ONTOLOGY POPULATION USING HUMAN COMPUTATION

A THESIS SUBMITTED TO
THE GRADUATE SCHOOL OF COMPUTER ENGINEERING DEPARTMENT
OF
MIDDLE EAST TECHNICAL UNIVERSITY

BY

GENCAY KEMAL EVİRGEN

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR
THE DEGREE OF MASTER OF SCIENCE
IN
COMPUTER ENGINEERING

JANUARY 2010

Approval of the thesis:

ONTOLOGY POPULATION USING HUMAN COMPUTATION

submitted by **GENCAY KEMAL EVİRGEN** in partial fulfillment of the requirements for the degree of
Master of Science in Computer Engineering Department, Middle East Technical University by,

Prof. Dr. Canan Özgen
Dean, Graduate School of **Natural and Applied Sciences** ________________

Prof. Dr. Müslim Bozyiğit
Head of Department, **Computer Engineering** ________________

Assoc. Prof. Dr. Ferda Nur Alpaslan
Supervisor, **Computer Engineering Department, METU** ________________

Examining Committee Members:

Prof. Dr. Mehmet R. Tolun
Computer Engineering Department, Çankaya University ________________

Assoc. Prof. Dr. Ferda Nur Alpaslan
Computer Engineering Department, METU ________________

Assoc. Prof.Dr. Nihan Kesim Çiçekli
Computer Engineering Department, METU ________________

Assist. Prof.Dr. Pınar Şenkul
Computer Engineering Department, METU ________________

Dr. Ayşenur Birtürk
Computer Engineering Department, METU ________________

Date: ________________

I hereby declare that all information in this document has been obtained and presented in accordance with academic rules and ethical conduct. I also declare that, as required by these rules and conduct, I have fully cited and referenced all material and results that are not original to this work.

Name, Last Name: GENCAY KEMAL EVİRGEN

Signature :

ABSTRACT

ONTOLOGY POPULATION USING HUMAN COMPUTATION

Evirgen, Gencay Kemal
M.S., Department of Computer Engineering
Supervisor : Assoc. Prof. Dr. Ferda Nur Alpaslan

January 2010, 56 pages

In recent years, many researchers have developed new techniques on ontology population. However, these methods cannot overcome the semantic gap between humans and the extracted ontologies. Words-Around is a web application that forms a user-friendly environment which channels the vast Internet population to provide data towards solving ontology population problem that no known efficient computer algorithms can yet solve. This application's fundamental data structure is a list of words that people naturally link to each other. It displays these lists as a word cloud that is fun to drag around and play with. Users are prompted to enter whatever word comes to their mind upon seeing a word that is suggested from the application's database; or they can search for one word in particular to see what associations other users have made to it. Once logged in, users can view their activity history, which words they were the first to associate, and mark particular words as misspellings or as junk, to help keep the list's structure to be relevant and accurate. The results of this implementation indicate the fact that an interesting application that enables users just to play with its visual elements can also be useful to gather information.

Keywords: Ontology Population, Human Computation, Web Applications

ÖZ

İNSANLA BERİM KULLANARAK ONTOLOJİ TÜRETME

Evirgen, Gencay Kemal
Yüksek Lisans, Bilgisayar Mühendisliği Bölümü
Tez Yöneticisi : Doç. Dr. Ferda Nur Alpaslan

Ocak 2010, 56 sayfa

Geçtiğimiz yıllarda, birçok araştırmacı ontoloji türetmeyle ilgili yeni yöntemler geliştirmişlerdir. Ancak, bu metotlar, insanlar ile çıkarılmış ontolojiler arasındaki anlambilimsel farkın üstesinden gelememişlerdir. Words-Around şu ana kadar bilinen herhangi verimli bir algoritmanın çözemediği ontoloji türetme problemine veri sağlamak için muazzam internet nüfusunu yönlendirmek amacıyla kullanıcı dostu bir ağ uygulaması oluşturmaktadır. Uygulamanın temel veri yapısı insanların doğal olarak birbirine bağladığı kelimeler listesidir. Uygulama, bu listeyi sürüklemenin ve oynamanın zevkli olduğu bir kelime bulutu içerisinde gösterir. Kullanıcılar uygulamanın veritabanından çekilen bir kelimeyi görünce akıllarına gelen başka bir kelimeyi girmeye yönlendirilir ya da belli bir kelimeyle başkalarının ilişkilendirdiği diğer kelimeleri görmek için arama yapabilirler. Siteye giriş yaptıktan sonra, kendi işlem geçmişlerini ve hangi kelimeleri ilk ilişkilendirdiklerini görebilir ya da kelimeleri imlası yanlış veyahut anlamsız olarak işaretleyebilir ve böylece listenin yapısının alakalı ve doğru kalmasına yardımcı olabilirler. Bu uygulamanın sonuçları kullanıcıların oynamaktan hoşlandığı görsel unsurlar içeren ilginç uygulamaların aynı zamanda bilgi toplamak için de kullanılabildiğini göstermiştir.

Anahtar Kelimeler: Ontoloji Türetme, İnsanla Berim, Ağ Tabanlı Uygulamalar

To the memory of my father, Muzaffer M. Evirgen

ACKNOWLEDGMENTS

I would like to start by thanking my supervisor Assoc. Prof. Dr. Ferda Nur Alpaslan. Her support, advice and guidance were invaluable during the course of this thesis.

Special thanks go to my friends (in alphabetical order), Sertan Alkan, Çağatay Çallı, Umut Eroğul, Doğacan Güney, Atıl İşçen, Çelebi Kocair, Bahar Pamuk, Selma Süloğlu and many others for their ideas, constant help and understanding.

I would also like to thank all the jury members for their insightful comments and suggestions on this thesis.

I would like to thank all staff at the Department of Computer Engineering at METU for providing a warm and friendly environment during this study.

A long list of users, too long to mention here, is thanked for an enormous amount of valuable feedback on the performance of the site. Without their assistance, this thesis would probably not looked the way it does now.

Finally, I would also like to express my gratitude to my brother, my mother and the loved ones for all the encouragement, support and love they provided. Without them, this thesis would never be finished.

TABLE OF CONTENTS

ABSTRACT iv

ÖZ v

ACKNOWLEDGMENTS vii

TABLE OF CONTENTS viii

LIST OF TABLES xi

LIST OF FIGURES xii

CHAPTERS

1 INTRODUCTION 1

1.1 Problem Definition and Motivation 1

1.2 Background 2

1.2.1 Ontology 2

1.2.1.1 Types of ontologies 2

Domain Ontology 2

Upper Ontology 2

1.2.1.2 Components of ontologies 2

1.2.1.3 Ontology Population 3

1.2.2 Human Computation 3

1.3 Contributions 3

1.4 Thesis Outline 4

2 RELATED WORK 5

2.1 Ontology Engineering 5

2.1.1 Information extraction approach 6

2.1.2 Machine learning approach 7

2.2 Human Computation 7

2.2.0.1 ESP Game 7
2.2.0.2 Verbosity 8
2.2.0.3 reCAPTCHA 9
2.2.0.4 Phetch 10
2.2.0.5 KissKissBan 11
2.2.0.6 OntoPronto 12
2.2.0.7 TagCAPTCHA 12
2.2.0.8 Thumps-Up 12
2.2.0.9 Page Hunt 13
2.2.0.10 Phrase Detectives 13
3 WORDS AROUND APPLICATION AND ITS RESULTS 15
3.1 Overview 15
3.1.1 Flash Client 18
3.1.1.1 Authentication Dialog 18
3.1.1.2 Word List Dialog 19
3.1.1.3 Junk Word / Not a Word Dialog 19
3.1.1.4 New Word Dialog 20
3.1.1.5 Word Association Dialog 20
3.2 The Algorithms and the Architecture 22
3.2.1 The Algorithm 22
3.2.2 The Architecture 24
3.2.2.1 Class Diagram 25
3.2.2.2 Association Overview 25
3.2.3 Service Layer 26
3.2.3.1 Model 26
User Profile 27
SynSet 27
JunkSet 29
Association 29
First 29

Misspell 30
Junk 30
Settings 30
Deathklok 31
3.2.3.2 Lex 34
3.2.3.3 Display 34
3.2.3.4 User 37
3.2.3.5 BaseView 38
3.3 The Results 39
3.3.1 User Feedback 39
3.3.2 Statistics 39
3.3.3 Foreign language issues 40
3.3.4 Comparision with WordNet 41
4 CONCLUSION AND FUTURE WORK 42
4.1 Conclusion 42
4.2 Future Work 42
REFERENCES 45
APPENDIX
A VISUAL DESIGN DETAILS 48
A.1 BaseView 48
A.2 BaseApplication 49
A.3 Model 52
A.4 Application 54

LIST OF TABLES

TABLES

Table 3.1 Synset of awesome 22

Table 3.2 Results of Words-Around in 1 Week 39

LIST OF FIGURES

FIGURES

Figure 2.1 A screen shot from the ESP Game. 8

Figure 2.2 A screen shot from Verbosity. 9

Figure 2.3 A view from reCAPTCHA system. 10

Figure 2.4 A screen shot from Phetch. 11

Figure 2.5 A screen shot from Page Hunt. 13

Figure 2.6 A screen shot from Phrase. 14

Figure 3.1 Overview of Words Around. 15

Figure 3.2 Cloud of Totallywild. 16

Figure 3.3 Log in panel. 18

Figure 3.4 Log out panel. 19

Figure 3.5 View of Word List. 20

Figure 3.6 Junk Word / Not a Word panel. 21

Figure 3.7 New Word panel. 21

Figure 3.8 Word Association panel. 21

Figure 3.9 Logic behind word associations 24

Figure 3.10 3-tier architecture of the application 24

Figure 3.11 Class diagram 25

Figure 3.12 Overview of the transactions between layers while associating words. . . . 25

Figure 3.13 Files that are in the service layer. 26

Figure 3.14 UserProfile class. 27

Figure 3.15 SynSet class. 28

Figure 3.16 JunkSet class. 29

Figure 3.17 Association class. 30

Figure 3.18 First class. 31

Figure 3.19 Misspell class. 31

Figure 3.20 Junk class. 32

Figure 3.21 Settings class. 32

Figure 3.22 Algorithm of Deathklok. 33

Figure 3.23 Associations of *dark*. 40

Figure 4.1 URL Change 44

Figure A.1 BaseView class. 48

Figure A.2 BaseApplication class. 49

Figure A.3 BaseApplication class continued. 50

Figure A.4 BaseApplication class continued. 51

Figure A.5 BaseApplication class continued. 52

Figure A.6 Model class. 52

Figure A.7 Model class continued. 53

Figure A.8 Model class continued. 53

Figure A.9 Model class continued. 54

Figure A.10 Application class. 55

Figure A.11 Application class continued. 56

Figure A.12 Application class continued. 57

Figure A.13 Application class continued. 57

Figure A.14 Application class continued. 58

CHAPTER 1

INTRODUCTION

1.1 Problem Definition and Motivation

One of the key elements of the Semantic Web is the ontology. The purpose of an ontology is to formally represent concepts within a domain and the relationships between those concepts. In terms of description logic, ontology can be defined as the combination of T-Box and A-Box, where T-box defines the classes and relations in an ontology, while A-box contains the individuals in the domain.

Without a populous A-Box containing many individuals, an ontology cannot be said to be complete. Therefore, a significant number of individuals are needed in an ontology. However, population of the individuals is very time consuming and the methods proposed by the researchers so far cannot overcome the semantic gap between humans and computers. Therefore, actual human-beings are needed for populating ontologies.

Most research in information extraction discovery requires heavily on the availability of datasets. With the rapid growth of user generated content on the internet, there is now an abundance of sources from which data can be drawn. Compared to the amount of work in the field on techniques for automated ontology population, there has been so little or no effort shown at the study of effective methods for ontology population using human beings.

In this thesis, my goal is to develop a web-based application that forms a user-friendly environment that channels the vast Internet population to provide data towards solving ontology population problem that no known efficient computer algorithms can yet solve.

1.2 Background

1.2.1 Ontology

Ontology is, as cited by Gruber [19] is a formal specification of a shared conceptualization. It deals with entities and their relations. Philosophically, ontology studies what entities exist or can exist in the world, what kind of relations those entities can have to each other and how such entities can be grouped. In computer science, an ontology is defined as the formal representation of a set of concepts within a domain and the relationships between those concepts. Ontologies can be used for reasoning purposes about the properties of that particular domain and also, they can be used for defining that domain.

Ontologies are used in many different areas, such as the Semantic Web, artificial intelligence, software engineering as a representation of some of the world.

1.2.1.1 Types of ontologies

Domain Ontology is about a specific domain. For example, the word *turkey* means differently in various domains. For example, if it would mean a country in the domain of country names, while it would mean a North-American bird in the birds domain.

Upper Ontology is about a model of the common objects and it contains a glossary, where the users can benefit from it while describing other domains. [31]

1.2.1.2 Components of ontologies

- **classes**: Classes group similar elements, with similar properties together.
- **individuals**: Individuals are objects in the universe of things.
- **properties**: Properties are binary relations. [22]

1.2.1.3 Ontology Population

Ontology population is achieved in different ways, such as term recognition and information extraction. While these are similar tasks in many parts, the techniques used performing these are quite different. Generally, term recognition benefits from primarily statistical techniques, which is usually with the combination of basic linguistic information. On the other hand, information extraction usually uses a rule-based approach or a machine learning technique, or a hybrid of these two. [27]

1.2.2 Human Computation

Human computation is a very new research area. After the paper published by Luis von Ahn [41] *et. al.* in 2004 about image labeling via humans, this topic has began to emerge. It simply suggests that people should be used to outsource to gather, process, or manipulate data while automated techniques are not enough. There have been several developments in the recent years that benefit from this massive work force.

1.3 Contributions

My contributions in this thesis are:

1. I propose a new tool for ontology population.
2. I propose a new approach where instead of using computer based systems, I use the vast Internet population to gather information for populating ontologies..
3. I propose a new algorithm that allows the populated ontology to be dynamically shaped according to the user interaction.
4. I, furthermore, propose that in web-based applications stimulating the competitive spirit between users is not the only source for gathering information from the people but a wellfavored application that enables its users to drag and play with its visual elements is also competent.

5. I also show that the quality of the end result varies greatly depending on the number of people using the application.

1.4 Thesis Outline

This thesis is organized as follows: In Chapter 1, I define the problem, mention about the motivations behind this thesis and provide the necessary background knowledge to understand the problem domain and the solutions. In Chapter 2, I mention about the related work. In Chapter 3, technical details of the work is given. In Chapter 4, experimental results which demonstrate the utility of the proposed methods are shown. In Chapter 5, the thesis is concluded with a summary and future directions.

CHAPTER 2

RELATED WORK

2.1 Ontology Engineering

During the last years, many ontology population techniques have been discoreved. However, ontology population task is still a costly and resource-intensive. Therefore, new methodologies that reduce this cost are highly needed. As the data is available in various forms, researchers have developed data-diriven techniques that have become the be known as ontology learning.

Ontology learning has been thought that it has the potential to decrease the amount of effort spent on creating and maintaining ontologies. Therefore, researchers have come up with different tools over the time, such as TextToOnto [8] (integrated with KAON [39] ontology), OntoLt [33] (integrated with Protege [14]) and Text2Onto [7] (integrated with NeOn [30]).

The data that the ontology techniques can be applied for are:

- structured (for example databases)
- semi-structured (for instance HTML or XML)
- unstructured (such as textual data)

documents.

The techniques utilizes several computer science disciplines. However, all of these approaches regarding ontology learning is still semi-automatic, meaning that human intervention is required.

I will not go into detail about learning the T-BOX of an ontology, rather I will survey on populating the A-BOX.

There have been some ways in population ontologies; one of which is done by using NLP techniques [27].

Using NLP, ontology population is mainly done by a type of ontology information extraction tool. In details, it is about identifying and relating the key terms in the text (named entities, technical terms) with the concepts. Generally, this is performed by linguistic preprocessing, followed by a named entitiy recognition tool.

2.1.1 Information extraction approach

There have been several approaches regarding the ontology population using the NLP techniques. One of them is GATE [10], the General Architecture for Text Engineering framework [28], that supports a variety of language engineering tasks. It includes an information extraction system, ANNIE [4] (consists of a *tokeniser, sentence splitter, POS tagger, gazetteer, finite state transduction grammar and orthomatcher*), and various plugins for ontology support, information retrieval, WordNet, machine learning algorithms, and so on.

Another tool, ANNIC [2] (ANNotations In Context) has functionalities like advanced search and visualisation of linguistic information, which provides alternative methods of identifying patterns and searching the textual data in the corpus.

AeroDAML [23], by linking proper nouns and common types of relations with classes and properties in a DAML ontology, applies information extraction techniques to automatically generate DAML annotations from web pages.

During the information extraction process, all these systems' target output is ontology. Some of them also use class and instance information during the information extraction process. Some others benefit from ontology structures. A few of them are uses rule-based approaches.

2.1.2 Machine learning approach

There are two paradigms in ontology population while using machine learning techniques. One of them is about context feature approaches [9] and the other is about using patterns and rules [15].

Context feature approaches (either superficial [13] or syntactic [24] [1]) utilize a corpus for extracting features in all contexts. Comparison between these shows that better performance can be gathered via syntactic features [9]. In this approach, feature weights are calculated by several machine learning algorithms or statistical measures [13] [25].

The systems that populate ontologies using patterns or rules, show that there is an is-a or other relation between two words, but it is not possible to find that kind of relations in all corpora. Therefore, some approaches benefit from the Web [36]. A weakly supervised pattern-based system learns generalized linear patterns which express a certain relation [35]. Another system is presented which uses patterns to extract relations in a weakly supervised manner and maps them to WordNet [32].

Also, there are some hybrid approaches which uses pattern based, term structure, and context feature methods together [6].

2.2 Human Computation

Human computation suggests that people should be used to outsource to gather, process, or manipulate data while automated techniques are not enough. There has been several developments in the recent years that benefit from this massive work force.

2.2.0.1 ESP Game

The touchstone of these developments can be said to be started with Luis von Ahn's ESP Game [41]. In this game, two random partners who are located somewhere in the world are shown random images that were crawled from web. They are trying to guess what the opposite one is typing for the image shown on the web. An example scene from the game is given in Figure 2.1.

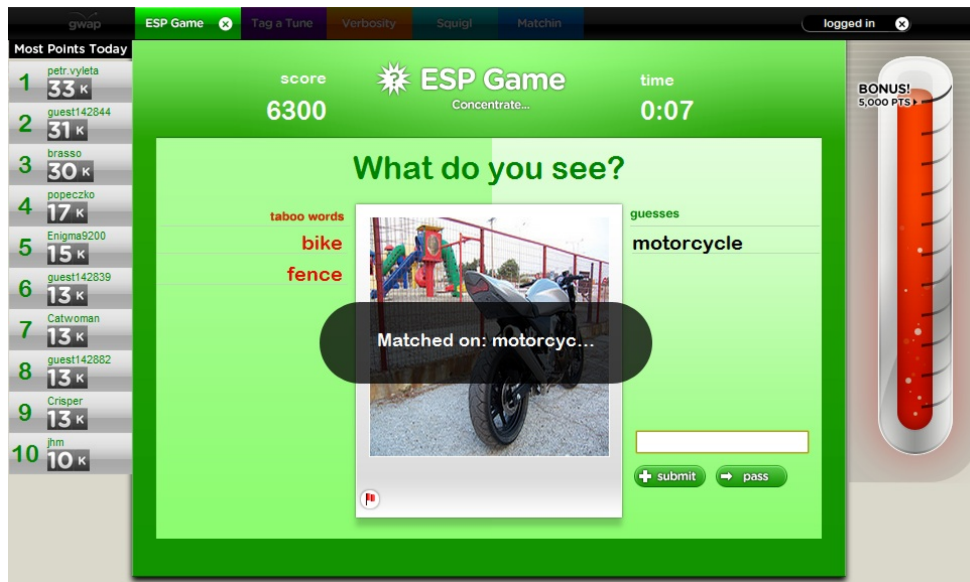

Figure 2.1: A screen shot from the ESP Game.

When they agree on a word, the photo is labeled with that agreed word. A key element which is shown in the advanced levels is that the ”taboo words” concept. Images, which were previous labeled with these taboo words by other players are shown to the player, while these taboo words aren’t allowed to be typed as a label. This makes the pictures labeled with more specific words.

There is a threshold for the labels. A label should have been agreed upon by at least an arbitrary number of pairs before it will be used as a taboo word.

This game’s effect on developers was striking, that Google later on acquired this game and it became Google Image Labeler [18].

2.2.0.2 Verbosity

Another application using humans is called Verbosity [44]. This game, like ESP game, was developed by the same team. There is a narrator and a guesser. The narrator is given a secret word and he is expected to make the guesser type the word by sending hints. These hints are (while ”it” is referring the secret word) in the form of

- it is

- it is a type of
- it has
- it looks like
- it is used for
- it is the opposite of

Therefore, the narrators' hints give the seekers common sense information about the word while the application is collecting data about hierarchical categorization, purpose of a word, spatial info, and basic relations between words. An example scene from the game is given in Figure 2.2.

Figure 2.2: A screen shot from Verbosity.

The reliability of the data depends on how much this game is played basically. The more this game is played, the more reliable data is found.

2.2.0.3 reCAPTCHA

CAPTCHA uses a digital system called OCR to recognize scanned words. However, OCR cannot recognize 20% of the words. But it is stated by the developers that humans can

recognize 99% of the words. The reCAPTCHA (re-Completely Automated Public Turing Test to tell Computers & Humans Apart) system, differently, displays 2 words, rather than CAPTCHA's 1 word [45]. An example scene from the application is given in Figure 2.3.

Figure 2.3: A view from reCAPTCHA system.

In the application, one of the words is called as the control word. This word was scanned by different OCR programs, and checked in an English dictionary, and with 96% accuracy this word is truly scanned by a computer. As the control word's answer is known, the answer which does not match with the control word is eliminated. The second word is a word that was not able to be scanned correctly by computers. Therefore, it is asked to humans. If the user answers correctly to the control word, the user's other answer is recorded as a potential answer for the unknown word. If three humans answer in the same way, then this word is saved as a control word. After one year of running system, humans had solved 1.2 billion of CAPTCHA's, that is amounting to over 17600 books.

2.2.0.4 Phetch

The purpose behind this game is to tag the images with explanatory descriptions, mainly with the reason of improving accessibility [43]. In the game, there is a Describer and 2 to 4 Seekers. Given an image, the Describer broadcasts a description regarding the image. Then the Seekers, while using an image search engine, seek images related to their queries. The first Seeker to click on the correct image becomes the winner. An example scene from the game is given in Figure 2.4.

This game utilizes the ESP Game's image database, which contains over 100000 images labeled with 9 keywords. The keywords used in the query is thought to be related with the

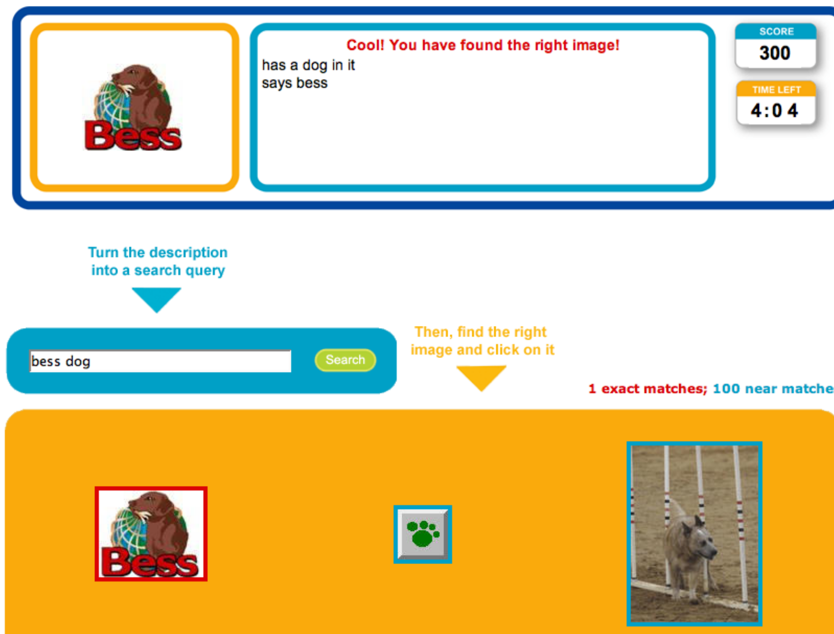

Figure 2.4: A screen shot from Phetch.

labels associated with images and description given by the Describer, as all of this data is the outcome of human computation. The accuracy of this app depends on the amount of time used by the Seekers to find the correct image. If it took long or even the Seekers fail, then the description becomes useless. Also, the recorded sessions of describers are shown to many Seekers to determine whether a Seeker is accurate. During the trial, 129 people played the game in a week and they generated 1436 captions for images. On average, they generated captions for almost 7 images in one session.

2.2.0.5 KissKissBan

This game with a purpose can be thought as an extraction over the ESP Game [21]. Taboo word concept is not available in this game. Instead, they use a new concept, called Blocker. The game is played with 3 people; 1 Blocker and 2 other players. In the beginning of the game, the Blocker is provided with an image and then he provides a list of blocked words, that he thinks the other player might match on. After this stage, the couple tries to guess what each other labeled the image. However, if they guess a word from the block list, they lose a point. If a player guesses a word from the blocked list, this is a label for the picture. If a

couple agree upon a word, that is another label. The outcome of this game is structured in this way. The developers used *Amazon Mechanical Turk* for their experiments and on average in 537 games; the application gathered a total of 5231 labels (3296 from block list and 2225 agreed upon).

2.2.0.6 OntoPronto

In OntoPronto [37], two players, located in different places, are given an an the first paragraph of a randomly chosen Wikipedia article. They are expected to select whether this paragrapgh is related to a set of objects or an individual. If they agree on one article, they move on to another. In two minutes, they try to earn as much points as possible. Using the agreed paragrapghs, the developers are trying to derive an OWL DL ontology at the background.

2.2.0.7 TagCAPTCHA

This is an image-based CAPTCHA instead of words [29]. People are given a set of images, one of which; as it is in the case of reCAPTCHA; comes from a verified image set. The players are expected to tag these words correctly. As one of the images come from a reliable source, players' answers to this image are important. This known image validates the players other annotations. A newly tagged image is promoted to pending status, until it can be verified by another player. Researchers conducted a small test of 12 people. Each player was asked to solve 20 instances. The result rate gave a 70% success rate on average.

2.2.0.8 Thumps-Up

This game is developed by researchers at Yahoo! [11] Two players are shown a search engine query and two digital images that are gathered from web pages related with this query. If the players agree on the web page, they move on to another web page and the query that they agreed upon gets is a higher level in that particular search query.

2.2.0.9 Page Hunt

This game is developed by researchers at Microsoft [26]. It is about annotating web pages. In the game, the player is shown a random web page and he tries to set up a query that would bring this page in the top 5 results of a search engine. An example scene from the game is given in Figure 2.5.

Figure 2.5: A screen shot from Page Hunt.

As this research has been conducted in Microsoft, they used their own search engine, Bing. People, in this way, while competing with each other, tag the web sites with an appropriate key word. The developers conducted a test on 100 queries and it turned out 78% queries to be OK.

2.2.0.10 Phrase Detectives

The purpose of this game is to experiment with human computation though the WWW to create large scale linguistically annotated corpora [5]. The players are given a word or a phrase and they must look for any evidence of it appearing earlier in the text. Also, in another mode of this game, players check the other players' decisions. The latter mode can be called as a tool for the accuracy of the game. An example scene from the game is given in Figure 2.6.

This game, however, lacks from usability a lot. It expects a lot from a regular flash-game

Figure 2.6: A screen shot from Phrase.

player. Therefore, results of this game are not pleasant. In one year, they were just able to get only 3000 annotations.

CHAPTER 3

WORDS AROUND APPLICATION AND ITS RESULTS

3.1 Overview

The Words-Around application is a Flash based word association game. It is a tool designed for both fun and lingual learning. Users are given a word to associate, and can enter anything that comes to their mind and submit it to the server.

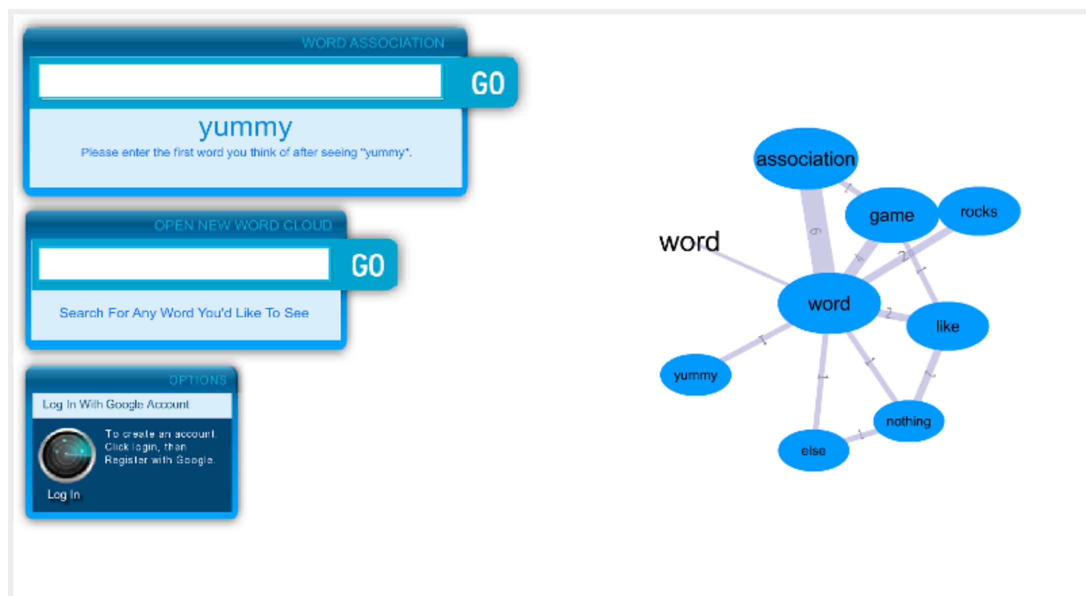

Figure 3.1: Overview of Words Around.

If the connection between the two words does not exist, I create one; if the words have been associated before, I increment a counter to add *weight* to the particular association. I then display this information as a word cloud, with higher strength associations being made larger

and more prevalent than lower strength associations. If a user has a word or phrase they would like to know more about, they can search for it and see a cloud of related words or phrases. Seeing and interacting with other users' grasp of a particular language can be an invaluable tool for increasing one's vocabularity and familiarity for that language. The current implementation is language agnostic; there is no translation mechanism, but it is planned to add a polylingualism module so words and phrases can be linked across languages. With this functionality, a user can search for a particular phrase in their native language, and then request that translations for a particular language be included. This would not only provide a direct translation mechanism as many websites currently offer, but a more abstract, loosely coupled selection of translation choices. This would greatly increase the fluidity and fluency of one's vocabulary, not to mention it's potential as a tool for writers of all kinds who have an idea of what they want to say, but cannot find *the perfect phrase*. If the phrase they want to link is not yet in the system, they can add it themselves, and other users will eventually give their own personal association for the particular entry. The remainder of this document shall be dedicated to how and why I have created this game, and give some technical insight as to how it all works *under the hood*.

Figure 3.2: Cloud of Totallywild.

To start with, consider the phrase, *totally wild* shown in Figure 3.2. Depending on where

you live and how you speak, this phrase could mean a variety of things. You may think of terms like, *savage* or *feral*, whilst another person might say, *really neat* or *awesome*, and still another user might think, *uncontrollable* or *uninhibited*. Because all of these phrases are linked to 'totally wild', as well as to each other, I get grouped word associations, with inter-word / phrase cross references to increase linguistic proficiency. English is especially notorious for having lots of words or phrases that mean more than the sum of the letters used. *I'm cool*, for example, literally means *I am cold*, but in most Western translations, it means *I'm popular* or *people like me*. There are also phrases that when directly translated make little to no sense; *I rocked out*, *I am nuts*, *I am on fire*, *That sucks*, *This blows*. Because the direct, word-to-word translations of these terms won't maintain the intended meaning of the phrases, I need to know how native-speakers of a language use these phrases, as well as phrases similar to them, in order to make an accurate translation.

English is not the only highly-colloquial language; in French, for example, it is common to call a loved one a small vegetable as a term of endearment; phrases like *My pretty tomato* or *My delicate cabbage* don't really make sense in English. Without colloquial translations to more literal terms like, *My dear friend*, or *I adore you*, these living language phrases could quite easily be confused by non-native speakers of the language. Although I am discussing cross-language translations to highlight the ambiguity of the diverse and ever-evolving languages, these principles still apply to everyday use of a single language; the more words and phrases you are familiar with, the better you can communicate your thoughts.

The core datatype of the Words Around application is a *Synset*, or *Synonym Ring* in linguistic science, and it contains all the statistics I need to create word clouds in the application. A Synset is a list of words that people naturally link to each other, and this application displays them as a word cloud that is fun to drag around and play with. Please note that the terms *word*, *phrase* and *Synset* are semi-interchangeable, so I will try to use Synset only when I mean the data object with statistics and other metadata. There is one Synset object for every word or phrase that has been entered into the Words-Around datastore, and it tracks how many times that particular word or phrase has been linked to other Synset words or phrases.

I create a variety of other data types for each Synset, including records for the first time a word is ever used, a record for misspelling reports, junk reports, and a record for who submitted what association, and when. I keep a String id that is the exact word spelling,

and a numeric id of each Synset, to help reduce wasted disk space and increase application performance. Storing and using long integers is always more efficient and less error prone than using text ids. All other data stored on Synset instances are composite counters, meaning they are numeric fields that simply count how many times a word has been associated, or marked as misspelled or junk. I only allow logged in users to mark words as garbage, as the system auto-deletes words whenever *total association count* is less than *total junk count*. Although I use these other data types individually, I use running counts composite counters in the Synset object itself so it can be quickly grabbed and used in the client application without counting hundreds of records each time it is accessed. I shall now explore how a user can interact with Synsets, as well as how and why I store this data.

3.1.1 Flash Client

3.1.1.1 Authentication Dialog

I perform the login Figure 3.3 and logout Figure 3.4 routines using the Google Accounts User API. Clicking login will redirect the user to a Google login page, where they can sign in with or register for a free account.

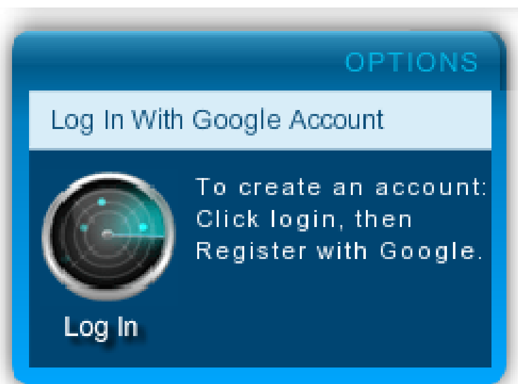

Figure 3.3: Log in panel.

When a user does login, the flash client will show all the user-only panels, like the Word Association List and the Junk/Misspell controls. I only allow logged in users to mark Junk so I can detect spammers.

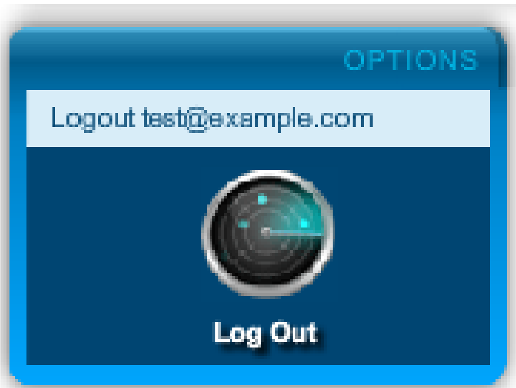

Figure 3.4: Log out panel.

3.1.1.2 Word List Dialog

The Word Association Lists panel is used to display all of the data that I tie directly to the user's email. This includes every association they have made, all the junk / misspelled words they have marked, as well as which words they thought of first. I use the GAE SWF Project [3] to get this data to flash from the Google AppEngine [17] Datastore [16] , and then I build lists of links for each word, so the user can click on one to open it's associated word cloud. This allows users to check back and see if any there are any new submissions on words they have entered, and it will allow users to flag words as garbage multiple times, by simply opening the word cloud and hitting the appropriate button.

Note that this panel shown in Figure 3.5 is only displayed for logged in users.

3.1.1.3 Junk Word / Not a Word Dialog

As shown in Figure 3.6, to help keep the records accurate, I allow users to notify us of garbage words that slip in. I make a distinction between misspelling and absolute garbage so I can implement a spelling correction mechanism to attempt to correct poor spelling.

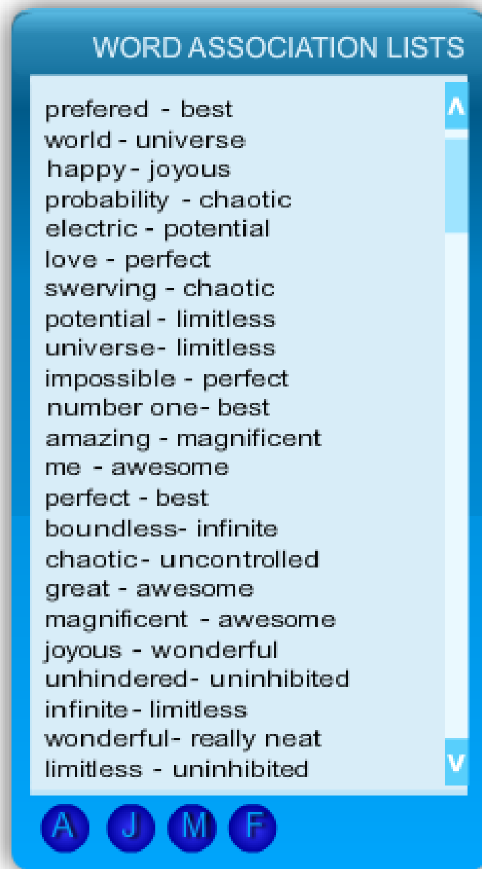

Figure 3.5: View of Word List.

3.1.1.4 New Word Dialog

A web application without a search functionality isn't much of a web application at all so I use this panel (shown in Figure 3.7) to prompt users to open a new word cloud. Entering text that doesn't exist yet will yield a random result.

3.1.1.5 Word Association Dialog

The most important panel of all is the word association submission panel.

The word displayed, in this case *awesome* is selected at random from the database (shown in Figure 3.8). Whatever the shown word is, users can choose to submit whatever text they naturally link to it. The word *awesome* already exists in the datastore, and in this case the user

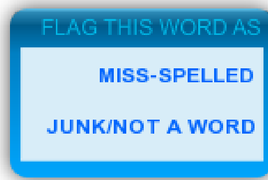

Figure 3.6: Junk Word / Not a Word panel.

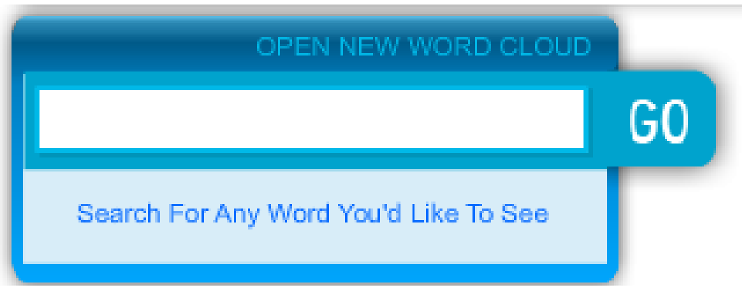

Figure 3.7: New Word panel.

has decided to associate it with the word *sweet*.

When they press enter or click go, I send the text *sweet* and the ID number of *awesome* to the server, to create a new association. The user's word may or may not be in the datastore, and the two words may or may not already be associated. In Table 3.1, I provided the structure for the Synset from the local data store, and in this case, *awesome* already exists, but *sweet* does not yet have an entry. Thus, I must create one.

The process of creating Synset objects is somewhat complex and encompasses a number of classes in the python data model {which are explained below} so I will summarize with the

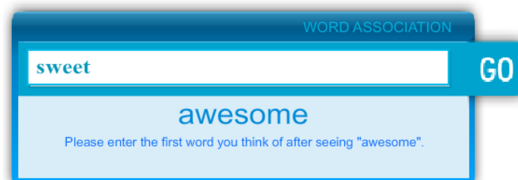

Figure 3.8: Word Association panel.

Table 3.1: Synset of awesome

Association strength	Associated to	Junk	Misspelled	ID #	Word
1	5	0	0	0	word
1, 2, 1, 1, 2, 1, 1	0, 6, 7, 9, 11, 12, 10	0	0	5	totally wild
2, 3, 2	5, 7, 8	0	0	6	savage
3, 1	6, 5	0	0	7	feral
2	6	0	0	8	beastly
1, 3	5, 10	0	0	9	really neat
3, 1	9, 5	0	0	10	awesome
2, 1	5, 12	0	0	11	without control
1, 1	11, 5	0	0	12	uninhibited

following pseudo code:

Algorithm 1 SynSet Association

SynSet *user_word* ← *txt*

SynSet *control_word* ← *id*

if ! *user_word* **then**

create and save a new SynSet for *user_word*

end if

link *user_word* and *control_word* to each other

Return *control_word*

Note, I am using ECMAScript-style pseudo code for data type clarity. All server code is actually written in python.

3.2 The Algorithms and the Architecture

In this section, the algorithms and the architecture of the application are presented.

3.2.1 The Algorithm

The algorithm is based on the idea of outsourcing, which is a popular trend in the business management. In this algorithm, all primary operators are outsourced. In a way, humans are

delegated to do what computers have been doing in other similar applications. This totally outsourced algorithm uses both evaluation and innovation abilities of humans. Therefore, using the business concept of outsourcing is natural here, as outsourcing takes place when an organization transfers the ownership of a business process to an external service. In other words, if a system is outsourcing a service then it is choosing an external agent to fulfill that particular service.

Here, one outsourced agent is a brain of a human-being. If the brains of the humans are considered as the processors in a distributed system, each of these brains can perform a small part of a gigantic computation. The advantage of this approach is its ability to address the complex ontology population problem that computers cannot overcome on their own.

The approach that this application uses, in a way, encourages the creative potential of users in the form of creation and evaluation. In Words-Around application, all but the core operators are outsourced and delegated to human-beings. The core operator just defines the rules for how the human participation will flow.

The mechanism provided by the computer is:

1. **Initial setup** User enters to the site;
2. **Rules** User is assigned to be an associator and she is prompted a word;
3. **Final condition** User produces the output, which is an association between the prompted word and the entered word.

As it is seen, the algorithm behind the association mechanism depends on the user contribution. The more words added, the better the system works. Words have their strength points, in other words, weights.

Consider the situation in Figure 3.9, the prompted word is the same (i.e. input_x) for all users. If all the outputs provided by the users are the same (i.e.

$$\forall_{i=2}^{i=n}(output_{1,x} = output_{i,x})$$

for input_x) then the association between input_x and output_x has the most strength points and on the other hand, as it can be inferred, the association between the input and the output that

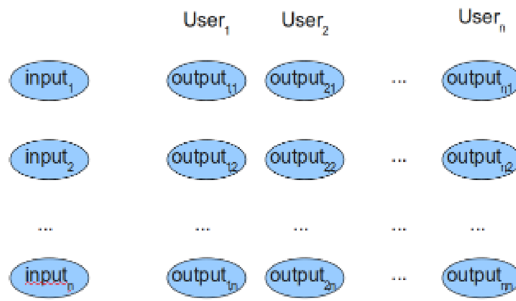

Figure 3.9: Logic behind word associations

is provided by only one user (i.e.

$$\forall_{i=2}^{i=n}(output_{1,x} \neq output_{i,x})$$

for input$_x$) is the weakest and has the least strength points.

3.2.2 The Architecture

The architecture of the applications is not complex. It is a 3-tier application. The users are interacted with a *Web Client* that is written in Flash and Python. The web client then talks with the *Web Application* which is located on Google's servers. If anything related to database happens (and almost all of the time it happens), then it communicates with the *Data* tier where the Datastore is located on.

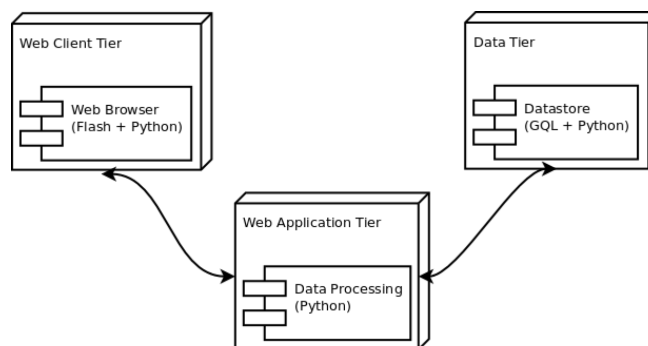

Figure 3.10: 3-tier architecture of the application

3.2.2.1 Class Diagram

The class diagram of the application can be seen at Figure 3.11.

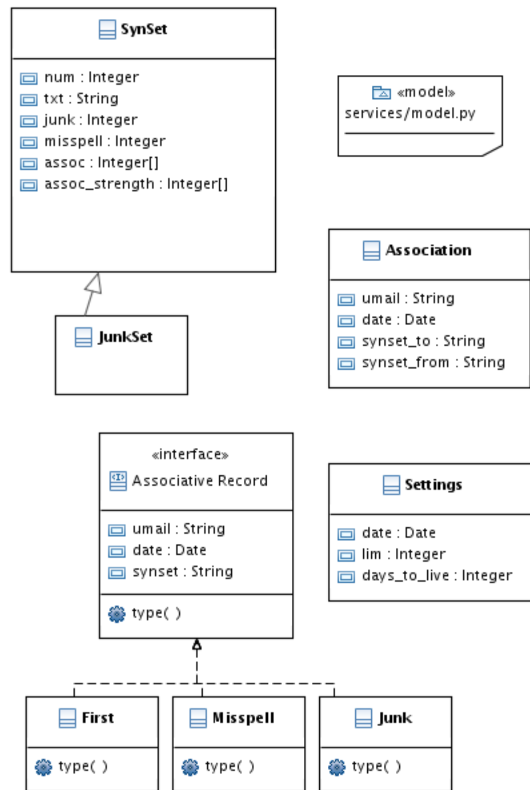

Figure 3.11: Class diagram

3.2.2.2 Association Overview

The main mechanism of the Words-Around Application is the word association ability. The overview of transactions while associating words of this ability can be seen at Figure 3.12 .

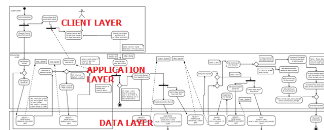

Figure 3.12: Overview of the transactions between layers while associating words.

Now, I continue with the detailed description of the architecture.

3.2.3 Service Layer

On the server side, Words-Around runs on the Google Appengine python framework, and communicates to the Flash Actionscript 3 client front-end using the Gaeswf / PyAMF library for Remote Procedure Calls {RPC}. I use custom python to interact with an administrative front end written in Google Web Toolkit, but extract all the core data management into the services.* python classes. Every method of every .py file in the services directory becomes instantly available to be called as a remote procedure within the actionscript front end, and through proxy wsgi (*Web Server Gateway Interface*) pages for the GWT front end. I will first explore the data types themselves, then detail how the RPC mechanism works, and finish off with the RPC methods, what they do, and how the flash client responds.

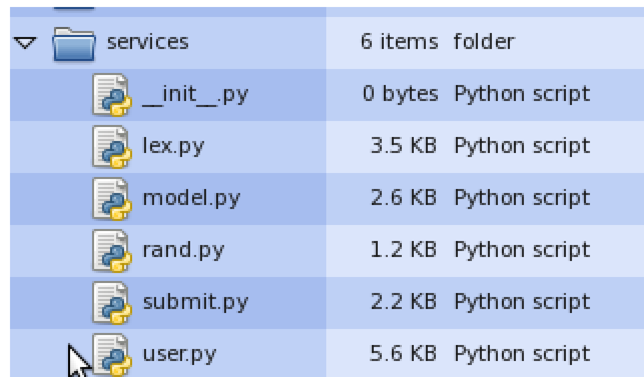

Figure 3.13: Files that are in the service layer.

As shown in Figure 3.13, these six files make up the entire service layer; __init__.py is only here to satisfy python import statements; model.py defines the Appengine datastore class objects, which are used in the remaining four files to submit word associations, track user data, request word data, etc.

3.2.3.1 Model

Every class within the model file extends google.appengine.ext.db.Model, either directly or by extending one of the other model classes.. This allows the classes to be persisted to and retrieved from the BigTable database on the Google Cloud server.

I will explore each class of the data model individually, with notes about how it is used in the

application. Once I have gone over each data type, I will explore how and where they are used in the Service Layer, then I will show how the Flash client uses these functions in production mode. To begin, I will explore the UserProfile object, as it is a simple, atomic object that does not depend on any other objects.

User Profile The UserProfile class is used to add the own metadata to the basic information provided by the com.google.appgengine.api.users.user object. The base Google object, which I use as a primary key, is made available by using Google Accounts Authentication. When a user is logged into the application, I can access the user's email address and nickname. The UserProfile class, that is shown in Figure 3.14 is created to allow us to add more fields to this basic data, and give users a more personalized experience.

```
8   from google.appengine.ext import db
9   from google.appengine.api import users
10
11  #An extended user object that supports the Google User object.
12  #The UserProfile class is leftover from the original Gaeswf model,
13  #which words-around uses.  We preserve it in case we use it later.
14 - class UserProfile(db.Model):
15      user = db.UserProperty()  #A Google user object we can query on.
16      name = db.StringProperty() #A site-personalized user name
17      url = db.LinkProperty() #A url link in case the user has a website
18      description = db.StringProperty() #Some biography text
19      createdAt = db.DateTimeProperty(auto_now_add=True) #creation time
20      modifiedAt = db.DateTimeProperty(auto_now=True) #modification time
```

Figure 3.14: UserProfile class.

SynSet The SynSet or Synonym-set object that is shown in Figure 3.15 is a composite object of all the data surrounding a particular word. Each SynSet has a text id and an integer id, the text id being the lowercase spelling of the core word in the SynSet, and the integer id which is used to reduce memory bloat and prevent programming language index errors when I need to use a SynSet id as a key in a dictionary / native object. For every word in the database, I store multiple types of data: What words is SynSet the associated too? How many times has each particular association been created? How many times has it been labeled junk or misspelled?

I store individual data objects for each specific kind of information, and then asynchronously update the list of composite SynSet objects with the weights and values of all relevant meta data that needs to be used on the client.

```
22  #The word SynsSet is short for Synonym Ring / Set.
23  #This object represent a single word, as well as metadata for the word.
24  #Most importantly, it stores an ordered list of associations to other words,
25  #As well as an array of counters, which shares its indices in the same order
26  #as the association list.  It also has counters for junk / misspell markings.
27  class SynSet(db.Model):
28      txt = db.StringProperty(required=True )#String id, word value
29      #numeric id, so references use ints
30      num = db.IntegerProperty(required=True )
31      #The ordered list of numeric ids this Synset is associated to
32      assocs= db.ListProperty(long)
33      #An array of counters for each association.
34      assoc_strength= db.ListProperty(long)
35      #A counter for misspellings; updated by cron.
36      misspell = db.IntegerProperty(default=0)
37      #A seperate convenience counter for junk as well.
38      junk= db.IntegerProperty(default=0)
39      #A convenience method to return the sum of all strengths.
40      def weight(self):
41          return sum(self.assoc_strength)
```

Figure 3.15: SynSet class.

To clarify, the assocs array stores integer pointers to every associated word, and the assoc_strength array stores how many times the particular index has been associated. For example, suppose I had the following three SynSets:

- A = { txt = 'ehh' , num = 1 , assocs = [2,3] , assoc_strength = [1, 2] }
- B = { txt = 'bee' , num = 2 , assocs = [1] , assoc_strength = [1] , misspell = 1}
- C = { txt = 'see' , num = 3 , assocs = [1] , assoc_strength = [2] }

You see that the first object is associated to the second and third objects, and that the other two are not associated to each other. The first object, A, has assocs.length == 2, and the strength

of each association is stored in the same order. Looking at B and C, I see that they have the same association strength to A as A does to B and C. I also through in an a misspell counter so I can continue to use these three example objects to explain how and why the rest of the data model works the way it does.

JunkSet Whenever I delete a SynSet, or one is autodeleted by having too many misspell and junk flags, I want to hold onto the data in case I want to implement an auto-delete functionality. I do this by creating a new class (which is a new table in the database) which extends the SynSet object and adds a boolean field so I can determine whether the deletion was automatic or done by an administrator. Details of this class can be seen at Figure 3.16.

```
43  #The JunkSet class extends the SynSet class with a single boolean property.
44  #The autojunked flag is used to determine how a SynSet was deleted.
45  #When a word is deleted from the datastore, it's value is copied to a JunkSet.
46  #This will allow us to undelete words.
47  class JunkSet(SynSet):
48      #Whether or not the SynSet was junked manually by an administrator.
49      autojunked = db.BooleanProperty(default=True)
```

Figure 3.16: JunkSet class.

Association An Association object is created for every link between two words. Whenever I create an Association record, I also update both corresponding SynSets for faster response times. Users query this object to populate their recent activity list, and I may be able to use this to trace system spamming to a particular user. Details of this class can be seen at Figure 3.17.

First Whenever a user submits an associative word link, I try to retrieve that object from the datastore, and when it fails I have to create a new SynSet. When this happens, I also create a First record so the user can track which words they were the first to think of. This data is displayed in the user's recent activity list. Details of this class can be seen at Figure 3.18.

```
51  #In order to display a record of associations created by a user,
52  #We create an Association object every time a user sends us a link.
53  #Note that we also update the assocs array in SynSet to prevent excess db calls.
54  #Appengine does not allow joins, so we keep copies of every Association object,
55  #Which is a very long list, AND we adjust the assocs and assoc_strength arrays.
56  #The Association table size == The sum of all SynSet.assoc_strength[]  / 2
57  class Association(db.Model):
58      #user email id, from google user object
59      umail = db.StringProperty(default='')
60      #We store the date to sort on for use in client / admin UIs.
61      date = db.DateTimeProperty(auto_now=True, required=True)
62      #The text id of the word linked to.
63      synset_to = db.StringProperty(required=True)
64      #Linked from id.  We don't store int id here to prevent excess db calls.
65      synset_from= db.StringProperty(required=True)
```

Figure 3.17: Association class.

Misspell The Misspell class that is shown in Figure 3.19 is used to track and store which words were marked as misspelled, by which user at what time. It has exactly the same field types as model.First, so I simply extend the object and add an overriding method named type() to maintain correct python syntax (every extended object must have at least one extra field or some kind of method to be compilable syntax). Unlike the First object, which is only created if the SynSet does not exist, I create a new Misspell object for each and every misspell submisstion. I do not delete these records when a word is deleted, so I can mark profanity and unacceptable language with many misspelling records so it will be autodeleted any time it is created.

Junk The Junk class that is shown in Figure 3.20 is exactly the same as the Misspell class, except it denotes words that are such garbled nonsense that there is no need to try to correct them into proper associations. I mark this difference so I can implement the 'spell check' described above without wasting dictionary entries on garbage.

Settings The Settings class that is shown in Figure 3.21 is a datastore object that I use to change application behaviour without having to modify source files and redeploy to Ap-

```
67  #The very first time a word is submitted,
68  #We want to store a special, singleton record of that word.
69  #The flash client then requests this list on a per-user basis.
70  #The First object is used to track user id, the time and the text id
71  #of the word.  We do not use int ids for fast translation to the client.
72  class First(db.Model):
73      umail = db.StringProperty(default='')#user email id.
74      date = db.DateTimeProperty(auto_now=True, required=True)#timestamp
75      #text id of marked synset; String is used to prevent db lookups from int ids.
76      synset = db.StringProperty(required=True)
77      #This class gets extended, and we override this method to keep python happy.
78      def type(self):
79          #This will probably never be used.
80          return 0;
```

Figure 3.18: First class.

```
82  #Misspell records have exactly the same data format as the First class,
83  #Except we store many of these for every time a user hits the Misspell button.
84  #We extend First and add a method to keep python syntax correct.
85  #When we create a Misspell record, we can choose whether or not
86  #to allow multiple records for certain users {if (user.isadmin()?}.
87  class Misspell(First):
88      def type(self):
89          return 1;
```

Figure 3.19: Misspell class.

pengine. The lim field is used by the Flash client to limit the number of words displayed in a cloud before hiding the lowest strength associations. days_to_live is how many ways to delete a SynSet after its misspell + junk counters ¿ sum(assoc_strength). For details, see the Deathklok class below.

Deathklok Whenever a new Misspell or Junk record is created, I schedule a recheck of that word for autodeletion. All the Association, Misspell and Junk records are counted and updated in the SynSet record, then the sums of these opposing weights are calculated, and if there is more bad than good, a single Deathklok object is created {unless it already exists}

```
91  #The Junk class is another extension of the First class.
92  #It is essentially the same as Misspell, but it doesn't
93  #warrant admin attention to possibly fix a misspelling.
94  class Junk(First):
95      def type(self):
96          return 2;
```

Figure 3.20: Junk class.

```
98  #The Settings class represents a singleton instance in the datastore.
99  #We insert these values into a user's login response to initialize
100 #the client Flash application with values we may want to tweak,
101 class Settings(db.Model):
102     #This is our primary key, we order by date and .get() to fetch singleton.
103     date = db.DateTimeProperty(auto_now=True, required=True)
104     #The number of words to display in a word cloud before hiding some.
105     lim = db.IntegerProperty(default=15)
106     #How many days will we wait before auto deleting a word / SynSet?
107     days_to_live = db.IntegerProperty(default=11)
```

Figure 3.21: Settings class.

to note that the object is to be autodeleted. The death field is a date that is set to now + Settings.days_to_live * day / millisecond. When the cron jobs checks a Deathklok that is past it's expiry date, it will recount the records and autodelete the object if the total Associations is less than the sum of Misspell and Junk. The algorithm behind this deathklok mechanism can be seen at Figure 3.22.

This ends the data model of Words-Around, so I will continue the explanation of the Service Layer with the various Remote Procedure Call methods, on both the server and client side. I will start with getting Synset data from the server, and then move on to flagging junk and misspells, then to submitting new associations. For low level details of RPC mechanics, please refer to Appendix A.

SERVER LAYER
{Python}

DATA LAYER
{GQL}

Get SynSet to operate on

id : num

The id is supplied as a parameter to /junk, which is where this process is mounted

(SELECT * FROM SynSet WHERE num = id) .get()

syn : SynSet

Update Junk Count

(SELECT keys_only FROM Junk WHERE synset = syn.txt) .count()

syn.junk =

Update Junk Count

(SELECT keys_only FROM Misspell WHERE synset = syn.txt) .count()

syn.miss =

Save SynSet Updates

syn.put()

If the SynSet has more total associations than garbage count, Stop processing. SynSet.weight() returns the sum of SynSet.assoc_strength

Check if syn.weight() < syn.junk + syn.miss

Note, when we autodelete a SynSet, we wait for a set period of days after junk + misspell > weight() before we delete the record. This gives words a chance to be saved from the Deathklok.

Check for an existing autodelete timer

(SELECT * FROM Deathklok WHERE txt = syn.txt) .get()

autodelete : Deathklok

Check Deathklok State

Make new Deathklok

Test autodelet Deathklok

Check date.today > autodelete.death

Note, when deleting a SynSet, we do not actually call syn.delete() here; instead we use taskqueue to defer a call to /rm, which is an entire process that handles SynSet deletion.

Delete Deathklok

autodelete.delete()

Copy SynSet to JunkSet

JunkSet(num = syn.num txt = syn.txt assoc=syn.assoc assoc_strength = syn.assoc_stregth) .put()

Delete SynSet

syn.delete()

Get days_to_live from datastore Settings

(SELECT * FROM Settings ORDER BY date) .get().days_to_live

timeout : Integer

Save new Deathklok

Deathklok(txt = syn.txt death = date.today() + timedelta(days=timeout)).put()

Figure 3.22: Algorithm of Deathklok.

3.2.3.2 Lex

I shall begin the detailed inspection of the service methods with the methods in services.lex.* This is where I perform get queries for the various model data types. I use it to retrieve SynSet, Association, First, Junk and Misspell methods. I will start with the First, Junk and Misspell methods as they share the extend the same data model class, and also share a single lookup method, monorec(). These are used to populate the Word Association List for logged in users to view their activity.

First, the three methods that are used in RPC. Each one supplies the class object they query to a single lookup function. Remember, all three of these classes extend First anyway, and they are all displayed in the client in the same way as well. The actual workhorse of these lookup methods is called monorec.

Because the Word Association List panel is hidden for users that are not logged in, I just return null to save time. This method should not even be called for users unless they are logged in, but I stay safe anyway.

I query for all objects of the supplied type and order them by date. Because BigTable generates all the indexes used, I never actually sort these results. They start sorted and stay sorted.

Next I will see how Flash uses this object.

3.2.3.3 Display

I use a single method to parse and display all three user record classes. There are nearly equivalent methods for the Association object on both server and client, but for the sake of brevity, they will be omitted. The only difference between the two is that instead of returning lists of String results, Association records use a dictionary object as follows: {'one': 'first word' , 'two' : 'second word'}.

All of these records are used to create html links that I add to the Word Association List panel. Clicking #first word causes the application to show the 'first word' cloud, and set the suggested word to associate in the Word Association panel. This allows users to suggest multiple associations for a specific Synset by opening it multiple times to submit multiple

words. I may want to check for users who spam a specific Synset with the same word (by checking for an existant Association object by Synset txt and user id).

The remainder of the lex.py service methods return SynSet records, which is done in two different ways. lex.lemma accepts a text id, and called to open a new word cloud. It returns a raw SynSet object that is then used to repeatedly call lex.synset to load all of the associated words into the cloud.

Note that if a user requests a word I do not have in the database, I create a new SynSet and a First record of this submission. The client will then set the Word Association dialog value to the new SynSet so they can also suggest possible links to that word.

Once the client has a SynSet from the lex.lemma method, they will have lists of numeric ids and association strengths, but they will need the SynSet text and any cross-cloud links to build the word cloud. They must send the desired number id, the strength of the association (to avoid extra lookups on the client), as well as the parent id from the lemma request. I must pass this data through so I can check that the lemma is still open before adding new associative links.

The process of creating a new word association link is complex, as I must perform lots of data integrity checks, create and get many datastore entries and update word association strengths before returning the control SynSet (the suggested word) to the client.

I need the users logged in email to create Association and First records later on in the method. ”text@example.com” is used for anonymous user submisstion.

I first try to get the user suggested SynSet using a GQL query object, using the String id as the lookup key.

When the SynSet is not yet present in the database, I create one, as well as a special First record to let the user know they were the first to suggest the word. This will be especially useful for detecting misspellings once most of the common words are in the datastore.

When I get the control SynSet, I also want to check to make sure it is not null. This can only happen if I delete a record in the time between when a user is given a control word, and when the new association is submitted.

In the case that the suggested word does not exist, I just return a random SynSet id for a new control word.

Once I have retrieved or created both SynSets, I must increment both assocs and assoc_strength arrays using the numeric ids.

When a SynSet object is sent to the ink() method, I must first ensure that the numeric id of the associated SynSet is in the assocs array. In the case that it is, I increment the correct index in the assoc_strength array. When it's not there yet, I simply push the id and the weight 1 onto the associativate arrays.

Every time I create an associative link, I store a record of both text ids, the user's email and the current time in an Association record

There is a wsgi page mounted at /sort which takes a SynSet id for a parameter, and sorts the word association arrays based on descending association strength. This would be wasteful to do now, while the user is waiting for returned data, so I use the Google taskqueue API to queue up a task. The /sort method also counts Junk and Misspell records, and potentially schedules a SynSet for autodeletion.

The final step is to return a new SynSet id (text and numeric) to load as a control word. I also return the old text id to tell the client they should load the updated word cloud. When the control word does not exist, I exclude this variable to prevent requests for SynSets that no longer exist.

The wsgi page at /sort forwards a SynSet id for data integrity procressing. This could be used by RPC, but I do not bother.

I have to sort two arrays at the same time, so I use a proxy list that is ordered by the strength of each association. Each object in the proxy list is itself a list of every id that has the current strength value. I must iterate through all associations twice, thus I avoid calling this inside methods that clients must wait for.

Once I have filled the proxy object with the association data, I want to reset the SynSet arrays to all zeroes. Note I set the lim variable again. This is to avoid an elusive python error.

I now iterate through the proxy object's list of lists, and push the sorted data onto the SynSet

object's associative arrays in order of descending strength. I take this opportunity to recount the Junk and Misspell counts on the Synset object.

Finally, if the SynSet has a higher garbage count than the total strength of associations, it is scheduled for autodeletion at /junk.

3.2.3.4 User

The only method used in user.py is the login method, which does a number of important things. It checks whether the user is already logged in or not, generates special login / logout urls to use Google Accounts authentication, gets or creates a UserProfile object for logged in users, and sets any global flash variables that I want to control in the datastore Settings class. For now, the only setting I use is the limit value for the number of associations to show {in descending order of association strength}. I will first investigate the login procedure itself, and then show how it is used as a Remote Proecudure Call within the flash client. I will show the basic setup needed to implement RPC, and explain the base classes used by Gaeswf.

The login procedure starts as simple as it gets; check for a logged-in user, and validate the parameters.

The next step is to create response object. A native python dictionary object is translated directly to a flash native object during RPC marshalling, so all the packing / unpacking overhead is never repeated on the client side.

I want to control the default initialization parameters with variables in the datastore. This allows us to tweak settings without having to recompile the .swf. To add more settings, just extend Settings to include the new fields, then add them to this response object and do something with them in the flash client. I will explore this process in greater detail shortly.

If the user object is not equal to none, the client is already authenticated with Google. I want the response object to include only the relevant metadata, which I store in the UserProfile class.

Since I do not need to copy all the data from the user object to the client, I create a dictionary object in user_vo.profile with only the field values I might use.

Finally, the system returns the response object to be translated to RPC wire format and sent to the client.

Now, I will show the basic set up requirements to use Gaeswf RPC in flash.

The first and most easily missed step is to include the javascript dependency, swfaddress.js. This is a small javascript library that parses history token changes and calls a notification method in javascript Within the flash client, Gaeswf uses the Flash ExternalInterface to register an Actionscript method to listen in on these functions.

The SWFAddress module contains the .js controls to access basic browser history functionality (reading / setting tokens, going forward or back and listening for history changes), as well as a Flash Actionscript file that wraps all this functionality into static Flash methods. I use this so I can generate urls as html anchors inside of flash and use the swfadress event listeners like global catch-all onclick methods for link.

3.2.3.5 BaseView

In Flash Actionscript 3, all .swf instances use a root MovieClip instance, defined in the source .fla. In the application, it is com.wordgame.Application, which ultimately extends org.gaeswf.flash.BaseView. This global root is available to all attached MovieClips on their own .root field, and thanks to the Gaeswf library, I also have a private static copy at Application.rpc.

The base class for all rpc-enabled applications is BaseView. It defines a single convenience method, execute, which takes one or more strings as parameters. The serviceName parameter is a two-part string, seperated by a period. "user.login" corresponds to the python services/user.py file, and login corresponds to the method in that file. Whatever strings are included in the args list become the string parameters to the python method on the server. The actual mechanics within org.gaeswf.Service go beyond the scope of this article, and include all the event listenning, error handling, parameter marshalling and response unmarshalling required to talk to python. They interact with the PyAMF module that is a server-side dependency of the Gaeswf library.

About the application's design details regarding the visualisation please refer to Appendix A.

3.3 The Results

In this section, the experimental results of the proposed method is represented.

3.3.1 User Feedback

I collected evidence showing that Words-Around is fun to play around and people provide correct word associations. Words-Around should be designed to be enjoyable and it is, indeed very enjoyable. I have collected so much feedback from the users regarding its addictive and entartaining structure.

Some users claimed that, they just wanted to take a look at the site but stayed for over 20 minutes while playing with the word clouds and as well as associating more than 200 words.

In one week since its debut to the small community of the Department of Computer Engineering at METU, Words-Around was able to collect more than 8,000 words and 20,000 associations. Considering this fact that it would only take quite a few months to collect an excessive number of associations.

3.3.2 Statistics

The throughput of human computation applications is defined by Luis von Ahn [42] *et. al.* as the average number of problem instances solved, or input-output mappings performed, per human hour. This calculation in Words-Around is done by examining the number of individuals logged into the site using their Google accounts.

Table 3.2: Results of Words-Around in 1 Week

Entity Type	**Amount**
Association	20,365
SynSet	8,290
Unique User	104
Misspell count	66
Junk count	32
Deathklok count	3

Figure 3.23: Associations of *dark*.

The time intervals between log in and log out dates of each individual have been calculated and divided into the number of associations made. Roughly said, it takes a minute for a user to associate 9 words.

3.3.3 Foreign language issues

However, there are some issues about the meaningfulness of the associations. Some words from foreign languages are used in Turkish. Also, English is not the native language of the users who have tested the application, as the testers of the application are mainly the students of the Department of Computer Engineering at METU. Therefore, upon seeing a word, they sometimes think as if it is a Turkish word and associate it with another Turkish word.

For example, in Figure 3.23 upon seeing *dark* on the screen, a user has entered *bira*. As it is known, *dark* is also used in Turkish, mainly to describe the kind of a beer that is dark-colored and heavy-bodied. Even, another user, after he was prompted with *dark*, entered *efes* as the association. From this outcome, it can be said that the reliability of the data mainly depends on the linguistic abilities of the users.

3.3.4 Comparision with WordNet

When Words-Around is compared with WordNet [12], it can easily be seen that, the association clouds in Words-Around are not complete as WordNet. There are many words associated to each other but English is a huge language. In Oxford English Dictionary [34], which is accepted as an authority on English, there are over a half million words. WordNet covers the about one third of this number, and this is quite a big number, also. However, there are also words in Words-Around which are not included in WordNet, too. For instance, WordNet does not include the -ing form of a verb unless it is used as an adjective. Moreover, Words-Around is more dynamic then WordNet. Even though WordNet is being developed continuosly, its development speed is less then Words-Around because more users are contributing to Words-Around.

CHAPTER 4

CONCLUSION AND FUTURE WORK

4.1 Conclusion

I have presented Words-Around, a game to collect word associations. I have shown data indicating that Words-Around is enjoyable and it produces correct data. Although it has been only a week since the release of the game, I have collected a great number of associations from a just few players compared to the vast Internet population. The major contribution of this application is the rapid ontology population abilit that it presents.

Words-Around is insipired by Luis Von Ahn's ESP Game [41] as it also considers the brains of human beings as processors in a distributed system [40]. It benefits from the fact that ontology population is a trivial task for human beings but continues to challenge even super computers. Its purpose is to channel the combined mind power of Internet users to populate ontologies. Populating a colossal ontology that will be in use of the civilization would be a glorious achievement and I believe, Words-Around will be immensely positive in doing so.

In summary, in this thesis, the goal is to develop a web-based application that forms a user-friendly environment that channels the vast Internet population to provide data towards solving ontology population problem that no known efficient computer algorithms can yet solve.

4.2 Future Work

As this application evolves, I intend to add a translation mechanism so bilingual users can provide translation maps, and pupils of new languages can access secondary language phrases submitted by native speakers.

It is possible to change the color scheme / theme of the flash client UI by using ColorTransforms on the existing MovieClip objects. I can do this by adding three integer fields for each RGB color I wish to transform. Recommended transforms are for background color, text color and foreground colors. I may also use the existing url field to point to a background image / watermark that users can select to add some eye candy where the plain white background once stood. I may also need to store the user's preferred languages in this object, once translations have been implemented.

To implement translations of SynSets, I may introduce the following SynSet field:

- `db.ListProperty(String, default=[], required = False)`

Then, I can push translations of this word to that list, with *EN_*, *FR_* or *ETC_* prefixes to denote the language of the translation. This way, the client can pick and choose which translation SynSet to lookup. Alternatively, I could just create a new data class that users can query as needed. This second approach will be more efficient, and help keep response times to an absolute minimum.

I may wish to create an extension called JunkAssociation, and move junk objects to and from the different table to help speed up queries and avoid wasting datastore diskspace.

I may want to add a utility to the GWT front end (and maybe the flash one as well) for users to submit corrections for a word, and copy all the misspelling objects into association objects for the correct word, and thus preserve the association strength of misspelled words. If I do this, I may also wish to store misspelling translations in a new class, and use them to generate "did you mean..." suggestions for quick correction.

I currently interpret all history changes as a request to open a word cloud with the text content of the token. If I use special syntax, such as prefixing a token with a restricted symbol, I can test for this case whenever I am notified of a urlChange. For example, to generate a list of possibly misspelled / junk words from the visible cloud of words, I can create a list of links like: from the stored global SynSet for the visible cloud. Since *?* is not allowed in a SynSet text, I can test for it in urlChange, and instead of opening a new cloud, I submit a misspell/junk notification (using a different prefix for each). Generating html in flash is much more efficient than manually creating MovieClips and registering hordes of inline event listener functions to

```
<a href="#?txt">txt</a>
```

Figure 4.1: URL Change

call the rpc manually.

Any new RPC methods defined in the python source folder services/ must be registered com.wordgame.Model. The syntax is *python_file_name.python_method*. The parameters to these methods must be zero or more strings, as all data must be serialized for http transfer. Note that I do NOT define the parameter maps, and simply assume that the client code will supply the correct number of String arguments to avoid errors.

Any new data types can be added to model.py for consistency. Any data management functions should also be placed in one of the existing Service Layer files, or a new .py file should be created in this folder. This will ensure availability to Flash RPC and the GWT control panel.

REFERENCES

[1] A. Almuhareb and M. Poesio. Attribute-based and value-based clustering: An evaluation. In *Proceedings of Empirical Methods in Natural Language Processing (EMNLP)*, pages 158–165, Barcelona, Spain, 2004.

[2] N. Aswani, V. Tablan, K. Bontcheva, and H. Cunningham. Indexing and Querying Linguistic Metadata and Document Content. In *Proceedings of Fifth International Conference on Recent Advances in Natural Language Processing (RANLP2005)*, Borovets, Bulgaria, 2005.

[3] A. Balkan. The gae swf project. http://gaeswf.appspot.com/, last visited on January 2010.

[4] K. Bontcheva, H. Cunningham, D. Maynard, and V. Tablan. Annie. http://www.aktors.org/technologies/annie/, last visited on January 2010.

[5] J. Chamberlain, M. Poesio, and U. Kruschwitz. A web-based collaborative annotation game. In *I-Semantics'08: Proceedings of the International Conference on Semantic Systems*, 2008.

[6] P. Cimiano, A. Pivk, L. Schmidt-Thieme, and S. Staab. Learning taxonomic relations from heterogeneous sources of evidence. *Ontology Learning from Text: Methods, Evaluation and Applications*, 2005.

[7] P. Cimiano and J. Volker. Text2onto. http://code.google.com/p/text2onto/, last visited on January 2010.

[8] P. Cimiano and J. Volker. Texttoonto. http://www.ohloh.net/p/texttoonto, last visited on January 2010.

[9] P. Cimiano and J. Volker. Towards large-scale, open-domain and ontology-based named entity classification. In *Proceedings of Recent Advances in Natural Language Processing (RANLP)*, pages 166–172, 2005.

[10] H. Cunningham, D. Maynard, K. Bontcheva, and V. Tablan. Gate: A framework and graphical development environment for robust nlp tools and applications. In *Proceedings of the 40th Annual Meeting of the ACL*, 2002.

[11] A. Dasdan, C. Drome, and S. Kolay. Thumbs-up: A game for playing to rank search results. In *18th International World Wide Web Conference (WWW2009)*, April 2009.

[12] C. Fellbaum, editor. *WordNet An Electronic Lexical Database*. The MIT Press, Cambridge, MA ; London, May 1998.

[13] M. Fleischman and E. Hovy. Fine grained classification of named entities. In *Proceedings of the 19th international conference on Computational linguistics*, pages 1–7, Morristown, NJ, USA, 2002. Association for Computational Linguistics.

[14] S. C. for Biomedical Informatics Research. Protege. http://protege.stanford.edu/overview/, last visited on January 2010.

[15] R. Girju, A. Badulescu, and D. Moldovan. Automatic discovery of part-whole relations. *Comput. Linguist.*, 32(1):83–135, 2006.

[16] Google. Datastore. http://code.google.com/appengine/docs/python/datastore/, last visited on January 2010.

[17] Google. Google app engine. http://appengine.google.com/, last visited on January 2010.

[18] Google. Google image labeler. http://en.wikipedia.org/wiki/Google_Image_Labeler, last visited on January 2010.

[19] T. R. Gruber. Toward principles for the design of ontologies used for knowledge sharing. *International Journal of Human-Computer Studies*, 43:907–928, 1993.

[20] N. Guarino. Concepts, attributes and arbitrary relations. *Data Knowledge Engineering*, 8:249–261, 1992.

[21] C.-J. Ho, T.-H. Chang, J.-C. Lee, J. Y.-j. Hsu, and K.-T. Chen. Kisskissban: a competitive human computation game for image annotation. In *HCOMP '09: Proceedings of the ACM SIGKDD Workshop on Human Computation*, pages 11–14, New York, NY, USA, 2009. ACM.

[22] I. Horrocks, P. P. Schneider, and F. van Harmelen. From shiq and RDF to OWL: The making of a web ontology language. *Journal of Web Semantics*, 1(1):7–26, 2003.

[23] P. Kogut. Aerodaml: Applying information extraction to generate daml annotations from web pages. In *First International Conference on Knowledge Capture (K-CAP 2001). Workshop on Knowledge Markup and Semantic Annotation*, 2001.

[24] D. Lin. Automatic retrieval and clustering of similar words. In *Proceedings of the 17th international conference on Computational linguistics*, pages 768–774, Morristown, NJ, USA, 1998. Association for Computational Linguistics.

[25] D. Lin. Automatic retrieval and clustering of similar words. In *COLING-ACL*, pages 768–774, 1998.

[26] H. Ma, R. Chandrasekar, C. Quirk, and A. Gupta. Page hunt: using human computation games to improve web search. In *HCOMP '09: Proceedings of the ACM SIGKDD Workshop on Human Computation*, pages 27–28, New York, NY, USA, 2009. ACM.

[27] D. Maynard, Y. Li, and W. Peters. Nlp techniques for term extraction and ontology population. In *Proceeding of the 2008 conference on Ontology Learning and Population: Bridging the Gap between Text and Knowledge*, pages 107–127, Amsterdam, The Netherlands, The Netherlands, 2008. IOS Press.

[28] D. Maynard, V. Tablan, H. Cunningham, C. Ursu, H. Saggion, K. Bontcheva, and Y. Wilks. Architectural elements of language engineering robustness. *Nat. Lang. Eng.*, 8(3):257–274, 2002.

[29] D. Morrison, S. Marchand-Maillet, and E. Bruno. Tagcaptcha: annotating images with captchas. In P. Bennett, R. Chandrasekar, M. Chickering, P. G. Ipeirotis, E. Law, A. Mityagin, F. J. Provost, and L. von Ahn, editors, *KDD Workshop on Human Computation*, 2009.

[30] E. Motta. Neon. http://www.neon-toolkit.org/, last visited on January 2010.

[31] R. Navigli and P. Velardi. Learning domain ontologies from document warehouses and dedicated web sites. *Comput. Linguist.*, 30(2):151–179, 2004.

[32] P. Pantel and M. Pennacchiotti. Automatically harvesting and ontologizing semantic relations. In *Proceeding of the 2008 conference on Ontology Learning and Population: Bridging the Gap between Text and Knowledge*, pages 171–195, Amsterdam, The Netherlands, The Netherlands, 2008. IOS Press.

[33] M. S. Paul Buitelaar. Ontolt - middleware for ontology extraction from text. http://olp.dfki.de/OntoLT/OntoLT.htm, last visited on January 2010.

[34] O. U. Press. Oxford english dictionary. http://www.oed.com/about/, last visited on January 2010.

[35] M. Ruiz-Casado, E. Alfonseca, M. Okumura, and P. Castells. Information extraction and semantic annotation of wikipedia. In *Proceeding of the 2008 conference on Ontology Learning and Population: Bridging the Gap between Text and Knowledge*, pages 145–169, Amsterdam, The Netherlands, The Netherlands, 2008. IOS Press.

[36] S. Schlobach, M. Olsthoorn, and M. D. Rijke. Type checking in open-domain question answering. In *In Proceedings of European Conference on Artificial Intelligence*, pages 398–402. IOS Press, 2004.

[37] K. Siorpaes and M. Hepp. Games with a purpose for the semantic web. *IEEE Intelligent Systems*, 23(3):50–60, 2008.

[38] U. S. U. Hustadt, B. Motik. Kaon2. http://kaon2.semanticweb.org/, last visited on January 2010.

[39] R. Volz, Y. Sure, R. Studer, D. Oberle, S. Handschuh, A. Hotho, and S. Staab. Kaon - the karlsruhe ontology and semantic web tool suite. http://kaon.semanticweb.org/, last visited on January 2010.

[40] L. von Ahn. Games with a purpose. *Computer*, 39:92–94, 2006.

[41] L. von Ahn and L. Dabbish. Labeling images with a computer game. *Proceedings of the SIGCHI conference on Human factors in computing systems*, pages 319–326, 2004.

[42] L. von Ahn and L. Dabbish. Designing games with a purpose. *Commun. ACM*, 51(8):58–67, 2008.

[43] L. von Ahn, S. Ginosar, M. Kedia, R. Liu, and M. Blum. Improving accessibility of the web with a computer game. In *CHI '06: Proceedings of the SIGCHI conference on Human Factors in computing systems*, pages 79–82, New York, NY, USA, 2006. ACM.

[44] L. von Ahn, M. Kedia, and M. Blum. Verbosity: a game for collecting common-sense facts. In *CHI '06: Proceedings of the SIGCHI conference on Human Factors in computing systems*, pages 75–78, New York, NY, USA, 2006. ACM.

[45] L. von Ahn, B. Maurer, C. Mcmillen, D. Abraham, and M. Blum. recaptcha: Human-based character recognition via web security measures. *Science*, pages 1465–1468, August 2008.

APPENDIX A

VISUAL DESIGN DETAILS

A.1 BaseView

In Flash Actionscript 3, all .swf instances use a root MovieClip instance, defined in the source .fla. In the application, it is com.wordgame.Application, which ultimately extends org.gaeswf.flash.BaseView. This global root is available to all attached MovieClips on their own .root field, and thanks to the Gaeswf library, an instance is stored at global root. I also have a private static copy at Application.rpc

```
import flash.display.MovieClip;

import org.gaeswf.Service;

public class BaseView extends flash.display.MovieClip
{
    // Execute: Makes a service call
    protected function execute(serviceName:String, ...args)
    {
        // Add the current
        args.unshift(this);
        args.unshift(serviceName);
        Service.call.apply(Service.call, args);
    }
}
```

Figure A.1: BaseView class.

The base class for all rpc-enabled applications is BaseView. It defines a single convenience method, execute(), which takes one or more strings as parameters. The serviceName pa-

rameter is a two-part string, seperated by a period. User.login corresponds to the python services/user.py file, and login corresponds to the method in that file. Whatever strings are included in the args list become the string parameters to the python method on the server. The actual mechanics within org.gaeswf.Service go beyond the scope of this article, and include all the event listenning, error handling, parameter marshalling and response unmarshalling required to talk to python. They interact with the PyAMF module that is a server-side dependency of the Gaeswf library.

Next, I will explore the BaseApplication class, which adds support for SWFAddress handling:

A.2 BaseApplication

```
import flash.net.URLRequest;
import flash.net.navigateToURL;
import flash.events.Event;

import lt.uza.utils.Global;

import com.mimswright.sync.Synchronizer;        // KitchenSyncLib

import org.gaeswf.flash.BaseView;

public class BaseApplication extends BaseView
{

    private var app:Global;

    function BaseApplication()
    {

        // Only access stage-related properties when added to stage.
        addEventListener(Event.ADDED_TO_STAGE, init);
    }
```

Figure A.2: BaseApplication class.

Note that I have excluded many of the import statements for brevity.

I extend the init() method in com.wordgame.Application, and I make sure to call super.init(event) when I do. This method does a few important things, like registering the application to listen to SWFAddress history token changes, sets a global instance of the root MovieClip in the public static lt.urza.util.Global.getInstance() dictionary, and initializes some important fields

```
// Override the init method to carry out initialization logic.
// You can do whatever you like here (with stage clips, etc.
// without worrying about triggering errors when this app
// is loaded into another one.)
protected function init(event:Event):void
{
    // Listen for SWFAddress updates.
    SWFAddress.addEventListener(SWFAddressEvent.INIT, handleSWFAddressInit);

    // Set up the KitchenSyncLib synchronizer.
    Synchronizer.initialize(this);

    // Using Uza's Global class.
    app = Global.getInstance();

    // Save the root in the global object
    // (where it _should_ be, guys!) and a few other convenience items.
    // (Don't abuse this.)
    app.root = root;

    var url:String = Global.getInstance().root.loaderInfo.url;

    // Is the app running locally?
    app.isLocal = (url.substr(0, 7) == "file://");

    // Save the base url (e.g. http://localhost:8080 or http://myapp.gaespot.com)
    // Notice: There is _no_ trailing space.
    app.baseURL = url.substr(0, url.indexOf('/', 7));
}
```

Figure A.3: BaseApplication class continued.

that determine how the application is to perform.

Figure A.4 shows the inital handler attached to the SWFAdress CHANGE event, and it calls the urlInit() method, which is not actually implemented in org.gaeswf.flash.BaseApplication. I override this method in com.wordgame.Application to test for an inital history token as soon as possible in the application load sequence. This will allow users to creates bookmarks of a word cloud, or send links to each other of a particular word cloud. Note that this listener overrides itself with a new listener, below:

The urlChange() method is extended in the global root class, com.wordgame.Application. It is used to listen for and respond to history token changes. I will not delve too deeply into this method just yet, as it ties in closely with an RPC method I will be exploring later.

I currently interpret all history changes as a request to open a word cloud with the text content of the token. If I use special syntax, such as prefixing a token with a restricted symbol, I can test for this case whenever I am notified of a urlChange(). For example, to generate a list of

```
//
// SWFAddress handlers and hook methods to override in the subclass.
//

private function handleSWFAddressInit(event:SWFAddressEvent):void
{
    urlInit();
    SWFAddress.addEventListener(SWFAddressEvent.CHANGE, handleSWFAddress);
}

// Override this to execute logic when the app's url becomes available
// (e.g. before calling GAE login.)
protected function urlInit():void
{
    trace("Warning: urlInit() not handled in Application class.");
}

// Important view states are mapped to URLs via SWFAddress.
protected function handleSWFAddress(event:SWFAddressEvent):void
{
    urlChange();
}
```

Figure A.4: BaseApplication class continued.

possibly misspelled / junk words from the visible cloud of words, I can create a list of links from the stored global SynSet for the visible cloud. Since ? is not allowed in a SynSet text, I can test for it in urlChange(), and instead of opening a new cloud, I submit a misspell / junk notification (using a different prefix for each). Generating html in flash is much more efficient than manually creating MovieClips and registering hordes of inline event listener functions to call the rpc manually.

This is a convenience method to open a url in the current browser window. It is syntactically equivalent to setting window.location = url; in javascript. I use this to open the login / logour urls for authentication, and to create history token changes. I could use the public static SWFAddress.href() method, but it introduces the unneeded overhead of calling through the SWFAddress ExternalInterface.

This concludes the relevant sections of org.gaeswf.flash.BaseApplication. Before I move on to the extension of this class, com.wordgame.Application, I will first explore com.wordgame.Model, which registers all the RPC methods I use, and defines the static RPC callback methods. com.wordgame.Application is cluttered enough without the addition of these methods and static fields.

```
// Get requested URL in the same browser window.
protected function getURL(url:String):void
{
    var request:URLRequest = new URLRequest(url);

    try
    {
        navigateToURL(request, '_self');
    }
    catch (e:Error)
    {
        trace("Error occurred while trying to navigate to " + url);
    }
}
```

Figure A.5: BaseApplication class continued.

A.3 Model

```
import org.gaeswf.Service;

public class Model
{
    //These are set to correct values on production server after the login rpc request
    public static var loginURL:String = '_ah/login?continue=/'; //dev server uri
    public static var logoutURL:String = '_ah/logout?continue=/'; //dev server uri

    //Some global statc state variables
    public static var profile:Object = null;
    public static var user:Object = null;
    public static var auth:Boolean = false;
```

Figure A.6: Model class.

First off, I create some public static variables so all of the Actionscript can quickly and easily access any data I glean from RPC responses.

Any new RPC methods defined in the python source folder services/ must be registered here. The syntax is ”python_file_name.python_method”. The parameters to these methods must be zero or more strings, as all data must be serialized for http transfer. Note that I do NOT define the parameter maps, and simply assume that the client code will supply the correct number of String arguments to avoid errors.

For every RPC method registered by org.gaeswf.flash.Service, I define FOUR listener meth-

```
/**
Called in Application.as to set up remote procedure calls
*/
public static function init():void
{
    // Hook up listeners for all the rpc used in the app
    listen("user.login");
    listen("user.updateProfile");
    listen("rand.word");
    listen("submit.link");
    listen("submit.junk");
    listen("submit.misspell");
    listen("lex.synset");
    listen("lex.lemma");
    listen("lex.record");
    listen("lex.misspells");
    listen("lex.junks");
    listen("lex.firsts");

    //Note that these identify python classes in the /services folder.
    //The first segment is filename, after period is function name
}
```

Figure A.7: Model class continued.

```
// Service listener helper
private static function listen(serviceName):void
{
    // Add the model as a listener for service responses.
    Service.listen(serviceName, Model);
}
```

Figure A.8: Model class continued.

ods. For the ”user.login” service, I create the following:

- wordgame.Model: user_loginResponse
- wordgame.Model: user_loginFailure
- wordgame.Application: function user_loginResponse
- wordgame.Application: function user_loginFailure

The Model methods are static and are required. The instance methods on the Application root class are optional, but excluding them will throw Flash trace errors. Most of the real work is

done in the instance methods. I have excluded the many blank static Model method stubs for brevity.

```
// Model function to intercept login response.
//Note how listen('file.function') corresponds to file_functionResponse
public static function user_loginResponse(result:Object):void
{
    //store returned python state information for use in application
    auth = result.auth;
    loginURL = result.login;
    logoutURL = result.logout;

    //if login success
    if (result.user != null)
    {
        //store user
        user = result.user;
        //if profile defined {not used}
        if (result.profile != null)
        {
            //store it as well
            profile = result.profile;
        }
    }
}
```

Figure A.9: Model class continued.

The user_loginResponse is the only static method that I actually implement in Model, and in it I scrape the basic authentication response data into static variables that can be accessed anywhere in the Actionscript. This concludes all relevant section of com.wordgame.Model; I now move on to com.wordgame.Application to see how I tie all of this boilerplate together into an RPC call that is actually rather simple in comparison to all the Gaeswf initializations.

A.4 Application

The first and most obvious step is to extend the org.gaeswf.flash.BaseApplication class. Much of the Application.as code is omitted for brevity.

This method is called in response to the Stage.INIT event. I make sure the BaseApplication gets first crack at the event object, then I initialize the model, which registers all of the RPC methods. Because com.wordgame.Application is set to Global.getInstance().root, it is automatically registered for instance-level callback of all RPC methods. The rest of init() sets

```
import com.wordgame.visuals.Manager;
import com.wordgame.visuals.Nodegroup;
import com.wordgame.visuals.Node;
import com.wordgame.events.WordSubmittedEvent;

import com.wordgame.Model;
import org.gaeswf.flash.BaseApplication;
import org.gaeswf.Service;
import SWFAddress;

import flash.external.ExternalInterface;

/**
    By extending BaseApplication from the GAESWF project,
    Our root instance can execute rpc commands,
    And operate directly on all the .fla MovieClip instances.
*/
public class Application extends BaseApplication
```

Figure A.10: Application class.

some basic Stage parameters, and calls Application.Random(), which I will explore in the next section of the Service Layer.

The next important method that I override is urlInit(), which is called when the SWFAddress ExternalInterface is available, and all other SWFAddress initialization is complete. I test the current SWFAddress value to see if there is NOT a current history token. If there is a token, urlChange() will get called automagically, so I only take action here is urlChange() is NOT going to be called immediately. That action is to load a new word cloud with the New() method, which is explained later. I also use this callback to execute the "user.login" Remote Procedure Call. This is an asynchronous process, and it calls services.user.py.login (base_url, login_return_url). I simply return to the website root: http://words-around.appspot.com/ as that is where I serve up the Flash .swf.

This comment simply explains what kind of data to expect from this rpc response. It is the same dictionary object I built in user.login, except I exclude the .profile object, as it currently has no use in the implementation. It is not removed, as I will be using it in future implementations.

```
/*
    This overrides the BaseApplication initializer from Gaeswf module
*/
override protected function init(event:Event):void
{
    //Actually sets up RPC and SWFAddress stuff
    super.init(event);

    // Initialize our model.  Defines rpc calls that target the Application object.
    Model.init();

    //Load a random word ASAP!
    Random();


    // Set Stage properties.
    stage.frameRate = 30;
    //Scale all to fit,
    stage.scaleMode=StageScaleMode.SHOW_ALL;
    //Anchor to the top left.
    stage.align=StageAlign.TOP_LEFT;
}
```

Figure A.11: Application class continued.

user_loginResponse() method performs a number of vital initialization procedures. To start with, it updates the Application.lim : Number variable, which is used in gui code to limit the number of word bubbles to show in a cloud before hiding the weakest Associations. Then, depending on whether I're logged in or not, I show / remove the user-only panels. These panels {the user activity word list and the junk / mispell tagging panel} are hidden by default, but I remove them to clear up memory if they will not be used. I also show or hide the login / logout buttons; logout is shown if I am logged in, and vice versa. When I show the hidden panels, I also call a repaint method to update the gui and an RPC method to load the user's recent activity list {both explored later}.

This ends the inital journey into Gaeswf RPC methods, and I will continue the explanation of the Service Layer with explanations of every method signature I called listen() on in Model.init().

```
/*
    urlInit is called from BaseApplication to notify us that rpc is ready to rock.
*/
override protected function urlInit():void
{
    //If there is no particluar #token to load
    if (SWFAddress.getValue()=="/")
        //Open a new random word
        New("");
    //Note, urlChange is called automatically if there is a token

    //Now that we can rpc, try logging in so we can finish initializations
    execute("user.login", Model.BASE_URL, '');

    //Note, you can leave Model.BASE_URL set to the appengine url.
    //All RPC automatically uses the server that hosts the swf.
    //This is only sent to the login response so the server can generate a login url.

    //I tried generating my own urls server side, but google adds a hash of the request
    //as some kind of security parameter; so even a well formed url is useless unless
    //it was generated on the server.

}
```

Figure A.12: Application class continued.

```
/*
    This method was hijacked from Gaeswf example application.

    Model.as gets access to this result before we do, and stores some data from it.

    result : {
        lim : Number //Server set display limit
        auth : Boolean //Whether we are logged in or not
        user : Object //User data; unused
        login : String //Server-generated login url
        logout : String //Server-generated logour url
    }
*/
```

Figure A.13: Application class continued.

```
public function user_loginResponse(result:Object):void
{
    //If the server sent us a request limit
    if(result.lim)
        //Overwrite the default 5 limit
        lim = result.lim;

    // If we are logged in
    if (Model.auth)
    {
        //Set status
        var user:Object = Model.user;
        mc_login.status.htmlText = "Logout " + user["_User__email"];
        // Log user details
        outputTF.text = user["_User__email"];

        //Show the logout button, since we're logged in
        mc_login.showDetails();

        //Unhide the user-only panels
        mc_wordlist.visible = true;
        mc_junker.visible = true;

        //Fire a repaint / resize event to update ui
        resizing(stage.stageWidth,stage.stageHeight);

        //Since we're logged in, rpc request the user's activity log
        LoadHistory();

    }
    else //we aren't logged in
    {
        //Remove unneeded MovieClip panels.
        removeChild(mc_wordlist);
        removeChild(mc_junker);

        //Make sure the login button is showing
        mc_login.hideDetails();

        //Let user's know what the log in button is for
        mc_login.status.htmlText= "Log In With Google Account";
    }

}
```

Figure A.14: Application class continued.