Intelligent Software Development Environments: Integrating Natural Language Processing with the Eclipse Platform

René Witte, Bahar Sateli, Ninus Khamis, and Juergen Rilling

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

Abstract. Software engineers need to be able to create, modify, and analyze knowledge stored in software artifacts. A significant amount of these artifacts contain natural language, like version control commit messages, source code comments, or bug reports. Integrated software development environments (IDEs) are widely used, but they are only concerned with structured software artifacts – they do not offer support for analyzing unstructured natural language and relating this knowledge with the source code. We present an integration of natural language processing capabilities into the Eclipse framework, a widely used software IDE. It allows to execute NLP analysis pipelines through the Semantic Assistants framework, a service-oriented architecture for brokering NLP services based on GATE. We demonstrate a number of semantic analysis services helpful in software engineering tasks, and evaluate one task in detail, the quality analysis of source code comments.

1 Introduction

Software engineering is a knowledge-intensive task. A large amount of that knowledge is embodied in natural language artifacts, like requirements documents, user's guides, source code comments, or bug reports. While knowledge workers in other domains now routinely make use of natural language processing (NLP) and text mining algorithms, software engineers still have only limited support for dealing with natural language artifacts. Existing software development environments (IDEs) can only handle syntactic aspects (e.g., formatting comments) and some basic forms of analysis (e.g., spell-checking). More sophisticated NLP analysis tasks have been proposed for software engineering, but so far have not been integrated with common software IDEs and therefore not been widely adopted.

In this paper, we argue that software engineers can benefit from modern NLP techniques. To be successfully adopted, this NLP must be seamlessly integrated into the software development process, so that it appears alongside other software analysis tasks, like static code analysis or performance profiling. As software engineers are end users, not experts in computational linguistics, NLP services must be presented at a high level of abstraction, without exposing the details of language analysis. We show that this kind of NLP can be brought

to software engineers in a generic fashion through a combination of modern software engineering and semantic computing approaches, in particular service-oriented architectures (SOAs), semantic Web services, and ontology-based user and context models.

We implemented a complete environment for embedding NLP into software development that includes a plug-in for the Eclipse[1] framework, allowing a software engineer to run any analysis pipeline deployed in GATE [1] through the Semantic Assistants framework [2]. We describe a number of use cases for NLP in software development, including named entity recognition and quality analysis of source code comments. An evaluation with end users shows that these NLP services can support software engineers during the software development process.

Our work is significant because it demonstrates, for the first time, how a major software engineering framework can be enhanced with natural language processing capabilities and how a direct integration of NLP analysis with code analysis can provide new levels of support for software development. Our contributions include (1) a ready-to-use, open source plug-in to integrate NLP services into the Eclipse software development environment (IDE); (2) novel NLP services suitable for interactive execution in a software engineering scenario; and (3) an evaluation of a software comment quality assurance service demonstrating the usefulness of NLP services, evaluated against annotations manually created by a large group of software engineering students.

2 Software Engineering Background

From a software engineer's perspective, natural language documentation contains valuable information of both functional and non-functional requirements, as well as information related to the application domain. This knowledge often is difficult or impossible to extract only from source code [3].

One of our application scenarios is the automation of source code comment quality analysis, which so far has to be performed manually. The motivation for automating this task arises from the ongoing shift in development methodologies from a document-driven (e.g., waterfall model) towards agile development (e.g., Scrum). This paradigm shift leads to situations where the major documentation, such as software requirements specifications or design and implementation decisions, are only available in form of source code comments. Therefore, the quality of this documentation becomes increasingly important for developers attempting to perform the various software engineering and maintenance tasks [4].

Any well-written computer program should contain a sufficient number of comments to permit people to understand it. Without documentation, future developers and maintainers are forced to make dangerous assumptions about the source code, scrutinizing the implementation, or even interrogating the original author if possible [5]. Development programmers should prepare these comments when they are coding and update them as the programs change. There

[1] Eclipse, http://www.eclipse.org/

exist different types of guidelines for in-line documentation, often in the form of programming standards. However, a quality assurance for these comments, beyond syntactic features, currently has to be performed manually.

3 Design of the NLP/Eclipse Integration

We start the description of our work by discussing the requirements and design decisions for integrating NLP with the Eclipse platform.

3.1 Requirements

Our main goal is to bring NLP to software engineers, by embedding it into a current software development environment used for creating, modifying, and analysing source code artifacts. There are a number of constraints for such an integration: It must be possible to use NLP on existing systems without requiring extensive re-installations or -configurations on the end user's side; it must be possible to execute NLP services remotely, so that it is not necessary to install heavy-weight NLP tools on every system; the integration of new services must be possible for language engineers without requiring extensive system knowledge; it must be generic, i.e., not tied to a concrete NLP service, so that new services can be offered by the server and dynamically discovered by the end user; and the services must be easy to execute from an end user's perspective, without requiring knowledge of NLP or semantic technologies. Our solution to these requirements is a *separation of concerns*, which directly addresses the skill-sets and requirements of computational linguists (developing new NLP analysis pipelines), language engineers (integrating these services), and end users (requesting these services). The Web service infrastructure for brokering NLP services has been previously implemented in the open source Semantic Assistants architecture [2] (Fig 1).

Developing new client plug-ins is one of the extension points of the Semantic Assistants architecture, bringing further semantic support to commonly used tools. Here, we chose the Eclipse platform, which is a major software development framework used across a multitude of languages, but the same ideas can be implemented in other IDEs (like NetBeans).

3.2 An Eclipse Plug-in for NLP

Eclipse is a multi-language software development environment, comprising an IDE and an extensible plug-in system. Eclipse is not a monolithic program but rather a small kernel that employs plug-ins in order to provide all of its functionality. The main requirements for an NLP plug-in are: (1) a GUI integration that allows users to enquire about available assistants and (2) execute a desired NLP service on a set of files or even complete projects inside the workspace, without interrupting the user's task at hand. (3) On each enquiry request, a list of NLP services relevant to the user's context must be dynamically generated and presented to the user. The user does not need to be concerned about making any changes

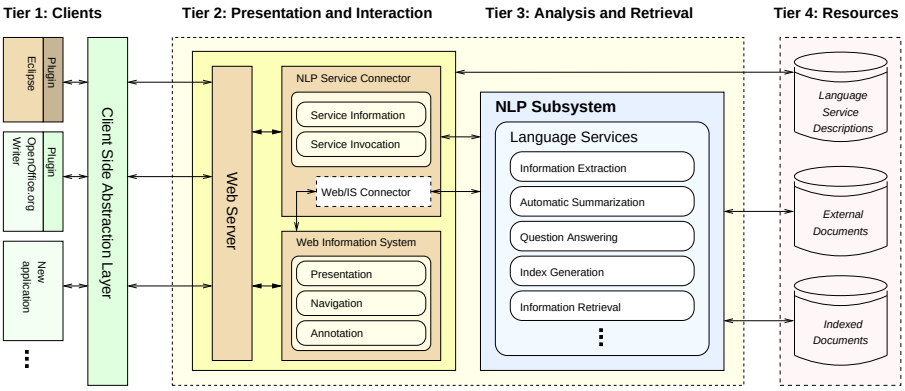


Fig. 1. The Semantic Assistants architecture, brokering NLP pipelines through Web services to connected clients, including the Eclipse client described here

on the client-side – any new NLP service existing in the project resources must be automatically discovered through its OWL metadata, maintained by the architecture. Finally, (4) NLP analysis results must be presented in a form that is consistent with the workflow and visualization paradigm in a software IDE; e.g., mapping detected NL 'defects' to the corresponding line of code in the editor, similar to code warnings displayed in the same view.

3.3 Converting Source Code into an NLP Corpus

A major software engineering artifact is *source code*. If we aim to support NLP analysis in the software domain, it must be possible to process source code using standard NLP tools, e.g., in order to analyze comments, identifiers, strings, and other NL components. While it is technically possible to load source code into a standard NLP tool, the unusual distribution of tokens will have a number of side-effects on standard analysis steps, like part-of-speech tagging or sentence splitting. Rather than writing custom NLP tools for the software domain, we propose to convert a source code file into a format amenable for NLP tools.

In the following, we focus on Java due to space restrictions, but the same ideas apply to other programming languages as well. To convert Java source code into a standard representation, it is possible to apply a Java fact extraction tool such as JavaML, Japa, or JavaCC and transform the output into the desired format. The tool that provides the most information regarding the constructs found in Javadoc comments [6] is the Javadoc tool. 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. When loading HTML documents generated using the standard doclet into an NLP framework (Fig. 2, left), the elements of an HTML tag are interpreted as being entities of an annotation. For example, the Java package (org.argouml.model) is interpreted as being of the type `h2`. This is because

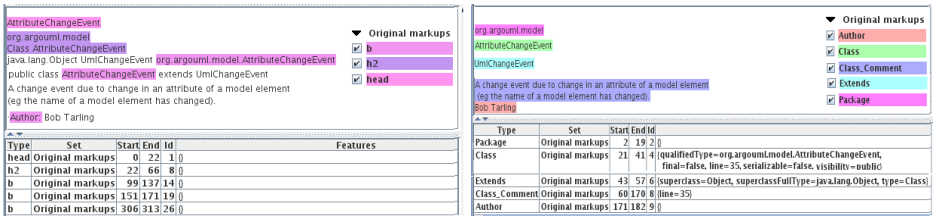


Fig. 2. Javadoc generated documentation loaded within an NLP Framework

the Javadoc standard doclet extraction tool marked up the package using the `<h2></h2>` tags. As a result, additional processing is required in order to identify the entity as being a package. In contrast, an XML document (Fig. 2, right), where the elements of the XML tags coincide with the encapsulated entity, clearly identifies them as being a `Package, Class,` etc. For transforming the Javadoc output into an XML representation, we designed a doclet capable of generating XML documents. The SSL Javadoc Doclet [7] converts class, instance variable, and method identifiers and Javadoc comments into an XML representation, thereby creating a corpus that NLP services can analyse easier.

4 Implementation

The Semantic Assistants Eclipse plug-in has been implemented as a Java Archive (JAR) file that ships with its own specific implementation and an XML description file that is used to introduce the plug-in to the Eclipse plug-in loader. The plug-in is based on the Model-View-Controller pattern providing a flexibility towards presenting annotations to the user generated from various NLP services. The user interaction is realized through using the Eclipse Standard Widget Toolkit and service invocations are implemented as Eclipse Job instances allowing the asynchronous execution of language services.

On each invocation of an NLP service, the plug-in connects to the Semantic Assistants server through the Client-Side Abstraction Layer (CSAL) utility classes. Additional input dialogues are presented to the user to provide NLP service run-time parameters after interpreting the OWL metadata of the selected service. Then, the execution will be instantiated as a job, allowing the underlying operating system to schedule and manage the lifecycle of the job. As the execution of the job is asynchronous and running in the background (if so configured by the user), two Eclipse view parts will be automatically opened to provide real-time logs and the retrieved annotations once NLP analysis is completed.

Eventually, after a successful execution of the selected NLP service, a set of retrieved results is presented to the user in a dedicated 'Semantic Assistants' view part. The NLP annotations are contained inside dynamically generated tables, presenting one annotation instance per row providing a one-to-one mapping of annotation instances to entities inside the software artifacts. The plug-in also offers additional, Eclipse-specific features. For instance, when executing source

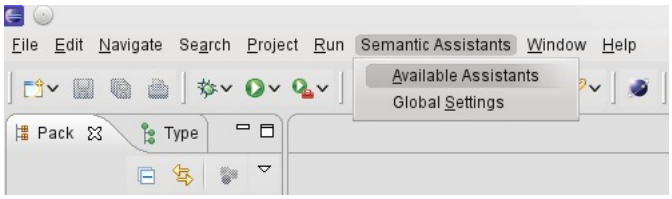
code related NLP services, special markers are dynamically generated to attach annotation instances to the corresponding document (provided the invocation results contain the position of the generated annotations in the code). This offers a convenient way for users to navigate directly from annotation instances in the Semantic Assistants view to the line of code in the project where it actually belongs, in the same fashion as navigating from compiler warnings and errors to their location in the code.

5 Applications: NLP in Software Development

In this section, we discuss application examples, showing how software engineers can benefit from integrated NLP services. One of them, the quality analysis of source code comments, is presented with a detailed evaluation.

5.1 Working with NLP Services in Eclipse

Once the Semantic Assistants plug-in is successfully installed, users can start using the NLP services directly from the Eclipse environment on the resources available within the current workspace. One of the features of our plug-in is a new menu entry in the standard Eclipse toolbar:



This menu entry allows a user to enquire about available NLP services related to his context. Additionally, users can manually configure the connection to the Semantic Assistants server, which can run locally or remote. Upon selecting the 'Available Assistants' option, the plug-in connects to the Semantic Assistants server and retrieves the list of available language services generated by the server through reading the NLP service OWL metadata files. Each language service has a name and a brief description explaining what it does. The user then selects individual files or even complete projects as input resources, and finally the relevant NLP service to be executed. The results of a successful service invocation are shown to the user in an Eclipse view part called "Semantic Assistants". In the mentioned view, a table will be generated dynamically based on the server response that contains all the parsed annotation instances.

For example, in Fig. 5, the JavadocMiner service has been invoked on a Java source code file. Some of the annotations returned by the server bear a *lineNumber* feature, which attaches an annotation instance to a specific line in the Java source file. After double-clicking on the annotation instance in the Semantic Assistants view, the corresponding resource (here, a .java file) will be opened in an editor and an Eclipse warning marker will appear next to the line defined by the annotation *lineNumber* feature.

5.2 Named Entity Recognition

The developed plug-in allows to execute any NLP pipeline deployed in GATE, not just software engineering services. For example, standard information extraction (IE) becomes immediately available to software developers. Fig. 4 shows a sample result set of an ANNIE invocation, a named entity recognition service running on the licensing documentation of a Java class. ANNIE can extract various named entities such as *Person, Organization,* or *Location*. Here, each row in the table represents a named entity and its corresponding resource file and bears the exact offset of the entity inside the textual data so it can be easily located. NE recognition can allow a software engineer to quickly locate important concepts in a software artifact, like the names of developers, which is important for a number of tasks, including traceability link analysis.

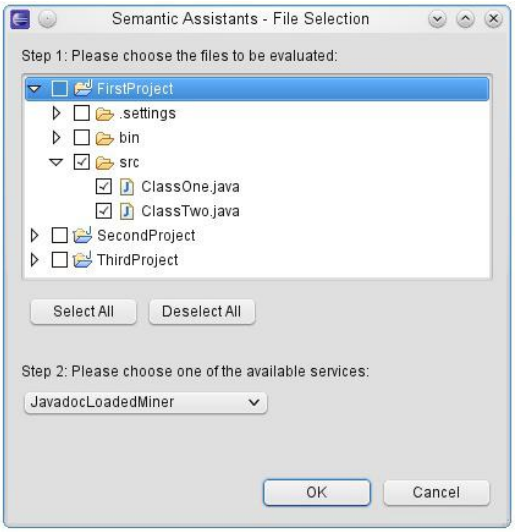


Fig. 3. Semantic Assistants Invocation dialogue in Eclipse, selecting artifacts to send for analysis

5.3 Quality Analysis of Source Code Comments

The goal of our JavadocMiner tool [4] is to enable users to automatically assess the quality of source code comments. The JavadocMiner is also capable of providing users with recommendations on how a Javadoc comment may be improved based

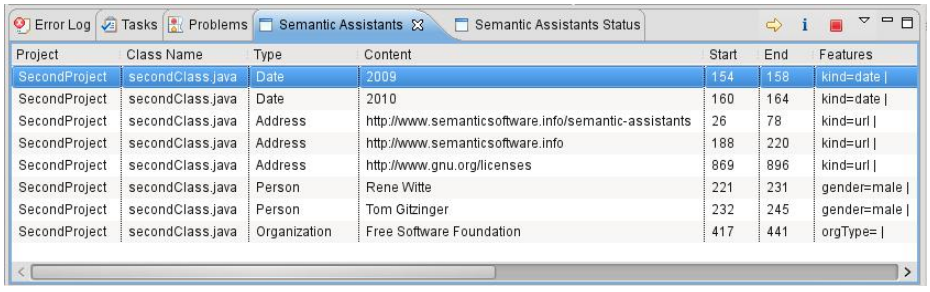


Fig. 4. Retrieved NLP Annotations from the ANNIE IE Service

on the *"How to Write Doc Comments for the Javadoc Tool"* guidelines.[2] Directly integrating this tool with the Eclipse framework now allows software engineers to view defects in natural language in the same way as defects in their code.

In-line Documentation and Javadoc. Creating and maintaining documentation has been widely considered as an unfavourable and labour-intensive task within software projects [8]. Documentation generators currently developed are designed to lessen the efforts needed by developers when documenting software, and have therefore become widely accepted and used. The Javadoc tool [6] provides an inter-weaved representation where documentation is directly inserted into Java source code in the form of comments that are ignored by compilers.

Different types of comments are used to document the different types of identifiers. For example, a class comment should provide insight on the high-level knowledge of a program, e.g., which services are provided by the class, and which other classes make use of these services [9]. A method comment, on the other hand, should provide a low-level understanding of its implementation.

When writing comments for the Javadoc tool, there are a number of guideline specifications that should be followed to ensure high quality comments. The specifications include details such as: (1) Use third person, declarative, rather than second person, prescriptive; (2) Do not include any abbreviations when writing comments; (3) Method descriptions need to begin with verb phrases; and (4) Class/interface/field descriptions can omit the subject and simply state the object. These guidelines are well suited for automation through NLP analysis.

Automated Comment Quality Analysis. Integrating the JavadocMiner with our Eclipse plug-in provides for a completely new style of software development, where analysis of natural language is interweaved with analysis of code.

Fig. 5, shows an example of an ArgoUML[3] method `doesAccept` loaded within the Eclipse IDE. After analyzing the comments using the JavadocMiner, the developer is made aware of some issues regarding the comment: (1) The *PARAMSYNC* metric detected an inconsistency between the Javadoc *@param* annotation and the method parameter list: The developer should modify the annotation to begin with the name of the parameter being documented, "objectToAccept" instead of "object" as indicated in *PARAMSYNC_Explanation*. (2) The readability metrics [4] detected the Javadoc comment as being below the Flesch threshold *FLESCHMetric and FleschExplanation*, and above the Fog threshold *FOGMetric* and *FOGExplanation*, which indicates a comment that exceeds the readability thresholds set by the user. (3) Because the comment does not use a third person writing style as stated in guideline (1), the JavadocMiner generates a recommendation *MethodCommentStyle* that explains the steps needed in order for the comment to adhere to the Javadoc guidelines.

[2] http://oracle.com/technetwork/java/javase/documentation/index-137868.html
[3] ArgoUML, http://argouml.tigris.org/

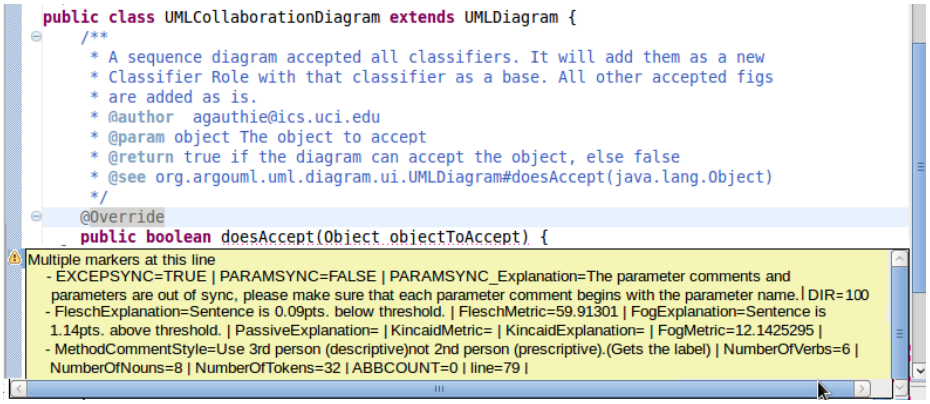


Fig. 5. NLP analysis results on a ArgoUML method within Eclipse

End-User Evaluation. We performed an end-user study to compare how well automated NLP quality analysis in a software framework can match human judgement, by comparing the parts of the in-line documentation that were evaluated by humans with the results of the Javadoc-Miner. For our case study, we asked 14 students from an undergraduate level computer science class (COMP 354), and 27 students from a graduate level software engineering course (SOEN 6431) to evaluate the quality of Javadoc comments taken from the ArgoUML open source project [10]. For our survey, we selected a total of 110 Javadoc comments:

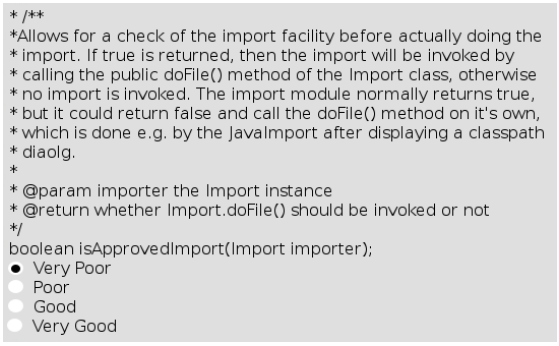


Fig. 6. A Sample Question from the Survey

15 *class and interface comments*, 8 *field comments*, and 87 *constructor and method comments*. Before participating in the survey, the students were asked to review the Javadoc guidelines discussed earlier. The students had to log into the free online survey tool *Kwik Surveys*[4] using their student IDs, ensuring that all students completed the survey only once. The survey included a set of general questions such as the level of general (Table 1, left) and Java (Table 1, right) programming experience.

The students were able to rate the comments as either *Very Poor, Poor, Good,* or *Very Good* as shown in Fig. 6, giving the comments a 50% chance of being positively or negatively classified. This also enabled us to know how strongly the participants felt about their sentiments, compared to using just a *Good* or *Bad*

[4]Kwik Surveys, http://www.kwiksurveys.com/

Table 1. Years of general and Java programming experience of study participants

	General Experience			Java Experience		
Class	**0 Years**	**1-2 Years**	**3+ Years**	**0 Years**	**1-2 Years**	**3+ Years**
COMP 354	11%	31%	58%	7%	61%	32%
SOEN 6431	02%	22%	76%	10%	49%	41%

selection. From the 110 manually assessed comments, we selected a total of 67 comments: 5 *class and interface comments*, 2 *field comments*, and 60 *constructor and method comments*, that had strong agreement ($\geq 60\%$) as being of either good (39 comments) or bad (28 comments) quality.

When comparing the student evaluation of method comments with some of the NL measures of the JavadocMiner (Table 2), we found that the comments that were evaluated negatively contained half as many words (14) compared to the comments that were evaluated as being good. Regardless of the insufficient documentation of the bad comments, the readability index of *Flesch*, *Fog* and *Kincaid* indicated text that contained a higher density, or more complex material, which the students found hard to understand. All of the methods in the survey contained parameter lists that needed to be documented using the *@param* annotation. When analysing the results of the survey, we found that most students failed to analyze the consistency between the code and comments as shown in Fig. 7. Our JavadocMiner also detected a total of 8 abbreviations being used within comments, that none of the students mentioned.

Justify:
3, +C, +D, +F //Comment indicates how process the error.
parameters are specified,but control flow should not be mentioned
2, +d
missing @param monitor
1, -B, +E
1, +B, -C, +D, +E, +F
1, +B, -C, +D, +E, +F
1, -C, +D, -E
2, +B, -C, -D
1, +C, +E

Fig. 7. A Sample Answer from the Survey

Finally, for twelve of the 39 comments that were analyzed by the students as being good, 12 of them were not written in third-person according to the guidelines, a detail that all students also failed to mention.

6 Related Work

We are not aware of similar efforts for bringing NLP into the realm of software development by integrating it tightly with a software IDE.

Some previous works exist on NLP for software artifacts. Most of this research has focused on analysing texts at the specification level, e.g., in order to automatically convert use case descriptions into a formal representation [11] or detect inconsistent requirements [12]. In contrast, we aim to support the roles of software developer, maintainer, and quality assurance engineer.

Table 2. Method Comments Evaluated by Students and the JavadocMiner

Student Evaluation	Avg. Number of Words	Avg. Flesch	Avg. Fog	Avg. Kincaid
Good	28.03	39.2	12.63	10.55
Bad	14.79	5.58	13.98	12.66

There has been effort in the past that focused on analyzing source code comments; For example, in [13] human annotators were used to rate excerpts from Jasper Reports, Hibernate and jFreeChart as being either More Readable, Neutral or Less Readable, as determined by a "Readability Model". The authors of [14] manually studied approximately 1000 comments from the latest versions of Linux, FreeBSD and OpenSolaris. The work attempts to answer questions such as 1) what is written in comments; 2) whom are the comments written for or written by; 3) where the comments are located; and 4) when the comments were written. The authors made no attempt to automate the process.

Automatically analyzing comments written in natural language to detect code-comment inconsistencies was the focus of [15]. The authors explain that such inconsistencies may be viewed as an indication of either bugs or bad comments. The author's implement a tool called `iComment` that was applied on 4 large Open Source Software projects: Linux, Mozilla, Wine and Apache, and detected 60 comment-code inconsistencies, 33 new bugs and 27 bad comments.

None of the works mentioned in this section attempted to generalize the integration of NLP analysis into the software development process, which is a major focus of our work.

7 Conclusions and Future Work

We presented a novel integration of NLP into software engineering, through a plug-in for the Eclipse platform that allows to execute any existing GATE NLP pipeline (like the ANNIE information extraction system) through a Web service. The Eclipse plug-in, as well as the Semantic Assistants architecture, is distributed as open source software.[5] Additionally, we presented an example NLP service, automatic quality assessment of source code comments.

We see the importance of this work in two areas: First, we opened up the domain of NLP to software engineers. While some existing work addressed analysis services before, they have not been adopted in software engineering, as they were not integrated with common software development tools and processes. And second, we demonstrate the importance of investigating *interactive NLP*, which so far has received less attention than the typical offline corpus studies. Our case study makes a strong case against a human's ability to manage the various aspects of documentation quality without (semi-)automated help of NLP tools such as the JavadocMiner. By embedding NLP within the Eclipse IDE,

[5] See http://www.semanticsoftware.info/semantic-assistants-eclipse-plugin

developers need to spend less efforts when analyzing their code, which we believe will lead to a wider adoption of NLP in software engineering.

Acknowledgements. This research was partially funded by an NSERC Discovery Grant. The JavadocMiner was funded in part by a DRDC Valcartier grant (Contract No. W7701-081745/001/QCV).

References

1. Cunningham, H., Maynard, D., Bontcheva, K., Tablan, V.: GATE: A framework and graphical development environment for robust NLP tools and applications. In: Proceedings of the 40th Annual Meeting of the ACL. (2002)
2. Witte, R., Gitzinger, T.: Semantic Assistants – User-Centric Natural Language Processing Services for Desktop Clients. In: 3rd Asian Semantic Web Conference (ASWC 2008). Volume 5367 of LNCS., Bangkok, Thailand, Springer (2008) 360–374
3. Lindvall, M., Sandahl, K.: How well do experienced software developers predict software change? Journal of Systems and Software **43**(1) (1998) 19–27
4. Khamis, N., Witte, R., Rilling, J.: Automatic Quality Assessment of Source Code Comments: The JavadocMiner. In: NLDB 2010. Number 6177 in LNCS, Cardiff, UK, Springer (June 23–25 2010) 68–79
5. Kotula, J.: Source Code Documentation: An Engineering Deliverable. In: Int. Conf. on Technology of Object-Oriented Languages, Los Alamitos, CA, USA, IEEE Computer Society (2000) 505
6. Kramer, D.: API documentation from source code comments: a case study of Javadoc. In: SIGDOC '99: Proceedings of the 17th annual international conference on Computer documentation, New York, NY, USA, ACM (1999) 147–153
7. Khamis, N., Rilling, J., Witte, R.: Generating an NLP Corpus from Java Source Code: The SSL Javadoc Doclet. In: New Challenges for NLP Frameworks, Valletta, Malta, ELRA (May 22 2010) 41–45
8. Brooks, R.E.: Towards a Theory of the Comprehension of Computer Programs. International Journal of Man-Machine Studies **18**(6) (1983) 543–554
9. Nurvitadhi, E., Leung, W.W., Cook, C.: Do class comments aid Java program understanding? In: Frontiers in Education (FIE). Volume 1. (Nov. 2003)
10. Bunyakiati, P., Finkelstein, A.: The Compliance Testing of Software Tools with Respect to the UML Standards Specification - The ArgoUML Case Study. In Dranidis, D., Masticola, S.P., Strooper, P.A., eds.: AST, IEEE (2009) 138–143
11. Mencl, V.: Deriving behavior specifications from textual use cases. In: Proceedings of Workshop on Intelligent Technologies for Software Engineering, Linz, Austria, Oesterreichische Computer Gesellschaft (2004) 331–341
12. Kof, L.: Natural language processing: Mature enough for requirements documents analysis? In: NLDB (LNCS 3513), Alicante, Spain, Springer (2005) 91–102
13. Buse, R.P.L., Weimer, W.R.: A metric for software readability. In: Proc. int. symp. on Software testing and analysis (ISSTA), New York, NY, USA (2008) 121–130
14. Padioleau, Y., Tan, L., Zhou, Y.: Listening to programmers Taxonomies and characteristics of comments in operating system code. In: ICSE '09, Washington, DC, USA, IEEE Computer Society (2009) 331–341
15. Tan, L., Yuan, D., Krishna, G., Zhou, Y.: /*icomment: bugs or bad comments?*/. In: SOSP '07: Proceedings of twenty-first ACM SIGOPS symposium on Operating systems principles, New York, NY, USA, ACM (2007) 145–158