Architectural Decision Records

March 7, 2022

Richard Thomas

Presented for the Software Architecture course
at the University of Queensland

THE UNIVERSITY
OF QUEENSLAND
AUSTRALIA

Architectural Decision Records

Software Architecture

March 7, 2022

Richard Thomas

1 Introduction

Documenting reasons why a decision was made about some aspect of the software architecture is important for the long-term maintainability of the system. Architecture decision records (ADR) are a simple but informative approach to documenting these reasons.

Michael Nygard was one of the early proponents of ADRs. His argument is that no one reads large documents but not knowing the rationale behind architecturally important decisions can lead to disastrous consequences, when later decisions are made that defeat the earlier decisions [1]. Nygard created ADRs as a light-weight approach to documenting important architecture decisions, to suit agile development processes. The ideas were based on Philippe Kruchten's discussion of the importance of architecture decisions in his article "The Decision View's Role in Software Architecture Practice" [2]. In this article Kruchten discusses extending his "4+1 View Model of Software Architecture" [3] to capture the rationale for important decisions while designing each view of the architecture.

Architecture decision records should capture important decisions that are made about the design of the architecture. These include decisions that influence the

- structure of the architecture,
- delivery of quality attributes,
- dependencies between key parts of the architecture,
- interfaces between key parts of the architecture or external interfaces, and
- implementation techniques or platforms.

These decisions provide important information for developers who have to work with the architecture, so that they know why decisions have been made and know how to design their code to fit into the architecture. They also provide important information for those who have to maintain the software in the future. ADRs help maintainers to know why decisions were made, so that they do not inadvertently make decisions that result in breaking expectations about the software.

ADRs are short notes about a single decision. The intent is to make it easy to record the information about the decision, so that it is more likely to be documented. ADRs are usually kept with the key project documentation. The argument is often made that this should be within the version control system (e.g. git) holding the project's source code. For example, a directory `doc/architecture/adr` in the project repository to contain the ADRs. The C4 model recommends you create a directory in the project repository to hold the C4 diagrams with a subdirectory for documentation, and another subdirectory for ADRs (e.g. `c4-model`, `c4-model/docs` and `c4-model/adrs`). Note that the C4 modelling tools do not like having the subdirectory containing the ADRs within the documentation subdirectory.

Each ADR is written in a separate file, using a simple notation like markdown. The file names are numbered so the history of decisions can be viewed. It is recommended that the file name describe the decision, to make it easier to scan for information about specific types of architectural decisions. Examples of meaningful file names include:

- `0001-independent-business-logic.md`
- `0002-implement-JSF-webapp.md`
- `0003-choose-database.md`

The directory containing these ADR files is the history of all architectural decisions made for the project.

2 Template

There are a few templates available to help provide consistent formatting of an ADR. The recommended template format contains the following sections.

Title Short phrase that describes the key decision.

Date When the decision was made.

Status Current status of the decision (i.e. proposed, accepted, deprecated, superseded or rejected).

Summary Summarise the decision and its rationale.

Context Describe the facts that influence the decision. State these in value-neutral language.

Decision Explain how the decision will solve the problem, in light of the facts presented in the context.

Consequences Describe the impact of the decision. What becomes easier and harder to do? There will be positive, neutral and negative consequences, identify all of them.

3 ADR Example

The following is an example ADR from the Sahara eCommerce application from section 4 of the *Architectural Views* notes.

1. Independent Business Logic
Date: 2022-01-06
Status: Accepted
Summary
In the context of delivering an application with multiple platform interfaces, *facing* budget constraints on development costs, *we decided* to implement all business logic in an independent tier of the software architecture, *to achieve* consistent logical behaviour across platforms, *accepting* potential complexity of interfaces to different platforms.
Context

- The system is to have both mobile and web application frontends.
- Marketing department wants a similar user experience across platforms.
- Delivering functional requirements requires complex processing and database transactions.
  - Product recommendations based on both a customer's history and on purchasing behaviour of similar customers.
  - Recording all customer interactions in the application.
- Sales department wants customers to be able to change between using mobile and web applications without interrupting their sales experience.
- Development team has experience using Java.

Decision
All business logic will be implemented in its own tier of the software architecture. Web and mobile applications will implement the interaction tier. They will communicate with the backend to perform all logic processing. This provides clear separation of concerns and ensures consistency of business logic across frontend applications. It means the business logic only needs to be implemented once. This follows good design practices and common user interface design patterns.

The business logic will be implemented in Java. This suits the current development team's experience and is a common environment. Java has good performance characteristices. Java has good support for interacting with databases, to deliver the data storage and transaction processing requirements.

Consequences

Advantages

- Separation of concerns, keeping application logic and interface logic separate.
- Ensures consistency, if business logic is only implemented in one place.
- Business logic can execute in a computing environment optimised for processing and transactions.
  - Also makes load balancing easier to implement.

Neutral

- Multiple interfaces are required for different frontend applications. These can be delivered through different Java libraries.

Disadvantages

- Additional complexity for the overall architecture of the system.

4 Quality

A well written ADR explains the reasons for making the decision, while discussing pros and cons of the decision. In some cases it is helpful to explain reasons for not selecting other seemingly good options. The ADR should be specific about a single decision, not a group of decisions.

The context should provide all the relevant facts that influence the decision. The decisions section should explain how business priorities or the organisation's strategic goals influenced the decision. Where the team's membership or skills has influenced the decision, these should be acknowledged. All consequences should be described. These may include decisions that need to be resolved later or links to other ADRs that record decisions that were made as a result of this decision.

The summary follows the format of a Y-statement [4]. It includes key information about the decision for quick consumption, leaving the details for the rest of the ADR. The template for a Y-statement is:

In the context of *functional requirement or architecture characteristic*,
facing *non-functional requirement or quality attribute*,
we decided *selected option*,
to achieve *benefits*,
accepting *drawbacks*.

The Y-statement can be expanded to include **neglected** *alternative options* after the **we decided** clause. A **because** clause providing *additional rationale* can be added to the end of the statement. Some teams use this expanded form of the Y-statement as the complete ADR for simple decisions.

ADRs should be reviewed after about one month. Consider how the consequences have played out in reality. Revise bad decisions before too much has been built based on the decision. The ADR may need to be updated to include any consequences that have been uncovered in practice.

ADRs should be immutable. Do not change existing information in an ADR. An ADR can have new information added to it, like adding new consequences. If the decision needs to be changed, create a new ADR. The original ADR should have its status changed to *deprecated*, and then *superseded* once the new decision is accepted and put into practice. Add a link in the original ADR to the new ADR, and the new ADR should link back to the original.

Sometimes an ADR is not superseded by a new ADR, instead it is extended or amended by another ADR. In these cases, the original ADR still has an *accepted* status but a link is added indicating the other ADR that amends or extends the original ADR. The new ADR then includes a link back to the original ADR, indicating that the new ADR extends or amends the original.

5 What to Put in an ADR?

Only document important decisions that affect the architecture's structure, non-functional characteristics, dependencies, interfaces, or construction techniques. Michael Nygard calls these "architecturally significant" decisions [1]. The types of decisions that usually should be documented are:

- Critical or important to delivering system functionality.
- Helps deliver important quality attributes.
- Unconventional or risky approach to a solution.
- Has expensive consequences.
- Has long lasting effects or will affect a large part of the design.
- Will affect a large number of stakeholders or an important stakeholder.
- Took a long time or significant effort to decide.
- Unexpected decisions that were required late in the project. These may also be important learning experiences.

6 Conclusion

Decisions that affect the software architecture need to be recorded for reference in the future. In a large project it is not possible for everyone to be aware of every decision that is made. ADRs provide a mechanism for team members to find reasons for decisions to help them understand how to work within its constraints. ADRs give new team members a starting point to learn about the architecture and understand how it evolved into its current state. They also provide a reminder for yourself when you are wondering why you did something a particular way, or when you need to explain your decision to someone else, months after you made the decision.

The takeaway is, write an ADR whenever you make an important decision. You, or a colleague, will need to refer to it at some point in the future. Eli Perkins has a good summary of why you should write ADRs at the GitHub Blog[1] [5].

References

[1] M. Nygard, "Documenting architecture decisions." https://cognitect.com/blog/2011/11/15/documenting-architecture-decisions, November 2011. Accessed: 2022-01-27.

[2] P. Kruchten, R. Capilla, and J. C. Duenas, "The decision view's role in software architecture practice," *IEEE software*, vol. 26, no. 2, pp. 36–42, 2009.

[3] P. Kruchten, "Architectural blueprints — the '4+1' view model of software architecture," *IEEE software*, vol. 12, no. 6, pp. 42–50, 1995. https://www.cs.ubc.ca/~gregor/teaching/papers/4+1view-architecture.pdf.

[4] O. Zimmermann, "Architectural decisions – the making of." https://ozimmer.ch/practices/2020/04/27/ArchitectureDecisionMaking.html, March 2021. Accessed: 2022-02-02.

[5] E. Perkins, "Why write adrs." https://github.blog/2020-08-13-why-write-adrs/, August 2020. Accessed: 2022-02-02.

[1] https://github.blog/2020-08-13-why-write-adrs/