Supporting Agile Software Development by Natural Language Processing

Barbara Plank[1], Thomas Sauer[2], and Ina Schaefer[3]

[1] University of Trento, Italy
[2] rjm business solutions GmbH, Lampertheim, Germany
[3] Technische Universität Braunschweig, Germany

Abstract. Agile software development puts more emphasis on working programs than on documentation. However, this may cause complications from the management perspective when an overview of the progress achieved within a project needs to be provided. In this paper, we outline the potential for applying natural language processing (NLP) in order to support agile development. We point out that using NLP, the artifacts created during agile software development activities can be traced back to the requirements expressed in user stories. This allows determining how far the project has progressed in terms of realized requirements.

Keywords: Agile Software Development, Project Management, Machine Learning, Natural Language Processing.

1 Introduction

Over the last decade, agile software development has evolved from a fiercely debated novelty into standard practice of many organizations. When properly applied, agile software development methodologies such as Scrum [14] help to develop software more predictably, more reliably, and with higher overall quality. This is mainly achieved by a iterative, incremental approach: the development process is split into iterations, which in Scrum are also known as *sprints*. In each sprint, a working increment of the system is realized. The development activities to do so are split into manageable chunks of work, so-called tasks. Developers are kept motivated by avoiding excessive documentation and unnecessary tools or procedures. An overview of agile software development using Scrum is given in Figure 1.

In Scrum, requirements are expressed as *user stories*, which describe a certain system feature from the perspective of a stakeholder [4]. User stories can refer to all sorts of requirements, including functional as well as non-functional system properties. When starting a new sprint, i.e., a new development cycle, the development team chooses as many user stories as they believe can be turned into a working increment of the product.

The *product owner* is responsible for providing enough user stories such that a working increment of the product can be actually implemented during an iteration. That is, before a new sprint can start, the product owner needs to have the user stories readily identified that are most important at the current stage of the development. After the sprint is done, the product owner is also responsible for deciding whether a user story has been sufficiently realized, or whether there is work remaining.

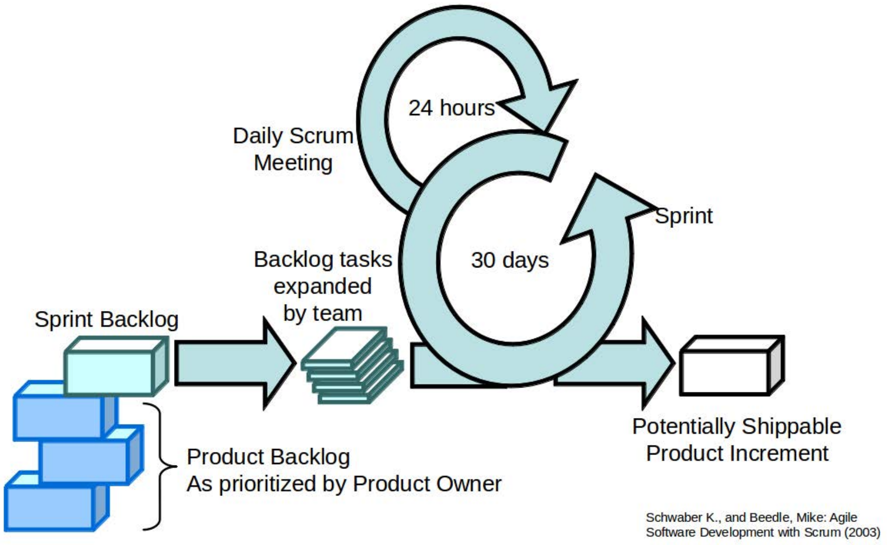
24 hours
Daily Scrum Meeting
Sprint
30 days
Backlog tasks expanded by team
Sprint Backlog
Product Backlog As prioritized by Product Owner
Potentially Shippable Product Increment
Schwaber K., and Beedle, Mike: Agile Software Development with Scrum (2003)

Fig. 1. Agile Development with Scrum

This requires that the product owner has a deep understanding of what the team has actually produced during the sprint. Especially when the product owner has to fulfill other duties in the organization, it can be overwhelmingly complex and time-consuming to keep up with the current status of the development. During a sprint, developers typically coordinate themselves in daily Scrum meetings, personal communication, etc., but the product owner is usually only a passive participant. For agile software development by Scrum to work, however, it is crucial that the product owner has the relevant requirements and user stories for the system to be developed and their priorities available when needed. Clearly, the product owner is not omniscient and may sometimes not be aware of recent development activities. Therefore, a monitoring process that can help the product owner to take decisions can prevent taking wrong priorities and thus a waste of resources.

In this position paper, we propose the use of natural language processing (NLP) techniques to overcome the problem that the product owner needs to keep track of the current status of development and the completed or non-completed user stories. By analyzing the artifacts created by the developers, such as source code, code comments or bug reports, connections can be established between the user stories that are planned to be completed during the sprint and the actual progress achieved by the development team. As the artifacts that are produced usually are not captured by some formal representation, natural language processing techniques are promising in order to automatically discover these connections. We propose a two step approach: in the first phase, *linking*, connections between artifacts and agile practices (user stories) need to be established; we can here build on prior work on traceability between software artifacts, e.g. [3,9] and initial work on linking user stories to lines of code [12], to be further discussed in Section 4. In the second phase, *information aggregation*, the previously connected data will be used for information aggregation. The goal is to abstract from the single information items found and advance current technologies in order to support the project owner by automatically providing information on the progress status to support agile project management decisions.

This paper is structured as follows: In Section 2, we review the background on development artifacts produced in Scrum and on natural language processing. In Section 3, we describe our approach. In Section 4, we discuss related work. In Section 5, we summarize this paper with an outlook to future work.

2 Background

User stories in Scrum often follow a certain template to express the roles involved, the goals to achieve, and the business value connected with the requirement. For instance, [4] suggests to express user stories in the format:

"As a (role) I want (some goal) so that (benefit)"

Figure 2 shows an example user story for implementing a string processing method. When starting a new sprint, the development team splits each user story into smaller tasks that can be accomplished in a single day. Typical tasks include implementation activities, writing unit tests, or reviewing code. For the example, there could be two tasks: the first task includes the implementation of the fancy case method and the second task concerns testing the implemented method.

User Story 101:.

```
As a string manipulation library user, I want to have a
fancy case method in order to gain fancy cased strings.
- The fancy case method should print the characters of a
string alternating in upper and lower case.
- Whitespace should be ignored.
```

Fig. 2. Example of a user story for string manipulation taken from [8]

Many development teams prefer to store the user stories and tasks for a sprint using physical task boards, index cards etc. But, there are numerous project management applications that allow keeping track of the user stories, tasks and their allocation to the different team members. For our approach, we assume that at least the product owner uses an electronic backlog to keep track of the user stories.

2.1 Development Artifacts

During a sprint, developers typically use a large variety of tools for their development activities, including IDEs, code repositories, bug tracking systems, etc. This leads to a large number of artifacts that are created besides the actual implementation when the Sprint team works on their tasks. Some of them follow some formal representation including a well-defined structure while others are more ad-hoc and mainly consist of natural language text without an external structure. The artifacts that we consider in our approach are the following. They are listed approximately in the following order: from more to less structured information (e.g. from code comments to instant messages) and according to the closeness to the source code. Some example artifacts for our running example are given in Figure 3.

Code comments:
```
// fancy case method, alternated casing
```

Unit test methods:
```
testFancyCaseMethod()
```

Commit messages:
```
commit #123: implemented user story 101
...
commit #145: fixed bug in fancycase method
```

Fig. 3. Example Artifacts for the user story given in Figure 2

Code Comments: In order to obtain code that is easy to debug and to maintain, code comments should be introduced at the relevant places to document the functionality of single methods or classes. For instance, using JavaDoc, these comments are placed within the actual code using special annotations from which an external documentation can be generated. Also unfinished issues in the code can be marked with `ToDo` such that these remaining issues can be tracked by the IDE. Both code comments and todo entries are natural text which can refer to tasks or to user stories.

Unit Test Definitions: Along the lines of test-driven development [2], the tests are written together with the implementation or even before the actual implementation. The tests refer to methods or classes in the implementation and may contain comments in natural language as well which state which scenarios from user stories are tested with the defined test cases. In the Java world, these test definitions are usually written using JUnit[1] such that they can conveniently be executed from the IDE.

Commit Messages: When a versioning system, such as Subversion or Git, is used, each developer adds the code he has implemented to complete a task or a smaller chunk of work, into a central repository. Each commit is usually accompanied

[1] `http://junit.sourceforge.net`

with a *commit message* which is usually natural language text stating which changes or additions have been made. The commit message may for instance refer to the addressed task.

Bug Reports: Deficiencies found while testing or reviewing the implementation are usually stored in a bug tracking system, such as Bugzilla[2]. Each issue found is commonly described with a unique identifier and a detailed description when and how the software misbehaves. When the problem is fixed, this is also entered into the bug tracking system, e.g., with a reference to the corresponding commit in the versioning system.

Build Scripts and Reports: Continuous integration systems such as Hudson[3] are often used to automatically integrate and compile the different components of a software system. Further, automatic testing can be triggered. In combination with the results reported, the integration steps currently configured can provide insight about the status the project is in and which parts are already finished.

Task Lists: In order to keep track of the tasks and their allocation to different team members, project management tools such as TinyPM[4] are typically used. Furthermore, the status of each task is maintained, e.g., whether a task is still pending, has been already started, or is already completed. Tasks can also be associated to the user stories within such a system. Thus, a project management system can provide the most detailed input from the management perspective, assuming that the team members keep the status of the tasks up-to-date. This facts needs to be validated from the other artifacts that are created during development.

Wiki Pages: Many organizations use Wiki systems to manage the knowledge obtained while working on a project. This may include best practices, lessons learned or specific design decisions. Wiki pages usually consist of natural language text that is only weakly structured by, e.g., marking section headings.

Calendar Entries: Group calendars are ubiquitous tools for most development teams. They may store information about meetings concerning specific issues within the development, such as decisions how to solve specific tasks. The information about the date and time of the meeting and its content, and maybe also the according meeting minutes, can provide insights into the progress of the project.

IM messages and social network postings: Instant messaging and social networks can be used by developers for quick communication with colleagues about specific issues during development, e.g., if a framework or API does not work as expected. These messages and posting may refer to tasks or the outcome of tasks and, thus, may be valuable in determining progress.

[2] `http://www.bugzilla.org`
[3] `http://www.hudson-ci.org`
[4] `http://www.tinypm.com`

This is by far not an exhaustive list, but it shows the possible artifacts that can we exploited for the proposed approach. As we have seen, the artifacts contain various levels of textual information (from short descriptions to entire Wiki pages), thereby presenting an interesting challenge for NLP, which we will introduce next.

2.2 Natural Language Processing

Natural Language Processing (NLP) [7] is an interdisciplinary field between computer science, artificial intelligence, machine learning and linguistics concerned with the study of computational approaches to understand and/or produce human (natural) language. Building systems that are able to do so is a difficult task, given the intrinsic properties of natural language. One of the major challenges for NLP is the *ambiguity* of language, exemplified in the following sentence: *The product owner gave her user stories.* Humans usually have no trouble identifying the intended meaning (that the product owner gave some user stories to 'her', presumably a software developer), while a computer usually identifies many possible readings. For example, an alternative reading is that the product owner gives some kind of stories to 'her user', thus identifying 'her' as possessive pronoun and splitting the compound noun 'user story'. Ambiguity is pertaining to all levels of linguistic processing. For instance, structural ambiguity (whether 'her' attaches to the verb or noun) or word-level ambiguity (whether her is a personal or a possessive pronoun).

While early approaches to NLP were mainly symbolic and rule-based, the field has changed dramatically since the development of annotated corpora (text collections), the introduction of machine learning and the associated growth and availability of computational power, leading to data-driven statistical approaches for learning. Current research largely focuses on the use of data-driven approaches to learn from annotated (supervised learning), partially labeled data (semi-supervised) or unlabeled data (unsupervised learning/clustering).

Some of the NLP tasks include, amongst others: part-of-speech (POS) tagging (determining the part of speech, or word-class, for each word in a sentence), named entity recognition (NER, given a text, determine which items in the text refer to, e.g. proper names, locations, geopolitical entities), parsing (extracting the syntactic structure of natural language sentences), relation extraction (RE, identify relationships between entities in text, e.g. who is working for whom), semantic role labeling (SRL, sometimes also called shallow semantic parsing, the detection of the semantic arguments associated with the predicate or verb of a sentence and their classification into their specific roles, e.g. agent, patient), Machine Translation (automatic translation between texts in different languages) and sentiment analysis (also known as opinion mining; extracting subjective information from text, e.g. opinion statements, overall polarity).

We here propose to use NLP to analyze the natural language-based artifacts created during the software development process. For instance, natural language parsing is the task of uncovering the syntactic structure of natural language sentences, which is represented in forms of trees. For example, if we apply a constituent parser (a parser that provides a hierarchical structure in which smaller parts are combined into larger parts called phrases, e.g. a *noun phrase* denoted NP) to the user story shown in Figure 2, we obtain the syntactic parse tree shown in Figure 4.

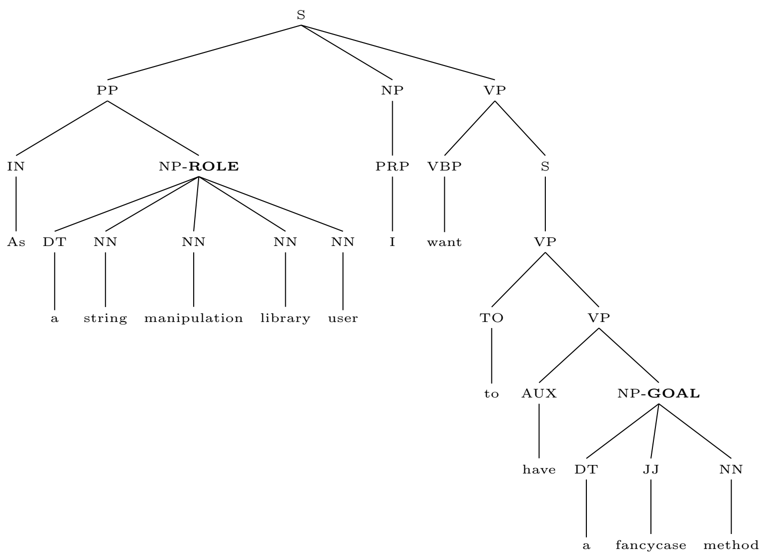

Fig. 4. A syntactic parse tree for the sentence "As a string manipulation library user, I want to have a fancycase method" from the example user story (punctuation omitted, abbreviated for space reasons). The tree is enriched with the target entity information (in bold face).

The same process can be applied to the artifacts: parsing the commit messages, the code comments, etc. Based on the syntactic structure, a classifier can be trained that determines the constituents that encode **ROLE**, **GOAL** or **BENEFIT** of a user story (indicated in bold face in Figure 4) and similarly of the artifacts. This leads to a possible structured instance representation that can be exploited, as discussed in the next section.

3 Approach

In order to establish a connection between the user stories on one side and the artifacts on the other site, we need a mechanism to associate them based on their similarity. In this section, we outline our proposed approach to apply NLP to artifacts obtained during agile software development in order to support the product owner's decisions.

To this end, we propose a two-step approach as depicted in Figure 5: In the first *linking* step, we establish connections between user stories and the development artifacts (cf. Section 2.1). In the second *information aggregation*

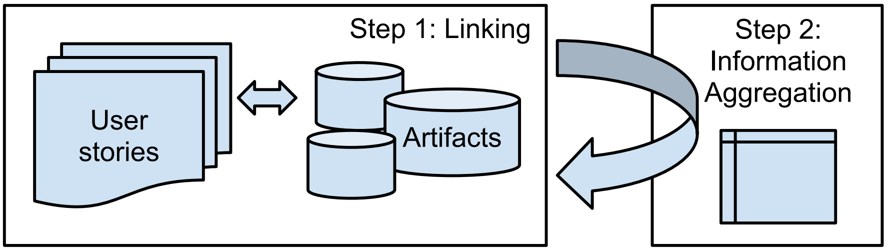

Fig. 5. Overview of the proposed approach

step, we classify user stories according to their status (to be implemented/not yet started, in progress, completed) based on the artifacts found. This helps the product owner to get a better understanding of the current status of the project at the user story level.

In the first linking step (cf. Figure 6), the information contained in the development artifacts is analyzed in order to discover which artifacts belong to the realization of which user story. For instance, a code comment or a commit message can refer to the implementation of the fancy case method of the example user story in Figure 2 allowing to link it to the first task of the user story. Additionally, the comments of a JUnit test can reference parts of the user story such that the test case can be associated to the second task of this user story. The artifacts that have tight links to the code, such as code comments or commit messages, can be augmented with information derived from bug reports or development Wiki. Also other sources of information might be exploited (which are less structured and more distant to the code, as shown in Figure 6), such as instant messaging (IM) within the company network or social network posts.

To make the linking step technically more concrete from the NLP perspective, we need to reason about i) possible instance representations of the artifacts and the user stories, and ii) possible learning mechanisms able to identify similar objects.

For the instance representation, a first attempt might consist in applying information retrieval [10] techniques: representing the information contained in the artifact or user story in a simple *bag-of-words* model in the *vector space* (i.e. counting how often a word appeared in a user story, possibly weighted). If we also want to link actual source code to user stories, then it will be also necessary to identify and split source code identifiers into actual words [9]. Then, similarity between these unstructured objects (vectors) can be calculated based on the angle between the feature vectors in the vector space (e.g. their *cosine similarity*). Alternatively, deep natural language processing might be applied to gather structured objects. For instance, the example user story could be represent as shown in Figure 7, where natural language parsing and argument classification has been applied. This representation could be further enriched with other NLP tools like a semantic role labeler, a named entity recognizer, or distributional semantic techniques. Then, machine learning algorithms able to deal with

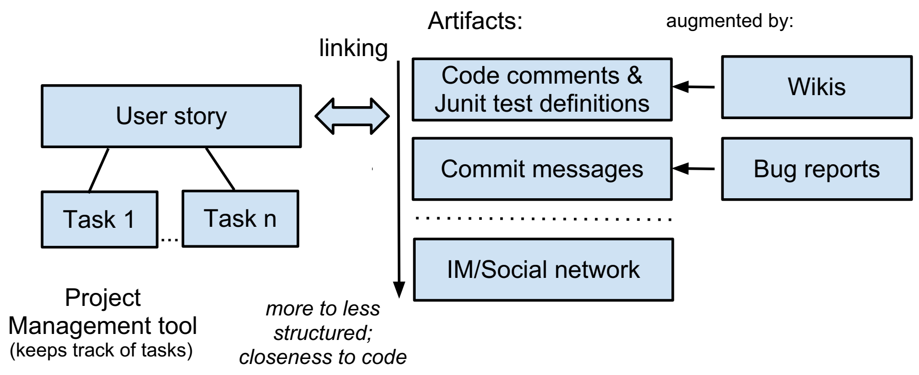

Fig. 6. Step 1: Linking User Stories with Artifacts

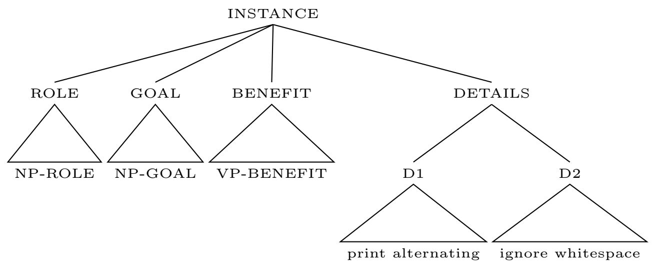

Fig. 7. Possible instance representation. The roof is a compact representation to represent tree information.

structured input data, like tree-kernel based support vector machines [5,11] could be applied to learn a similarity function in the structured space.

Once a mapping between artifacts and user stories has been established, the second information aggregation step is performed (cf. Figure 8): a classifier is trained to determine the status of the user story: “to be implemented/not yet started”, ”in progress“, “completed”. The amount of artifacts found in the first stage, as well as related meta-data (e.g. number of lines of code associated with a commit message, amount of JUnit tests related to the user story, status of unit tests, number of bugs fixed, etc.) could be exploited to train a system to classify user stories into the three categories, while further giving aggregated information on the collected artifacts. For instance, if in the example user story (cf. Figure 2), code comments and commit messages referring to the first task of implementing the fancy case method are found the user story is classified as ”in progress”. If also test cases are found with a positive reporting and no bug reports referring to the fancy case method are found, the user story can be labeled as ”completed”.

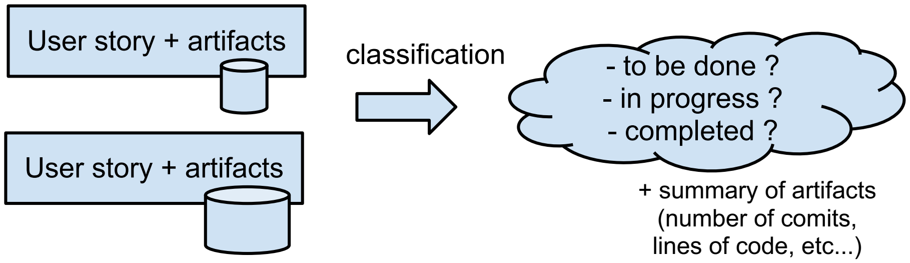

Fig. 8. Step 2: Classifying User Stories

4 Related Work

Monitoring development activities for supporting project management has been discussed before as *software project telemetry* [6]. The development environment is instrumented by software "sensors" attached to editors, test suites or bug-tracking databases. The sensors continuously send data to a central analysis component, where metrics of interest such as code churn or build failures are calculated. This enables detecting unwanted development in time. In contrast to our approach, management roles such as product owners have to draw their own conclusions how current activities are connected to specific requirements.

Connecting user stories with concrete agile development activities is discussed in [12]. The authors present a tool for associating newly created or recently modified lines of code with the individual tasks of a user story. While the initial association has to be made manually, subsequent development activity is automatically tracked by analyzing revision control usage. However, links between user stories and higher-level development artifacts other than source code are not supported.

For reducing the efforts required to link requirements with development results, *automated traceability* [3] has been suggested. By applying information retrieval algorithms, the likelihood of connections between specific requirements and code documents, UML diagrams, etc. is determined by, e.g., calculating the similarity of terms. A survey of applicable techniques can be found in [15]. The NLP approach presented in this paper augments these techniques by specifically supporting the concept of user stories in agile software development. If we also include actual source code, then an important preprocessing step related to this is the work on automatically splitting source code identifiers into component terms (e.g. *drawRectangle* or *drawrect* into *draw* and *rectangle*), as done in [9].

NLP has been already found useful for supporting specific agile techniques, such as behavior driven development [8]. Here, product owners provide an abstract test script for each user story. Using an ontology, these scripts are related with their corresponding implementation. When the development team is working on a new user story and its test script, NLP techniques are applied for extracting the nouns and verbs contained in the story. The extracted entities enable to find similar test steps by consulting the ontology, fostering efficient test code reuse.

5 Conclusion and Future Work

In this paper, we have presented the idea of using natural language processing techniques for supporting agile development. By analyzing the artifacts created during development activities, such as writing code, committing a patch, or filing a bug report, connections are established between the user stories which represent the system requirements. This supports the roles representing the stakeholders, such as product owners in a Scrum project, to understand what the team has actually produced during a development cycle.

Although user stories are expressed in free-form text, they are typically not free of form. Instead, certain templates are followed, which encode roles, goals or organizational benefits. Similar applies to artifacts such as source code, commit messages, or bug reports. This allows using proven NLP techniques to create structured representations, which in turn enables finding interdependencies.

The next step and challenge is to create a suitable training set to evaluate the presented approach. For example, an agile open source software project can be taken as a starting point.

Acknowledgements. This research has been supported by the European Communitys Seventh Framework Programme (FP7/2007-2013) under the grants #247758 (EternalS) and #288024 (LiMoSINe).

References

1. Ambriola, V., Gervasi, V.: On the systematic analysis of natural language requirements with circe. Autom. Softw. Eng. 13(1), 107–167 (2006)
2. Beck, K.: Test Driven Development By Example. Addison-Wesley (2002)
3. Cleland-Huang, J., Settimi, R., Romanova, E.: Best practices for automated traceability. Computer 40(6), 27–35 (2007)
4. Cohn, M.: User Stories Applied for Agile Software Development. Addison-Wesley (2004)
5. Collins, M., Duffy, N.: Convolution kernels for natural language. In: Proceedings of NIPS (2001)
6. Johnson, P.M., Kou, H., Paulding, M., Zhang, Q., Kagawa, A., Yamashita, T.: Improving software development management through software project telemetry. IEEE Software 22(4), 76–85 (2005)
7. Jurafsky, D., Martin, J.H.: Speech and Language Processing. Prentice Hall Series in Artificial Intelligence. Prentice Hall (2008)
8. Landhäußer, M., Genaid, A.: Connecting user stories and code for test development. In: Proc. of the 3rd International Workshop on Recommendation Systems for Software Engineering (RSSE 2012), pp. 33–37 (2012)

9. Madani, N., Guerrouj, L., Di Penta, M., Gueheneuc, Y., Antoniol, G.: Recognizing words from source code identifiers using speech recognition techniques. In: 2010 14th European Conference on Software Maintenance and Reengineering (CSMR), pp. 68–77 (March 2010)
10. Manning, C., Raghavan, P., Schütze, H.: Introduction to information retrieval. Cambridge University Press (2008)
11. Moschitti, A.: A study on convolution kernels for shallow semantic parsing. In: Proceedings of the 42nd Meeting of the ACL, Barcelona, Spain (2004)
12. Ratanotayanon, S., Sim, S.E., Gallardo-Valencia, R.: Supporting program comprehension in agile with links to user stories. In: AGILE Conference, pp. 26–32. IEEE Computer Society (2009)
13. Sawyer, P., Rayson, P., Garside, R.: Revere: Support for requirements synthesis from documents. Information Systems Frontiers 4(3), 343–353 (2002)
14. Schwaber, K., Beedle, M.: Agile Software Development with Scrum. Prentice Hall (2001)
15. Winkler, S., von Pilgrim, J.: A survey of traceability in requirements engineering and model-driven development. Software and Systems Modeling 9, 529–565 (2010)