Generating an NLP Corpus from Java Source Code: The SSL Javadoc Doclet

Ninus Khamis, Juergen Rilling, and René Witte

Department of Computer Science and Software Engineering
Concordia University, Montréal, Canada

Abstract

Source code contains a large amount of natural language text, particularly in the form of comments, which makes it an emerging target of text analysis techniques. Due to the mix with program code, it is difficult to process source code comments directly within NLP frameworks such as GATE. Within this work we present an effective means for generating a corpus using information found in source code and in-line documentation, by developing a custom doclet for the *Javadoc* tool. The generated corpus uses a schema that is easily processed by NLP applications, which allows language engineers to focus their efforts on text analysis tasks, like automatic quality control of source code comments. The SSLDoclet is available as open source software.

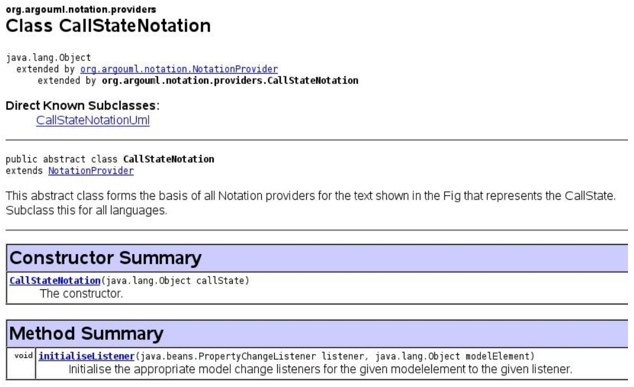
1. Introduction

One of the main challenges in Software Engineering is performing software maintenance tasks on an application that a developer is unfamiliar with. An important software engineering artefact used by developers and maintainers to assist in software comprehension and maintenance is source code documentation. Source code documents provide the insight needed by developers and maintainers to effectively perform their tasks, and therefore ensuring the quality of this documentation is extremely important. In-line documentation is at the forefront of explaining a programmer's original intentions for a given implementation. The blocks of text are inserted directly in source code, and are designed to efficiently assist others in understanding the source code. Since in-line documentation is written in natural language, they are a prime candidate for applying NLP and text mining methods, e.g., for quality assurance, ontology population, or traceability recovery. However, inline comments are mixed with source code in a way that makes them difficult to work with directly when using standard NLP tools such as GATE (Cunningham et al., 2002). *Javadoc* (Kramer, 1999) is a standard inline documentation tool that extracts and transform source code comments to HTML. This format is very well suited for human end users, but again not ideal for machine processing as many semantics are lost in this translation. For NLP processing, a meta-data language such as XML (Ray, 2003) is preferable.

In this paper, we introduce the *Semantic Software Lab Doclet (SSLDoclet)*, which is a custom Javadoc doclet that is able to generate a corpus using information found in source code and in-line documentation. The goal of the SSLDoclet is to covert the information found in source code by encoding it using an XML schema that is specifically designed for NLP applications. Our doclet is available under an open source license.[1]

2. Background

Javadoc (Kramer, 1999) is an automated tool that generates API documentation using Java source code and in-line documentation. In Figure 1 we show part of an API documentation generated using the Javadoc tool and the standard


```
org.argouml.notation.providers
Class CallStateNotation

java.lang.Object
  extended by org.argouml.notation.NotationProvider
      extended by org.argouml.notation.providers.CallStateNotation

Direct Known Subclasses:
      CallStateNotationUml

public abstract class CallStateNotation
extends NotationProvider

This abstract class forms the basis of all Notation providers for the text shown in the Fig that represents the CallState.
Subclass this for all languages.

Constructor Summary
CallStateNotation(java.lang.Object callState)
      The constructor.

Method Summary
void | initialiseListener(java.beans.PropertyChangeListener listener, java.lang.Object modelElement)
     |     Initialise the appropriate model change listeners for the given modelelement to the given listener.
```

Figure 1: Excerpt of an API Documentation generated using the Standard Javadoc Doclet

doclet.

Javadoc comments added to source code are distinguished from normal comments by a special comment syntax (`/**`). The `javadoc` tool extracts the source code and comments in order to transform the information into a variety of output formats, such as HTML, LaTeX, or PDF.

Javadoc provides an API that enables users to implement their own doclets[2] in order to generate documents using a desired output format.

3. Design & Implementation

Javadoc's standard doclet generates API documentation using the HTML format. While this is convenient for human consumption, automated NLP analysis applications require a more structured XML format.

The Javadoc library is in charge of parsing a source directory and providing an interface to a set of objects that is created as a result of the static source code analysis. Generation of the XML documents is then made possible by developing a custom doclet that uses the Javadoc library.

Implementing a custom doclet enabled us to (i) control what information from the source code will be included in the corpus, and (ii) mark up the information using a schema that NLP applications can easily process.

[1] SSLDoclet, http://www.semanticsoftware.info/javadoclet

[2] Java Doclet Overview, http://java.sun.com/j2se/1.4.2/docs/tooldocs/javadoc/overview.html

3.1. SSL Javadoc Doclet Design

In this section, we discuss in detail the level granularity and structure used when marking up Java source code using our SSLDoclet. As a running example, consider Figure 2, which shows an *Abstract Class* declaration taken from the open source project ArgoUML.

```
package org.argouml.notation.providers;

import java.beans.PropertyChangeListener;
import org.argouml.model.Model;
import org.argouml.notation.NotationProvider;

/**
 * This abstract class forms the basis of all Notation providers
 * for the text shown in the Fig that represents the CallState.
 * Subclass this for all languages.
 *
 * @author mvw@tigris.org
 */
public abstract class CallStateNotation extends NotationProvider
```

Figure 2: An Abstract Class Declaration taken from ArgoUML's Source Code

In Figure 3, we show the same declaration marked up using our XML meta-data tags, attributes, and elements.

```
<Abstract_Class_Block>
<Abstract_Class>CallStateNotation</Abstract_Class>
<Package>org.argouml.notation.providers</Package>
<Extends_Block>
        <Extends superclass="Object"
        qualifiedType="org.argouml.notation.NotationProvider"
                superclassFullType="java.lang.Object"
                        type="Abstract_Class">
        NotationProvider
        </Extends>
        <Extends_Comment>
                A class that implements this abstract class manages a
                text shown on a diagram. This means it is able to
                generate text that represents one or more UML objects.
                And when the user has edited this text, the model may be
                adapted by parsing the text.
                Additionally, a help text for the parsing is provided,
                so that the user knows the syntax.
        </Extends_Comment>
</Extends_Block>
<Class_Comment_Block>
        <Class_Comment>
                This abstract class forms the basis of all Notation
                providers for the text shown in the Fig that represents
                the CallState.
                Subclass this for all languages.
        </Class_Comment>
        <Author>mvw@tigris.org</Author>
</Class_Comment_Block>
```

Figure 3: A Section of the Corpus, generated using an Abstract Class Declaration

What makes XML superior over HTML for representing information that needs to be analysed by NLP applications, is that XML is much more versatile than HTML, and enables users to use custom *tags* and *attributes* to mark up the information of the XML elements (Ray, 2003), whereas with HTML we are limited to pre-defined tags such as `<p>` or `<head>`, and predefined attributes such as `font-size`. Such tags are designed to be rendered by a browser for human consumption (Antoniou and van Harmelen, 2008).

3.2. Marking Up Source Code

Our *SSLDoclet* is able to model both the syntactic and semantic information found in Java source code, such as:

- Parent/Child relationships between generalized and specialized *Classes*.
- The *Package* an *Interface* or *(Abstract) Class* belongs to.
- *Fields*, *Constructors* and *Methods* of a *Class*.
- The *types*, *modifiers (private, public, protected)*, and *constant* values of the fields.
- The *return types*, *parameter list*, and *thrown exceptions* of a *method*.

In Figure 3, we show how our doclet represents the information for the `CallStatNotation` abstract class, using the `<Package>` and `<Extends>` tags to model the package the abstract class belongs to, and the superclass that it extends. Figure 4 shows how the parameters of the `intialiseListener` method are modelled using the XML tag `<Parameter>`.

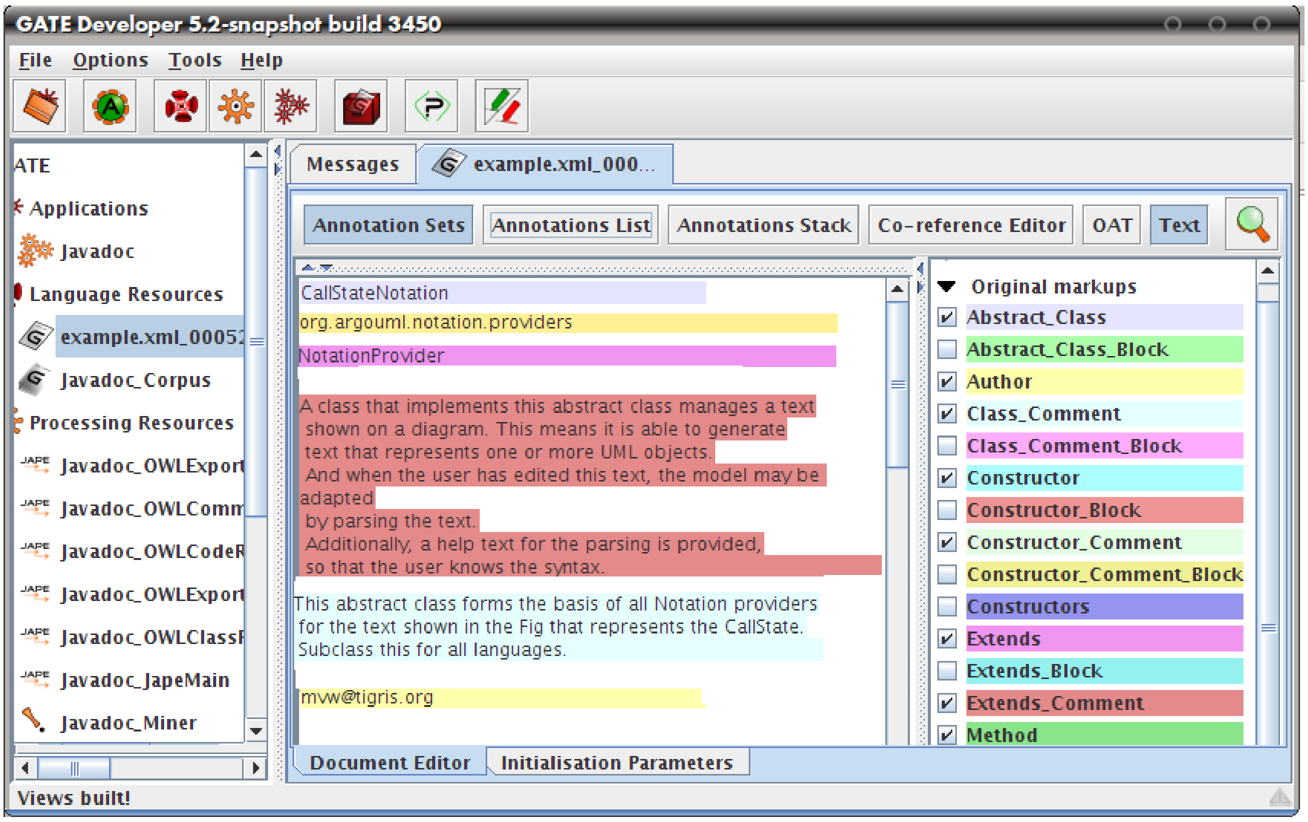
```
<Methods>
<Method_Block>
<Method modifier="public"
          visibility ="public" signature="()">
                enable
</Method>
<Method_Comment_Block>
        <Method_Comment>
                Method to enable the module.&lt;p&gt;
                 If it cannot enable the module because some other
                module is not enabled it can return
                &lt;code&gt;false&lt;/code&gt;.
                In that case the module loader will defer this
                attempt until all other modules are loaded (or until
                some more of ArgoUML is loaded if at startup). Eventually
                 it is only this and some other modules that is not loaded
                and they will then be listed as having problems.
        </Method_Comment>
</Method_Comment_Block>
<Return_Block>
        <Return>boolean</Return>
        <Return_Comment>true if all went well</Return_Comment>
</Return_Block>
</Method_Block>
</Methods>
```

Figure 4: A Section of the Corpus Generated using a Method Declaration

Both figures demonstrate how our doclet is able to represent more information effectively using XML attributes, compared to the standard HTML output. For example, we now also know that the parent of the `CallStatNotation`'s superclass is `Object`, and that the `listener` parameter of the `intialiseListener` method has the type *PropertyChangeListener*.

3.3. Marking Up Source Code Comments

Our SSLDoclet is also able to mark up the natural language information found in Javadoc comments, such as the *docComment*, *block*, and *in-line* tags.
Figure 2 shows an example of a Javadoc comment that includes a *docComment*, and uses the *@author* in-line tag. And in Figure 3, we show how Javadoc comments are



```
ATE
Applications
  Javadoc
Language Resources
  example.xml_00052
  Javadoc_Corpus
Processing Resources
  Javadoc_OWLExport
  Javadoc_OWLComm
  Javadoc_OWLCodeR
  Javadoc_OWLExport
  Javadoc_OWLClassF
  Javadoc_JapeMain
  Javadoc_Miner
```

```
CallStateNotation
org.argouml.notation.providers
NotationProvider

A class that implements this abstract class manages a text
 shown on a diagram. This means it is able to generate
 text that represents one or more UML objects.
 And when the user has edited this text, the model may be
adapted
 by parsing the text.
 Additionally, a help text for the parsing is provided,
 so that the user knows the syntax.
This abstract class forms the basis of all Notation providers
 for the text shown in the Fig that represents the CallState.
 Subclass this for all languages.

mvw@tigris.org
```

Original markups
- [x] Abstract_Class
- [ ] Abstract_Class_Block
- [x] Author
- [x] Class_Comment
- [ ] Class_Comment_Block
- [x] Constructor
- [ ] Constructor_Block
- [x] Constructor_Comment
- [ ] Constructor_Comment_Block
- [ ] Constructors
- [x] Extends
- [ ] Extends_Block
- [x] Extends_Comment
- [x] Method

Figure 5: Corpus Generated using the SSLDoclet loaded within GATE

marked-up using the `<Extends_Comment>` tag, which contains the comment belonging to a super class. Additionally, the figure shows how the class comment belonging to `CallStatNotation` is represented using the `<Class_Comment>` and `<Author>` tags.

The SSLDoclet uses a schema that maintains the relationships found in source code, and represents the information using a combination of XML tags, attributes and elements. In Figure 6, we show how the relationships found in the sample source code is modelled. We eliminated the XML elements and attributes for readability purposes.

The information generated using the SSLDoclet could have been modelled in a number of different ways; however, it is important to keep in mind that when a corpus is loaded within an NLP framework such as GATE, the XML tags are interpreted as *annotations*, the XML elements are interpreted as *entities* of the annotation they belong to, and finally the XML attributes are interpreted as *features* of the annotation.

Our SSLDoclet is designed to generate a corpus using a schema that best utilizes how NLP frameworks interpret the information when initially loaded within the environment. In Figure 5, we show how *annotations*, *features*, and *entities* are created using only the original markups supplied by the corpus.

It is important that the information within a corpus be represented using a structure that enables the NLP environment to form an initial foundation that acts as the starting point for the automated NLP analysis.

```
<Abstract_Class_Block>
    <Abstract_Class/>
        <Package/>
            <Extends_Block>
                <Extends/>
                <Extends_Comment/>
            </Extends_Block>
            <Class_Comment_Block>
                <Class_Comment/>
                <Author/>
            </Class_Comment_Block>
            <Constructors>
            <Constructor_Block>
                <Constructor/>
                    <Constructor_Comment_Block>
                        <Constructor_Comment/>
                    </Constructor_Comment_Block>
                    <Constructor_Block>
                        <Constructor/>
                        <Parameter_Block>
                        <Parameter/>
                        <Parameter_Comment/>
                        </Parameter_Block>
                    </Constructor_Block>
            </Constructors>
            <Methods>
                <Method_Block>
                    <Method/>
                    <Parameter_Block>
                        <Parameter/>
                        <Parameter_Comment/>
                    </Parameter_Block>
                </Method_Block>
            </Methods>
</Abstract_Class_Block>
```

Figure 6: SSLDoclet Schema

4. Application and Evaluation

To execute the SSLDoclet, it is passed as a parameter to `javadoc` when processing a source directory. In Figure 7

Table 1: Open Source Project Versions, Lines of Code (LOC), Number of Comments and Identifiers, and Process Duration

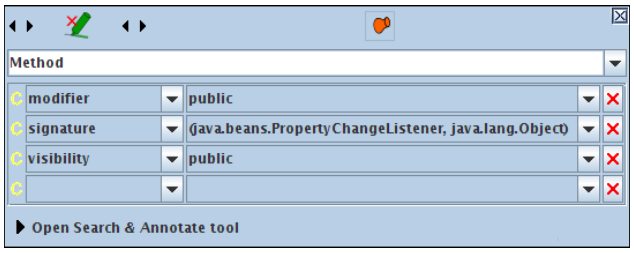
Project	LOC	Number of Comments	Number of Identifiers	Duration (sec.)
ArgoUML v0.24	250,000	6,871	13,974	3.4
ArgoUML v0.26	600,000	6,875	14,262	8.9
ArgoUML v0.28.1	800,000	7,168	14,789	12.2
Eclipse v3.3.2	7,000,000	32,172	158,009	93.1
Eclipse v3.4.2	8,000,000	33,919	163,238	115.7
Eclipse v3.5.1	8,000,000	34,360	165,945	123.1

we show an example of a Javadoc `ant` task that indicates (i) the path and the name of the doclet, (ii) the path to the source directory, (iii) the name of the package in the source directory that needs to be processed, and finally (iv) any other additional parameters, for example, to increase the default Java heap space.

```
<target name="docs" depends="jar">
        <javadoc docletpath = "${doclet.dir}/
                                        ${ant.project.name}.jar"
        doclet          = "${doclet}"
        sourcepath      = "${src.dir}"
        packagenames = "info.semanticsoftware.doclet"
        additionalparam = "-J-Xmx256m"
        />
</target>
```

Figure 7: Javadoc Ant Task that accepts the SSLDoclet as a Parameter

An NLP framework such as GATE can now process the generated XML meta-data as annotations, entities and features, which form the basis for the automated NLP analysis. In Figure 8, we show how GATE interprets the meta-data found within the corpus for the `intialiseListener` method.



Method

modifier	public
signature	(java.beans.PropertyChangeListener, java.lang.Object)
visibility	public

Figure 8: Annotations and Features created by GATE for a Method Declaration generated by the SSLDoclet

4.1. SSLDoclet Benchmarks

We performed a performance evaluation of our doclet to assess the time needed for creating a corpus from source code. Here, the SSLDoclet is passed as a parameter to the Javadoc parser. The parser is extremely efficient compared to other parsers when processing an entire source directory. In Table 1, we show the time required to process different versions of the ArgoUML and Eclipse open source projects.

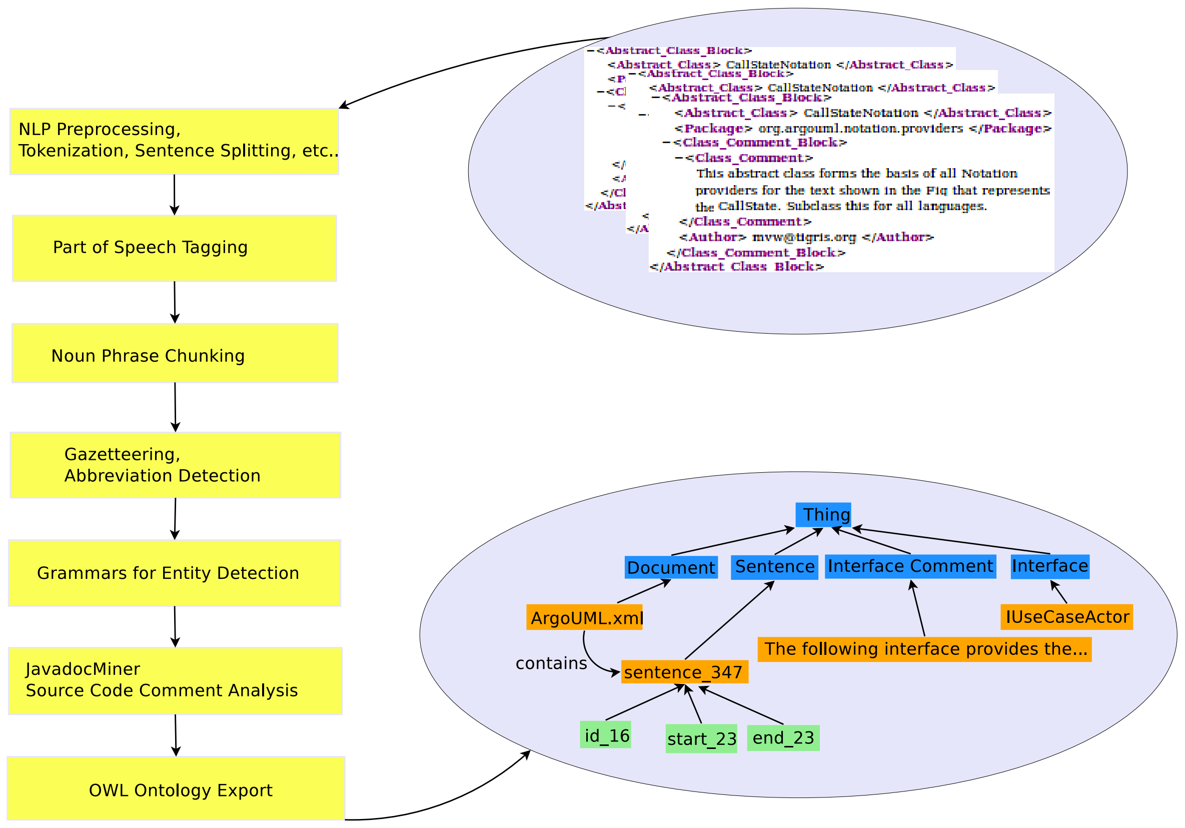
4.2. Example Application: The JavadocMiner

We developed several NLP applications that currently use a corpus generated by the SSLDoclet as input. The *JavadocMiner* is a GATE application that assesses the quality of in-line documentation written in natural language found in source code. In Figure 9, we show an illustration of the processing resources that currently make up the JavadocMiner GATE pipeline. Each processing resource adds additional information in form of annotations to the corpus.

5. Related Work

A number of other doclets exist that can create XML files using `javadoc` and information found in source code, such as the `xml-doclet`,[3] Mavens's `XMLDoclet`,[4] and finally the `jeldoclet`.[5]
However, when looking at the schema generated by these doclets, we observed that the doclets were not necessarily designed for generating a corpus to be used within NLP applications.
For example, the `xml-doclet` marks up information using only XML tags and elements and does not make use of XML attributes to represent information. As mentioned earlier, XML attributes are interpreted by NLP frameworks as features of an annotation.
A doclet that generates a schema that closely resembles the SSLDoclet is the `jeldoclet`. The `jeldoclet` however does not attempt to differentiate between the different types of comments, which could minimize the descriptiveness of the corpus. The `jeldoclet` also does not capture the information provided by Javadoc when a certain class implements or extends another class, as shown in Figure 3.
The source data being represented, and the output format is the same for all XML generating doclets, and the XML documents generated using the doclets mentioned herein can be loaded within an NLP framework. However, how the information is marked-up can drastically change the number of annotations, features and entities that are created, which can have a cascading effect on the rest of processing resource within the NLP application.
Having the most number of annotations, features or entities as result of how the information is marked up within an XML document is not necessarily beneficial. Providing a schema that enables NLP frameworks to differentiate between what is an annotation, feature, and entity is important when generating an XML document that is to be used as a corpus. None of the existing doclets that we examined were capable of doing so. For example, since the `xml-doclet` marks up all the information using XML tags only, no features are created when the document is loaded within an NLP framework and the number of annotations would exceed

[3]XML-Doclet, http://code.google.com/p/xml-doclet/
[4]Maven Doclet, http://maven.apache.org/maven-1.x/
[5]jeldoclet, http://jeldoclet.sourceforge.net/



```
-<Abstract_Class_Block>
    <Abstract_Class> CallStateNotation </Abstract_Class>
    <Package> org.argouml.notation.providers </Package>
  -<Class_Comment_Block>
    -<Class_Comment>
        This abstract class forms the basis of all Notation
        providers for the text shown in the Fig that represents
        the CallState. Subclass this for all languages.
     </Class_Comment>
     <Author> mvw@tigris.org </Author>
   </Class_Comment_Block>
</Abstract_Class_Block>
```

Figure 9: The JavadocMiner Pipeline for analysing the Quality of Source Code Comments

that of the SSLDoclet for the same amount of information. This will actually have a negative impact on the amount of work needed by the language engineers to make use of the generated corpus.

To conclude, even though there exists a number of XML generating doclets that can be downloaded from the net, we feel that our SSLDoclet differs from the rest due to its ability to generate XML output using a schema that is optimized for further NLP processing, which is an application scenario not targeted by existing efforts.

6. Conclusion & Future Work

In this paper, we presented a novel approach for using a custom doclet and Javadoc to generate a corpus that can be used as input to an NLP application. We emphasized the benefits of representing the information found in Java source code and in-line documentation using XML meta-data over HTML to facilitate automated NLP analyses.

We discussed how the SSLDoclet is able to generate a corpus using information found in both source code and in-line documentation. We also showed how the corpus can be used within existing text mining applications such as the JavadocMiner (Khamis et al., 2010).

And finally, we compared our SSLDoclet with other doclets that are currently published, and pointed out that it is the first to be explicitly designed for generating a corpus that is to be used within NLP applications. In particular, it is able to better differentiate between an annotation, feature, and entity, which the existing doclets are unable to do.

Future work is specifically needed in two areas: first, marking-up more of the information provided by Javadoc (for example, the information that exist in *enumerations*). This can be achieved by implementing more services using the Javadoc API. And second, enabling the user to generate multiple documents (a corpora) using a single source directory containing multiple files. The SSLDoclet will parse each source code file separately and generate an AST for each Java class. While this will add functionality to the SSLDoclet, we believe that it already provides significant functionality for language engineers targeting NLP analysis of source code and its inline comments.

7. References

Grigoris Antoniou and Frank van Harmelen. 2008. *A Semantic Web Primer*. The MIT Press, 2 edition, March.

H. Cunningham, D. Maynard, K. Bontcheva, and V. Tablan. 2002. GATE: an Architecture for Development of Robust HLT Applications. *Proceedings of the 40th Anniversary Meeting of the Association for Computational Linguistics (ACL)*.

Ninus Khamis, René Witte, and Juergen Rilling. 2010. Automatic Quality Assessment of Source Code Comments: The JavadocMiner. In *15th International Conference on Applications of Natural Language to Information Systems (NLDB 2010)*. Cardiff University, June 23-25.

Douglas Kramer. 1999. API documentation from source code comments: a case study of Javadoc. In *SIGDOC '99: Proceedings of the 17th annual international conference on Computer documentation*, pages 147–153, New York, NY, USA. ACM.

Erik T. Ray. 2003. *Learning XML*. O'Reilly & Associates, Sebastopol, California, 2nd edition, September.