Ina Schaefer, Dimitris Karagiannis, Andreas Vogelsang, Daniel Mendez (Hrsg.): Modellierung 2018,
Lecture Notes in Informatics (LNI), Gesellschaft für Informatik, Bonn 2018 1

Implementing Knowledge Management in Agile Projects by Pragmatic Modeling

Harald Störrle[1]

Abstract: BACKGROUND: Team knowledge is diluted and destroyed through domain evolution and staff turnover. A challenge to any project, agile projects are particularly vulnerably as they rely more on tacit knowledge than plan-based approaches. Increasing project sizes and durations deteriorate this situation. Introducing documentation and modeling to turn tacit into explicit knowledge as exercised in traditional approaches is perceived as costly, and impeding with agility.
OBJECTIVE: We want to improve agile practices for large, long-running projects by adopting and adapting long-standing modeling practices, challenging these practices in the process. We aim to establish a more pragmatic view of what should be considered a model, and how complex system models could be organized to better support their usage.
METHOD: We propose several additions and changes to existing agile practices, and a new notion of model. We highlight how models are used in industry, and how existing modeling languages and tools might be improved to better support these usage modes.
RESULTS: We have successfully implemented our approach in a large project. A transfer to a smaller, more typical agile project in a different environment is under way.
CONCLUSIONS: Modeling can be a valuable and appreciated addition to agile development projects. However, this requires a pragmatic approach beyond the conventional wisdom of MDE and academic modeling practices. A broader view on what models and modeling are is useful in practice, and offers relevant new research questions.

1 Introduction

Over the past 20 years, agile approaches to software development have evolved from innovation to main-stream. The *"home ground"* [Bo02, p. 64] of agile approaches are smaller projects in dynamic environments with few external constraints. Pushing for the limits, though, practitioners have attempted to apply agile practices also to large and long-running projects in complex domains or highly regimented environments, facing substantial *"barriers"* [BT05, p. 30]. In this paper we argue that the commonality in all these barriers is the (un)availability of knowledge in a team.

Obviously, software engineering is a highly knowledge-driven activity, making software engineers prototypical knowledge workers [CM04], and underscoring the importance of knowledge sharing. Agile approaches acknowledged this from the start, aiming to *"reduce*

[1] Dr. Harald Störrle, Principal IT Consultant, QAware GmbH, Aschauer Str. 32, 82152 München, Harald.Stoerrle@qaware.de

doi:18.18420/to-be-provided-by-editor

the cost of moving information between people" [CH01, p. 131] by striving to "*replace documents with talking in person and at whiteboards*" (op cit.).

Therefore, agile approaches tend to *"focus on individual competency as a critical factor in project success"* [CH01, p. 131], relying critically on tacit knowledge ("in the heads of the team members") [Bo02], and de-emphasizing the role of explicit knowledge as expressed in documents, models, and other artifacts.[2] Clearly, relying on tacit knowledge presents a critical challenge for agile approaches when faced with large teams, high staff turnover, or long running projects [NMM05, BT05, Hi03]. Consider these factors in how they affect knowledge in a development team.

- **Team Size:** With increasing team size, it is more and more unrealistic to assume that all team members are exchangeable, with the same level and profile of expertise, interests, strengths, and experience. Thus, larger teams will exhibit an increasing differentiation of roles filled within a team. Inevitably, team *structures* will emerge, be they formal or informal.
- **Staff turn over:** Clearly, when knowledge is primarily tacit ("in the heads"), removing experienced and knowledgeable members from the team also removes knowledge from the team. Explicit knowledge codified in documents, on the other hand, is much less affected by staff turn over.
- **Long-running projects:** Regardless of staff turnover, passing time alone brings about changes in the application domain, system structure, and technologies used. Therefore, knowledge decays over time: similar to technical debt accumulating with interest over time, technology and domain "inflation" depreciate the value of knowledge. Unlike staff turn over, however, these changes affect tacit and explicit knowledge in similar ways: both of them decay and need active effort to keep up to date.

In short: staff turn over amounts to knowledge loss and the inevitable differentiation of growing teams impedes with agile practices to compensate loss of knowledge. Conventional measures to turn tacit knowledge into explicit knowledge, i. e., documentation and modeling, are no solution: Simply transplanting such approaches and saddling developers with these tasks risks agility [BT05, p. 34], and does not solve the issue of knowledge decay, thus limiting their expected benefit. This leads to the research question we explore in this paper:

RQ: How can we adapt existing methods to reduce loss or decay of knowledge in large, long-running agile projects, yet maintain agility as far as possible?

In this paper we report on a project that overcame this challenge by re-interpreting what it means to model, and how models can be used as an effective knowledge management medium. Our observations generalize to projects that start out highly agile, and evolve into a more moderate pace as they mature, addressing the needs of increasing business criticality.

[2] Davenport and Prusak define knowledge as *"a fluid mix of framed experience, values, contextual information, and expert insight that provides a framework for evaluating and incorporating new experiences and information. [...] In organizations, it often becomes imbedded (sic.) [...] in documents or repositories [...]"* [DP98, p. 5]. See [Po66] for an explanation of the dichotomy of tacit and explicit knowledge.

2 Documentation in practice

In order to turn implicit knowledge into explicit knowledge, it must be committed to writing (or drawing) in one form or another. This inevitably results in (large) sets of documents in a multitude of more or less organized storage spaces. Typically, there would be a mixture of shared drives with a folder hierarchy of many documents in different formats, quite possibly with some "proper" models[3] mixed in. Additionally, there might be Wikis, paper documents, posters and drawings on walls and whiteboards, and the proverbial personal drawer.

Clearly, in such a form of documentation, it is difficult to find information, and even more difficult be sure that a particular bit of information is missing. This makes adding and updating documents difficult and error-prone, increasing the burden of documentation further. As a consequence, information gaps, duplicates, and diverging variants of documents will appear increasingly. In short, the cost-benefit proposition of such a form of documentation is not very attractive.

Another way of documenting large scale systems is by creating comprehensive models. However, as such models get large, structuring them offers a challenge of its own [St10]. Also, whatever modeling language is used, it is not as universally (and proficiently) adopted as natural language, and improvised sketches. Then, all existing modeling languages have more or less severe shortcomings, and often are not a good match for the application domain. For instance, UML has been criticized repeatedly for the lack of perceptual support and conceptual clarity [FS07]. Finally, existing modeling tools fall short of users' legitimate expectations regarding functionality and usability.

Among the many forms of documentation frequently found in real projects, two stick out. First, there are many informal diagrams which team members would use to illustrate a point, or supplement communication with one another, clients, or other stakeholders. In order to clearly separate them from the more regimented and restricted diagrams found in UML and similar languages, we will use the name *sketch* for these artifacts. Sketches are very widely used [BD14], and often resemble well-known notations such a

Second, one often discovers documents with model-like content in a fairly rigid structure, with no attached visualizations but created mainly for human users. For instance, data models might be described as text documents with a section per entity containing a table of fields/columns and their respective properties. Likewise, one often sees spreadsheets with lists of REST-endpoints, use cases, or states and applicable triggers. So, following generic definitions of models in software development [St73, Lu03, Ma09], we maintain that for all intents and purposes, structured text or tables can act as models. In the remainder, we call such documents *modeloids*.

With sufficient effort, both sketches and modeloids could be transformed into "proper" models, at least to a substantial degree. Quite possibly, we might have to add either comments

[3] In the remainder, we use the term model to refer to such "proper" models, i. e., expressed in one of the common languages like UML, created by a specific tool, and offering diagrams along with semantic structures.

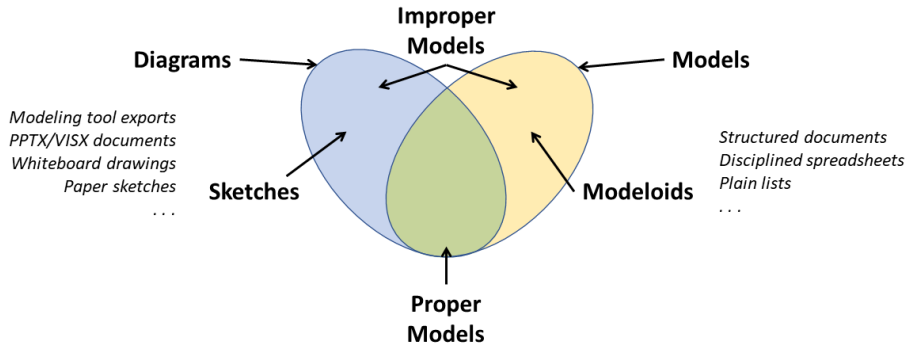

Fig. 1: Models, Diagrams, and Modeloids: differentiating the content from the presentation aspect

to capture the prose description of modeloids, or visual annotations and colors to mimic the visual appeal of sketches. Of course, it is questionable whether spending this effort is justified by the benefit, and whether it is realistic to expect this to happen in practice. In both cases, in our experience, the answer is no. This begs the question how we could use sketches and modeloids as elements in knowledge management of software development projects rather than "proper" models.

3 Pragmatic Agile Documentation

Our approach combines three major elements: a single unified repository, improved access paths, and content curation.

3.1 Single unified repository

Development projects typically use shared drives, version controlled repositories like svn or git, wikis, ticket systems, chats, emails, various forms of dedicated collaboration software (e. g. Slack, Google Docs), along with conventional modeling tools, and analog media. The existence of multiple storage spaces is obviously an impediment to finding explicit knowledge, and keeping it consistent and up to date. So, the first goal is to integrate all existing sources of documentation into a single Project and System Knowledge Base (PSKB). See **??** for data from a case study project.

In practice, there will never be a single, unified storage for all documents. We can strive to reduce their number, though, and impose clearly defined usage roles. We suggest to use an enterprise strength wiki system as the information backbone. It offers several advantages over the classic documents-in-subversion approach.

- **comprehensive** Wikis allow all forms of documents, from simple wiki pages with embedded images and links, to attachments (including modeloids);
- **contextuality** Wikis easily allow to provide context to all documentation elements;
- **changeable** Grace to the simplicity of mark-down and the built-in straightforward version tracking on individual pages, it simplifies changing it and is forgiving in the face of errors;
- **collaborative** Many wikis facilitate distributed collaboration across organizational boundaries.

In our case, we use the pre-existing Confluence Wiki. Apart from already being in place and widely used, it offers rich functionality, including integration with Jira and HipChat.

3.2 Improved access paths

In order to cater for different capability levels and tastes of different users, we provide multiple alternative access methods to the information in the knowledge base. The Confluence wiki comes with a built-in conventional full-text search and sophisticated filtering. Information spaces are organized in a tree-structured page hierarchy, that allows users to navigate the tree very much the same way they would navigate a directory hierarchy.

Additionally, we also provide a visual path to knowledge through Visual Access Maps (VAM). VAMs are (large) diagrams representing important aspects or parts of the overall system or project, where diagram elements are instrumented with links to pages or attached documents that provide more detailed information to that element. For instance, the system architecture map affords links to the major neighboring systems, subsystems, interfaces, data items and use cases. It is irrelevant whether these diagrams follow any particular well-defined syntax, whether they overlap, or what aspect they cover – organization charts or marketing can be as useful as ER diagrams or UML Assembly Diagrams[4]. Particularly, informal sketches can be used as well as "proper" diagrams. That way, existing diagrams can be reused with little additional effort.

Likewise, it does not matter what the elements link to. A link's target my be another diagram (i. e., zooming into a refinement), a wiki page with some description, an attached slide set with a high-level description, or a lengthy table with detailed information. Or, in fact, a combination of the above. Additional elements that do not naturally fit into given model structures (e. g., a glossary) may be included by simply adding a document icon with a link.

Upgrading existing diagrams to VAMs allows user to more easily relate to the new structure. Also, reusing existing material makes for a faster bootstrap of the new documentation approach and thus supports overall project agility. Contrary to intuition, it is not cheaper, though, as most of the effort stems from stratifying the diagram (improving layout, closing gaps), and adding links to it. Obviously, the more abstract and high-level these diagrams

[4] Also known as Part-Port-Diagrams

are, the lower the change rate, and, thus, the smaller the maintenance effort. Either way, keeping such diagrams up to date is part of the ongoing curation process (see below).

Using all forms of improper models directly, we not only save the effort of transforming them into proper models, but also ensure that the stakeholders who created them can also maintain them, even if they do not speak whatever modeling language is "the right one". Working with what we actually find, and changing as little as possible also allows us to progress much more quickly—no big up-front efforts are required. This way, it is possible adopt a modeling approach in an agile project.

By focusing on existing documents we ensure that the right pieces are documented: if people found it necessary to write the documents, they are obviously worthwhile. Also, there is no risk of impeding agility by burdening a team with unwanted documentation.

All in all, the PSKB created this way very much resembles a large, well-organized UML model with many diagrams in a proper modeling tool, except that it readily incorporates all kinds of diagrams, and all kinds of information. Syntactic and semantic restrictions embedded in UML (or any other language) no longer apply. This new-won freedom must be used with care, though, to ensure a prolonged usefulness of the overall structure.

3.3 Content curation

Content curation consists of two parts. First, building the knowledge base requires an initial, one-time effort to transform existing elements of documentation and consolidate them into an the PSKB. In our case, the pre-existing Wiki-page hierarchy needed to be re-organized, cleansed, and consolidated. The pre-existing multiple large documents needed to be split up and inserted into the PSKB in suitable places.

Second, in order to maintain the quality level reached, ongoing curation is indispensable. We suggest to add the dedicated role of Knowledge Manager, supplemented by cyclic team efforts ("Documentation Day") similar to refactoring of source code and system structure. Repeated reminders in sprint retrospectives serve to maintain a focus on the quality of documentation. We used a list of knowledge gaps for identified "undocumented important knowledge" and "known unknowns", and a ticket system for larger issues with the existing documentation.

4 Case Study: RepairResearch

In this section we describe the RepairResearch (RR) project in which we developed and tested our approach. RR has been launched by a major premium car manufacturer to support after sales activities, mainly maintenance, repair, and upgrade of vehicles. Each week, information on over 220.000 vehicles is delivered to approx. 5.000 parties world wide via RR.

The RR system consists of over 200KLoC of source code, mostly Java, and employs substantial amounts of third party code. Launched in 2011, over 130 person-years have been spent creating RR (not counting client efforts). In total, over 60 different people have worked on this project. The staff size ranged from 25 to 35 people (currently 26 people, amounting to 19.3 full time equivalents), with some more loosely attached on a need-to basis.

Starting with hardly any documentation, the number of people involved and the (inevitable) staff turnover required more and more elements of explicit knowledge. Also, realizing the criticality of the application and the knowledge monopoly of particular individuals, the client demanded conventional documentation. Initially, there was a Wiki with over 500 pages, more than 900 attachments (more than 700 images, 145 documents), and a total volume of almost 300MB. There were three shared drives with a combined volume of over 200GB, in almost 15,000 folders with approx. 100,000 files. The so called System Handbook alone comprised 400 printed pages, and was complemented by handbooks for architecture, maintenance, configuration management, and users. There were several overlapping glossaries, and many versions of project plans, organization charts and similar diagrams. Many team members had (outdated) copies of subsets of these resources.

While a great deal of information was *technically* available, many team members (client side as well as supplier side) felt that actually very little information was practically available. This particularly felt by new team members and those whose work required familiarity with multiple topics. Probing deeper, it appeared that the reasons for this sentiment included the following.

- **Location** The multiple storage locations contained overlapping or complementary information.
- **Access** The stores had different, and sometimes difficult to use access paths; full-text search covered only part of the storage.
- **Obsolescence** Some (parts of) information resources were outdated/obsolete.
- **Ambiguity** Information in different places was partially inconsistent.
- **Gaps** Even considering all information stores, there were many actual gaps in the documented knowledge.

Judging from past projects, we consider this size and state typical for a project of this scope and age, irrespective of whether they apply agile practices or not. However, agile practices with the emphasis on code rather than other artifacts are bound to suffer more from the impact of unregulated proliferation of documentation. This is when the RR project decided to meet the challenge by a dedicated knowledge management effort. Knowledge management has been an official role in RR since 2014, though the activities were low-key. In early 2016, a large number of experienced team members were shifted out, both client side and supplier side, leading to increased demand of the remaining experts, and requests for more explicit knowledge. By autumn 2016, knowledge management activities were ramped up resulting in the solution described in this paper. The projects adopted the approach in late 2017.

Our experience so far is promising: team members report improved quality and availability of system and project knowledge, which is felt almost immediately. However, it might be that the benefits we experience are due to the initiative as such (Hawthorne effect). Also, it appears that the investment up to this point was well-spent and yields a positive return on invest. However, it is too early to say whether our approach is truly sustainable and self-perpetuating. This can only be judged in hindsight, that is, in a few years. Finally, we have no hard evidence on the relative sizes of cleaning up obsolete and overlapping documents, filling actual gaps, improving access by visual access, and unifying storage locations. Thus, this case is but a first exploration into the opportunities.

We observed that implementing the PSKB did not in any way displace the existing system and domain experts—they were just as sought after for sharing personal insight as the were before. However, they reported ‘questions to become “more interesting” as many simple questions can now be answered by other team members. The effort of building and maintaining the PKSB are so far smaller than expected. The initial set up effort amounted to roughly 40 person-days (8% of a project-month), and ongoing curation comes at 1-2 person days per month (2-3% of a project month). Periodic “documentation days” cost roughly a full person-day of a quarter the team every three months (1-2% of a project month). Altogether, this amounts to 10% of a project month once, and 2-4% of the budget continuously, on top of whatever effort was spent by individuals for creating specific bits of knowledge, which did not change.

5 Related Work

Davenport and Prusak’s seminal work [DP98] defines knowledge in the sense used in this paper, and Polanyi supplements the distinction between tacit and explicit knowledge [Po66]. Schneider provides a comprehensive and accessible introduction to knowledge management in the context of software engineering [Sc09]. Scientific reviews of the field are provided in [BD08, RL02], though both are somewhat dated and shed little light on KM practices in agile projects.

Primary research about KM for SE in SMEs using agile practices goes back to [Di02], where post-mortems, pair programming, and team rotation are highlighted as effective knowledge sharing practices. As we have outlined in the introduction, these practices are limited in scope to small projects with stable teams and little role differentiation. Maurer [Ma02] examines limiting factors to scaling agile approaches, but considers only alternatives to co-location, sidestepping growing team size and project duration.

There are three popular flavors of applying agile principles in larger scale projects: Large

Scale Scrum (LeSS)[5], Disciplined Agile Delivery (DaD)[6], and the Scaled Agile Framework (SAFe)[7] (see [He] for a comparison of these approaches).

Chau et al. [CMM03] compared the knowledge management mechanism of agile and plan-based development projects. Dorairaj et al. [DNM12] acknowledge that knowledge management is an important part of agile development projects, and report on a qualitative study of practitioners experienced in large scale agile projects.

6 Conclusion

Agile approaches to software development live by the assumption that *"direct communication is more effective than documentation"* [Ma02, p. 14] which is a "*limiting factor for scalability*" (op cit.). As projects grow in size and lifetime, the mechanisms proposed to promote knowledge sharing in agile environments (stand ups, retrospectives, rotation, wiki) become more and more ineffective. Traditional formats of explicit knowledge (i. e., documentation) like models or prose documents, however, are not considered cost-effective. So, we propose a novel approach that amalgamates elements from both worlds into a Wiki hypertext structure, integrating proper diagrams and sketches, as well as proper models and model-like documents ("modeloids"). Arbitrary elements may be attached to wiki pages. Visual access maps facilitate finding elements of knowledge. Our approach is practical, and has proven its viability in a real world, very large agile project.

We are currently applying our approach to a second project with a very different context. In particular, this new case study has a much smaller team size, more like the typical agile project. This project will teach us how to adapt our approach for smaller projects, and whether it is still economically (and practically) viable under those circumstances.

References

[BD08] Bjørnson, Finn Olav; Dingsøyr, Torgeir: Knowledge Management in Software Engineering: A Systematic Review of Studied Concepts, Findings and Research Methods Used. Information and Software Technology, 50(11):1055–1068, 2008.

[BD14] Baltes, Sebastian; Diehl, Stephan: Sketches and Diagrams in Practice. In: Proc. 22nd ACM SIGSOFT Intl. Symp. Foundations of Software Engineering (FSE). ACM, pp. 530–541, 2014.

[Bo02] Boehm, Barry: Get ready for agile methods, with care. IEEE Computer, 35(1):64–69, 2002.

[BT05] Boehm, Barry; Turner, Richard: Management challenges to implementing agile processes in traditional development organizations. IEEE Software, 22(5):30–39, 2005.

[5] `https://less.works/`

[6] `http://www.disciplinedagiledelivery.com/agility-at-scale/large-agile-teams/`

[7] `http://www.scaledagileframework.com/`

[CH01] Cockburn, Alistair; Highsmith, Jim: Agile Software Development: The People Factor. Computer, pp. 131–133, 2001.

[CM04] Chau, Thomas; Maurer, Frank: Knowledge Sharing in Agile Software Teams. Logic versus approximation, pp. 173–183, 2004.

[CMM03] Chau, Thomas; Maurer, Frank; Melnik, Grigori: Knowledge sharing: Agile methods vs. tayloristic methods. In: Proc. 12th IEEE Intl. Ws. Enabling Technologies: Infrastructure for Collaborative Enterprises (WETICE). IEEE, pp. 302–307, 2003.

[Di02] Dingsøyr, Torgeir: Knowledge management in medium-sized software consulting companies. Empirical Software Engineering, 7(4):383–386, 2002.

[DNM12] Dorairaj, Siva; Noble, James; Malik, Petra: Knowledge management in distributed agile software development. In: Proc. Agile Conf. (AGILE). IEEE, pp. 64–73, 2012.

[DP98] Davenport, Thomas H.; Prusak, Laurence: Working Knowledge: How Organizations Manage What They Know. Harvard Business School Press, 1998.

[FS07] Fish, Andrew; Störrle, Harald: Visual qualities of the Unified Modeling Language: Deficiencies and Improvements. In (Cox, Phil; Hosking, John, eds): Proc. IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC). IEEE CS, pp. 41–49, 2007.

[He] Heusser, Matt: Comparing scaling agile frameworks. CIO Magazine. originally published 2015-08-21, last acessed at 2017-12-07.

[Hi03] Highsmith, Jim: Agile Project Management: Principles and Tools. Cutter Consortium Reports, 4(2), February 2003.

[Lu03] Ludewig, Jochen: Models in Software Engineering – an introduction. J. Softw. Syst. Model., 2(1):5–14, 2003.

[Ma02] Maurer, Frank: Supporting Distributed Extreme Programming. In (Wells, Don; Williams, Laurie, eds): Proc. 2nd XP Universe, Proc. 1st Agile Universe Conf. (XP/Agile Universe). Springer Verlag, pp. 13–22, 2002.

[Ma09] Mahr, Bernd: Information Science and the logic of models. J. Softw. Syst. Model., pp. 365–383, 2009.

[NMM05] Nerur, Sridhar; Mahapatra, Radha-Kanta; Mangalaraj, George: Challenges of Migrating to Agile Methodologies. CACM, 48(5):72–78, 2005.

[Po66] Polanyi, Michael: The Tacit Dimension. Doubleday & Company, 1966.

[RL02] Rus, Ioana; Lindvall, Mikael: Knowledge management in software engineering. IEEE Software, 19(3):26, 2002.

[Sc09] Schneider, Kurt: Experience and Knowledge Management in Software Engineering. Springer Verlag, 2009.

[St73] Stachowiak, Herbert: Allgemeine Modelltheorie. Springer Verlag, 1973.

[St10] Störrle, Harald: Structuring very large domain models: experiences from industrial MDSD projects. In (Gorton, Ian; Cuesta, Carlos E. Babar, Muhammad Ali, eds): Proc. 4th Eur. Conf. Sw. Architecture (ECSA): Companion Volume. ACM, pp. 49–54, 2010.