Editor: **George Fairbanks**
gf@georgefairbanks.com

Love Unrequited: The Story of Architecture, Agile, and How Architecture Decision Records Brought Them Together

Michael Keeling

SOFTWARE ARCHITECTURE HAS long sought the attention of agile developers, but its love is unrequited. For decades, architects have offered up gifts to agile teams. In response, agile developers have responded with little more than a raised eyebrow. As a result, teams that blend architecture principles and agile practices are rare.

Over time, the agile teams have warmed to architecture's attention. One gift in particular has put a sparkle in their eyes: architecture decision records (ADRs). While previous architecture offerings have fallen flat, ADRs have that special something. Teams that had ignored architecture modeling and documentation practices are increasingly embracing ADRs.

Understanding why ADRs appeal to today's agile teams will lead to a stronger, mutual relationship between the architecture and agile communities. This is the story of architecture, agile, and a practice that is bringing them together: ADRs.

Digital Object Identifier 10.1109/MS.2022.3166266
Date of current version: 20 June 2022

Design Decisions and ADRs

Making good design decisions is hard, and changing them is harder. Architecture decisions set the direction for a project and guide smaller decisions in the code, so it's critical that developers understand them. Documentation helps, but how exactly?

ADRs are typically small text files, each describing a single design decision and rationale. ADR templates commonly include three parts: context, decision, and consequences. The context of an ADR describes the technical, business, social, or political circumstances that directly influence a design decision. A brief description of the decision itself outlines the selected course of action for the design. Consequences describe the expected outcomes that result once the decision is applied.

As an example, say a team of experienced Java developers needs to deliver a web service on a tight schedule. The team's experience, the expected project timeline, and a technical constraint that the team is to deliver a Java-based web service are contextual forces. In response to this context, the team decides to use a popular framework as the backbone for the architecture. As a consequence of this decision, the team expects to deliver a highly maintainable solution in only a few weeks and to hoist several other desirable quality attributes into the web service's architecture. However, adopting the framework introduces a risk that the team may someday encounter a problem that is awkward or impossible to solve due to the framework's constraints.

In this example, the rationale primarily focuses on quality attributes (maintainability and time to market), engineering risks, and scheduling. The consequences describe positive and negative outcomes but do not pass judgment on the hypothesized outcomes. While the decision allows the team to ship quickly and promotes desirable quality attributes, a new risk is introduced by the decision that must be managed. Accepting this decision means that the team accepts all of these consequences.

Discussing and writing down design decisions is not a new idea. For as long as we've developed software, teams have described, debated, and shared their decisions. What is new is treating decisions as artifacts that the team writes down. Looking back in time can help us understand why this shift is happening now and why agile teams are willing to write ADRs.

The Seeds of a One-Sided Romance

Despite the importance of decisions, when software architecture was first studied carefully, in the 1990s, researchers focused primarily on structure and abstraction. Perry and Wolf were an exception when they included design rationale prominently in their formula, "Architecture = {Elements, Form, Rationale}."[1] A more typical treatment of architecture is found in David Garlan and Mary Shaw's early publications,[2] which referred to decision making only indirectly. Decisions were seen as something you made to arrive at the key architecture abstractions, like components, connectors, and modules, but not as one of those key abstractions

As we look back with fresh eyes, we should not forget that software architecture was still a new discipline in the 1990s. After decades of muddling through increasingly complex software systems using ad hoc models, the software industry had finally arrived at an initial set of useful abstractions for describing how to arrange software systems to promote desirable system properties. Software architects of the day gained access to tremendous explanatory power in the form of views, view models, and architectural styles. This was a game changer.

> Instead of inadvertently gatekeeping design, ADRs gave developers direct access and empowered them to own it.

For pre-agile teams of the day (the Agile Manifesto was not published until 2001), these powerful new ideas were challenging to adopt. Contemporary design practices were time consuming and required deep expertise to apply well. Pre-agilists felt that the work necessary to document multiple views of the architecture, especially using the notations, tools, and practices common in the early 1990s, was cost-prohibitively time consuming. Thus, working software was valued over comprehensive documentation, and the seeds of a one-sided romance were sown.

As both industry and research gained experience with the emerging software architecture discipline, there was a growing recognition that successfully implementing a system's architecture required more than only a correct and complete architecture description. Teams who understood the trail of decisions leading to a design were better able to scale up their organizations, improve design quality, handle staff turnover, and evolve the system over time. The resulting system design was certainly important, but understanding the rationale behind a design had practical significance.

Given that so much state of the art was established during this time, it seems only reasonable that some building blocks would not be fully understood or appreciated at the time. Naturally, these missed foundational concepts would be investigated in the years to follow.

Architects Love Views

Throughout the late 1990s and early 2000s, design decisions were discussed increasingly often in the architecture community. The topic is briefly mentioned in the Software Engineering Institute's series of books on software architecture as an approach for describing models more richly and as an analysis tool. Both researchers and practitioners—including Anton Jansen, Dana Bredemeyer, Jan Bosch, Olaf Zimmerman, Paris Avgeriou, Rich Hilliard, Ruth Malan, Uwe Van Heesch, and others—shared their observations about the increasing prominence and promise of design decisions and the evolving software architect's role in decision making.

During this time, there was a clear turning point in how software architects thought about design decisions. The available software architecture abstractions, while powerful and expressive, were beginning to be seen

as necessary but not sufficient for designing and describing a software system. From this new perspective, design decisions were a key architecture abstraction on a par with components and modules.[3]

If design decisions are a new abstraction, how do they relate to the others, and how should we express them? Many in the software architecture community attempted to incorporate design decisions by creating decision-focused views.[4,5]

Agile teams were unimpressed. Describing decisions as views required them to fully embrace the very architecture formalisms they had already rejected. Jeff Tyree and Art Akerman, noting the challenges agile teams encountered when incorporating architecture ideals, attempted to strike a balance by cataloging design decisions using a structured template.[6]

Despite the awkward fit of decision views, the architecture community's interest in design decisions kept growing. In 2009, Philippe Kruchten, Rafael Capilla, and Juan Dueñas synthesized more than a decade of research on design decisions into a single call to action for the software architecture community to create practical and useful decision-focused viewpoints.[7] The age of design decisions had officially arrived, and agile would soon show signs of warming to software architecture.

ADRs Make Sparks Fly

After more than two decades, architecture finally found a gift that excited the agile community. That gift was packaging decisions as ADRs. Decision-centric design complemented existing architecture abstractions and helped teams describe a new dimension of the architecture: change over time. Any agile team eager to embrace change would be excited by this idea.

Throughout the early 2010s, a number of practicing, agile teams, inspired by Kruchten's, Capilla's, and Dueñas's call to action, shared their experiences with design decisions. In late 2011, Michael Nygard published a blog post describing his team's experiences writing ADRs in the shape of patterns, following a lightweight template. Each decision record was added to an immutable decision log that, over time, built a history of a system's design.[8]

ADRs were materially different from other approaches coming out of the architecture community up to that point. Anyone on the team could assume the architect's role by writing an ADR. Instead of inadvertently gatekeeping design, ADRs gave developers direct access and empowered them to own it. ADRs achieved this in three ways.

First, no special tools are required to write an ADR. ADRs are stored as plain text documents, written in markdown. Anyone with a text editor can create an ADR. Diagrams are created with any tool, formal or informal, and might be as simple as a picture of a sketch on a whiteboard. Tools that treat diagrams as code have become especially popular for this purpose.

Second, ADRs are stored in the same version control repository as the code to which those decisions apply. Storing ADRs close to the code makes it easier for developers to discover them and increases the likelihood that developers will read and be guided by past design decisions. Since ADRs are stored in the version control system, they are subject to the same peer-review process as code. This makes it easier to solicit feedback and share knowledge.

Third, ADRs do not require special notations or knowledge. ADRs rely heavily on plain prose descriptions. A lightweight template provides structure and guidance for authors. Developers can write their first ADR after only brief training. ADR authors with deep software architecture knowledge or experience can still use what they know to write concise and comprehensive ADRs, but knowledge and experience are not prerequisites for participation. Anyone with a passion for writing an ADR need only describe a design decision to the best of their ability.

These new ways of thinking about design decisions made sparks fly between agile and software architecture. Documenting even only a single decision provides a strong return on investment. The cost of each ADR is measured independently as it is written. Investment is easily justified, even for systems with rapidly evolving architectures. By these economics, creating something akin to a decision view, such as a decision log depicting the evolving history of the architecture, is practically free.

In the decade since Nygard's blog post, practitioners and researchers have continued to explore design decisions and ADRs. In fact, Nygard's take on ADRs is not the only one out there. One example described by Olaf Zimmerman emphasizes not only past decisions but also future ones yet to be made.[9] Thanks to contributions from Heiko Koziolek, Joe Runde, Lukas Wegmann, Nat Pryce, Oliver Kopp, Paulo Merson, Rafael Capilla, Thomas Goldschmidt, and so many others, there is now a rich knowledge base of pragmatic advice available to help teams use ADRs effectively. Search the web today, and you'll find a robust discussion about design decisions, templates, and techniques for describing design rationale.

ADRs Are the Gateway to Better Design Practice

It may not have been love at first sight for agile and architecture, but you wouldn't know that seeing them together today. Design decisions were not invented with ADRs, but ADRs made design decisions accessible in a way other software architecture design methods did not. This broad accessibility made it easy for practicing agile teams to have an active hand in helping design decisions cross over from research to practice.

ADRs are quickly becoming a standard practice across the software industry. New advice, templates, and variations on ADRs emerge regularly as the practice is explored and refined by practicing software development teams. Of course, as more teams improve their design and documentation practice by writing ADRs, new problems emerge. Knowledge management is increasingly a problem with which software development teams must contend.[10]

Just like any partner, ADRs are not perfect. Unstructured prose is notoriously difficult to analyze. ADRs strongly emphasize technical stakeholders' perspectives, especially developers, at the sacrifice of nontechnical stakeholders. While not requiring particular notations and tooling makes ADRs easy to use, it also makes ADRs difficult to use well, especially by novices. From a certain point of view, this is a feature, not a bug.

ADRs' greatest strength is their low barrier to entry. Since anyone on the team can write an ADR, everyone who wants can fill the role of software architect. That anyone can write ADRs creates an opportunity to grow software architects over time. In my experience, ADRs create a gateway to increasingly sophisticated architecture design practice. Teams who write ADRs, it seems, can't help but become better software architects over time. Even if it isn't true love, agile and architecture seem to have finally found a common interest upon which a stronger relationship can be built.

ABOUT THE AUTHOR

MICHAEL KEELING is a software engineer at Kiavi, Pittsburgh, Pennsylvania, 15208, USA, and the author of *Design It!: From Programmer to Software Architect*. Contact him at http://neverletdown.net/ or mkeeling@neverletdown.net.

References

1. D. E. Perry and A. L. Wolf, "Foundations for the study of software architecture," *ACM SIGSOFT Softw. Eng. Notes*, vol. 17, no. 4, pp. 40–52, Oct. 1992, doi: 10.1145/141874.141884.
2. D. Garlan and M. Shaw, "An introduction to software architecture," in *Proc. Adv. Softw. Eng. Knowl. Eng.*, V. Ambriola and G. Tortora, Eds. Hackensack, NJ, USA: World Scientific, 1993, vol. I, pp. 1–39, doi: 10.1142/9789812798039_0001.
3. A. Jansen and J. Bosch, "Software architecture as a set of architectural design decisions," in *Proc. 5th Work. IEEE/IFIP Conf. Softw. Architecture (WICSA'05)*, 2005, pp. 109–120, doi: 10.1109/WICSA.2005.61.
4. J. C. Dueñas and R. Capilla, "The decision view of software architecture," in *Software Architecture*, vol. 3527, R. Morrison and F. Oquendo, Eds. Berlin, Germany: Springer-Verlag, 2005, pp. 222–230, doi: 10.1007/11494713_15.
5. U. van Heesch, P. Avgeriou, and R. Hilliard, "Forces on architecture decisions - A viewpoint," in *Proc. 2012 Joint Work. IEEE/IFIP Conf. Softw. Architecture Eur. Conf. Softw. Architecture*, pp. 101–110, doi: 10.1109/WICSA-ECSA.212.18.
6. J. Tyree and A. Akerman, "Architecture decisions: Demystifying architecture," *IEEE Softw.*, vol. 22, no. 2, pp. 19–27, Mar./Apr. 2005, doi: 10.1109/MS.2005.27.
7. P. Kruchten, R. Capilla, and J. C. Dueñas, "The decision view's role in software architecture practice," *IEEE Softw.*, vol. 26, no. 2, pp. 36–42, Mar./Apr. 2009, doi: 10.1109/MS.2009.52.
8. M. Nygard. "Documenting architecture decisions." Cognitect. https://cognitect.com/blog/2011/11/15/documenting-architecture-decisions (Accessed: Apr. 3, 2022).
9. O. Zimmerman, "Making architectural knowledge sustainable - The Y approach industrial practice report and outlook," in *Proc. SATURN Conf.*, May 2012. [Online]. Available: https://resources.sei.cmu.edu/library/asset-view.cfm?assetID=31345
10. G. Hohpe, I. Ozkaya, U. Zdun, and O. Zimmermann, "The software architect's role in the digital age," *IEEE Softw.*, vol. 33, no. 6, pp. 30–39, Nov./Dec. 2016, doi: 10.1109/MS.2016.137.