

Sound Pattern Recognition in Texts
by
Oliver Crease (CHR)

Fourth-Year undergraduate project in
Group F, 2006/2007

I hereby declare that, except where specially indicated, the work submitted herein is my own original work.

Signed:

Date:

Technical Abstract

This report discusses the program called SPRIT, short for Sound Pattern Recognition in Texts. SPRIT is a tool for analysing patterns in text, mainly aimed at the analysis of poetry. The program is written in Java and was designed with modularity and expandability in mind. Java was chosen as the programming language due to its multiplatform support, integrated graphical user interface toolkits, Unicode support and its support for XML processing.

The program can convert text to its phonetic transcription and then it can perform an analysis on the results. The main program does not do much of the analysis, it is mostly done by plug-ins which extend the functionality of the main program. Plug-ins are dynamically loaded, so the program can be compiled without having any knowledge of external classes apart from a common interface. The dynamic class loading is done through runtime polymorphism based on interface inheritance. Since the program knows about the interface, it can call the methods on objects of that implement that interface. The classes are loaded using a class loader, and after a series of checks, the loaded class can be instantiated and its methods are available to the main program. Dynamic class loading through interface inheritance is suitable when the design of the external classes can be influenced by an interface.

SPRIT currently supports two types of plug-ins. The first is for form detectors which can calculate a score for a poem based on how closely it matches a particular form. By loading various form detectors, detection for a variety of forms is possible. The second use of dynamic class loading is for loading the text to phoneme (TTP) converter. Only an English TTP converter is provided with SPRIT, but more could be added as long as they implement the necessary interface. Unfortunately, TTP converters have complicated rules to do the conversion, so they are not common. TTP converters for other languages would allow analysis of poems in a foreign language.

The analysis done by SPRIT can be saved to XML. The XML output could be used by a batch processor to get statistics on a set of poems. In the future more features could be added to the program, including metre and rhyme identification and a batch processor.

Table of Contents

Technical Abstract	i
1. Introduction	3
2. Background and Literature Review	3
3. Purpose	4
4. Project Aims	5
5. Technology Choices.....	5
5.1. Java.....	5
5.2. Swing.....	6
5.3. XML.....	6
5.4. Javadoc.....	7
5.5. NetBeans.....	7
6. Program Structure.....	9
6.1. Word, Line, Stanza and Syllable counting.....	10
6.2. XML Processing using DOM and JAXP	10
6.3. Dynamic Class Loading.....	11
6.3.1. Text to Phoneme Conversion	12
6.3.2. Form Detection	14
6.4. Document Chooser	15
7. Project Achievements and Shortcomings	15
8. Future Work	16
8.1. Batch Facility	16
8.2. Text to Phoneme conversion	16
8.3. Metre and Rhyme Identification.....	17
8.4. Phoneme Analysis	17
8.5. Full Javadoc documentation	17

8.6. Error checking and exception handling	17
9. Conclusion	18
10. Acknowledgements.....	18
11. Bibliography	19
12. Appendix	20
12.1. Word count	20
12.2. iPhoneme Interface.....	20
12.3. iFormIdModule interface	22
12.4. Dynamic Class Loading – TTP Example	22
12.5. Example XML output.....	24

Table of Figures

Figure 1 – NetBeans IDE 5.5	9
Figure 2 – Options Dialog	13
Figure 3 – Form Identification Dialog.....	14
Figure 4 – Document Chooser Dialog.....	15

1. Introduction

Poetry and computer analysis don't appear to be a good combination. After all, poetry is an art form that goes back for centuries and poems have many different perceived meanings depending on many variables like the reader's mood. Computers are incapable of helping to determine a poem's meaning, but they can be used to examine the structure of poems and to look for patterns in them.

A computer program that can look for forms and patterns in the metre, rhyme and phoneme distributions could be used to research both new and old poetry. Another good use for a computer program would be to batch process a large set of poems and then calculate statistics relating them. Researchers could use this to look for trends in a poet's works, trends in poetry of a particular period or to compare poetry over the years, amongst others.

SPRIT, short for Sound Pattern Recognition in Texts, is a program that aims to examine poems for patterns. The following report discusses the design and features of the program.

2. Background and Literature Review

The desire to analyze poetry using a computer is not new, there have been some attempts in the past. Clearly computer analysis of a poem is limited to patterns and structures rather than meaning. It would be unreasonable to expect a computer to tell the reader of a poem what exactly the poet meant, although technology may advance in the future to make this possible. For the time being, computers can be a great aide in analysing patterns in text. A computer program could find patterns which are not immediately obvious to the reader. A simple example of this is a square poem, where either the number of words or syllables per line are the same as the number of lines (Growney, 2006).

Information on two programs was found while researching the field. The first one is called InVerse, developed at the University of East Anglia. However, it would appear that development of this program has stopped as the most recent information on it was from 1997 (Fraser, 1997).

The second of the two programs is much more recent and appears to continue in development. The program AnalysePoems developed at the University of Toronto uses the

Representative Poetry Online (RPO)¹ database to identify patterns in poems (Plamondon M. R., 2006). The program looks up the syllable decomposition and stress in a database. When it finds a word that it does not recognize, it prompts the user for the word's stress and syllabic decomposition. AnalysePoems is a classification tool that extracts information from poems in the RPO database.

In another article, Marc Plamondon (2005) discusses visualization techniques which can assist a reader in the phonetic analysis of a poem. The idea is to use colour to highlight groups of phonemes to visualize patterns. Another use is to label lines using colours to signify the phonetic density for a subset of phonemes, say all the plosive consonants. The user gets a choice of nine different phonetic pattern matching algorithms which change the ways the colours are applied.

This project is a follow on from a fourth year project which was done in 2003 / 2004 by Ruth. P. Jackson. The program, ASP, had similar aims to those of SPRIT and used the same text to phoneme converter (Jackson, 2004). It concentrated more on phonetic analysis and was not expandable using plug-ins. SPRIT is a completely new program

3. Purpose

A program that can analyse poetry in some basic manner has educational use at various levels. At GCSE and A-level standard, students could benefit from simple features such as form detection and rhyme identification. At this level, users are only likely to want to get information on a specific poem, rather than gather statistics from a large set of poems. Students could benefit from using a computer to identify patterns which would then help them understand some of the principles in poetry.

At university and research level, users are more likely to want to draw comparison between poems or gather statistics for a larger set of poems. A program which can look for patterns in the text combined with a batch processing facility could draw statistics on a large selection of poems. For example, a researcher or poetry critic could want to draw comparisons between the works of two poets. They could use the tool to get statistics on a set of poems

¹ The RPO database can be found at <http://rpo.library.utoronto.ca>

from each poet, and then the results could be compared. The results could show trends in a poet's works and conclusions could be made on their particular writing styles.

4. Project Aims

The aims of this project are:

- Produce a useful program to assist in the analysis of poetry
- Develop a program which is easily expandable and customizable
- Adhere to good software engineering principles
- Write two or more plug-ins to enhance the capabilities of the program and demonstrate the plug-in system

5. Technology Choices

The computing world has a very large range of technologies which could have been used for this project. The aims and requirements of the program narrowed down the choice of technologies which would be suitable. Among the most influential requirements were:

- Multi-platform portability
- Unicode support
- GUI libraries
- Opening, saving and manipulating files

The following sections describe the reasons behind the choices of technologies made.

5.1. Java

There are many programming languages to choose from. However, not many of them meet the requirements set above. Java is a fully featured programming language that allows programmers to develop stand-alone applications. The main reason for choosing Java is that it is platform independent. The Java compiler produces byte-code which is then run on the Java Runtime Environment (JRE). Sun Microsystems, the Java developers, produce JRE versions for Microsoft Windows, Solaris and multiple Linux distributions. Apple also produces the JRE for Mac OS.

The requirement for manipulating files means that the program needs to be a desktop application so that it can have access to the local file system. This and the need for a full graphical user interface essentially rules out web applets.

The version of Java chosen for the project is the Java Platform, Standard Edition 6 (Java SE 6) since this was the latest version available when the project was started. For developers, the Java SE Development Kit (JDK) includes the JRE and command line development tools which are required to create applications. In order to run SPRIT, end users will need to have the JRE installed on their computer. The program has only been tested using version 6, but the code should run on the Java SE 5 since there are no features that require version 6 functionality.

Another requirement from the language is that it has to support the Unicode character set. Unicode contains over 65000 characters providing support for many of the world's languages. It also has the characters used by the International Phonetic Alphabet (IPA) which are used to transcribe speech or to express text as a language independent sound. SPRIT uses the characters from the IPA to represent the sounds of the text being analysed. Java uses the Unicode character set to represent characters, so it has full Unicode support.

5.2. Swing

Swing is one of the user interface toolkits included in the Java SE. It is a group of classes that implement components for building a Graphical User Interface (GUI). It is completely programmed in Java and is platform independent, so programs behave in the same way regardless of the operating system (OS). The Swing classes support a pluggable look and feel which allows the program to look the same on any OS, or it can adapt its look and feel to match that of the current operating system. SPRIT tries to use the system's look and feel rather than forcing a specific look and feel. The inclusion of the Swing GUI toolkit in Java gives it a significant advantage over other languages such as C++ and Perl, which require separate libraries to create graphical user interfaces.

5.3. XML

The Extensible Markup Language (XML) is a general purpose markup language that allows programmers to customize its structure for a particular application. It is a text based language which is stored as text files, so it is readable by any text editor and it is platform

independent. XML is an international standard regulated by the World Wide Consortium (W3C)².

In addition to this, Java offers excellent support for XML, in the form of the Java API for XML Processing (JAXP) which means that handling XML documents is relatively straightforward. SPRIT uses XML for saving the output from the analysis. SPRIT has its own custom XML format specifically designed to hold the data that it produces. The output is intended to be for use by other applications, although by using stylesheets or XSLT transformations, the output could also be displayed on a web browser.

5.4. Javadoc

Javadoc is a tool for generating documentation directly from comments in Java source code. Comments starting with `/**` rather than the usual `//` or `/*` are detected by the Javadoc tool. The tool generates HTML files containing the documentation. The comments can be used to provide other developers with more information on a particular class, method, interface or field. Special tags such as `@param` and `@return` can be used to describe a method's input parameters and the meaning of its return value.

The Javadoc tool is excellent for documenting code that might be used by other developers. The output is standard across the Java world. Documentation for many packages is created using the Javadoc tool, including the Java SE reference. Since it is standard, developers will already be used to the style and will have no problem in understanding it. In SPRIT, it is particularly important that the interfaces with external classes are well defined and documented. Sections 12.2 and 12.3 in the appendix show two such interfaces, both commented using Javadoc comments.

5.5. NetBeans

The NetBeans IDE is an open-source Integrated Development Environment. It enables rapid application development by providing a set of tools that increases the ease with which software can be written. The tools can take care of the tedious and repetitive tasks of software development allowing the programmer to concentrate on the application's

² <http://www.w3.org/>

functionality and design. By reducing the amount of code that needs to be manually written, it can also reduce the number of errors and bugs introduced into the program.

The version of NetBeans used for this project is version 5.5. It includes many features, but among the most useful were the advanced code editor, debugger and Swing GUI Builder. The source code editor has intelligent code completion which is a popup that suggests possible variable or method names depending on the context. Not only does this feature increase the speed with which the application is developed, but it also reduces the likelihood of making mistakes while writing the code. The code completion popup also displays Javadoc information for classes and methods which are documented. The information that is included lets the programmer know what the function does and what parameters it requires. The code editor also provides assistance for refactoring code. It can take care of modifying references to a variable, class or method if it is renamed or moved. It can also help to encapsulate fields and extract methods so that they can be reused. Field encapsulation is where class field is provided with `get` or `set` methods which can control the reading and writing of the field.

The NetBeans Swing GUI builder greatly simplifies the design and creation of graphical user interfaces. Visual GUI editors improve GUI creation by immediately showing changes made without the need to recompile and run the program. Figure 1 shows the NetBeans IDE while editing the design of a form using the Swing GUI Builder. A palette containing the available Swing components makes it easy to drag and drop the components onto the forms or dialogs. The IDE takes care of all the component initialization, saving the programmer a substantial amount of time. The components are fully customizable through the properties panel. This makes it very easy to change the behaviour and appearance of components and since all the properties are shown, the programmer does not need to remember what the available properties of each component are. Visual guidelines assist the designer in placing components by suggesting optimal spacing and alignments between components.

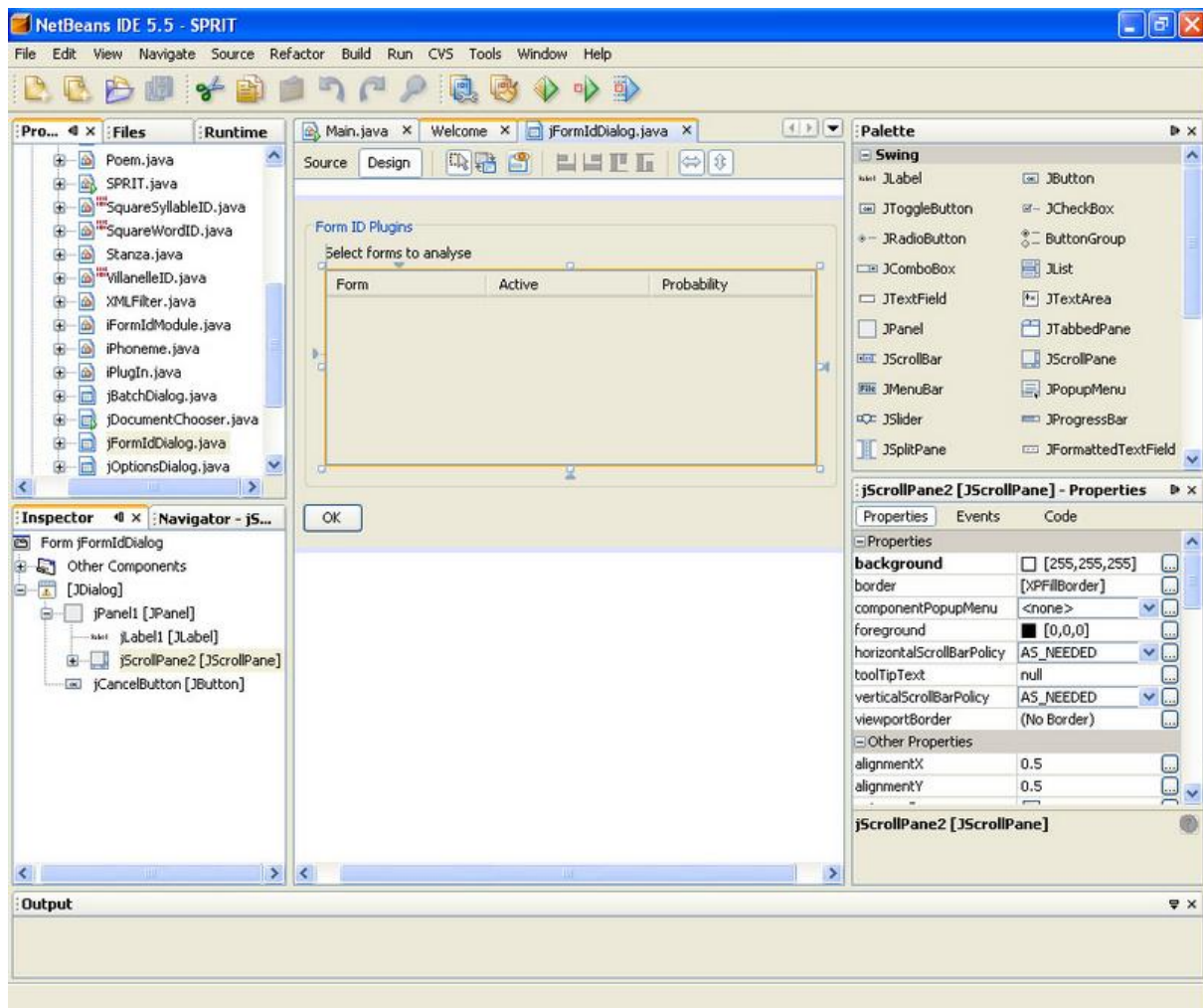


Figure 1 – NetBeans IDE 5.5

The integrated debugger is perfect for identifying problems during testing. The debugger allows the user to progress through the program execution using breakpoints or step by step so that problems can be analyzed.

6. Program Structure

The program structure was designed with modularity and expansion in mind. The program core does the necessary basic functions but it can be expanded using plug-ins. The main class that controls the program is the main GUI form. This class controls the execution of the program. The user can use the menus on the GUI to access the functionality of the program.

When a file is opened, the text is loaded into an instance of the `Poem` class. The poem class contains fields for information such as the author and title of the poem. It also has an

`ArrayList`³ of `Stanza` objects. The stanzas have an `ArrayList` of `Line` objects as well as various methods to get and set data. The lines then contain the text and the phonetic transcriptions.

6.1. Word, Line, Stanza and Syllable counting

The loading of the text into the `Poem` class hierarchy makes the task of counting lines and stanzas trivial. For the line count, the size of the `ArrayLists` containing the `Line` objects in each stanza simply needs to be added. For the stanza count, the size of the `ArrayList` containing the `Stanzas` needs to be found. The size of an `ArrayList` is found easily by calling its `size()` method.

For the word counts, a similar approach can be used, but first the text is split into an array of words by using the `split()` method of the `string` class and using a regular expression to match whitespace. Any punctuation characters are ignored in the count. The size of the array then gives the number of words in a line.

Syllable counting is also done by the `Line` class. The syllable counting algorithm is based on the knowledge that syllables usually only contain one vowel. Therefore the number of syllables in a line is simply the number of vowels in the line. Special care has to be taken with diphthongs, which are represented as two separate vowels, but are considered to be a single entity. The algorithm takes this into account. However, the results are not 100% accurate since some words have more than one way of splitting them into syllables

6.2. XML Processing using DOM and JAXP

SPRIT can save the analysis done on poems as an XML file. Creating the XML output is a two stage process. First the data needs to be made into a document based on the Document Object Model (DOM)⁴ and then it can be serialized using the Java API for XML Processing (JAXP).

The Document Object Model is a platform and language independent interface that allows programs to access and modify the content, style and structure of documents (W3C, 2005). In the process of saving the SPRIT output to XML, a document needs to be created with the

³ `java.util.ArrayList`

⁴ <http://www.w3.org/DOM/>

content and structure of the data. For this application, we are not concerned with the style of the document. Documents have one and only one root 'node'. This node then has 'elements' which can have other elements, 'text' and 'attributes'. With respect to XML, the elements are tags, text is the text inside a pair of tags, and attributes are tag attributes. In the SPRIT output, an element could be a line or stanza, an attribute is a line or stanza number and text is the text between `lineText` tags.

The second stage is the serialization of the DOM document to an XML document. The serializer uses the content and structure of the document to work out where to add the necessary symbols to make up XML tags and attributes.

6.3. Dynamic Class Loading

Dynamic class loading is one the more advanced topics in Java programming. It can be used to load classes which were unknown to a program when it was compiled. SPRIT uses this to load plug-ins and the TTP Converter.

There are three ways of using methods in dynamically loaded classes (Baldwin, 2006). These are:

- Reflection
- Runtime polymorphism based on interface inheritance
- Runtime polymorphism based on class inheritance

The choice between the three options is down to how much the programmer of the main application can influence the design of the external classes. The less influence the programmer has over the external classes, the more complicated the solution needs to be.

Reflection is the only choice when the programmer has no influence over the external classes and is therefore the most complicated. Reflection works by loading the class using a class loader and then getting a list of available methods. The methods then have to be invoked by calling their `invoke()` method.

Both solutions using runtime polymorphism are more straightforward than reflection. With these two methods, the programmer can influence the design of the external classes. In runtime polymorphism based on interface inheritance relies on defining a common interface

that the application and plug-in programmer both have access to. The class is loaded using the same method as before, but the names of the methods are already known, so invoking them can be done simply by calling them. Runtime polymorphism based on class inheritance is essentially the same as using interface inheritance, but instead of implementing a common interface, the new class extends a class known by the main program. Runtime polymorphism based on class inheritance is more restrictive than the more general method using interface inheritance.

In SPRIT, the most suitable method is using runtime polymorphism based on interface inheritance since it is possible to influence the plug-ins using a common interface. There are two uses for dynamic class loading in SPRIT, and these are discussed in more detail in the following sections.

Section 12.4 shows an extract of code for the loading of the TTP modules. First the program looks for the directory that contains the TTP modules and if it doesn't find it, it creates it. Then it translates the directory name to a URL⁵ which can be used to create a `ClassLoader` object. It then cycles through all the files in the directory, looking for files that have a `class` extension. Class files are then loaded using the class loader into the `loadedClass` object. The next step is to query the class for its interfaces. In this case, only those classes which implement the `iPhoneme`⁶ interface should be instantiated. An instance of the class is created by calling the `newInstance` method and then casting it to an `iPhoneme` object. Now an instance of the class exists and its methods can be called.

6.3.1. Text to Phoneme Conversion

The first example of dynamic class loading is for giving the user the option of which Text to Phoneme (TTP) converter to use. The TTP converter could have been hard coded into the program, but this would have limited analysis to texts in English only. By dynamically loading the TTP converter and allowing the user to choose which one to use, it is now possible to analyse poems in other languages. However, only an English TTP converter is available with the program, so although it would be possible to load a different one, none currently exist. Creating a set of conversion rules for a different language is not a trivial task and would

⁵ URL – Uniform Resource Locator

⁶ The `iPhoneme` interface is available in section 12.2

probably encompass a project in itself. The flexibility in SPRIT would allow the loading of new TTP converter as long as it adheres to the `iPhoneme` interface set out in section 12.2. The TTP modules are detected by SPRIT and then the user can select which one to use using the options dialog. Figure 2 shows the options dialog. TTP modules have to be in the correct directory to be detected⁷.



Figure 2 – Options Dialog

The English text to phoneme converter bundled with SPRIT is one originally developed by John A. Wasser and later translated to Java by Olivier Sarrat (Wasser & Sarrat, 2001). The conversion rules are based on a report⁸ from the U.S. Naval Research Institute. The converter transcribes English into phonemes using 41 phonetic symbols.

Plosives	/p/ /b/ /t/ /d/ /k/ /g/
Fricatives	/f/ /v/ /θ/ /ð/ /s/ /z/ /ʃ/ /ʒ/ /h/
Affricates	/m/ /ŋ/ /tʃ/
Nasals	/n/ /ŋ/
Oral Continuants	/l/ /w/ /j/ /ɹ/ /hw/

Table 1 – Consonants used in English

⁷ The directory where form detectors should be placed is `modules/TTPModules` under the main program directory

⁸ "Automatic Translation of English Text to Phonetics by means of Letter-To-Sound Rules" (NRL Report number 7948).

Pure Vowels	/i/ /ɪ/ /e/ /ɛ/ /æ/ /ɑ/ /ɔ/ /o/ /ʊ/ /u/ /ə/ /θ/ /ʌ/
Diphthongs	/aɪ/ /aʊ/ /ɔɪ/

Table 2 – Vowels used in English

6.3.2. Form Detection

Dynamic class loading is also used to load form detectors. There are many different Poetry forms so it makes sense to have a mechanism for adding new form detectors. SPRIT allows this using dynamic class loading. Bundled with the program are four form detectors that are used as a proof of concept. Figure 3 shows the form identification dialog. Four detectors are loaded, capable of detecting Villanelle, Haiku, Square Word and Square Syllable patterns.

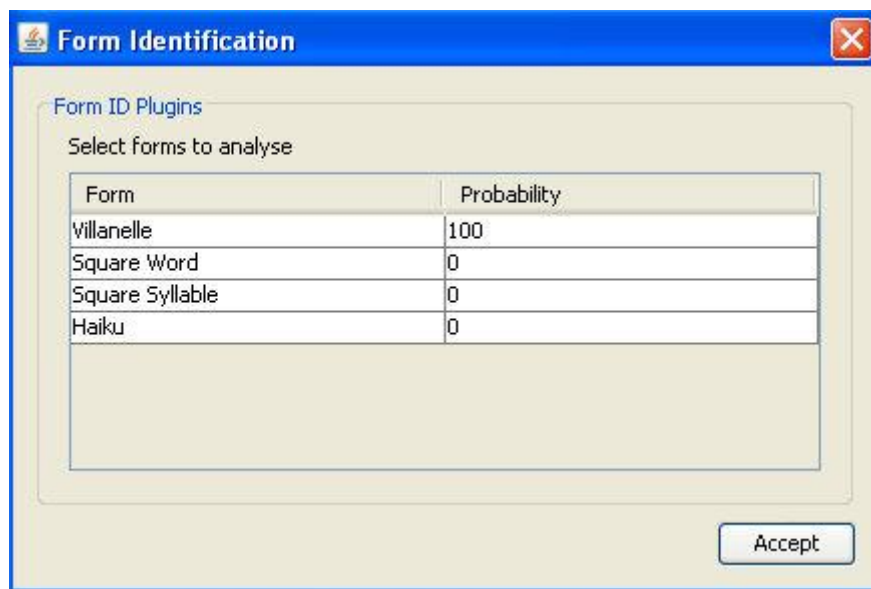


Figure 3 – Form Identification Dialog

The form detectors need to implement the `iFormIdModule` interface shown in section 12.2. The interface has only two methods; one for analysing the poem and returning an integer score and another for returning the name of the form which is being identified. The `analysePoem(Poem poem)` method has the poem as a parameter, so the form detector has complete access to the public methods of the `Poem` class. Form detectors are automatically loaded by SPRIT when they are located in the correct directory⁹.

⁹ The directory where form detectors should be placed is `modules/forms` under the main program directory

6.4. Document Chooser

SPRIT allows multiple documents to be loaded concurrently. The document chooser dialog lists all the open documents and allows the user to switch between them. The possibility of loading multiple poems was included so that batch support could be included at a later date. Currently only one poem can be analysed at a time.



Figure 4 – Document Chooser Dialog

7. Project Achievements and Shortcomings

The outcome of the project is a program which provides a framework for analysing patterns in poetry. It has a limited feature set, but has the capacity of increasing functionality using plug-ins. It has a well defined mechanism for adding form detectors and for changing the text to phoneme converter that is used. It comes bundled with form detectors for four simple forms.

The program is able to convert English text into its phonetic transcription. This information along with word, line, stanza and syllable counts is made available to plug-ins and it is possible to save the analysis to an XML file.

The project did not meet all the aims. A batch processing facility would have added tremendous value to the program, but this has not been fully implemented. Multiple poems can be loaded into SPRIT, but the logic for analysing and collating results for a set of poems has not been developed. Someone interested in using SPRIT's output to analyse a batch of poems could do so by finding or developing a tool that can analyse a set of XML files.

Along with batch processing, there are no facilities for drawing charts or graphs. These facilities would have had great value when used together with a batch processor, as they could easily highlight any trends. However, the lack of batch processing facilities meant that graphing facilities were of less importance.

From a personal point of view, the project has taught me many things from advanced Java programming to the software lifecycle and project management.

8. Future Work

8.1. Batch Facility

Batch processing would add significant value to the program in the view of researchers and literature critics. By analysing a set of poems and collating statistics, researchers could look for trends, in say a poet's works or poems from a particular period. Even simple statistics such as lines per poem could provide an insight into a set of poems.

For the users, it would be far more convenient to have the batch functionality built in to the program rather than relying on an external application that analyses the XML output. SPRIT already offers some functionality which would be useful for a batch processor. This functionality is the ability to load multiple poems at once. After the poems are loaded and automatically analysed, the batch processor can simply find the information it needs directly from the poems.

8.2. Text to Phoneme conversion

The text to phoneme conversion is not 100% accurate. There are words which have more than one possible phonetic transcription and the converter cannot detect these variations. The program should allow the user to modify the phonetic transcription and perhaps save the alternative, so that when the converter finds the word again, it can prompt the user for which transcription they would prefer.

Text to phoneme converters for other languages or other variations of English would be interesting. However, the conversion rules are complex and specialist knowledge of the

language would be required. This could be a joint project with, say, the Modern Languages department.

8.3. Metre and Rhyme Identification

Metre and rhyme identification should be a feature of any poetry analysis program. A similar mechanism to the form identification mechanism could be implemented using dynamic class loading. There are many different rhyme patterns, so a plug-in architecture would seem suitable.

Pattern identifiers need to take into account the inaccuracies of the TTP converter. Fuzzy matching algorithms with confidence values could be used to accept the inaccuracies and work around them.

8.4. Phoneme Analysis

Using a computer for phonetic analysis of a text greatly simplifies the task. Creating and analysing phonetic transcriptions would be very tedious jobs to do by hand. If instead a computer program does the phonetic transcription automatically and then highlights specific phonemes or sets of phonemes, the user should be able to extract more information and do it quicker. A system as suggested in (Plamondon M. R., 2005) could be investigated and implemented into SPRIT.

8.5. Full Javadoc documentation

Currently only the `iPhoneme` and `iFormIdModule` interfaces are properly commented using Javadoc comments. Developers trying to write plug-ins for SPRIT will also need to know the structure of the `Poem`, `Stanza` and `Line` classes. The remainder of the classes should also be documented using Javadoc to encourage code reuse.

8.6. Error checking and exception handling

The exception handling mechanism in SPRIT is very rudimentary. It has the bare minimum handling of exceptions, at best simply printing the stack trace. Error checking is only implemented at a basic level. Both error checking and exception handling need to be improved to increase the robustness of the application.

9. Conclusion

SPRIT is a poetry analysis program that was designed with flexibility and expandability in mind. It is written in Java and is designed for the Java Platform, Standard Edition version 6. It provides a framework for loading plug-ins currently limited to form detectors and text to phoneme converters. It does so by dynamically loading classes. This is a great way of expanding functionality since the main program does not have to know about the external class when it is compiled.

The analysis performed by the program can be saved as an XML file. This output could then be used by another program to do further analysis. In the future a batch processing facility could be implemented into the program.

10. Acknowledgements

I would like to thank my supervisor, Tim Love, for providing so much guidance and assistance throughout the year.

11. Bibliography

- Baldwin, R. G. (2006). *Dynamic Loading/Reloading of Classes, Part 2*. Retrieved from Gamelan.com - A developer.com:
<http://www.developer.com/java/other/article.php/3632536>
- Crystal, D. (1995). *The Cambridge Encyclopedia of The English Language*. Cambridge, UK: Cambridge University Press.
- Fabb, N. (2005). English iambic pentameter: an accentual-syllabic metre. *Conference: Metrics and Versification: the state of play*. Cambridge University.
- Fraser, M. (1997). *InVerse*. Retrieved April 30, 2007, from Guide to Digital Resources 1996-98: <http://users.ox.ac.uk/~ctitext2/resguide/resources/i140.html>
- Growney, J. (2006). Mathematics In Poetry. *Journal of Online Mathematics and its Applications* (Article ID: 1262).
- Plamondon, M. R. (2005). Computer-Assisted Phonetic Analysis of English Poetry: A Preliminary Case Study of Browning and Tennyson. *Text Technology*, 14 (2), 153-175.
- Plamondon, M. R. (2006). Virtual Verse Analysis: Analysing Patterns in Poetry. *Literary and Linguistic Computing*, 21 (Suppl Issue, 2006), 127-141.
- Sun Microsystems. (2006). *The Java Tutorials*. Retrieved from <http://java.sun.com/docs/books/tutorial/>
- Wasser, J. A., & Sarrat, O. (2001). *Text to Speech Program Source Code: Java Text to Phonemes*. Retrieved from <http://www.john-wasser.com/TextToSpeech/JavaSpeech/>

12. Appendix

12.1. Word count

This method is found in the `Line` class

```
public int getNumWords() {
    int numWords = 0;
    String[] words = text.trim().split("\\p{Space}+");
    for(int i = 0; i<words.length;i++){
        if(words[i].matches("\\p{Punct}+")){
            // Don't count if its a punctuation character'
        } else numWords++;
    }
    return numWords;
}
```

In the `Stanza` class, it becomes

```
public int getNumWords() {
    int numWords = 0;
    for(Line line : lines){
        numWords += line.getNumWords();
    }
    return numWords;
}
```

Similar methods are provided by the `Poem` and `Stanza` classes to count lines, stanzas and syllables.

12.2. iPhoneme Interface

```
package sprit;
import java.util.Locale;

/**
 * This interface defines the methods that a TTP
 * converter must provide to be accepted by SPRIT.
 * For a TTP converter to be automatically loaded
 * by SPRIT, it must implement this interface and
 * it must be located in the directory
 * <code>modules/TTPModules</code>
 * @author Oliver Crease
 */
```

```
public interface iPhoneme {

    /**
     * Returns the phonetic transcription of a string
     * @param text the string to be converted
     */
    public String toPhoneme(String text);

    /**
     * Returns true if the phoneme being tested is a vowel
     * @param ch the phoneme to be tested
     */
    public boolean isVowel(char ch);

    /**
     * Returns true if the phoneme being tested is a diphthong
     * @param ch the phoneme to be tested
     */
    public boolean isDiphthong(char ch1, char ch2);

    /**
     * Returns true if the phoneme being tested is a plosive
     * @param ch the phoneme to be tested
     */
    public boolean isPlosive(char ch);

    /**
     * Returns true if the phoneme being tested is a fricative
     * @param ch the phoneme to be tested
     */
    public boolean isFricative(char ch);

    /**
     * Returns true if the phoneme being tested is a affricate
     * @param ch the phoneme to be tested
     */
    public boolean isAffricate(char ch);

    /**
     * Returns true if the phoneme being tested is a nasal
     * @param ch the phoneme to be tested
     */
    public boolean isNasal(char ch);

    /**
     * Returns true if the phoneme being tested is an
     * oral continuant
     * @param ch the phoneme to be tested
     */
    public boolean isOralContinuant(char ch);

    /**
     * Returns the locale of the TTP converter
     */
    public Locale getLocale();
}
```

12.3. iFormIdModule interface

```
package sprit;

/**
 * This interface defines the methods that have to be
 * provided by a Form Identification module. To be
 * recognized by SPRIT, the module must implement this
 * interface. It must also be located in the
 * <code>modules/forms</code> folder.
 *
 * @author Oliver Crease
 */
public interface iFormIdModule {

    /**
     * Does the analysis on the poem. It returns a score
     * determining the likelihood of the poem being of
     * the specified form.
     *
     * @param poem the poem to be analysed
     * @return returns the probability of the poem being
     *         of the form being analysed
     */
    public int analysePoem(Poem poem);

    /**
     * Returns the name of the form being identified.
     */
    public String toString();

}
```

12.4. Dynamic Class Loading - TTP Example

The dynamic class loading mechanism is shown here. For brevity, the whole file is not shown. The code needs the two import statements to be present for it to work. The code is inside a function that finds the TTP modules and then puts them in a combo box so that the user can select the desired one. This code is based on (Baldwin, 2006).

```
import java.io.*;
import java.net.*;

Class loadedClass = null;
try{
    File targetDir = new File("modules" + File.separator
        + "TTPModules" + File.separator);
    if(!targetDir.exists()){
        targetDir.mkdir();
    }
}
```

```
    }

    URI uri = targetDir.toURI();
    URL url = uri.toURL();
    URL[] theUrl = new URL[]{url};

    ClassLoader classLoader = new URLClassLoader(theUrl);
    String[] fileList = targetDir.list();

    /* Load the specified class, creating a Class
     * object that represents the class in the process. */
    for (String filename : fileList){
        int length = filename.length();
        if(filename.endsWith(".class")){
            try{
                loadedClass = classLoader.loadClass(
                    filename.substring(0,length-6));
                Class[] interfaces =
                    loadedClass.getInterfaces();
                boolean flag = false;
                for (int i=0;i<interfaces.length;i++){
                    if(interfaces[i].getName() ==
                        "sprit.iPhoneme"){
                        flag = true;
                        break;
                    }
                }
                if(flag){
                    iPhoneme obj =
                        (iPhoneme)loadedClass.newInstance();
                    jTTPLanguageComboBox.addItem(obj);
                }
            }
            // If the file is not a class, then ignore it
            catch (ClassNotFoundException e){
                continue;
            }
            /*
             * If there is a file or directory
             * with less than 6 letters then ignore
             * it and continue to the next file
             */
            catch (StringIndexOutOfBoundsException e){
                continue;
            }
        }
    }
} catch (Exception e){
    e.printStackTrace();
}
```

