AARHUS UNIVERSITET

# Software Architecture in Practice

## Architectural Decisions and Rationale

# Warstory

- Major IT company in the region
- Customer requirement:
  - Convert **huge** streams of XML data in soft realtime
- Architect decides:
  - Use **SAX** (a *constant memory* parsing technique)
    - SAX: event-driven (callback) parsing
    - DOM: hold full document in memory ☠
- Developers:
  - Build test environment (=small XML files)
  - Solve functional requirement **using SAX to produce in-memory tree**

# Warstory

- Result?
  - Too late to change this innocent looking ”impl. choice”
  - Buy large costly machines
  - Severely constrain size of XML input

- Why?
  - Decision was not
    - Communicated!
    - (Documented)

# Architectural Decisions

The Short Version

AARHUS UNIVERSITET

# Architectural Knowledge

- Krutchen et al.

**Architectural Knowledge**

**=**

**Design Decision + Design**

AARHUS UNIVERSITET

# Design decision types

- Four types
  - **Implicit, undocumented**
    - SA@work: a lot of this!
  - **Explicit, undocumented**
  - **Explicit, explicitly undocumented**
  - **Explicit, documented**

- Exercise:
  - Why ”explicitly undocumented?”
- Warstory:
  - Jyske Bank never discussed security! Why not?

# M. Nygard

Two options when encounter decision without rationale documented

Agile methods are not opposed to documentation, only to valueless documentation. Documents that assist the team itself can have value, but only if they are kept up to date. Large documents are never kept up to date. Small, modular documents have at least a chance at being updated.

Nobody ever reads large documents, either. Most developers have been on at least one project where the specification document was larger (in bytes) than the total source code size. Those documents are too large to open, read, or update. Bite sized pieces are easier for for all stakeholders to consume.

1. **Blindly accept the decision.**

   This response may be OK, if the decision is still valid. It may not be good, however, if the context has changed and the decision should really be revisited. If the project accumulates too many decisions accepted without understanding, then the development team becomes afraid to change anything and the project collapses under its own weight.

2. **Blindly change it.**

   Again, this may be OK if the decision needs to be reversed. On the other hand, changing the decision without understanding its motivation or consequences could mean damaging the project's overall value without realizing it. (E.g., the decision supported a non-functional requirement that hasn't been tested yet.)

AARHUS UNIVERSITET

# Suggestion: ADR

- ADR
  - **Architectural Decision Record**
    - *For architecturally significant decisions*

"An architecture decision record is a short text file in a format similar to an Alexandrian pattern that describes a set of forces and a single decision in response to those forces."

**Title** These documents have names that are short noun phrases. For example, "ADR 1: Deployment on Ruby on Rails 3.0.10" or "ADR 9: LDAP for Multitenant Integration"

**Context** This section describes the forces at play, including technological, political, social, and project local. These forces are probably in tension, and should be called out as such. The language in this section is value-neutral. It is simply describing facts.

**Decision** This section describes our response to these forces. It is stated in full sentences, with active voice. "We will ..."

**Status** A decision may be "proposed" if the project stakeholders haven't agreed with it yet, or "accepted" once it is agreed. If a later ADR changes or reverses a decision, it may be marked as "deprecated" or "superseded" with a reference to its replacement.

**Consequences** This section describes the resulting context, after applying the decision. All consequences should be listed here, not just the "positive" ones. A particular decision may have positive, negative, and neutral consequences, but all of them affect the team and project in the future.

1-2 pages long

# ADR

- They are simple text files, under version control and in the project's source code
  - i.e. rationale is co-located with code
    - *So I can find it!*

We will keep ADRs in the project repository under doc/arch/adr-NNN.md

We should use a lightweight text formatting language like Markdown or Textile.

ADRs will be numbered sequentially and monotonically. Numbers will not be reused.

If a decision is reversed, we will keep the old one around, but mark it as superseded. (It's still relevant to know that it *was* the decision, but is *no longer* the decision.)

We will use a format with just a few parts, so each document is easy to digest. The format has just a few parts.

# Keeling/ECSA 2018

- About 70 ADRs in IBM Watson (?) project

https://agile2018.sched.com/event/EUBj/share-the-load-distributing-design-authority-with-lightweight-decision-records-michael-keeling-joe-runde

## ADR 1: Use the pipe-and-filter library

This ADR is closely related to ADR 2.

In the past while writing agents, we decided it was a good idea to use the pipe and filter architectural pattern. See this ADR for context (if it's still around).

We also found that when the engineers who like pipes and filters took their eyes off of a project, changes would creep in that violated the pattern. Eventually this defeated the purpose of trying to structure the code in an architecturally evident way.

Joe has written a pipe and filter library that will soon be open sourced by the team. Using a library like this enforces the architectural style on the project: there are few ways to code around the framework short of ditching it altogether. This stops engineers from accidentally violating key assumptions on accident.

### Decision

We will use the pipe-and-filter library to organize all application logic for the training agents.

### Status

Accepted

### Consequences

All training agents must fit within the context of pipe and filter. For the moment, we think that this is the case. There may be some pipelines that are very small, but will still work.

All filters must be completely decoupled from each other.

Concurrency can be controlled on a per-filter basis. This means we can write faster programs, but we also must be careful to write thread-safe filters if we want to run them concurrently.

Reacting to errors or expected failure modes will be much easier, and can be implemented with the message subscriber feature of the pipe and filter library.

There will be stricter controls on logging and metrics- they must be implemented as message subscribers. This decouples them from the filters entirely, limiting the possibility of mistakes and duplication of logging/metrics code for common error cases, but also means that more work is required to carefully pass good error responses around so that well-formed messages can be constructed and enqueued by the filters.

Uncaught race conditions or errors in the pipeline library may cause failures in production.

AARHUS UNIVERSITET

# Tyree and Akerman

- A more elaborate technique…

**Table 1**

**Architecture decision description template**

| | |
|---|---|
| Issue | Describe the architectural design issue you're addressing, leaving no questions about why you're addressing this issue now. Following a minimalist approach, address and document only the issues that need addressing at various points in the life cycle. |
| Decision | Clearly state the architecture's direction—that is, the position you've selected. |
| Status | The decision's status, such as pending, decided, or approved. |
| Group | You can use a simple grouping—such as integration, presentation, data, and so on—to help organize the set of decisions. You could also use a more sophisticated architecture ontology, such as John Kyaruzi and Jan van Katwijk's, which includes more abstract categories such as event, calendar, and location.[8] For example, using this ontology, you'd group decisions that deal with occurrences where the system requires information under event. |
| Assumptions | Clearly describe the underlying assumptions in the environment in which you're making the decision—cost, schedule, technology, and so on. Note that environmental constraints (such as accepted technology standards, enterprise architecture, commonly employed patterns, and so on) might limit the alternatives you consider. |
| Constraints | Capture any additional constraints to the environment that the chosen alternative (the decision) might pose. |
| Positions | List the positions (viable options or alternatives) you considered. These often require long explanations, sometimes even models and diagrams. This isn't an exhaustive list. However, you don't want to hear the question "Did you think about … ?" during a final review; this leads to loss of credibility and questioning of other architectural decisions. This section also helps ensure that you heard others' opinions; explicitly stating other opinions helps enroll their advocates in your decision. |
| Argument | Outline why you selected a position, including items such as implementation cost, total ownership cost, time to market, and required development resources' availability. This is probably as important as the decision itself. |
| Implications | A decision comes with many implications, as the REMAP metamodel denotes. For example, a decision might introduce a need to make other decisions, create new requirements, or modify existing requirements; pose additional constraints to the environment; require renegotiating scope or schedule with customers; or require additional staff training. Clearly understanding and stating your decision's implications can be very effective in gaining buy-in and creating a roadmap for architecture execution. |
| Related decisions | It's obvious that many decisions are related; you can list them here. However, we've found that in practice, a traceability matrix, decision trees, or metamodels are more useful. Metamodels are useful for showing complex relationships diagrammatically (such as Rose models). |
| Related requirements | Decisions should be business driven. To show accountability, explicitly map your decisions to the objectives or requirements. You can enumerate these related requirements here, but we've found it more convenient to reference a traceability matrix. You can assess each architecture decision's contribution to meeting each requirement, and then assess how well the requirement is met across all decisions. If a decision doesn't contribute to meeting a requirement, don't make that decision. |
| Related artifacts | List the related architecture, design, or scope documents that this decision impacts. |
| Related principles | If the enterprise has an agreed-upon set of principles, make sure the decision is consistent with one or more of them. This helps ensure alignment along domains or systems. |
| Notes | Because the decision-making process can take weeks, we've found it useful to capture notes and issues that the team discusses during the socialization process. |

AARHUS UNIVERSITET

# Bass at al.’s take…

Hmmm

AARHUS UNIVERSITET

# Bass et al.?

- 2nd Edition: *Rationale is important.*
  - And that was it…
- 3rd Edition: *Seven categories of design decisions*
  - *Allocation of responsibilities* — Cutting the cake
  - *Coordination model* — Comm. mechanisms
  - *Data model* — Persistence, lifecycle
  - *Mapping among architectural elements* — Across views
  - *Management of resources* — CPU, IO, Memory
  - *Binding time decisions* — Variant handling
  - *Choice of technology* — Technology stack

AARHUS UNIVERSITET

# QA Checklist (Availability)

**Table 5.4. Checklist to Support the Design and Analysis Process for Availability**

| Category | Checklist |
|---|---|
| Allocation of Responsibilities | Determine the system responsibilities that need to be highly available. Within those responsibilities, ensure that additional responsibilities have been allocated to detect an omission, crash, incorrect timing, or incorrect response. Additionally, ensure that there are responsibilities to do the following:<br>▪ Log the fault<br>▪ Notify appropriate entities (people or systems)<br>▪ Disable the source of events causing the fault<br>▪ Be temporarily unavailable<br>▪ Fix or mask the fault/failure<br>▪ Operate in a degraded mode |
| Coordination Model | Determine the system responsibilities that need to be highly available. With respect to those responsibilities, do the following:<br>▪ Ensure that coordination mechanisms can detect an omission, crash, incorrect timing, or incorrect response. Consider, for example, whether guaranteed delivery is necessary. Will the coordination work under conditions of degraded communication?<br>▪ Ensure that coordination mechanisms enable the logging of the fault, notification of appropriate entities, disabling of the source of the events causing the fault, fixing or masking the fault, or operating in a degraded mode.<br>▪ Ensure that the coordination model supports the replacement of the artifacts used (processors, communications channels, persistent storage, and processes). For example, does replacement of a server allow the system to continue to operate? |

Etc.

# Bass et al.?

- 4th Edition: *Tactics-based Questionnaire*
  - Tabular format, basically enumerating all tactics form the catalogue, and asking the questions
    - Does architecture support it (yes/no) ?
    - Design decision & Location
      - More elaborate decision (how to design/impl tactic?)
      - … in which module/package/service/…
    - Risk
      - Assessment on L/M/H scale (low/medium/high risk)
    - **Rationale**
      - Argumentation for Why, and assumptions made

AARHUS UNIVERSITET

# Tactics-based Q (Availability)

**Table 4.3** Tactics-Based Questionnaire for Availability

| Tactics Group | Tactics Question | Support? (Y/N) | Risk | Design Decisions and Location | Rationale and Assumptions |
|---|---|---|---|---|---|
| Detect Faults | Does the system use **ping/echo** to detect failure of a component or connection, or network congestion? | | | | |
| | Does the system use a component to **monitor** the state of health of other parts of the system? A system monitor can detect failure or congestion in the network or other shared resources, such as from a denial-of-service attack. | | | | |
| | Does the system use a **heartbeat**—a periodic message exchange between a system monitor and a process—to detect failure of a component or connection, or network congestion? | | | | |
| | Does the system use a **timestamp** to detect incorrect sequences of events in distributed systems? | | | | |

AARHUS UNIVERSITET

# Discussion

- Bass et al. apparently have a hard time to decide how to document *rationale* for decision.

- Frankly, I do not find any of them that practical
  - A large tabular list of ‘not supported’ with a small column to put the rationale into?

**Table 4.3** Tactics-Based Questionnaire for Availability

| Tactics Group | Tactics Question | Support? (Y/N) | Risk | Design Decisions and Location | Rationale and Assumptions |
|---|---|---|---|---|---|
| Detect Faults | Does the system use **ping/echo** to detect failure of a component or connection, or network congestion? | | | | |
| | Does the system use a component to **monitor** the state of health of other parts of the system? A system monitor can detect failure or congestion in the network or other shared resources, such as from a denial-of-service attack. | | | | |
| | Does the system use a **heartbeat**—a periodic message exchange between a system monitor and a process—to detect failure of a component or connection, or network congestion? | | | | |
| | Does the system use a **timestamp** to detect incorrect sequences of events in distributed systems? | | | | |

# Summary

- *Rationale is important to capture in a team of any size > 1*
  - Ok, even in a team of size 1 ☺

- Nygaard's take combines appealing features