting/priority matching
                        if($(this).find("topic").length < 1){
                            $(this).append("<temptopic>"+ topicName
                                +"</temptopic>"));
                        }
                        // push in the array of matched results
                        resultset.push(this);
                        found += 1;
                    }
                });
            }
        }
    });

...
- The word “X” (where X is a word of the input matched in a pattern) has lower priority than “_” but higher priority than “*”;
- The “*” wildcard has the lowest priority.

It can be derived that:
- a “_” appearing earlier in a pattern has highest priority;
- if the first “_” appears in the same position (index) in two different matching patterns, the one with more “_”s has higher priority;
- if two matching patterns have the same, amount of “_”s and they are at the same indexes in the patterns, the pattern with fewer “*”s has higher priority;
- if two matching patterns have the same, amount of “_”s and they are at the same indexes in the patterns, and also have the same amount of “*”s, the pattern with the first “*” having a higher index has higher priority;
- fully matched text with no wildcards has lower priority than any pattern with any positive number of “_”s but has higher priority than any pattern with any positive number of “*”s.

This is true both for patterns, <that> values and topic values, but topics have the highest specificity so, if present, they need to be checked first, <that> elements have the second highest specificity, and patterns have the lowest. It is also possible that different matched patterns have the same <that> and/or topic value, so duplicate values of those elements should not be discarded early.

The following code snippet shows the choosing algorithm for matched topic values, it is exactly the same for <that> values and for patterns (minus the first if statement):

```javascript
...  
for(var i = 0; i < set.length; i++){  
  var topic = $(set[i]).find("temptopic").text();  
  tempundcount = (topic.split("_").length - 1);  
  tempundindex = topic.indexOf("_");  
  tempstarcount = (topic.split("*").length - 1);  
  tempstarindex = topic.indexOf("*");  
  //same topic, add to the result for later matching  
  if(topic == chosenTopic && chosenTopic != ""){
    chosenOne.push(set[i]);  
  }
  //higher priority to lower indexes of "_"  
  else if(tempundindex < undindex && tempundindex >= 0){  
    undindex = tempundindex;  
    undcount = tempundcount;  
    starindex = tempstarindex;  
    starcount = tempstarcount;  
    chosentopic = topic;  
    chosenOne[0] = set[i];  
    //remove other elements if they exists  
    chosenOne.splice(1, chosenOne.length-1);
  }
  //higher priority to higher count of "_"  
  else if((tempundindex == undindex && tempundindex >= 0)  
    && tempundcount > undcount){
    //same code as in previous block, omitted for shortness
```
else if((tempundindex == undindex && tempundindex >= 0) 
        && tempundcount == undcount){
    //but fewer stars or stars that appear later 
    //in the pattern
    if((tempstarindex >= starindex && tempstarindex >= 0) 
         && tempstarcount <= starcount){
        //same code as in previous block, omitted for shortness
    }
}

else if(!((tempundindex != -1 || tempundcount != 0 ||
            tempstarindex != -1 || tempstarcount != 0))){
    //check that underscore value is still default (1000)
    // or not found (-1)
    if((undindex == 1000 || undindex == -1) 
        && undcount == 0){
        //same code as in previous block, omitted for shortness
    }
}

//if no matches with Underscores or perfect matches,
//look for matches with stars
//higher priority to lower count of "*"
else if((tempstarindex == starindex && tempstarindex >= 0) 
        && tempstarcount < starcount){
    if(((undindex == 1000 || undindex == -1) && undcount 
        == 0) && !perfectMatch){
        //same code as in previous block, omitted for shortness
    }
}

//higher priority to higher indexes of "*
else if(tempstarindex > starindex && tempstarindex >= 0){
    if(((undindex == 1000 || undindex == -1) && undcount 
        == 0) && !perfectMatch){
        //same code as in previous block, omitted for shortness
    }
}

//update the value of the set to match
set = $.extend(true, [], chosenOne);
//make sure there are no unneeded values still in the set
set.splice(chosenOne.length, (set.length - 
            chosenOne.length));
...
4.2.5 Output Processing

Once the matching category has been found, the last task is to process its <template> and all the AIML elements inside it and provide a textual result. This is achieved by recursively getting the tagName attribute of each element and call the function handler it refers to. Currently implemented elements are listed below with any additional information as a comment:

```javascript
var elemProcessors = {
  "bot": processBot, // same as <get>
  "condition": processCondition,
  "formal": processFormal,
  "gender": processGender, // has access to substitutions
  "get": processGet, // has access to predicates
  "gossip": processGossip,
  "input": processIn, // has access to input history
  "javascript": processJavascript, // evaluates a Javascript expression
  "learn": processLearn, // only accepts valid AIML code as text
  "li": processLi,
  "lowercase": processLowerCase,
  "person": processPerson, // has access to substitutions
  "person2": processPerson2, // has access to substitutions
  "random": processRandom,
  "text": processText, // for templates that only have text
  "sentence": processSentence, // has access to predicates
  "set": processSet,
  "sr": processSr,
  "srai": processSrai,
  "star": processStar, // has access to input history and substitutions
  "system": processSystem, // not implemented
  "template": processTemplate,
  "that": processThat, // has access to input stack and output history
  "thatstar": processThatstar, // has access to output history
  "temptopic": processTemptopic, // simply removes this non-AIML tag
  "think": processThink,
  "topicstar": processTopicstar, // has access to topic history
  "uppercase": processUpperCase
};
```

Furthermore, the <system> element cannot be implemented due to Javascript restrictions on its interactions with the Operating System.

4.3 Challenges Faced during Implementation

The main challenged faced during the implementation of the interpreter presented itself in the very early stages in the form of being able to save to memory and use the return value of AJAX callback functions, when this return value depended on the content of the XML/AIML files accessed with the AJAX call. This is due to the
fact that AJAX calls are executed in parallel or generally fetched at a different time than when the remaining Javascript code is running. This means that trying to utilize the results of said callbacks (even by copying the retrieved data to higher-scope variables) would result into unexpected behaviors (either empty variables or the values of the values before the AJAX calls are executed). While nesting them forces them to only process the nth call after nth-1 has returned successfully, this was not possible in the interpreter. After different stages of trial and error the only working solution turned out to be calling each AJAX calls synchronously (with the jQuery parameter async: false). This way, if a variable from the calling function was provided, the success callback of the AJAX request could update said value with the XML/AIML (entire or a portion of) that was fetched with the request. This was the only real coding-related problem that took more than a couple of days to figure out, and while no other major coding setback was encountered, a lot was learned in the fields of jQuery selectors, DOM traversing and even about Regular Expressions used for matching strings.

Additionally, the lack of the possibility to create permanent log files restricted the debugging abilities and error reporting to mere console.log() calls. A default variable selects the verbosity of the logging, either errors-only or everything (which includes logs of the main functions called and their result).

### 4.4 A Look at the Interface

A Screenshot of the HTML interface is provided to highlight the simplicity of the look:
Figure 3 - AIML.js Demo
Chapter 5

Testing and Evaluation

This chapter focuses on testing the implementation in terms of correctness and evaluating the user-perceived performance of the application. Evaluation of usability is skipped due to the very simple nature of the interface that should be readily understandable by any user.

5.1 Correctness

Correctness testing was divided into two steps. During the first step, given the same AIML file it was checked that for the same input AIML.js and PyAIML produced similar outputs (with a difference in the wrapping of the output string, obviously). This was done with one of the AIML files provided by the supervisor and a very restricted number of inputs (the AIML file had more than 2000 categories). The second step was creating custom made AIML files for testing purposes that had pattern and <that> fields made so that consecutive similar or equal inputs cycled through different matched patterns to make sure that pattern matching and wildcard priorities are implemented correctly. The last part of the second step included checking the correct processing of each template-side element on its own, and in the most common combinations (for example <think> elements containing <set> elements, <set> elements containing <star> elements, combinations of string-formatting elements). Considering that the possible combinations of elements are unlimited (there is no set maximum nesting depth in the AIML specification), exhaustive testing is not possible, and limiting the combinatory testing to the most common combinations was deemed acceptable.

5.2 Perceived Performance

It is important that the chatbot user feels no latency while using the application, therefore the only performance issues that are of concern are the ones that might constitute delays of half a second or more. In order to test for performance, real XML and AIML files are used in sequential similar test cycles of chatbot loading and input/output. These are analyzed with Google Chrome’s Development Tools (more specifically with the Javascript CPU Profiler23). This test was performed with the code running on a remote webserver, to remove possible advantages of a localhost setup.

This is the profile for a normal chat session (only first handful of input/output cycles is shown):

https://developers.google.com/chrome-developer-tools/docs/cpu-profiling
Figure 4 – DevTools overall Flamechart
Figure 5 - DevTools AJAX Flamechart
As it can be seen, the main delay happens between seconds 2 and 4 of this execution. This is due to the loading of the bot files; whereas other spikes in the chart (which are input/output cycles) are much thinner.

Taking a look at the loading part, the main culprit of a suboptimal performance is the synchronicity of the AJAX calls to load each file, if it was possible to make the requests asynchronously, this section would only take slightly longer than `load_aiml()` function, rather than the sum of that with the other loading functions, however this is not currently possible due to the restrictions noted in 4.3.

![DevTools i/o Flamechart](image)

It has to be noted that the overall execution time for this section of the code is however barely 0.8s, which is not too degrading considering that the user has not started interacting with the chat yet until this section is fully loaded. Hence it can be deemed an acceptable performance result, considering the goal of perceived performance.

Looking at the performance of the input/output cycle instead, the performance is very close to optimal, as even complex templates are resolved in around 20ms.
5.3 Notes on Compatibility

The interpreter has been tested remotely and in local environments on the versions available from June 2013 and newer of the following web browsers:

- Google Chrome
- Mozilla Firefox
- Internet Explorer 11
- Safari
- Opera

No issues have arisen on any of the aforementioned browsers. The application should also work on Internet Explorer 9 as jQuery is fully compatible with it and no additional Javascript functions with particular requirements were used. The application, however, will not work on Internet Explorer version 8 or older, due to incompatibility with jQuery and with some native Javascript functions (such as `Array.prototype.splice()` and `String.prototype.indexOf()`). If jQuery-specific errors should arise on any browser, activating the `jQuery-migrate` plugin included with the project (in the `/js` folder) could fix or at least show console warnings that will help resolve the encountered issues.

With regards to library compatibility, this application works on jQuery versions 1.10 and newer, including versions 2.x. Compatibility with older versions should exist, but it is not guaranteed. If the need to use an older version should arise and errors are encountered, these can be fixed by also loading the already mentioned `jQuery-migrate` library.

Finally, if errors are shown when using special characters (such as accented vowels, Cyrillic characters etc.), the appropriate charset and encoding needs to be set on the webserver and checked on all the XML/AIML files that display errors or unexpected behavior. The recommended encoding is UTF-8.
Chapter 6

Conclusions

The goals for this thesis were to develop an interpreter in Javascript for chatbots based on AIML.

The steps taken to achieve this started with learning about AIML and its specification and also taking a look at pre-existing AIML interpreters and how they functioned (mainly PyAIML). Afterwards, a careful definition of structural and functional requirements made it possible to lay solid foundations for the whole project. The main design choice for jQuery as the library to use in the interpreter further consolidated the project structure, due to its useful AJAX requests system and DOM manipulation tools.

The implementation focused on creating a robust interpreter that fulfilled all the requirements strictly and making full use of the jQuery toolset resulting in algorithms often radically different from the ones observed in PyAIML, while maintaining similar features. These algorithms exploit for the most part the jQuery's ability to manipulate AIML files' DOM tree, which result in easier-to-understand code which, in turn, should lead to easier code maintenance and modification even by third parties.

Finally, software testing revolved around the correctness of the output results and acceptable perceived performance of the framework. Even if jQuery functions are known to be slower than pure Javascript ones [19], the performance was found to be adequate to the requirements. Special attention also went to ensure cross-platform compatibility among up-to-date commonly used web browsers.

The end result is a fast and sleek framework that offers great input/output performance and rigorous AIML 1.0 compliance with a minimal and easy-to-understand interface.

6.1 Future Improvements

Future Improvements for the interpreter include support for newer versions of the AIML specification both in the parsing and template processing sections, introduction of HTML5 File API for logging sessions/warnings/errors on text files rather than just on the browser’s console, and possibility of running AJAX calls asynchronously without losing access to the values of the callback functions (when it will be possible). Future improvements on a broader scope revolve around the integration of the interpreter in the Research Group’s application and/or using its user management API to have personalized (and logged) access.
References

[1] F. Malvisi, Development of visual tools for the semantic marking-up of HTML5 documents via Microdata, B.Sc. in Computer Engineering, Università degli Studi di Parma, Italy, 2010.


Appendix A: Interaction Example

This appendix includes an example of how the interpreter works with working files and user input. The brain file used in this example is \textit{Galileo.aiml} and both the input and the patterns matched will be roughly translated in English for documentation purposes only. Here are the contents of the loaded files for this example (substitutions file and brain files are omitted, the relevant elements will be reported in the example as they are called):

\textbf{Predicates.xml}:

```xml
<?xml version="1.0" encoding="UTF-8"?>
<properties xmlns="http://ida.liu.se/aimljs/bot-configuration"
             xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
             xsi:schemaLocation="http://ida.liu.se/aimljs/bot-configuration http://ida.liu.se/aimljs/bot-configuration.xsd">
  <property name="name" value="Galileo Gallilei"/>
  <property name="gender" value="male"/>
  <property name="age" value="50"/>
  <property name="city" value="Pisa"/>
  <property name="default" value=""/>
  <property name="count-hej" value="0"/>
  <property name="user-compgame" value="NO"/>
</properties>
```

\textbf{Sentence-splitters.xml}:

```xml
<?xml version="1.0" encoding="ISO-8859-1"?>
<sentence-splitters xmlns="http://ida.liu.se/aimljs/bot-configuration"
                    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
                    xsi:schemaLocation="http://ida.liu.se/aimljs/bot-configuration http://ida.liu.se/aimljs/bot-configuration.xsd">
  <splitter value=".">
  <splitter value="!">
  <splitter value="?">
  <splitter value=";"
</sentence-splitters>
```

Once the bot is loaded, the first user input can be typed in the input field and submitted, in this example it is: “Hey! What’s your name?”

The input gets split into two sentences: “Hey” and “What’s your name”. The first sentence is run through the substitutions files and the following rule is found:

```xml
<replace find="\bh\ey\b" replace="hello"/>
```
The sentence after substitutions is “hello”, it is converted to uppercase and tries to get matched. The matching patterns for the sentence are “HELLO” and “*”, the first one is selected due to priority, this is its category:

<category>
  <pattern>HELLO</pattern>
  <template>
    <think><set name="topic">NO</set></think>
    <condition>
      <li name="count-hej" value="1">
        <srai>ASK USER</srai>
        <think><set name="count-hej">0</set></think>
      </li>
      <li><think><set name="count-hej">1</set></think>Hello</li>
    </condition>
  </template>
</category>

The topic is set to “NO”, and since the variable “count-hej” is 0, the default list item for the condition is parsed, setting the same variable to 1 and returning the partial output “Hello”.

The second sentence is then run through substitutions, finding this rule:

<substitute find="\bwhat's\b" replace="what is"/>

Therefore the string to be matched becomes (after substitutions and uppercase) “WHAT IS YOUR NAME”, with a “that” value of “Hello” and the topic “NO”. The matched categories have these patterns “WHAT IS YOUR NAME”, “*” and “* IS YOUR NAME”, the first one is chosen for priority.

<category>
  <pattern>WHAT IS YOUR NAME</pattern>
  <template>
    <think><set name="topic">NAME</set></think>
    <bot name="name"/>
  </template>
</category>

The topic is therefore set to “NAME” and the return value is the bot variable “name” which is “Galileo Galilei”. The output is recomposed and submitted to the user as: “Hello. Galileo Galilei.”

If the next input is “Hello”, the same pattern gets matched as the first sentence of the first input, as there are not any more specific patterns with the current “that” value or topic (respectively “Galileo Galilei” and “NAME”), but this time the other list item gets parsed since the variable “hej-count” is now set to 1. This means that now the <srai> element gets executed and “hej-count” is once again set to 0. The <srai> element matches the following patterns: “ASK USER” and “*”, once again the most specific one is selected:
<category>
  <pattern>ASK USER</pattern>
  <template>
    <condition>
      <li name="user-name" value="NO">
        <think><set name="topic">NAME</set></think>
        What is your name?<li>
      ...
      <li name="user-compgame" value="NO">
        <think><set name="topic">COMPGAME</set></think>
        Do you play computer games?<li>
      ...
      <srai>HAVENT ANY QUESTIONS LEFT</srai></li>
    </condition>
  </template>
</category>
(some list items are omitted)

Since the first variable (and all the omitted ones) is not set (so its default value is ""), the selected list item is the one that checks the value of "user-compgame" which is set to "0" in the predicates file; the topic is then set to "COMPGAME" and the output is "Do you play computer games?".

The next and final input for this example is “I do, do you?”, which is left unchanged through sentence splitting and substitutions since no matching separators or rules are found, gets converted to uppercase and gets sent to the pattern matching function with a “that” value of “Do you play computer games?” and the topic value of “COMPGAME”. In this case there are matched categories with the current topic value and “that” value, and they have patterns “* YOU” and “*”. The selected one is the first one.

<topic name="COMPGAME">
  ...
  <category>
    <pattern>* YOU</pattern>
    <that>DO YOU PLAY COMPUTER GAMES?</that>
    <template>
      <think><set name="user-compgame"><star/></set></think>
      <srai>SPEL A</srai></template>
  </category>
  ...

This sets the variable “user-compgame” as the star value of the input, which is “I do, do” and matches the string “GAME A” which matches the pattern that goes by the same name in the same topic.

<category>
  <pattern>GAME A</pattern>
  <template><random>
    <li>Mhm, computer games are cool</li>
    <li>Yes, quite often</li>
    <li>Most games on the internet</li>
  </random>
</category>

A random list item is selected, and its value is returned as the final output.
In English

The publishers will keep this document online on the Internet - or its possible replacement - for a considerable time from the date of publication barring exceptional circumstances.

The online availability of the document implies a permanent permission for anyone to read, to download, to print out single copies for your own use and to use it unchanged for any non-commercial research and educational purpose. Subsequent transfers of copyright cannot revoke this permission. All other uses of the document are conditional on the consent of the copyright owner. The publisher has taken technical and administrative measures to assure authenticity, security and accessibility.

According to intellectual property law the author has the right to be mentioned when his/her work is accessed as described above and to be protected against infringement.

For additional information about the Linköping University Electronic Press and its procedures for publication and for assurance of document integrity, please refer to its WWW home page: http://www.ep.liu.se/

© [Filippo Malvisi]