Markdown Architectural Decision Records: Format and Tool Support

Oliver Kopp[1], Anita Armbruster[1], and Olaf Zimmermann[2]

[1] Institute for Parallel and Distributed Systems, University of Stuttgart
Stuttgart, Germany
{lastname}@ipvs.uni-stuttgart.de
[2] Institute for Software, Hochschule für Technik (HSR FHO)
Rapperswil, Switzerland
olaf.zimmermann@hsr.ch

Abstract. Architectural decision records answer "why" questions about designs and make tacit knowledge explicit. Many architectural decisions are made during development iterations because they have a close connection to the code. It is challenging to come up with task-specific decision capturing practices and supporting tools that are not perceived as time wasters; context switches and media breaks that harm the productivity of coding architects and developers involved in the decision making have to be avoided. To integrate existing architect-centric approaches into the developer toolchain, this paper presents a Markdown-based decision capturing template that we derived from previous work to enhance an existing decision capturing tool for developers. Our early validation results in the context of an open source project suggest that format and tool promise to contribute to an integrated decision capturing practice, with further enhancements being required. Tool and template are available in public GitHub repositories.

1 Introduction

Source code needs to be documented. This typically leads to comments in code and to external documents. Well-written classes and methods, which have expressive names and understandable branches [6], make low-level code comments obsolete. On the other end of the spectrum, however, wide-ranging decisions of high architectural significance are made during development iterations; these decisions are not self-explanatory and not expressed in the code explicitly. An example of such an *architectural decision* is how to keep user session data consistent and current across Web shop instances. Typically, these kind of decisions are recorded in external documentation files, wikis, or tools [18]. The primary tool of developers, however, is their Integrated Development Environment (IDE) with integrated version control system support. Requiring developers to use more tools has a negative impact on productivity, quality, and motivation to capture architecturally significant decisions: opening another tool requires some setup and training effort and leads to undesired, time consuming context switches. Furthermore, model- and document-centric tools typically do not integrate themselves well in the developer's toolchain: The documents are not committed in plain text format into the version control system—if versioned along the code at all. Further, the documents might get out of sync with the

N. Herzberg, C. Hochreiner, O. Kopp, J. Lenhard (Eds.): 10th ZEUS Workshop, ZEUS 2018, Dresden, Germany, 8-9 February 2018, published at http://ceur-ws.org/Vol-2072

code base [13]. As a consequence, architectural decision capturing practices and tools oftentimes are perceived as chronophages (time wasters). This holds true particularly for agile and lean developer communities. We therefore can derive the following problem statement:

How to seamlessly integrate architectural decision making into developer tool landscapes — so that decision rationale can be collected under high coding velocity?

To describe how we propose to overcome this problem, we first provide some background and related work (Sect. 2). We then introduce the *Markdown Architectural Decision Records* (MADR) format as the conceptual contribution of this paper (Sect. 3). Next, we present a tool implementation for MADR integration that makes our conceptual solution available to practitioners and validates the novel format (Sect. 4). We have further validated the MADR format and tooling in an action research (Sect. 5). A discussion on the findings of MADR follows in Sect. 6. Finally, we conclude the paper (Sect. 7).

2 Background and Related Work

A large body of research work on capturing architectural decisions exists; the state of the art is for instance surveyed by Alexeeva et al. [2] and by Capilla et al. [4]. Specifically to the context of service orientation and service composition, the SOA Decision Modeling (SOAD) project [17] investigated architectural decisions recurring in SOA design and introduced a seven-step method to identify, make, and capture such decisions. SOAD used a fixed, rather comprehensive meta model. Taking that experience into account, our template and tool, to be introduced in Sect. 3 and 4, are designed in such as way that they are applicable on service-oriented middleware and tool development projects (as evidenced in the validation activity presented in Sect. 5), but not limited to such projects.

More recently, templates, practices, and tools specifically targeting coding architects and agile developers, who make certain architectural decisions and contribute to others, have been proposed. Seven different formats, including comprehensive and lean ones, are compared by Zimmermann et al. [18]. They also introduce ADMentor, a decision modeling add-in for Sparx Enterprise Architect. ADMentor supports two primary user stories and themes, a) problem space modeling and b) architectural decision capturing. Problem spaces model recurring decisions along with options to be considered. The architectural decision capturing capability then allows architects on projects to keep track of decisions made in a *decision log* as suggested by the ISO/IEC/IEEE 42010 standard [7] for architecture description. Other features include rich text editing, model linking and refactoring, and reporting/analysis. Decision capturing is streamlined by lightweight decision capturing templates such as *Y-Statements* [15]; Question, Option, Criteria (QOC) diagrams [9] are supported as well.

General best practice recommendations for decision documentation are presented by Zdun et al. [15], including the above mentioned Y-Statement format originally developed for—and first applied in–an industry project setting at ABB [16]. Y-Statements contain the aspects context, concern, the chosen option, other possible options, the expected positive results, and the accepted downsides as well as (optionally) additional decision

```
In the context of <use case/user story u>,
facing <concern c>
we decided for <option o>
and neglected <other options>,
to achieve <system qualities/desired consequences>,
accepting <downside / undesired consequences>,
because <additional rationale>.
```

Fig. 1. (WH)Y-Statement format: context and concern form the diagonal two lines at the top of the Y, the other five form the vertical bar of it.

rationale (Fig. 1). As a structured text format, Y-Statements can be put in external documentation, in code comments [6], or in Java annotations[3].

An example of such a Y-statement is: "In the context of the Web shop service, facing the need to keep user session data consistent and current across shop instances, we decided for the Database Session State pattern [5] (and against Client Session State [5] or Server Session State [5]) to achieve cloud elasticity, accepting that a session database needs to be designed, implemented, and replicated."

A rather popular[4] practitioner's tool is `adr-tools`[5]. It uses the format by Nygard [10], which covers less aspects than the Y-Statements. For instance, the neglected options are not shown. Both Y-statements and Nygard's Architecture Decision Records (ADRs) have been designed with a lean and agile mindset that is in line with the vision of software specification and documentation in Continuous Software Development (CSD) [14].

3 Markdown Architectural Decision Records (MADR)

To keep the architectural decisions close to common developer tools and artifacts, we propose to 1) use Markdown as decision capturing syntax (with a proposed format derived from Y-Statements) and 2) place the decisions in the folder `docs/adr` of code projects that are version-controlled.

Markdown is a text format, which enables common version control systems such as git to be used. This makes diffing within the IDE possible. Our decision to use Markdown as markup language (instead of other markup languages) is supported by the following rationale: 1) it eases writing, 2) Markdown is the markup language for comments by users within GitHub (such as in gists, issues, or pull requests), and 3) already available rendering tools can be leveraged.

We call the new format *Markdown Architectural Decision Records (MADR)*. Some early adopters of the Y-Statement syntax had commented that the sentences can get really long and are therefore hard to read for inexperienced readerships. As Markdown is a *structured* text format in which headings can mark sections, we decided to deviate from the pure Y-format and transferred it into a section-oriented one (similar to the successful approach of the `adr-tools` outlined in Sect. 2). The starting point was the "Decision Capture Template" [12], which we adapted to contain all elements of the Y-Statements.

[3] GitHub project "Embedded Architectural Decision Records," `https://adr.github.io/e-adr/`

[4] 630 stars on GitHub as of 2018-01-31

[5] `https://github.com/npryce/adr-tools`

We indicate the backward mapping to the Y-format when describing the new template below (in parenthesis).

Figure 2 shows the format of MADR. Each record takes a title (line 1) followed by the user story (line 3). The user story is made optional because its content further elaborates on the mandatory context (from Y-Statement syntax) and problem statement (line 5). More information on the context such as forces or decision drivers (Y-Statement's concerns and aims) can be appended (line 6). The considered alternatives (including the chosen and the neglected ones; Y-Statement) are listed as items (lines 8 to 11). The chosen alternative includes a justification (Y-Statement's "to achieve" rationale) and optionally consequences (Y-Statement's "accepting that" downsides). Follow-up decisions are are also listed as items (lines 13 to 19). If a longer pro/con evaluation of the alternatives makes sense, each option can be listed as a separate heading followed by an itemized list of pros and cons. In summary, all aspects of a Y-Statement are covered in the template, even though the consequences are left optional.

Note that MADR does not restrict the Markdown syntax. Thus, it is possible to include images, ASCII art, and PlantUML[6].

The folder `docs/adr` was chosen to enable rendering in GitHub pages. Since 2016, GitHub pages allows for rendering a homepage out of the `docs` folder [8]. When updating files in the `docs` folder, GitHub processes them using the Jekyll site generator[7], which basically converts markdown files into HTML files using a given template. As a consequence, when placing the ADRs into a subfolder, it is possible to make them accessible on the World-Wide Web.

4 Tool Implementation and Integration

To support MADR we extended `adr-tools` (made available at `https://github.com/adr/adr-tools`) and created `adr-log` (made available as npm package at `https://www.npmjs.com/package/adr-log`).

The original `adr-tools` support arbitrary formats when creating new architectural decisions by providing an appropriate `template.md` file. New ADRs are put in the format `nnnn - title-lowercased-with-dashes.md` in the directory, where `nnnn` is a number starting from 0001. Besides basic creation functionality, `adr-tools` allows for linking ADRs. For instance, a new ADR can supersede an existing ADR. For that the status of an ADR is tracked under a new heading "Status". In MADR, we record the status and the status changes in a table with the columns "Date" and "Status". We extended `adr-tools` to support the command `adr new docs/adr madr`, where `docs/adr` is the directory where architectural decisions are put and `madr` denotes that MADR should be used as template format. At each call of `adr new TITLE`, MADR is used as template instead of Nygard's template. Furthermore, we are working on supporting the status table in the beginning so that it is 1) created when an ADR is superseded by another ADR (e.g., `adr -s 1 Use SQL Database`, tells `adr-tools` to add a note at ADR-0001 that it is superseded) and 2) amended when there is a new link to an ADR

[6] `http://plantuml.com/`

[7] `https://help.github.com/articles/using-jekyll-as-a-static-site-generator-with-github-pages/`

```
# *[short title of solved problem and solution]*

**User Story:** *[ticket/issue-number]* <!-- optional -->

*[context and problem statement]*
*[decision drivers | forces]* <!-- optional -->

## Considered Alternatives

* *[alternative 1]*
* *[...]* <!-- numbers of alternatives can vary -->

## Decision Outcome

* Chosen Alternative: *[alternative 1]*
* *[justification. e.g., only alternative, which meets k.o. criterion decision
    driver | which resolves force force | ... | comes out best (see below)]*
* *[consequences. e.g., negative impact on quality attribute,
    follow-up decisions required, ...]* <!-- optional -->

## Pros and Cons of the Alternatives <!-- optional -->

### *[alternative 1]*

* `+` *[argument 1 pro]*
* `-` *[argument 1 con]*
* *[...]* <!-- numbers of pros and cons can vary -->
```

Fig. 2. MADR 1.0.0 format decomposing the Y-Statement elements into document sections.

(e.g., `adr -l "1:Amends:Amended by" Use PosgreSQL`, tells `adr-tools` that ADR-0001 is amended by the newly created ADR).

An index of existing architectural decision records is a welcome feature to gain an overview of the decision making status and be able to navigate the log efficiently. The existing `adr-tools` already offers the command `adr generate toc`. This, however, generates a completely new file and does not allow to add arbitrary text before or after the toc. For the generation of the table of contents of one markdown file, the tool `markdown-toc`[8] inserts the TOC after the token `<!-- toc -->`. Inspired by that idea, we implemented `adr-log`, which places the list of all ADRs after the placeholder `<!-- adrlog -->`. We chose the name "log" instead of "toc" to be consistent with the database terminology, where a set of records forms a log.

[8] `https://www.npmjs.com/package/markdown-toc`

5 Preliminary Validation

We validated MADR and tooling in action research [3] on the Eclipse Winery project that is driven by one architect, three coding architects, and two developers (students, staff members, and volunteers).

In action research, the researcher joins a project and influences it actively, for instance as coach, pacemaker, or technical reviewer[9]. Applying action research allowed us to experience the practical applicability of our concepts ourselves and to interact with and learn from other users while they used MADR. We followed a very basic study protocol of 1) define validation goals and approach, 2a) present MADR and `adr-tools` to the project team and create a first MADR record ourselves (lead by example), 2b) monitor usage and remind project participants, e.g. in sprint planning meetings and retrospectives, 2c) give feedback to project participants and offer coaching, and finally 3) collect data and seek suggestions for improvements from project participants.

A total of 16 MADR records were created.[10] Eight of these fully filled out the template, and eight used a shortened form without the explicit section "Pros and Cons of the Alternatives". Two of the short forms additionally include details of the solution. When working with code, it was easy to document the decision along with the code. A single file had to be copied and renamed (or `adr new TITLE` invoked). Then, one could start with writing down the context, options, chosen option, and the pros and cons.

Seasoned professionals did not have issues to fill out the template and even came up with their own. Inexperienced students were able to document their decisions ranging from supported writing to independent writing. A major issue for them was to understand how to replace the placeholders in markdown. For instance, in ADR-0005[11], the options listed there did neither have ids nor short titles. The chosen option was referred to as "Option D", but there was no explicit option D — only a fourth unnamed option. Some students also reported that they were afraid to be criticized for options not considered. Since MADR makes it explicit which options a solution was chosen from, it is easy to detect if an important option was missed. On the positive side, this leads to a teaching effect and allows supervisors to get to know which knowledge the students lack at a certain educational level. On the one hand, it was agreed, that is difficult to create an ADR if the technology itself is new. On the other hand, it was also agreed that is necessary to document decisions after one has enough knowledge (e.g., after experimenting longer with different options) to make it feasible for others to understand the decisions taken. This is in line with Parnas's view on a rational software process [11].

In summary, the users reported that the MADR template and tools helped them to be clear about the available options and to choose the best fitting one based on arguments. The template was filled during the discussions and helped to refine the pros and cons of alternatives.

---

[9] This is different from exposing selected research results to users and merely observing them (this would be done in a controlled experiment).

[10] `https://github.com/eclipse/winery/tree/d84b9d7b6c9828fd20bc6b1e2fcc0cf3653c3d43/docs/adr`

[11] `https://github.com/eclipse/winery/blob/d84b9d7b6c9828fd20bc6b1e2fcc0cf3653c3d43/docs/adr/0005-XML-editor-does-not-enforce-validation.md`

6 Discussion

Feedback from reviews and workshop raised some concerns whether placing ADRs in a single folder really scales: a complex system may consist of multiple microservices, and each microservice can itself be structured in different modules even written in different languages. Thus, the granularity of the decisions is different. Two possible solutions are: A) adding a category to each ADR and offer filtering. B) putting each ADR close to the source code where the decision is taken, e.g., `src/doc/adr` for a Java project.

A developer began to add longer explanations of code howtos to the ADR. The reasoning was that this code howto is very related to the ADR and that there is one place where the decision and the coding consequences can be found. Thus, an interesting question requiring further investigation and discussion would be whether close-to-code (M)ADR documentation leads to an increased use of documentation (in comparison to external documentation).

The presented version 1.0.0 of MADR uses slightly different terms than the Y-Statements (Sect. 3). We plan to refactor future versions of MADR[12] to be even closer to the terms of Y-Statements as these are proven in industry projects and have been gaining momentum recently [6].

In large projects, it is common to create a project management issue for each change. In MADR, the link to an issue is optional to enable application in small projects. These two different settings call for MADR *profiles*. For instance, one such profile could enforce the link to the ticket/issue number (pointing to an entry in task management system) and make the section "pros and cons of the alternatives" mandatory.

7 Conclusion and Outlook

This paper presented Markdown Architectural Decision (MADR) records, a decision capturing template derived from earlier work on a Y-statement format. We also presented an extension of existing `adr-tools` to enable command-line tools for handling MADRs as well as a new `adr-log` tool to generate a list of existing ADRs.

Based on the early feedback, we plan to improve the creation and review process. We also consider to develop a comprehensive yet lean getting started tutorial and quick reference card.

MADRs capture a concrete decision in the context of a single particular project. However, problems and options may reoccur and different options might be chosen in different contexts. For instance when a system runs normally in the absence of partitions, one choose between different trade offs between latency and consistency [1]. Each trade off has its pros and cons which are differently weighted in each context. Currently, it is possible to model this "problem space" using AD-Mentor [18], but not using Markdown. To come up with a corresponding Markdown format and tool integration for knowledge sharing and reuse therefore is an enhancement to be considered in the future evolution of MADR.

[12] The development of MADR takes place at `https://github.com/adr/madr/`.

Acknowledgments This work is partially funded by the BMWi projects SmartOrchestra (01MD16001F) and IC4F (01MA17008G).

References

1. Abadi, D.: Consistency Tradeoffs in Modern Distributed Database System Design: CAP is Only Part of the Story. Computer 45(2), 37–42 (2012)
2. Alexeeva, Z., et al.: Design Decision Documentation: A Literature Overview. In: Software Architecture, pp. 84–101. Springer International Publishing (2016)
3. Avison, D., et al.: Action Research. Communications of the ACM 42(1), 94–97 (1999)
4. Capilla, R., Jansen, A., Tang, A., Avgeriou, P., Babar, M.A.: 10 Years of Software Architecture Knowledge Management: Practice and Future. Journal of Systems and Software 116, 191–205 (jun 2016)
5. Fowler, M., Rice, D.: Patterns of Enterprise Application Architecture. Addison-Wesley, Boston, Mass. (2003)
6. Harrer, S., Lenhard, J., Dietz, L.: Java by Comparison: Become a Java Craftsman in 80 Examples. Pragmatic Bookshelf (2018), http://java.by-comparison.com
7. ISO/IEC/IEEE 42010:2011: Systems and software engineering – Architecture description. Standard (Dec 2011)
8. Leschner, J.: Simpler GitHub Pages publishing (2016), https://github.com/blog/2228-simpler-github-pages-publishing
9. MacLean, A., Young, R.M., Bellotti, V.M.E., Moran, T.P.: Questions, Options, and Criteria: Elements of Design Space Analysis. Hum.-Comput. Interact. 6(3), 201–250 (Sep 1991)
10. Nygard, M.: Documenting architecture decisions (2011), http://thinkrelevance.com/blog/2011/11/15/documenting-architecture-decisions
11. Parnas, D.L., Clements, P.C.: A rational design process: How and why to fake it. In: Formal Methods and Software Development, pp. 80–100. Springer Science + Business Media (1985)
12. Schubanz, M.: Full Decision Capture Template (2017), https://github.com/schubmat/DecisionCapture/blob/ca03429634ac2779b37e12aee34dd09a5fdbdcd3/templates/captureTemplate_full.md
13. ThoughtWorks: Technology Radar Vol. 17, https://thoughtworks.com/radar
14. Van Heesch, U., et al.: Software Specification and Documentation in Continuous Software Development: A Focus Group Report. In: 22nd European Conference on Pattern Languages of Programs (EuroPLoP'17). ACM (2017)
15. Zdun, U., et al.: Sustainable Architectural Design Decisions. IEEE Software 30(6), 46–53 (2013)
16. Zimmermann, O.: Making Architectural Knowledge Sustainable – Industrial Practice Report and Outlook (2012), presentation at SATURN 2012, http://www.sei.cmu.edu/library/abstracts/presentations/zimmermann-saturn2012.cfm
17. Zimmermann, O., Koehler, J., Leymann, F.: Architectural Decision Models as Micro-Methodology for Service-Oriented Analysis and Design. In: Workshop on Software Engineering Methods for Service Oriented Architecture 2007 (SEMSOA). CEUR (2007)
18. Zimmermann, O., et al.: Architectural Decision Guidance Across Projects – Problem Space Modeling, Decision Backlog Management and Cloud Computing Knowledge. In: 12th Working IEEE/IFIP Conference on Software Architecture (WICSA). IEEE (2015)

All links were last followed on February 14, 2018.