Opportunities for Agile Documentation Using Natural Language Generation

Håkan Burden, Rogardt Heldal and Peter Ljunglöf
Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg
Gothenburg, Sweden
E-mail: burden@cse.gu.se

I. Code vs. Text

Introducing cross-functional teams is one way of reducing the manual hand-overs in a project [1]. A challenge these teams face in large-scale software development is that the items pulled from the backlog refer to a product that has evolved in a way that makes their understanding of the system obsolete. In order to quickly understand the product and the impact of implementing the backlog item it is necessary to have an accurate description of the system. However, agile processes prioritize software development before writing documents [2], causing a tension between code and documentation. Over time the code and documentation diverge so that the only way to understand the system is to dig into the code. And – as our on-going interview study has found – when software developers find it difficult to decode software, the process excludes many of the stakeholders from active participation.

II. Developers Need Accurate Texts

During interviews in our on-going research project involving Ericsson AB, Volvo Cars Group and Volvo Group Trucks Technology on model-driven software development we encountered several interviewees who addressed the problems that arise when specification and implementation differ. The interviews were conducted from the beginning of January 2013 to the beginning of June the same year. In total 25 software developers have been interviewed during the project, representing more than 25 hours of recorded dialog. While conducting the interviews, we repeatedly came across engineers who raised the problem of inconsistency between requirements and implementations as well as the effort in manually retrieving the needed information from the implementation for further development. The following quote comes from the interaction between a software engineer, SE, and one of the principal investigators, PI. The engineer brings up how the textual specification is not representative of the software models used for code generation.

SE: But we have a text document that's about 300 or 400 pages in total if you take all the documents. And that hasn't been updated for a couple of years. So this is wrong. This document is not correct.

PI: So you'd have inconsistency problems between the model description and the textual description?

SE: Yeah. The textual description is really lousy. It's really, really bad.

The result is that whenever the software engineer needs to understand a specific part of the system the solution is to dig through the implementation in form of one or more UML models intertwined with C++ code to understand the functionality of the system, instead of searching and reading the textual specification. The problem is repeated by another software engineer who describes how it was necessary to come up with a work-around for obtaining the necessary information.

We have in our requirements a list of signals used in the requirement. Now that list is seldom updated. It's hardly ever, so they're always out of date. So I don't actually read them anymore. I just go in through the specific sub-requirements and I read what is asked for my functionality. This is asked - what do I need? I need this and this. So, yeah, so I do it manually, I guess.

The work-around allows the engineer to access the relevant information but at the cost of loosing the full picture as it once was intended by the requirements. To be able to automatically retrieve a document that describes which signals that need to be implemented, lists those that are already implemented and the relationship between the signals would save a lot of time and enable the specification of the signal database to be consistent with the implementation. Today, such a protocol model is developed by hand [3].

One of the engineers, a system architect, stated that not everyone in the software development process is familiar with source code. And modern development tools have complex user interfaces and functionality: *The tools are too unintuitive [. . .] the threshold for learning how to use them is high.*

As a consequence, understanding the implementations becomes both time consuming and difficult, impacting lead times in a negative way. And each time a team pulls a new product feature from the backlog the chance is that the product to be changed has evolved since their last visit. In both cases valuable time will be spent in deciphering the code in order to estimate the impact of a change on dependent systems or estimate the time needed to develop and integrate the new feature. In contrast, the system architect concludes that *everybody knows how to consume text [. . .] text can be consumed in your favourite editor*

III. Generating Text from Source Code

Natural language generation is a framework suitable for automatically retrieving the information encoded in the implementation and display it as texts in an appropriate format

[4], [5]. The generation is done in four main steps – first the decision is made on what information should be conveyed to the consumer, the second step is to decide on how to structure the text and order the information, the third step is choosing the syntactical structure of the sentences and deciding on what terminology to use. The last step is to choose how the text is to be encoded for presentation, e.g., adding appropriate tags for rendering html- or LaTeX-documents.

In a recent literature review on the generation of requirements from software models 17 out of the 24 cited publications generated natural languages, six generated formal languages and one publication generated a combination of natural and formal languages [6]. None of the cited publications relate their generated texts to agile software documentation.

Since the publication of the literature review a number of new contributions have emerged. Among those are three contributions on natural language generation from Java code [7], [8], [9]. The motivation behind these contributions is that understanding code is a time consuming activity and accurate descriptions can both summarise the algorithmic behaviour of the code and reduce the amount of code a developer needs to read for comprehension. The automatic generation of summaries from code mean that it is easy to keep descriptions and system synchronized. The generated texts are evaluated in an academic setting.

Rastkar et. al. [10] generate natural language specifications of crosscutting concerns. A crosscutting concern is a functionality that is defined in multiple modules and as a result of their scattered nature it is a complex task to understand how a change in functionality is going to spread across the logical structure of the source code. The authors conclude that having a natural language summary for each crosscutting concern and where it is implemented helps developers handle software change tasks. Again, the generated texts are evaluated in an academic setting.

The only evidence we have found so far of an industrial case study is reported by Spreeuwenberg et. al. [11]. Here, software models are used to define candidate legislature for the Dutch Immigration Office. Since the models are difficult to decode for the stakeholders with an expertise in legal and administrative issues a natural language representation of the models is generated to include all stakeholders in the validation procedure of encoding new laws and regulations.

IV. Agile Documentation through Generation

We have shown how the inconsistency between software specifications and implementations causes problems for the software developers. If it is a challenge for system architects and coders it is also a problem that excludes the active involvement of many stakeholders that do not have the necessary training for decoding software. It is also clear that there is a gap between the academic efforts put into text generation from source code and the application of the same in industry.

An operations document that summarizes the dependency relations to other systems, use of databases together with contact points etc., often follows a predefined template [3]. Such a text is feasible to generate given the advances shown in recent publications and would be helpful for the interviewed system architect who would get a textual representation that is up-to-date with the implementation. Another document that should be straight-forward to generate is a contract model that describes the technical interface of a (sub-)system [3]. Generating such a document with a list of the signals actually used by the system would be another useful document if it could be automatically generated, according to our interviewees.

Using the code for generating documents that describe different system properties enables a single source of information in the development process, aligning documentation with implementation. Different views could then be generated from the code during implementation, with content and structure depending on who the consumer of the document and what the purpose of the text is. This could be done together with the relevant stakeholders, enabling concise documents that are easy to keep consistent with the implementation. The opportunities for collaborations between industry and academia in text generation as a means for agile documentation are abundant!

Acknowledgment

The authors would like to thank the Graduate School in Language Technology, the Center of Language Technology and the Software Center for funding and support as well as the contact persons at respective company for facilitating the study.

References

[1] K. Schwaber and M. Beedle, *Agile Software Development with Scrum*, 1st ed. Upper Saddle River, NJ, USA: Prentice Hall PTR, 2001.

[2] J. Highsmith and M. Fowler, "The agile manifesto," *Software Development Magazine*, vol. 9, no. 8, pp. 29–30, 2001.

[3] S. Ambler, "Agile/Lean Documentation: Strategies for Agile Software Development," www.agilemodeling.com/essays/agileDocumentation.htm, accessed June 19th 2013.

[4] E. Reiter and R. Dale, *Building Natural Language Generation Systems*. Cambridge University Press, 2000.

[5] J. Bateman and M. Zock, "Natural Language Generation," in *The Oxford Handbook of Computational Linguistics*, ser. Oxford Handbooks in Linguistics, R. Mitkov, Ed. Oxford University Press, 2003, ch. 15.

[6] J. Nicolás and J. A. T. Álvarez, "On the generation of requirements specifications from software engineering models: A systematic literature review," *Information & Software Technology*, vol. 51, no. 9, pp. 1291–1307, 2009.

[7] E. Hill, L. L. Pollock, and K. Vijay-Shanker, "Automatically capturing source code context of nl-queries for software maintenance and reuse," in *Proceedings of the 31st International Conference on Software Engineering*. Vancouver, Canada: IEEE, 2009, pp. 232–242.

[8] G. Sridhara, E. Hill, D. Muppaneni, L. Pollock, and K. Vijay-Shanker, "Towards automatically generating summary comments for Java methods," in *Proceedings of the IEEE/ACM international conference on Automated software engineering*, ser. ASE '10. New York, NY, USA: ACM, 2010, pp. 43–52.

[9] G. Sridhara, L. Pollock, and K. Vijay-Shanker, "Automatically detecting and describing high level actions within methods," in *Proceedings of the 33rd International Conference on Software Engineering*, ser. ICSE '11. New York, NY, USA: ACM, 2011, pp. 101–110.

[10] S. Rastkar, G. C. Murphy, and A. W. J. Bradley, "Generating natural language summaries for crosscutting source code concerns," in *27th International Conference on Software Maintenance*. Williamsburg, VA, USA: IEEE, September 2011, pp. 103–112.

[11] S. Spreeuwenberg, J. Van Grondelle, R. Heller, and G. Grijzen, "Design of a CNL to Involve Domain Experts in Modelling," in *CNL 2010 Second Workshop on Controlled Natural Languages*, M. Rosner and N. Fuchs, Eds. Springer, 2010, pp. 175–193.