Editor: George Fairbanks
gf@georgefairbanks.com

# The Psychology of Architecture Decision Records

Michael Keeling

**WHEN I FIRST** discovered architecture decision records (ADRs) eight years ago, I thought they would be a useful documentation technique for describing design decisions and sharing them among my team members. Since then, I've written and reviewed hundreds of ADRs across multiple teams, organizations, industries, and software systems. Through this experience, I have learned that ADRs are indeed useful documentation but also something more. Writing ADRs facilitates meaningful cultural change by transforming developers into architectural thinkers who strongly value design.

ADRs accomplish this cultural shift by subtly influencing developers to change four key behaviors. First, ADRs change developers' perception regarding the value of design. Second, ADRs encourage developers of varying skill levels to participate in design. Third, ADRs increase the likelihood that developers will follow through on design decisions. Fourth, ADRs positively reinforce engagement in design.

*Digital Object Identifier 10.1109/MS.2022.3198195*
*Date of current version: 24 October 2022*

These four complementary changes in behavior evolve developers' self-identities. Over time, developers who write ADRs see themselves as thoughtful software architects who care deeply about design. This shift in personal identity also transforms the team culture. This meaningful cultural change is the real value of ADRs.

## Increase the Perceived Value of Design

For many software teams, design is a distant concern. The architecture is described in slide decks and other documents stored in a rarely used repository. Architects from outside the team might dictate the design, heavily implying that architecture is not for developers. Architecture descriptions can be riddled with jargon and unnecessary complexity, making the architecture abstruse and intimidating. When the distance between developers and design is too great, developers perceive design as having little value.

ADRs help developers see greater value in design by bringing design closer to them. ADRs are often written as small text files stored in the same version control repository as the code.[1] Each ADR describes a single design decision and the rationale for the decision (see "An Example ADR"). ADRs are added to an append-only decision log. When a new decision replaces an old one, both ADRs are kept, and links are created between them. Over time, the decision log forms a history of the architecture that describes how the design changed over time.[2]

ADRs change developers' perception regarding the value of design by associating design with code. The code is important. Code is stored in the version control and modified using a standard text editor. The architecture is in the version control, and ADRs are simple text files like code; therefore, by association, the design is also important.

Once that association is established, the team's behavior changes.[3] Design is treated with the same immediacy and care as code. Many teams choose to review ADRs by following the same process they use to review code. Developers strive to craft descriptive ADRs just like they strive to craft clean code. Previously distant ideas, such as architectural styles, technical debt, or quality attributes, become highly relevant topics worthy of careful consideration.

As developers increase their appreciation of design's value, more members of the team will want to

0740-7459/22©2022IEEE

## AN EXAMPLE ADR

All ADR templates include the same essential parts: context, decision, and consequences. The context describes the technical, business, social, or political circumstances that directly influence a design decision. A brief statement describing the design decision outlines the selected course of action. Consequences describe the expected outcomes—positive and negative—that result from applying the decision.

Here is an example of a simple ADR.

**ADR 21: Assign Additional Responsibilities to the Foo Service**

**Context**
We need to introduce Feature X into the system in time for a trade show in less than four weeks. We can deliver Feature X as a new web service, as a library, or by extending an existing web service. The team feels that the current web services are well factored, each with clear responsibilities. Feature X requires significantly more RAM and CPU compared to other services but is forecast to be used rarely (bursty traffic). The fastest the team has ever delivered a new service to production is four weeks.

**Decision**
We will extend the existing Foo Service to accept Feature X.

**Status**
Superseded

See ADR 26: Create Bar Service to be Responsible for Feature X

**Consequences**
We don't have time to create a new web service, the team's preferred choice. This is intended to be a temporary decision. Creating a library has too few benefits. Extending the Foo Service reduces schedule risks (we'll probably hit the date!) but increases the cost of rework. We can work to keep the new code decoupled so it's easier to extract into a new service later. There's a risk code coupling will be accidentally introduced, making later architecture changes more difficult. Adding additional responsibilities to the Foo Service increases load, so we'll need to increase the number of instances.

In this example, the development team has accepted code quality tradeoffs in response to schedule pressure. The team later refactored the architecture (as indicated by the superseded decision status, with a link to a different ADR) to improve code quality.

This example uses a basic ADR template. Different ADR templates emphasize different design details, such as rationale, team reflection, alternatives considered, or even the team's mood at the time of the decision.

participate in the design by writing, reading, and reviewing ADRs. Teammates start to notice when design decisions are made and encourage each other to write ADRs.

For many teams, designing architecture is a new skill. Not everyone will be prepared to participate. This is less problematic with ADRs compared to other design methods since ADRs provide scaffolding for teaching design just-in-time.

### Invite Broad Participation

Traditionally, software design is seen as an exacting discipline. Precise models are highly prized for how they support detailed analysis and clear communication. The burden of correctness is high. Mushy abstractions are thought to provide little value. Exacting formalisms and rigid notations erect barriers that prevent developers from participating in design. ADRs remove these barriers and invite contributions from experienced and inexperienced designers alike.

ADRs make design more accessible. Special tools or notations are not required. Instead of demanding precision, ADRs ask developers to do the best job they can to accurately describe a single design decision. Even a poorly written ADR can improve the team's communication about a design decision. Since an ADR describes a single decision, the next decision offers a fresh opportunity to write an even better ADR.[2]

Every ADR reflects the author's understanding of the underlying architecture and that author's mastery over essential design principles. Experienced architects are more likely to write concise, comprehensive, and nuanced ADRs, but knowledge of patterns or architectural abstractions is not a prerequisite to writing an ADR. Encouraging novices to write ADRs creates opportunities for practicing design, mentoring, and training that might not otherwise exist.

Each ADR is a learning opportunity that manifests at the ADR author's moment of need. For example, an experienced reviewer can help an ADR author replace a paragraph of text with a reference to a documented pattern, or to expand consequences so they demonstrate how a decision influences important quality attributes. ADRs can easily be peer reviewed, just like code. With practice and feedback, novice designers improve their skills and gain experience.

Broad accessibility to design has other benefits beyond skill building. It also increases awareness about the design. As developers become aware of the design decisions being made, they will expect the team to follow the design described.

ADRs, like promises, increase the likelihood that a team will follow through on a design decision.

## Follow Through on Design Decisions

For many teams, architecture is discussed often, but design decisions are rarely written down. Design decisions and the amazing, detailed, nuanced discussions that accompanied those decisions are forgotten once the meeting ends. If you missed it, then you might never know an important design decision had been made.

Writing an ADR makes design decisions real. Distributing an ADR for feedback asks the team to form an opinion on the proposed idea and builds support for it. Merging an ADR into the version control repository as an accepted decision is akin to making a public commitment to abide by the decision. Whether teammates agree wholeheartedly with the decision or agree only to disagree and commit, an ADR represents the planned, future direction of the architecture.

A published ADR is a written, public declaration that describes how the software system is intended to be changed. It's a promise to the team that the software will be changed as portrayed in the design decision. When people make a public promise, especially a written one, they feel compelled to follow through with the promise.[3] This is as true for software design as it is for someone who tells a friend they are trying to quit smoking. In psychology, this is known as the consistency principle.[3]

Once an ADR is published, developers on the team are empowered to hold one another accountable to that decision so the code remains consistent with the design promised in the ADR. I have seen teammates point out architecture violations in the code during peer reviews. They speak up when the architecture diverges from agreed decisions and discuss how best to reconcile that divergence. They refactor code to align it with ideas described in ADRs. They write new ADRs to reflect what's actually in the code, as a first step toward improving the architecture. I have even seen teammates direct new hires to the decision log as a part of their onboarding to endow them with knowledge of the software system.[4]

Promises have tremendous power over our behaviors. ADRs, like promises, increase the likelihood that a team will follow through on a design decision. As this happens repeatedly, developers on the team will begin to see themselves in a different light.

## Reinforce Engagement in Design

The existence of ADRs in the version control repository provides evidence that a team's behaviors have changed. Developers who see the ADRs will say to themselves, "others on my team write ADRs; maybe I should, too." Those who write ADRs understand their value and are more likely to read and share feedback on others' ADRs.

Those who read ADRs develop an increased awareness of when design decisions are being made and are themselves more likely to write ADRs.

On teams that write ADRs, a reinforcing feedback loop emerges that promotes architectural thinking, design, and communication. The more people who participate in design by writing and reading ADRs, the more value the team gets from design. As developers' behaviors change, individuals who might never have thought of themselves as software architects begin to self-identify as the kind of developers who think deeply about architecture and strongly value design. Every time an ADR is added to the decision log, the team receives a gentle reminder: "This is a team that values design," and "I am the kind of developer who thinks through design decisions."

This shift in self-identity evolves the team culture. Discussions about technical debt, quality attributes, and risk are encouraged and become common practice. Thanks to feedback on ADRs and coaching from teammates knowledgeable about design, design decisions become more nuanced and sophisticated. Teammates provide each other with increasingly thoughtful feedback and encourage deeper exploration of the context and consequences. ADR authors are eager to learn more about design and software architecture.

On a small team, it may take as little as 3–6 months for ADRs to become a standard practice. Every ADR written nudges the team's behaviors toward becoming a team of architectural thinkers. Once ADRs become a standard practice, no matter how much experience they had when they start, over time, developers on the team will become the architects the team needs.

## ADRs Are More Than Lean Documentation

There is a wonderful quote by artist Robert Henri I first learned from Woody Zuill that I think applies directly to ADRs: "The object isn't to make art, it's to be in that wonderful state which makes art inevitable."

An ADR isn't just a document; it's a vehicle for changing a team's design psychology. Through association with code, developers see the value of design. Removing barriers to participation invites all developers to contribute to the design. Following through on the promised design proves that time spent on design has value. Participation begets engagement. Engagement in design shifts developers' self-identity as architects. Team culture evolves with this shift in self-identity.

There is something almost magical about ADRs and how they can change team culture. It isn't the documents themselves that are responsible for this cultural shift but the act of creating them over time. ADRs challenge teams to change their behavior in a number of key areas. How should a team think about the value design brings? Who should participate in design activities? What does someone need to know to contribute to the design? What does it mean to realize a design decision? How often should a team engage with the design?

The changes in behavior are what are ultimately responsible for evolving the team's culture. ADRs are just the medium. The true object of ADRs isn't to document design decisions but to help software development teams be in that wonderful state that makes good design inevitable.

## References

1. M. Nygard. "Documenting architecture decisions." Cognitect. Accessed: Aug. 23, 2022. [Online]. Available: https://cognitect.com/blog/2011/11/15/documenting-architecture-decisions
2. M. Keeling, "Love unrequited: The story of architecture, agile, and how architecture decision records brought them together," *IEEE Softw.*, vol. 39, no. 4, pp. 90–93, Jul./Aug. 2022, doi: 10.1109/MS.2022.3166266.
3. R. B. Cialdini, "Influence: The psychology of persuasion," Harper Business, New York, NY, USA, Dec. 2006. [Online]. Available: https://www.goodreads.com/book/show/28815.Influence
4. M. Keeling and J. Runde. (Aug. 2018). "Share the load: Distribute design authority with architecture decision records." Presented at Agile2018 Conf., San Diego, CA, USA. [Online]. Available: https://www.agilealliance.org/resources/experience-reports/distribute-design-authority-with-architecture-decision-records/

## ABOUT THE AUTHOR

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