Share

ZIO Contact Consulting IT architect and part time lecturer.

A Definition of Done for Architectural Decision Making

May 22, 2020 (Updated: Nov 26, 2021)
Reading time: 8 minutes

Content Outline

It is good to know when the most responsible moment for an architectural decision about a pattern or technology has come. But when can a decision be considered done? This post suggests five criteria to help you decide whether you are ready to move on: *evidence*, *criteria*, *agreement*, *documentation* and *realization/review* plan.

Context: Definition of Done in Agile Sprints/Iterations

Definition of Done is an essential Agile practice that helps to avoid misunderstandings and keep the team focused. The standard Definition of Done (a.k.a. "done list") deals with product increments or features (often stories), but not with technical activities.

Technical tasks, including analysis and design work, can be brought into the iteration planning in several ways: architectural spikes, epics annotated with quality goals and technical stories.[1] Once you have started working on such tasks, how do you know that you have analyzed, designed and decided just enough?

Feature- and implementation-oriented Definitions of Done are not necessarily suited for technical tasks, no matter how these are handled. Hence, this post aims at establishing one — more precisely, one for *Architectural Decision (ADs)*. It follows up from my previous post that focussed on Architectural Decision (AD) capturing.

Proposal: Definition of Done for Architectural Decisions

I reflected a bit what I expect from a design/an AD when I review one, and discussed with peers. The result is inspired by the five SMART criteria used in project and people management (but also when eliciting non-functional quality requirements):[2]

1. *Evidence*: You have gained reason to believe that the chosen design (selected pattern, technology, product or open source asset and its configuration, that is) will work — which means: a) it helps satisfy specific, measurable quality requirements, b) it does not break previous ADs by accident, and c) it is actionable: implementable and deployable in the short term and manageable and maintainable in the long run (if these are important qualities). You can gain this evidence in several ways:
   - Implement a proof-of-concept or architectural spike yourself.[3]
   - Put such evaluation activity on the backlog, have a qualified team member work on it and analyze the results.
   - Ask somebody you trust to vouch for this design option.

2. *Criteria*: At least two alternatives have been identified and investigated, and compared by stakeholder concerns and other decision drivers (regarding short term/long term impact). One is chosen, and the other ones are rejected (or kept as fallbacks).[4]
   - You might want to apply a recognized, systematic evaluation technique, but also be pragmatic. It is not cost-effective to establish and evaluate 20+ criteria for 5+ alternatives per AD thoroughly (you might have to make 100s while sprinting!).
   - You might want to predefine the criteria across projects (portfolio or company level) to make architectures (and portfolio products) comparable.

3. *Agreement*: At least one mentor or peer and the team have challenged the AD and agree with outcome and rationale. The amount of "decision

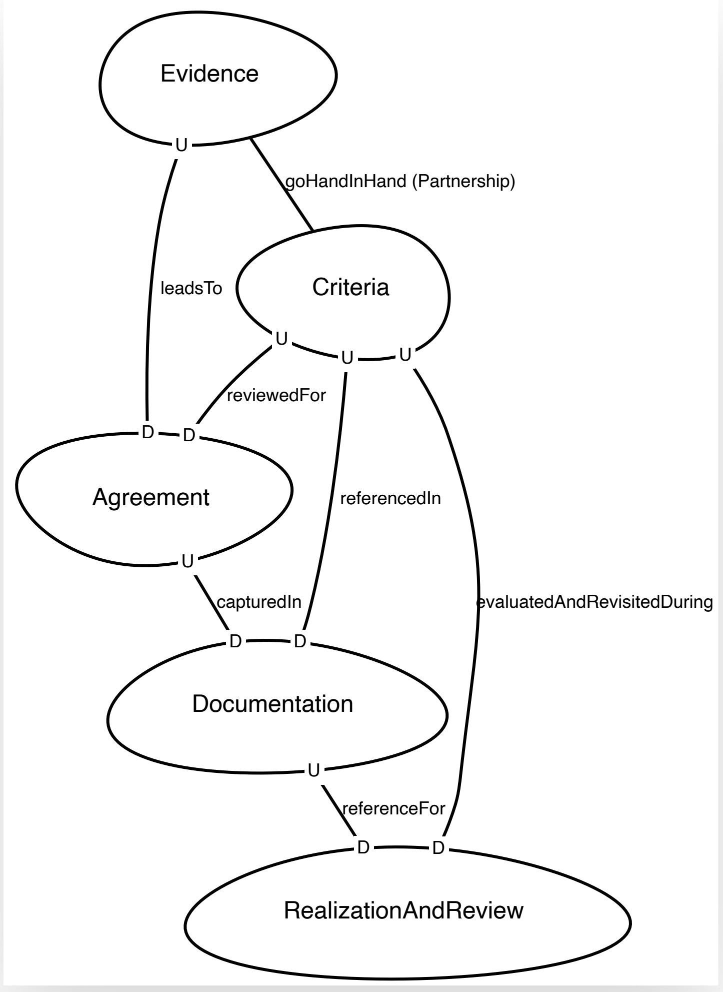
socialization" that is adequate depends on project context and decision making culture. Sometimes nobody objecting in a design workshop or review meeting are enough, sometimes explicit approvals or formal sign offs by the entire team or by external stakeholders (for instance, a design authority) may be required.

   - Agile teams may differ from those applying more traditional plan-driven methods. Often all team members participate in the decision making; decentralization and autonomy are often emphasized (as in microservices architectures, by the way).
   - The governance level (for instance, enterprise, portfolio, solution in SAFe terms) has an impact as well. The wider a decision reaches, the more buy in is required. All relevant stakeholders should be involved early, otherwise they might fight the decision because one of their key concerns is not considered (or simply because they felt left out). Early means early *enough* from a recipient point of view here: last-minute requests for comments and approval are usually not appreciated. In my experience, stakeholder involvement must be planned ahead at least a bit, otherwise it might be forgotten. People usually are more willing to comment if they know that something important is coming their way; they might even be willing to block time.

4. *Documentation*: The decision has been captured and shared, preferably in a lean and light template such as a Y-statement or a Markdown Architectural Decision Record (MADR). Other ADR notations are fine too, as long as they are used consistently and continuously.
   - The justification should provide convincing rationale referencing requirements and information gathered to meet `E`, `C` and `A`; see previous post for examples of good and bad justifications.
   - The decision record must be made available to all affected parties (for instance, announced and "published" in a collaboration or document sharing tool).

5. *Realization and review plan*: a) To be effective, a made decision must be executed upon (enacted); this work has been scheduled. It also has been identified when to evaluate whether the AD has been implemented as intended and that the resulting design indeed works as desired (this evaluation corresponds to testing in development). b) You have also looked ahead and planned when to talk about the AD in a review meeting or retrospective. Two elaborate forms of such reviews are ATAM and DCAR. In hindsight, you may want to answer questions such as:
   - Are we (still) content with the AD outcome? Are there new alternatives (options)?
   - When will we revisit and possibly revise this decision (expiration)?

Let's capture these five criteria and their relations in a *context map*:[5]



If the above criteria discussion was too verbose for your taste, how about a checklist:

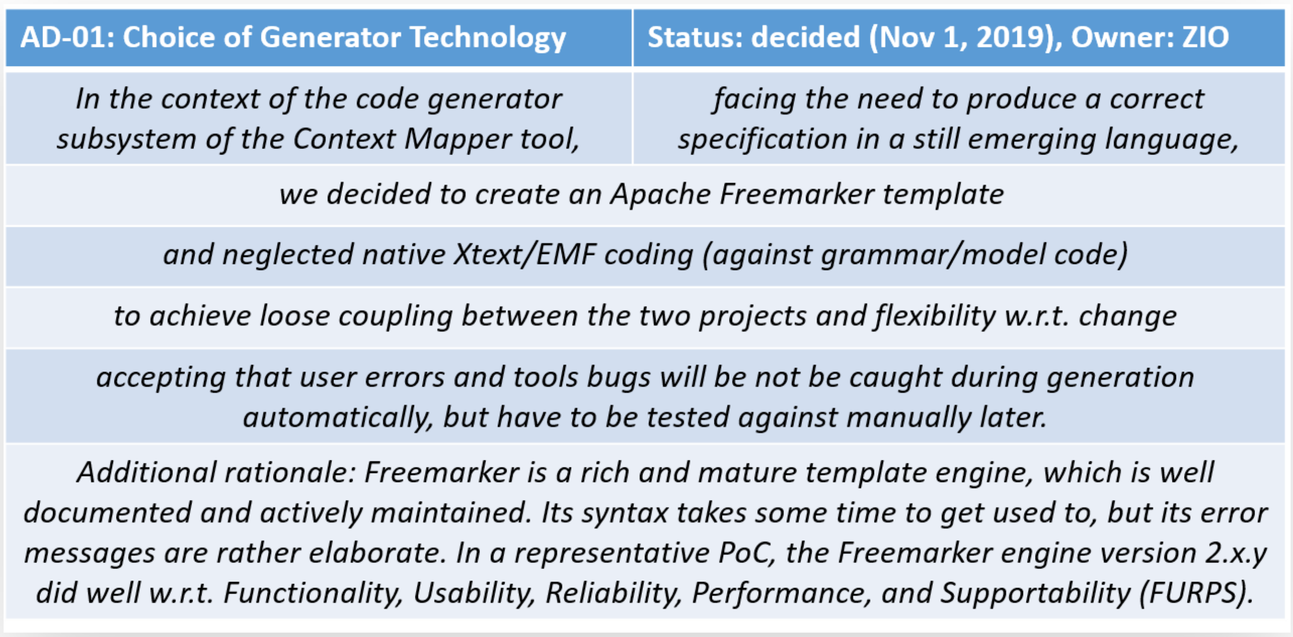
1. ✓ Are we confident enough that this design will work (`E`)?
2. ✓ Have we decided between at least two options, and compared them (semi-)systematically (`C`) ?
3. ✓ Have we discussed among each other and with peers just enough and come to a common view (`A`)?
4. ✓ Have we captured the decision outcome and shared the decision record (`D`)?
5. ✓ Do we know when to realize, review and possibly revise this decision (`R`)?[6]

If you can answer "yes" five times in this quick test, you are done with an AD. If any answer is missing, you might have to invest a bit more time — at least to justify why this criterion does not apply for this particular AD. The checklist is also featured in the short version of this post on Medium.

Example: Selection of Freemarker Template Engine

On the Context Mapper project, we had to decide how to generate service contracts from bounded contexts, a pattern in Domain-Driven Design, last year. The contexts are defined in CML. The target language is MDSL, another Domain-Specific Language (DSL) for microservices APIs that kept on changing at that time; no stable library offering an abstract syntax tree was available yet. Hence, we had to make an AD about the CML-to-MDSL mapping technology; its most responsible moment had come since the generator was supposed to be implemented in the next iteration.

> *(E) I had gained quite positive experience with Apache Freemarker on a previous (smaller) project, and our requirements were similar (for instance, Java was used on both projects, and the level of abstraction of the source and the target DSL were not too different). So I suggested this option to Stefan Kapferer, who was my master student at that time.*
>
> *(E, C) Stefan got acquainted with Freemarker and also looked for alternatives (with criteria such as vitality of community, documentation, expressivity). We briefly considered Eclipse ecore/EMF coding as alternative because MDSL uses Xtext, which in turn depends on EMF and ecore. We identified flexibility/modifyablity and loose coupling as advantages of Freemarker; the main advantage of Eclipse ecore/EMF is full and instant validation of the target model. Downsides were an extra dependency (Freemarker) and development effort and complexity (ecore/EMF).*
>
> *(A) In a meeting, we decided that templating is the way to go for this particular CML generator, and that Freemarker is an adequate, state of-the-art choice in Java land.*
>
> *(D) The decision was documented in the final project report (a Y-statement for it is below).*
>
> *(R) We agreed to reflect in an iteration review meeting. Later on, we revisited the decision when resuming work on Context Mapper this year (Stefan now works at the IFS). Due to the positive experience, we decided to stick to it and even use it further, for instance to generate JDL files from the tactic DDD models in Context Mapper (as input to the rapid application development framework/ecosystem JHipster).*



AD-01: Choice of Generator Technology	**Status: decided (Nov 1, 2019), Owner: ZIO**
In the context of the code generator subsystem of the Context Mapper tool,	*facing the need to produce a correct specification in a still emerging language,*
we decided to create an Apache Freemarker template	
and neglected native Xtext/EMF coding (against grammar/model code)	
to achieve loose coupling between the two projects and flexibility w.r.t. change	
accepting that user errors and tools bugs will be not be caught during generation automatically, but have to be tested against manually later.	
Additional rationale: Freemarker is a rich and mature template engine, which is well documented and actively maintained. Its syntax takes some time to get used to, but its error messages are rather elaborate. In a representative PoC, the Freemarker engine version 2.x.y did well w.r.t. Functionality, Usability, Reliability, Performance, and Supportability (FURPS).	

If you compare this decision record with the Y-template and example in this post, you will notice that the last part is new here, a free form sentence providing additional evidence and arguments from the criteria comparison and agreement discussion.[7]

Concluding Thoughts

The take-away messages from this post are:

- While it is important to know when the most (vs. last) responsible moment for an architectural decision has come, it is equally important to know when it has passed and you are DONE-done with an AD.
- A checklist or quick test can help the team to agree that it is actually time to move on.
- I proposed five criteria `E, C, A, D, R` here: presence of *evidence*, *criteria*, *agreement*, *documentation* and identification of *realization and review* tasks.
- A criteria-based checklist can remove ambiguities and cut unnecessary, inefficient discussions short by clarifying the difference between done and DONE-done.

Some ADs take longer than others (to make and agree upon). The strategic buy-or-build-or-rent decision for a company-wide customer relationship management system will require significant `E`, `C`, and `A` work, while the tactic decision to wrap calls to a messaging system or cloud provider to promote portability (hopefully) reaches the DONE-done state much sooner (but also might have to be revised more often). Cost, effort and risk guide the prioritization, as for instance RCDA teaches us.

You are never done with the entire decision making: one AD typically triggers others immediately, and the made ADs age and therefore might require your attention later. There always will be a backlog of less important/urgent ADs, new ones, ones to be revisited due to technology evolution and feedback from customers, operators or other stakeholders (the ultimate evidence for this observation can be found in the concepts of continuous and evolutionary architectures).

Unlike the Y-statement template used in the example, I have not applied the "ecADR" checklist proposed in this post as such in practice much yet (I certainly have applied all five criteria tacitly though). So your feedback is appreciated — do the above five criteria work for you? Did I miss a criterion (checklist item)? Contact me!

– Olaf (a.k.a. ZIO)

Acknowledgements

I would like to thank Mirko Stocker, Stefan Kapferer, Cesare Pautasso, Anton Jansen, Daniel Lübke, Uwe Zdun and Christian Ringler for participating in the "brain dump" experiments and/or fruitful discussions during the preparation of the first version of this post.

Notes

1. if you do not like the term "technical story", please use any term/concept that is commonly used for "non-story" type of tasks in your organization ↩

2. Many variants of SMART exist. For requirements, I go with *specific* (to a context), *measurable* (or testable), *agreed upon* (in the team, with the client), *realistic* (for instance, by establishing landing zones), and *time bound* (aligned with the project planning/the product roadmap). ↩

3. coding architects love this option and might argue that it is the only true one; however, it might not scale and fit each decision making context ↩

4. At SATURN 2010, I emphasized the importance of the alternatives part in my presentation, and had a lively discussion with Philippe Kruchten, who had talked about ADs in a tutorial and his keynote; we eventually agreed that it does not make sense to list "fake" alternative just to be template- or method-compliant. Sometimes, there only is one choice (which should still be justified). ↩

5. Context Map is a pattern in strategic domain-driven design. This map is not handmade, but generated from a DSL specification file with Context Mapper. ↩

6. Note that I switched from "I" to "we" here; this was deliberate, as many decisions are made by teams and architecting has become a virtual, shared responsibility in many teams (see this article for related insights). ↩

7. My second law of method adoption (after "if in doubt leave it out", see previous post) is "do not follow blindly but adopt to your needs". ↩

Category: Practices

Tags: #agile architecting #architectural decisions #checklists #software architecture #template

Previous: Getting Started with Microservice API Patterns (MAP)

Next: How to Write Review-Friendly Articles

Explore Blog by category →

index (5) authoring (6) practices (11) patterns (3) misc (1)