Software architecture evaluation – methods, experiences, issues

Turku 21.11.2013
Kai Koskimies
Tampere University of Technology
Contents

• Introduction & background
• Software architecture evaluation as information source
• Challenges in scenario-based evaluation
• DCAR: Decision-based software architecture evaluation
• Conclusions
Background

Two nationally funded research projects with global machine manufacturers (John Deere, Sandvik, Metso, Kone, Areva, + software subcontractors) in 2007-2011.

Aims: Improving architecture work practices for machine control systems

About 20 ”real-life” architectural evaluations carried out in industry
SA evaluation problem

Quality requirements

Software architecture
Scenario-based approach: use scenarios as test cases

- Quality requirements
- Software architecture
- Refining requirements
- Identification of solutions
- Scenarios
- Analysis
- Architectural decisions
Evaluation reports: first format

Grouped according to quality attribute

Architectural solutions extracted from analysis

Risks identified on the basis of analysis

Main problems:

Unstructured analysis

Few architectural solutions identified

<table>
<thead>
<tr>
<th>Scenario description</th>
<th>If components can not communicate due protocol version differences, this is noticed and shown as a fault.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Analysis</td>
<td>Drive and elevator software might have incompatible protocol version after a update. This might happen for example when a lift CPU is broken and it is replaced. This should be detected and an error message should be presented immediately so that the service personnel can notice the problem. Where is the information about the software compatibility? Not just a protocol versioning problem. All communicating partners should verify that they are compatible to make sure that the protocol version is compatible. On the other hand, incompatible elevator will go solo and the updater should notice that. In group controller case all the elevators go solo. In practise, the elevator software in large setting is updated in parts (subgroups) or everything is updated in one go during the night.</td>
</tr>
<tr>
<td>Architectural approaches</td>
<td>Protocol compatibility taxonomy. If incompatibility is found the system has to shutdown or resolve the state gracefully. Formal protocol specification.</td>
</tr>
<tr>
<td>Risks</td>
<td>Mixed set of software is expected to communicate when in reality they are incompatible.</td>
</tr>
</tbody>
</table>
Evaluation reports: second format

Flat list of analyzed scenarios (in priority order)

All solutions identified beforehand

Scenario-related solutions identified & analyzed

Risks, non-risks and trade-offs identified from solution analysis

Main problems:

The scenario "disappears": difficult to get an overview of the scenario (how is the scenario managed as a whole?)

Sometimes repeating "trivial" information
Scenario description

List of architectural decisions that are potentially involved in the scenario

The architect’s explanation on how to manage the scenario

Explanation of how the architectural decision is related to this scenario, identification of possible problems

**Scenario:** The feed control algorithm is moved to another controller to balance performance. This is done in two weeks.

<table>
<thead>
<tr>
<th>#</th>
<th>Architectural Decisions</th>
<th>T</th>
<th>R</th>
<th>N</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.2</td>
<td>Event publisher-subscriber</td>
<td>T2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3.4</td>
<td>System state objects wraps Variable Manager</td>
<td></td>
<td></td>
<td>N7</td>
</tr>
<tr>
<td>3.17</td>
<td>Commander is a facade for user commands</td>
<td></td>
<td></td>
<td>N8</td>
</tr>
<tr>
<td>3.18</td>
<td>Executor</td>
<td></td>
<td></td>
<td>N9</td>
</tr>
</tbody>
</table>

**Description:**
The feed algorithm function, that is one task, is moved to the other controller. The local executor is configured so that this task is removed and the remote executor is configured to include this task. The task communicates with the rest of the system with system state objects or events. The system state object can write either to local variable or to the CAN. If all the needed data is not on the bus, new message(s) must be introduced. The commander can send its events to the CAN and the remote end may read these from the bus. Therefore, the remote end sees the same commands as the original system.

**Argumentation:**
3.2 Event publisher-subscriber decouples the modules from each other. By reconfiguring the listeners and possibly adding a simple subscriber to serialize the event via CAN and adding receiving end deserializing publisher. This increases the bus load. Some events may be too big to fit in one message. This adds to the bus load and complexity.
Exploiting SA evaluation for information mining

Architecture -> Architecture evaluation

Requirements -> Architecture evaluation

? -> Risks
Example: Pattern mining

Architecture

Requirements

Architecture evaluation

Design patterns (domain-oriented)

Risks
Pattern language for machine control

Patterns identified as a sideprocess of architectural evaluation (as solutions)

Aiming at a full generative pattern language for machine control systems

Several novel patterns, many existing patterns

Patterns elaborated in VikingPLoPs and EuroPLoPs
Architecture Knowledge Management (AKM)

• AKM is about approaches, methods, and techniques for capturing and using architectural knowledge

• Design (and architectural) patterns are one way to express general architectural knowledge

• Many proposals exist for tools to store and retrieve architectural knowledge (see e.g. M. Ali Babar et al (eds.), Architecture Knowledge Management – Theory and Practice, Springer 2009)
Frustrating experience …

• In a recent research project with industrial partners, AKM approaches were developed involving attractive (we thought) benefits e.g. in automating document production.

• Still, in spite of generally positive attitude among the practitioners, the partners did not see the cost-benefit ratio attractive enough to actually try out the approaches in an industrial context.

• Are these people stupid or what?
Problem: Feeding the beast
Problem: Information bloat

Falessi et al. (TOSEM, to appear): only half of design rationale information in full documentation is probably needed
The management domination problem

• In any collaborative human activity that needs coordination, management and administration tend to take over in the long run

• In software engineering, the counter reaction has been agile & lean

• Can AKM be lean?
Lean AKM Manifesto

1. The producing and consuming of architectural information in AKM should not require extra effort. There should be no activities that are related only to AKM itself during the software system lifecycle.

2. AKM should be invisible from the viewpoint of information producers and consumers. The producers and consumers should not need to be aware of AKM. AKM should be seamlessly integrated with usual knowledge-sharing activities like documentation, reviews and project meetings.

3. Only potentially useful architectural information should be stored in AKM. All stored information is expected to be used in the context of a probable development or an evolution scenario. The burden of useless information surpasses the possible benefit of coincidental usage.
Architectural knowledge

Architectural Knowledge = Design Decisions + Design

Kruchten 2006, Bosch 2004, Tyree et. al 2005

• Can we produce (essential) design decisions information effortlessly?

• Can we produce (essential) design information effortlessly?

• Once we have this kind of information, can we satisfy the needs of stakeholders effortlessly?
Towards invisible AKM

Activities
- Requirements specification
- Architectural evaluation
- Meetings (e.g. retrospects)
- Coding

Tools

Architectural knowledge repository

Virtual Architecture Informaticist
Observations

- Companies actually use ATAM-style evaluation also for learning the architecture and for producing (more covering) SA documentation, or information to be used in SA documentation.
- A lot of information potentially relevant for SA documentation comes up during ATAM-style evaluation.
- Evolution scenarios capture probable situations where SA documentation is needed.
- Scenario analysis reveals more or less the relevant architectural information related to that evolution situation.
- Light-weight, "good enough" documentation of SA can be produced with very little extra work as a side-effect of evaluation.
Example

Architectural decisions that have to be understood if the scenario gets realized

Effectively instructions for changing/exploiting the architecture in case the evolution scenario gets realized

**Scenario:** The feed control algorithm is moved to another controller to balance performance. This is done in two weeks.

<table>
<thead>
<tr>
<th>#</th>
<th>Architectural Decisions</th>
<th>T</th>
<th>R</th>
<th>N</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.2</td>
<td>Event publisher-subscriber</td>
<td>T2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3.4</td>
<td>System state objects wraps Variable Manager</td>
<td></td>
<td></td>
<td>N7</td>
</tr>
<tr>
<td>3.17</td>
<td>Commander is a facade for user commands</td>
<td></td>
<td></td>
<td>N8</td>
</tr>
<tr>
<td>3.18</td>
<td>Executor</td>
<td></td>
<td></td>
<td>N9</td>
</tr>
</tbody>
</table>

**Description:**
The feed algorithm function, that is one task, is moved to the other controller. The local executor is configured so that this task is removed and the remote executor is configured to include this task. The task communicates with the rest of the system with system state objects or events. The system state object can write either to local variable or to the CAN. If all the needed data is not on the bus, new message(s) must be introduced. The commander can send its events to the CAN and the remote end may read these from the bus. Therefore, the remote end sees the same commands as the original system.

**Argumentation:**
3.2 Event publisher-subscriber decouples the modules from each other. By reconfiguring the listeners and possibly adding a simple subscriber to serialize the event via CAN and adding receiving end deserializing publisher. This increases the bus load. Some events may be too big to fit in one message. This adds to the bus load and complexity.
Architectural knowledge from ATAM

- Major architecturally significant requirements
- Architecture design overview
- Architectural decisions
- Architectural & design patterns
- Architect’s maintenance instructions (scenario analysis)
- Known weaknesses of the architecture

In an experiment, about half of the AK repository contents originated from ATAM
Shortcomings in ATAM based AK

- No detailed & precise component architecture
  Possible solution: using RE tools for extracting component structure from source code, when available
- Some relationships between information items may be difficult to create on the fly (e.g. component – requirement)
- Decisions are only identified and named
Towards invisible AKM

Activities

- Requirements specification
- Architectural evaluation
- Meetings (e.g. retrospects)
- Coding
- ...

Tools

- Architectural knowledge repository

Tools

- Virtual Architecture Informaticist

Software Architecture Course, Turku, November 2013

Department of Pervasive Computing
Possible approach for SA informaticist: architectural slicing

- **Task-based slicing**
  Derivation of an information package addressing a specific task related to software architecture

- **Process-based slicing**
  Derivation of an information package corresponding to an artifact defined by the software development process

- **Item-based slicing**
  Derivation of an information package collecting all information relevant to understand a specific item involved in software architecture

- **Role-based slicing**
  Derivation of an information package from the viewpoint of a specific stakeholder role

(Eloranta et al. 2012)
Example: Task-based slicing

During maintenance, a maintainer is faced with a task of modifying the system and needs architectural guidance. The maintainer asks the Virtual SA informaticist to provide all architectural information relevant for the task.

The Virtual SA informaticist compares the task to the existing evaluation scenarios, and if a sufficient match is found, generates an information package exploiting the relationships of the scenario item (e.g. related decisions and design items).
Example: Process-based slicing

At any time, the Virtual SA informaticist can be asked to provide a comprehensive software architecture documentation that reflects the system at a given time. The Virtual SA informaticist follows a predefined document template and fills in the contents required by the template.
Example: Item-based slicing

A person can be interested in a specific architectural item, for example a component. The Virtual SA informaticist can be asked to provide an information package focusing on that component, to be used e.g. by a person who is expected to implement or maintain that component. The Virtual SA informaticist makes use of the relationships of that item (e.g. usage contexts, decisions, requirements, interfaces etc.) and generates an architectural mini-document for the component.
Example: Role-based slicing

A manager needs an overview description of the architectural aspects of a product, emphasizing business drivers, very high-level decisions, technological decisions, possible risk areas etc. The Virtual SA informaticist can be asked to provide such an information package by filtering out items that are irrelevant from the viewpoint of the given role.
TopDocs: Generating specialized software architecture documentation

Implemented on top of Polarion, a commercial ALM platform

Information emerging in architectural evaluation
Example: maintenance scenario documentation

Need-oriented architecture documentation (Eloranta et al. ECSA 2012)
Challenges in scenario-based evaluation

• Trustworthiness problem: can we rely on the outcome of the evaluation?
• Heavy-weightness problem: can we afford the time & resources?
Trustworthiness problem

- Scenario-based evaluation can be self-deception: there is no inbuilt coverage concept, or characterization of the confidence level of the results of the evaluation.
- Would the next not analyzed scenario reveal a problematic architectural decision or a compromised quality attribute?
- Scenarios are prioritized by the stakeholders which may have their own agenda: “safe” scenarios often get prioritized.
Heavy-weightness problem

- Medium-sized ATAM can take up to 70 person days
- ATAM assumes the presence of many stakeholders that are difficult to get available at the same time
- A lot of time is used to refine requirements
- Incompatible with agile approaches (e.g. Scrum)
DCAR: decision-based architecture evaluation method

Design = balancing the forces associated with a design problem

Forces pull the solution of the problem towards a certain direction
For a decision, forces are positive or negative

Example:

Problem: traditional DB leads to slow response of an application

Solution: use an in-memory DB system
DCAR process

- Identify forces that have influence on the architecture
- Identify architecture design decisions
- Prioritize design decisions
- Document design decisions (with forces)
- Evaluate design decisions against forces
DCAR: decisions are analyzed against forces

<table>
<thead>
<tr>
<th>Name</th>
<th>Redundancy of controllers</th>
</tr>
</thead>
<tbody>
<tr>
<td>Problem</td>
<td>The application should run even if the server fails</td>
</tr>
<tr>
<td>Solution or description of decision</td>
<td>The system is deployed to two servers: one is active, the other one is inactive. The active server provides all system services, while the passive one is running in the background. When the active server fails, the inactive server becomes active. During the switch over, the active server tries to update the passive one to make sure that it has the same data and status. Both servers have an identical software configuration. This solution follows the Redundant Functionality Pattern.</td>
</tr>
<tr>
<td>Considered alternative solutions</td>
<td>Apply the Redundancy Switch Pattern: Both servers are active; external logic is used to decide which output is actually used in the control. In this case, cyclic data copying could be avoided. However, applying this solution would require major modifications to the system. Even though availability would be increased, it would also cause additional costs. The customers are not prepared for paying more for higher availability. Additionally, the external logic component could become a potential single point of failure. Therefore, this alternative was discarded.</td>
</tr>
<tr>
<td>Forces in favor of decision</td>
<td></td>
</tr>
<tr>
<td></td>
<td>• Easier to implement than the alternative solution.</td>
</tr>
<tr>
<td></td>
<td>• Scales easily to versions where redundancy is not used.</td>
</tr>
<tr>
<td></td>
<td>• No additional costs</td>
</tr>
<tr>
<td>Forces against the decision</td>
<td></td>
</tr>
<tr>
<td></td>
<td>• Slower switch over time than the alternative would have.</td>
</tr>
<tr>
<td></td>
<td>• Hard to offer higher availability than the current 99.99%</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Outcome</th>
<th>Green</th>
<th>Yellow</th>
<th>Yellow</th>
<th>Red</th>
</tr>
</thead>
<tbody>
<tr>
<td>Rationale for outcome</td>
<td>Current solution seems to be ok.</td>
<td>I am concerned about the slow switch over time.</td>
<td>Widely accepted solution. Availability might become a problem in the future</td>
<td>We should really reconsider this decision, as the next release is likely to have higher availability requirements.</td>
</tr>
</tbody>
</table>
DCAR is …

• **Lightweight**: does not use time to argue about (quality) requirements, only technical people need to participate (can be done in half a day – requires about 20 company person hours)

• **Incremental**: evaluation is carried out decision-by-decision, evaluation can be limited to any part of a system -> can be integrated with agile

• **Trustworthy**: the analysis of the analyzed decisions remains valid, “testing” aspect is missing. Coverage is clear: the results hold exactly for the analyzed decisions, nothing else

+ produces full decision information as a side-effect, to be stored in an AKM repository

**Drawbacks**:

• Prioritization problem remains: decisions can get “wrong” prioritization

• Less support for maintenance guidance

• No requirements refinement, requirements are assumed to be ok

van Heesch et al., IEEE Software 2014 (to appear)
Conclusions

• There is still a lot of things we should understand better in architecture evaluation and in its relationships with the development process and AKM

• ATAM-like scenario-based evaluation can be used to elicit essential information about software architecture, to be stored in an AKM repository and to automate architecture documentation

• ATAM-like scenario-based evaluation has several problematic issues, but even with its weaknesses, ATAM-like evaluation seems to be useful to the practitioners, perhaps more as a communication protocol than a comprehensive test

• DCAR is a new decision-based architecture evaluation method that avoids some of the problems of scenario-based methods
Thanks!
References


