Tampereen yliopisto
Tampere University

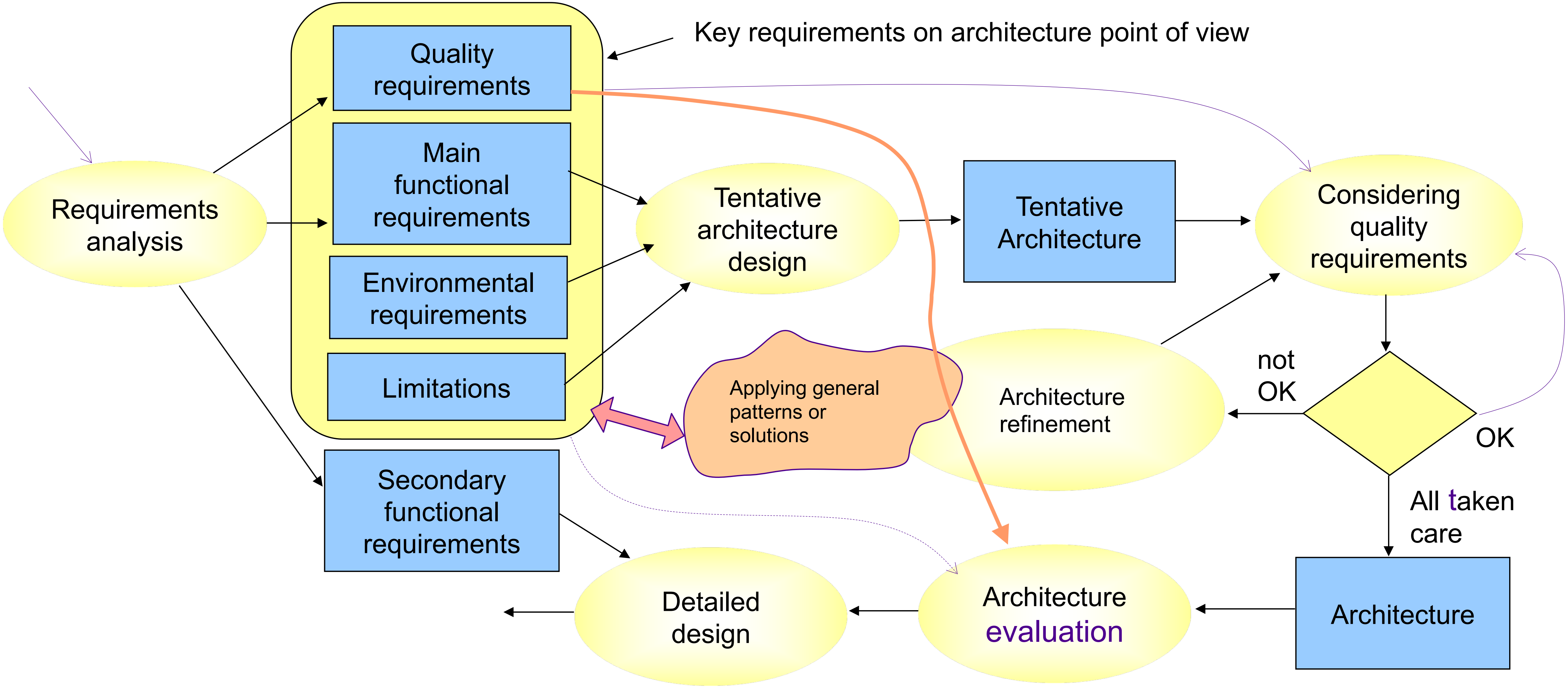
Large Scale Software Design
Architecture evaluation
ATAM, DCAR

Hannu-Matti Järvinen, David Hästbacka
Spring 2024

Evaluating software architectures

- Introduction
- ATAM method
- Example
- Practical experiences and problems
- DCAR
- Conclusions

Designing architecture during development process



Architecture and quality requirements

- Here, quality means the quality the system performs its logical functions, not correctness.
- Software architecture is a way to fulfil quality requirements of the system, i.e. architecture defines how quality requirements are fulfilled.
- Architecture description has to include all the information needed to decide if the quality requirement is met or not.
- Architecture is (normally) assessed against quality requirements.

What is evaluation of software architecture?

- Evaluation of a software architecture refers to an activity that can be used to draw conclusions about how well a particular software architecture supports the implementation of the requirements of the system in question.

Why software architectures need evaluation?

- Architecture is the first precise description of the system.
- The evaluation confirms good solutions and draws early attention to potential problems.
- The evaluation will help to better understand the system.

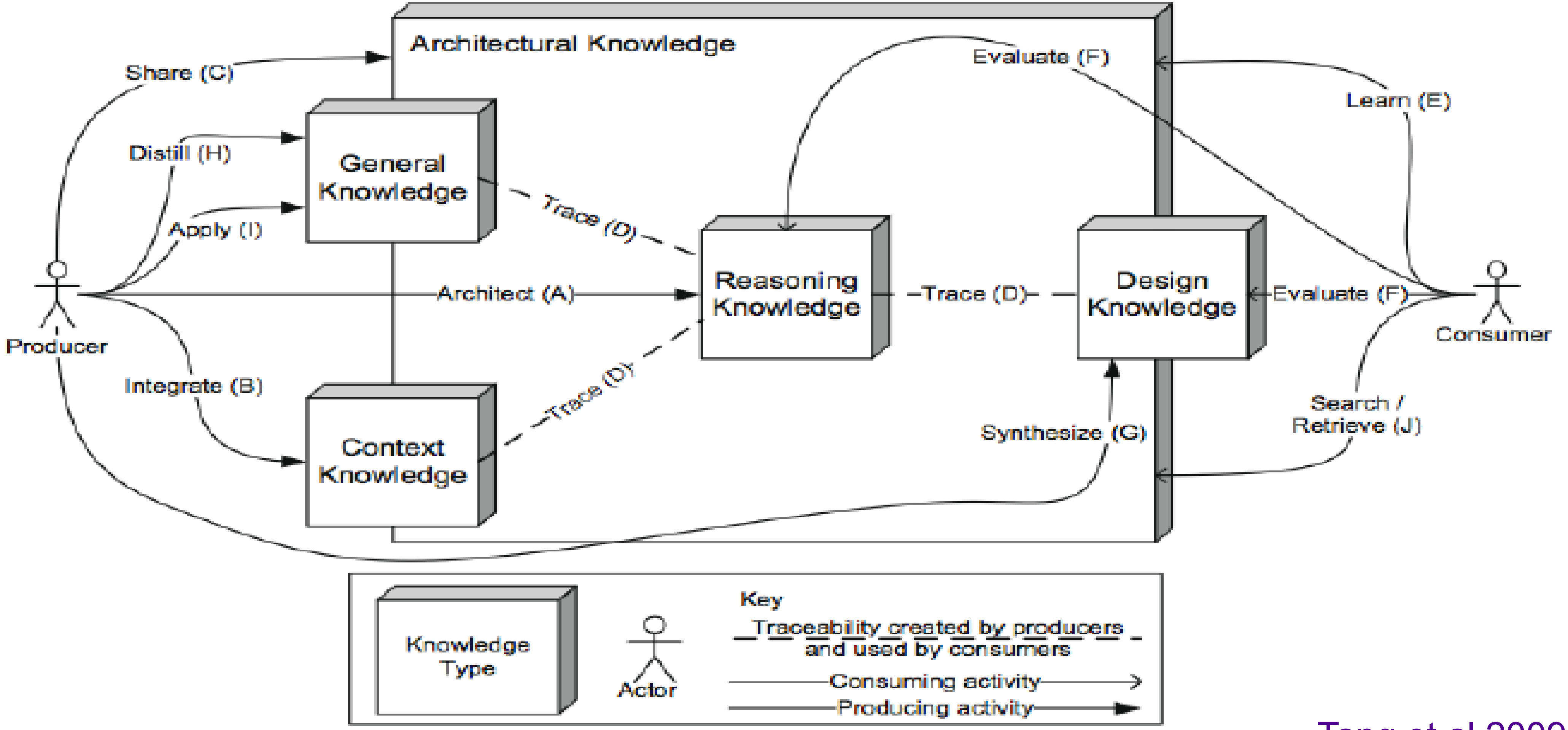
Other possible benefits

- Identification of development trends and potential development and the risk areas
- Software reform, identifying the main reform targets, and reviewing decisions.
- Opportunities to expand its operations into a new sector, the assessment of the necessary changes.
- The evaluation can be used to ensure the quality software made by others (eg. subcontracting).
- Recognition and refinement of quality requirements that direct the design.
- Recognition and documentation of architecture solutions and connecting them to the quality requirements.
- Improvement of architectural documentation
- Increasing communication

When to evaluate?

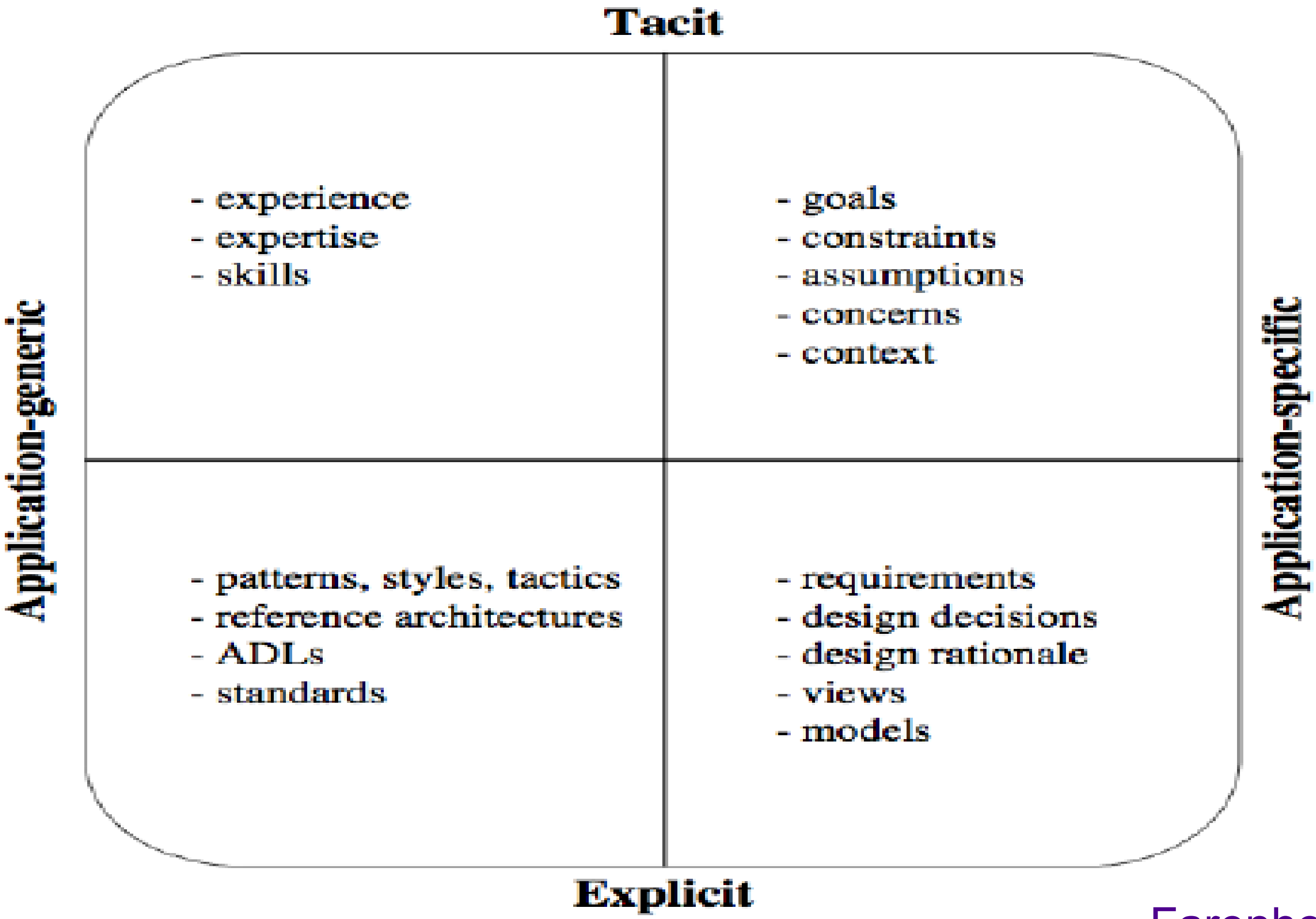
- On the basis of the first of (alternative) drafts (preliminary architectural document).
- After the architectural design, prior to the staring of implementation (system / subsystem architectural document).
- Existing system (eg. Renewing the old system)
  - Need for refactoring when problems are found

Architectural knowledge



Tang et al 2009

Categories of architectural knowledge



Farenhorst, de Boer 2009

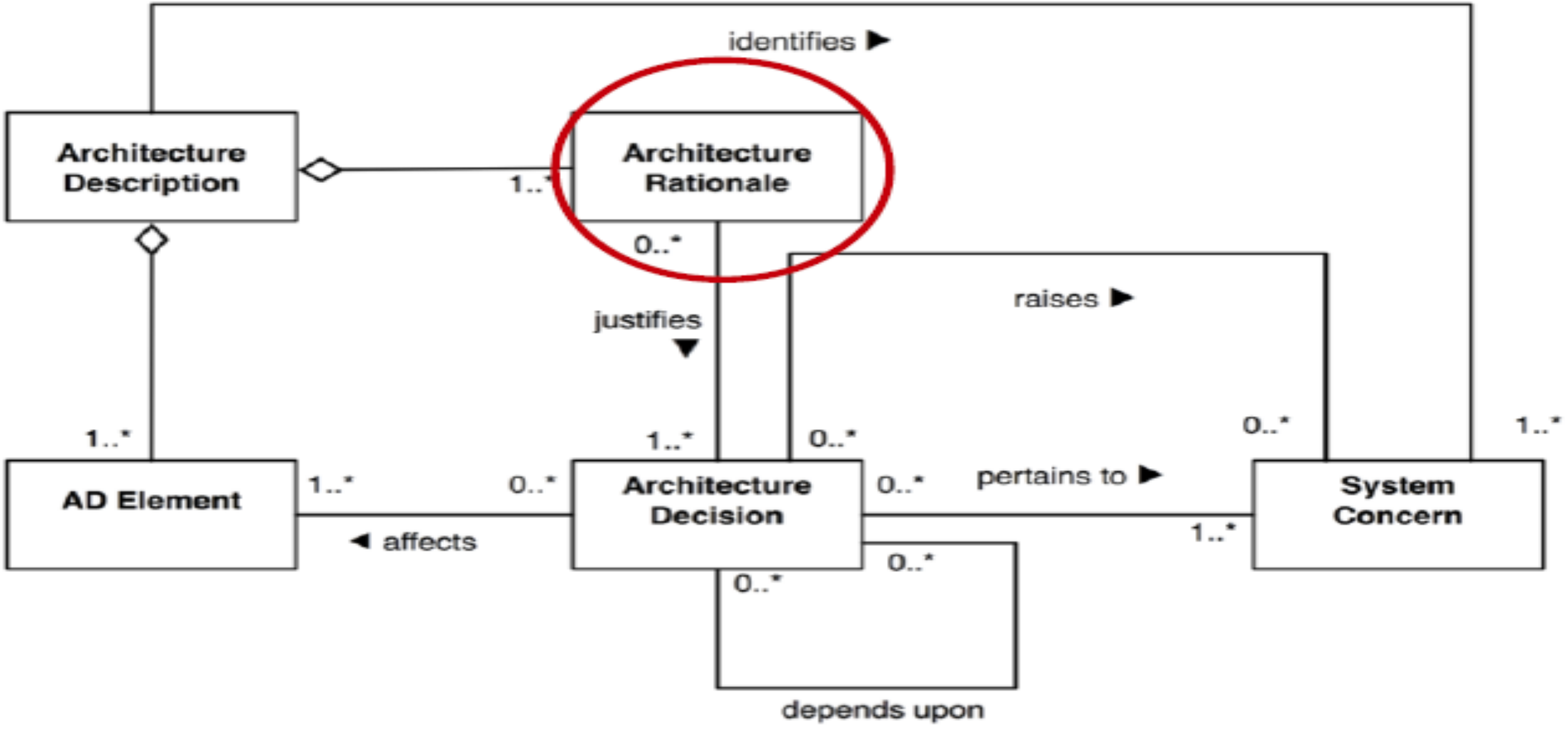
Problem

- Knowledge has feet. It can fall down from ladders or get better deal.

Tampereen yliopisto
Tampere University

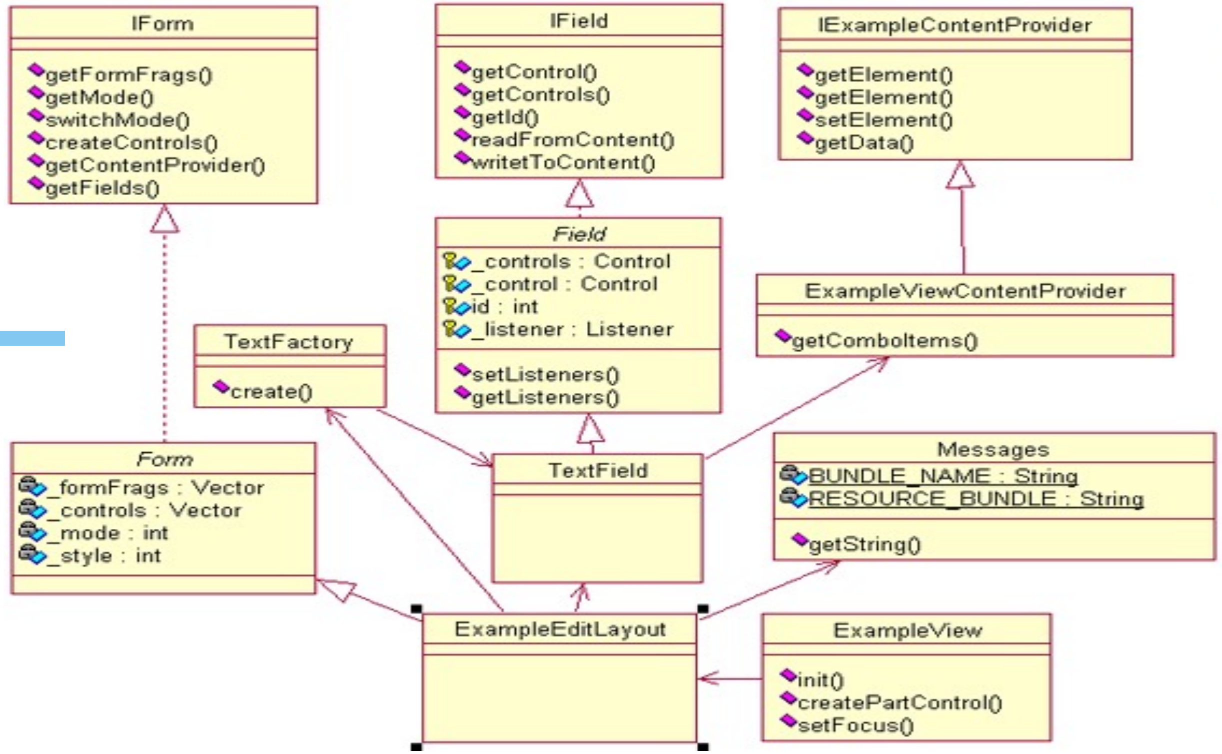
42010 Standard

42020 Standard (cont.)



Both should be documented

Rationale



Architectural decisions

- “Software architecture is the composition of a set of architectural design decisions” – ***Jansen & Bosch, 2005***
- “Architecting is making decisions. The life of a software architect is a long (and sometimes painful) succession of suboptimal decisions made partly in the dark” – ***Philippe Kruchten***

Documenting the decisions

- Decisions are documented when they are made
  - Rationale behind decisions available even after a long time
  - Increases the amount of documentation
  - May be too heavy
- Different kinds of documentation models

Tampereen yliopisto
Tampere University

Decision documentation example 1

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.

Tampereen yliopisto
Tampere University

Decision documentation example 2

Name				
Problem				
Solution / description of decision				
Considered alternative solutions				
Arguments in favour of decision				
Arguments against the decision				
Outcome				
Rationale for outcome				

Quality properties of software

- Run-time quality properties
  - Efficiency
  - Use of space
  - Reliability
  - Availability
  - Security
  - Usability
  - …
- Development and evolution time quality properties
  - Adaptability
  - Portability
  - Maintainability
  - Reusability
  - …
- Quality standards: e.g. ISO 25010

Detailed quality properties

Functional suitability
Functional completeness
Functional correctness
Functional appropriateness

Performance Efficiency
Time behaviour
Resource utilization
Capacity

Compatibility
Co-existence
Interoperability

Usability
Appropriateness recognisability
Learnability
Operability
User error protection
User interface aesthetics
Accessibility

Reliability
Maturity
Availability
Fault tolerance
Recoverability

Security
Confidentiality
Integrity
Non-repudiation
Accountability
Authenticity

Maintainability
Modularity
Reusability
Analysability
Modifiability
Testability

Portability
Adaptability
Installability
Replaceability

- The grouping does not have big significance in practise.
- The list is useful for selecting the assessment targets.

Architecture and business goals

Results of analysis

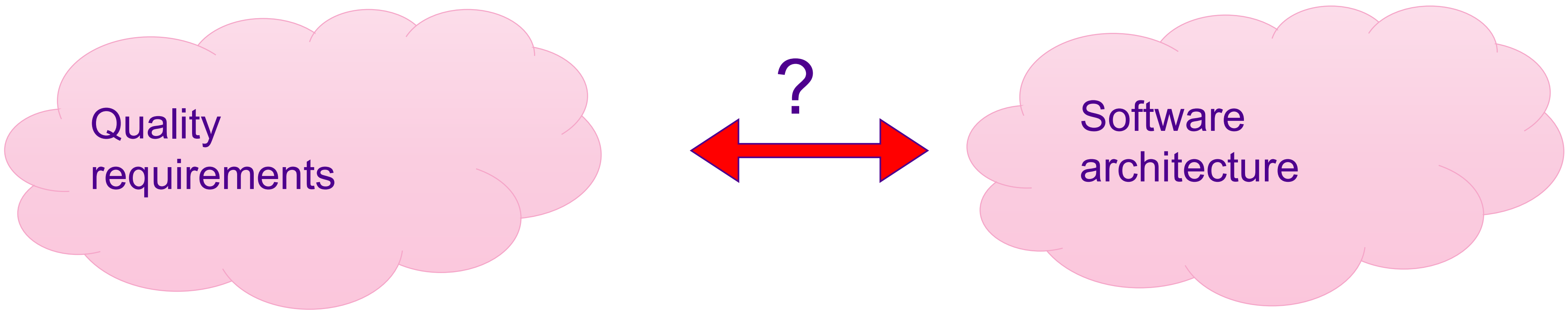
- Analysis of a software architecture answers typically to the following questions:

1. Does the designed architecture fulfil the essential quality requirements? If it does, why? If not, why?
2. Which of the alternative architecture solutions fits best for the system? Why?
3. How well can a given quality requirement be achieved by the designed architecture?

Notes

- Assessment is based on the description of the architecture, information available and activity of participants.
- The accuracy of the results depends on the accuracy of the given data.
- In assessment, sensible implementation has to be assumed, and the architecture must make sensible implementation possible.

The problem of software architecture analysis



Tampereen yliopisto
Tampere University

Quality requirements come from stakeholders

Efficient, reliable, good usability

Easy to maintain, portable

End user

Maintainer

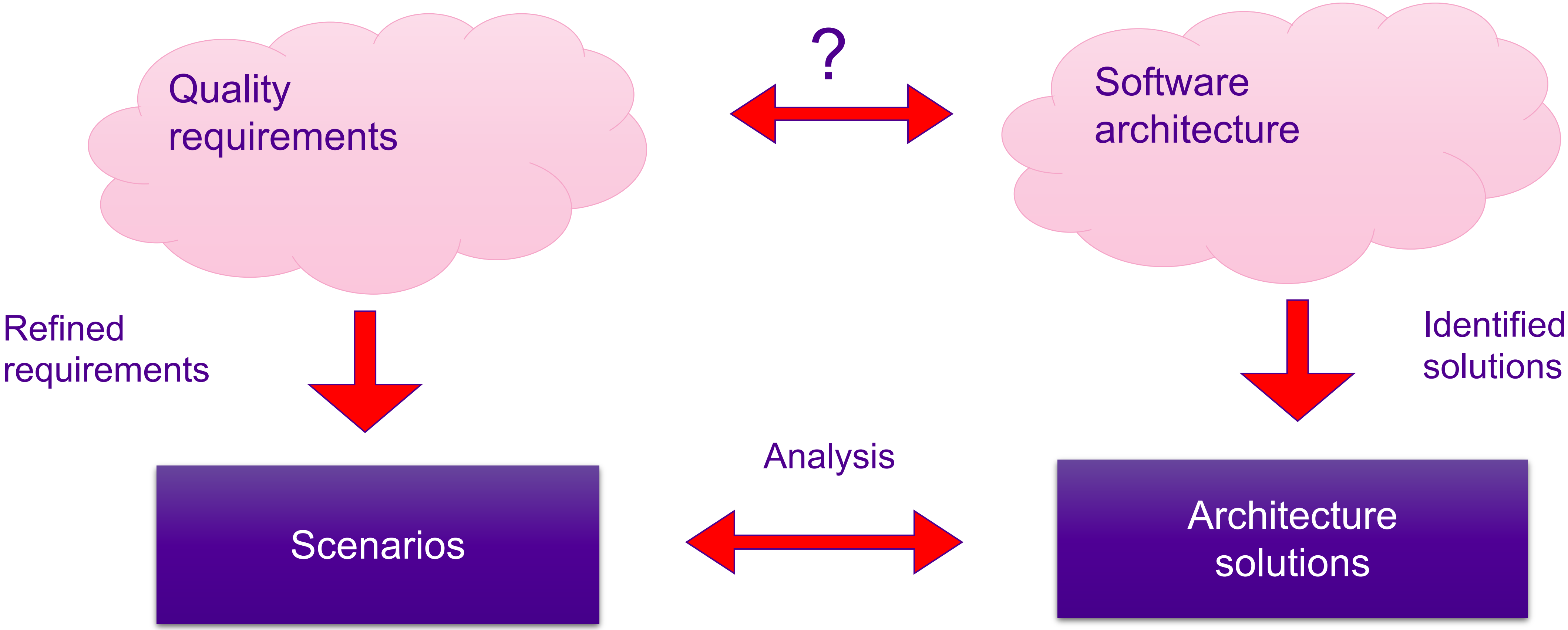
Assessment of quality properties

- There are no clear fulfilment criteria for quality properties.
- E.g. Maintainability: system change should be easy if its usage environment changes.
- How to assess a property if there are huge number of different kind of situations, where the property is potentially endangered?
- Compare correctness – testing.
- General method:
  - Define goals for the system, and derive the quality properties from them.
  - Refine the quality properties.
  - Give an example of each quality property
  - Examine, if the quality property is fulfilled in the example.

Refining quality requirements by scenarios

- Scenario = a situation or sequence of events that brings up if a quality requirement is fulfilled or not (on the view of a part of the system).
- Scenario makes the quality requirement concrete using example.
- Scenario has to be accurate enough to make assessment of the architecture possible – often precise numeric values.
- Compare traditional use case – functional requirement.
- Scenario = test case of the architecture.

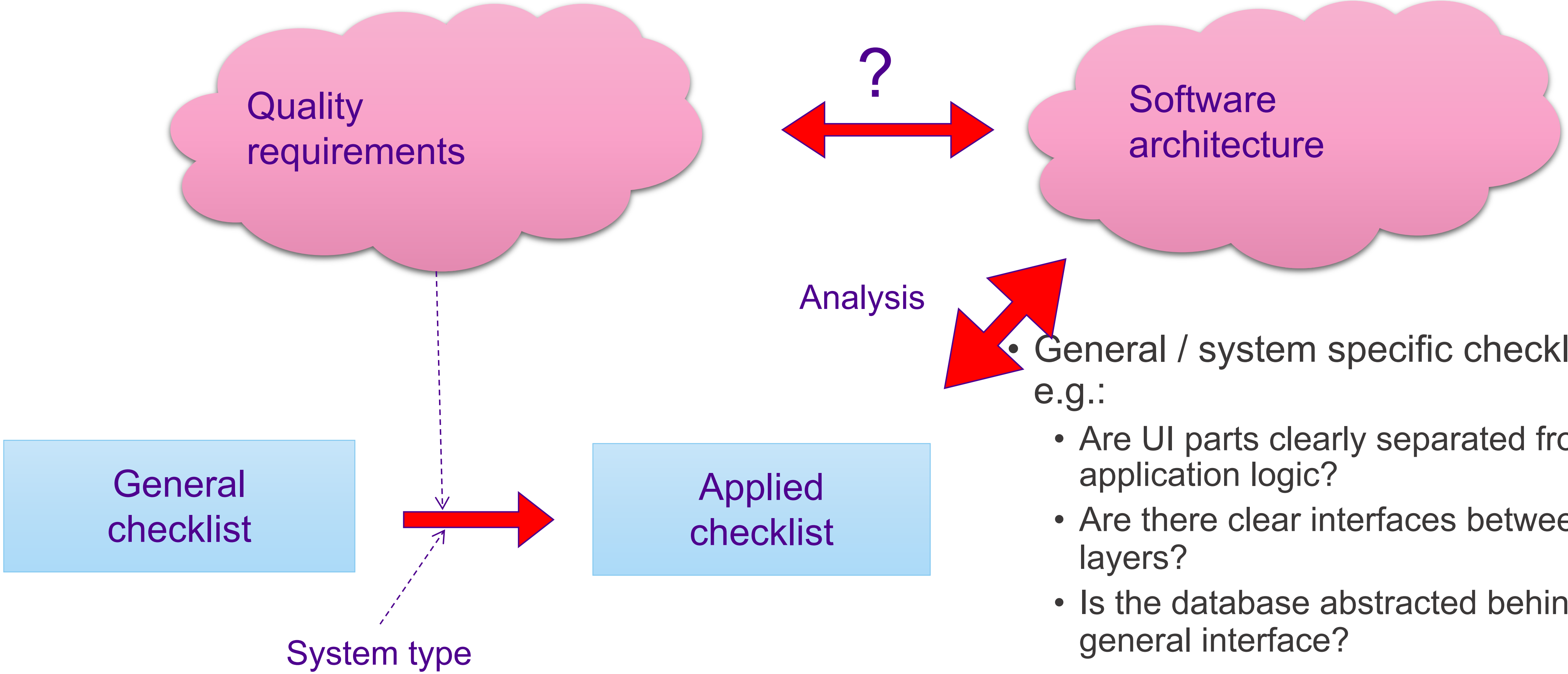
Solution for software architecture analysis: scenario-based assessment



Mining data out of architecture

- Experts' views
  - The main architect, architects that have designed similar systems, etc.
- Remodelling
  - The code can be abstracted by remodelling tool; this does not produce an actual architecture description but analyses different kinds of dependencies.
- Simulation
  - If there is an executable model, performance and reliability depending on the architecture can be examined; requires modelling of the system and a good tool.
- Metrics
  - Can be used as a rough tool to find out suspicious places (works mainly for maintainability)
  - Requires good tools.
  - E.g. Big classes, a lot of dependences between components.

Alternative way: checklist-based assessment

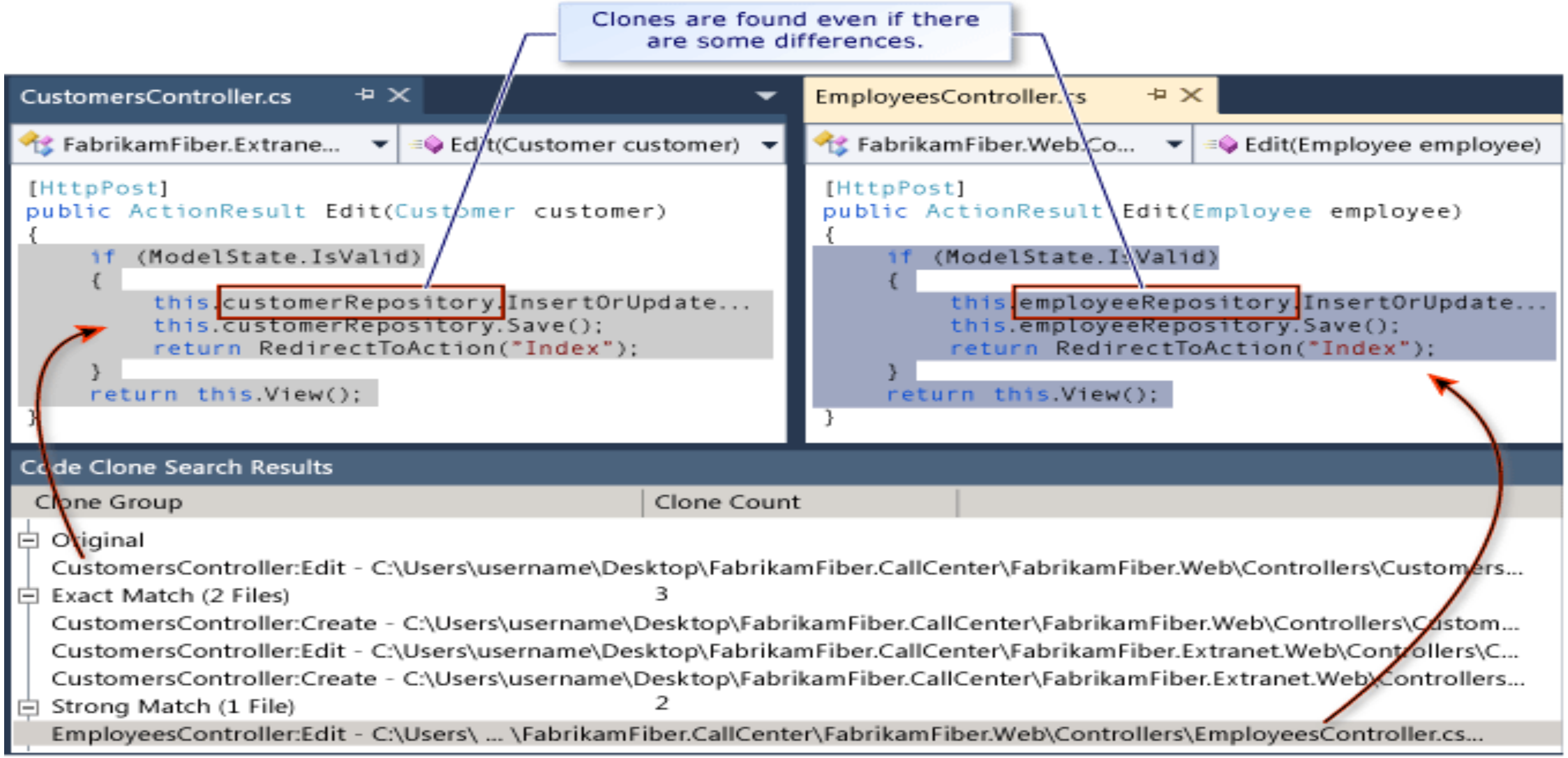


- General / system specific checklists, e.g.:
  - Are UI parts clearly separated from the application logic?
  - Are there clear interfaces between layers?
  - Is the database abstracted behind a general interface?

Utilising analysis tools

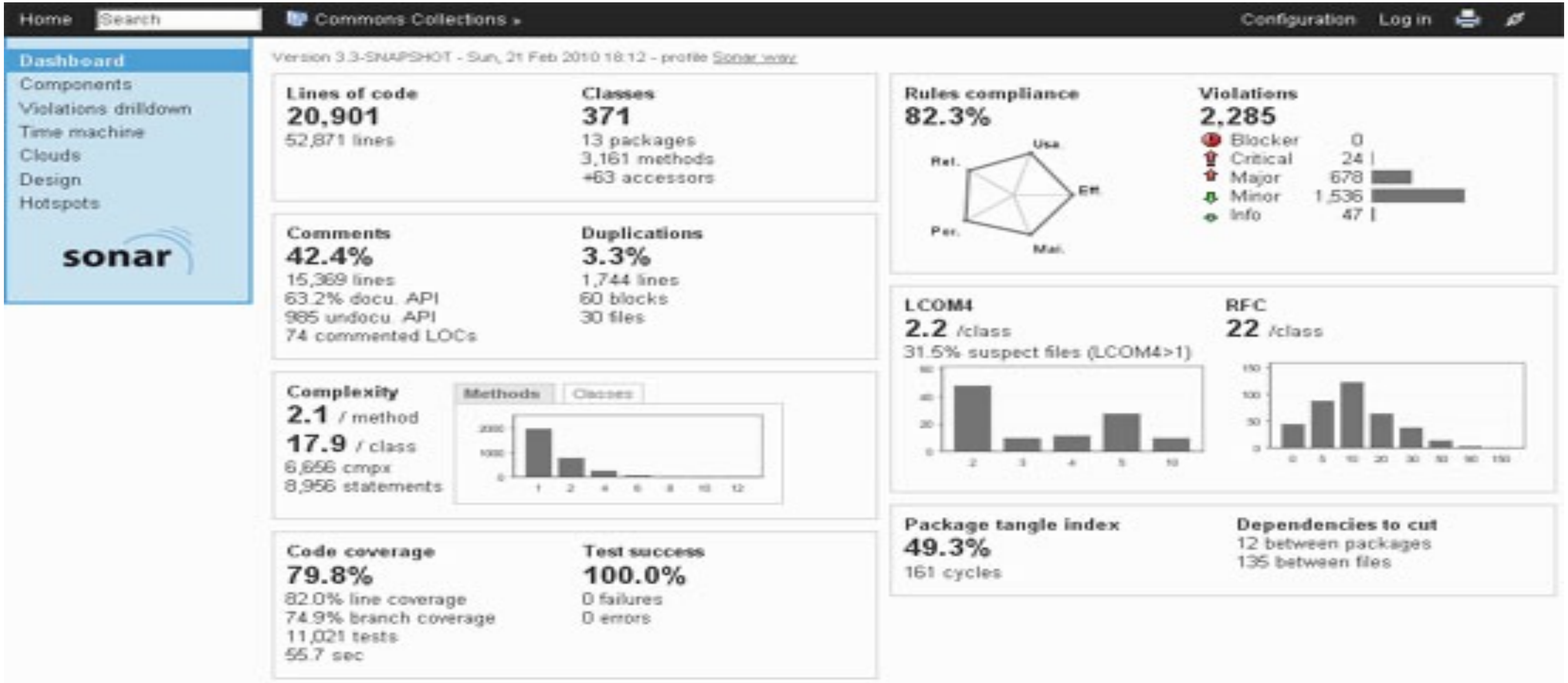
- For an existing architecture assessment, different kinds of tools can be used (e.g. metrics tools, rule-checking tools, visualisation tools, dependability analysts, analysts for copied code, remodelling tools).
- They are especially useful when analysing maintainability and adaptability.
- Many tools work on code (static analysis) -> might not produce architecture-level information.
- They can be utilised in scenario-based assessment e.g. retrieving and prioritising scenarios that target to ”suspicious” parts of the system.

Code copy, Visual Studio (code analysis tools)



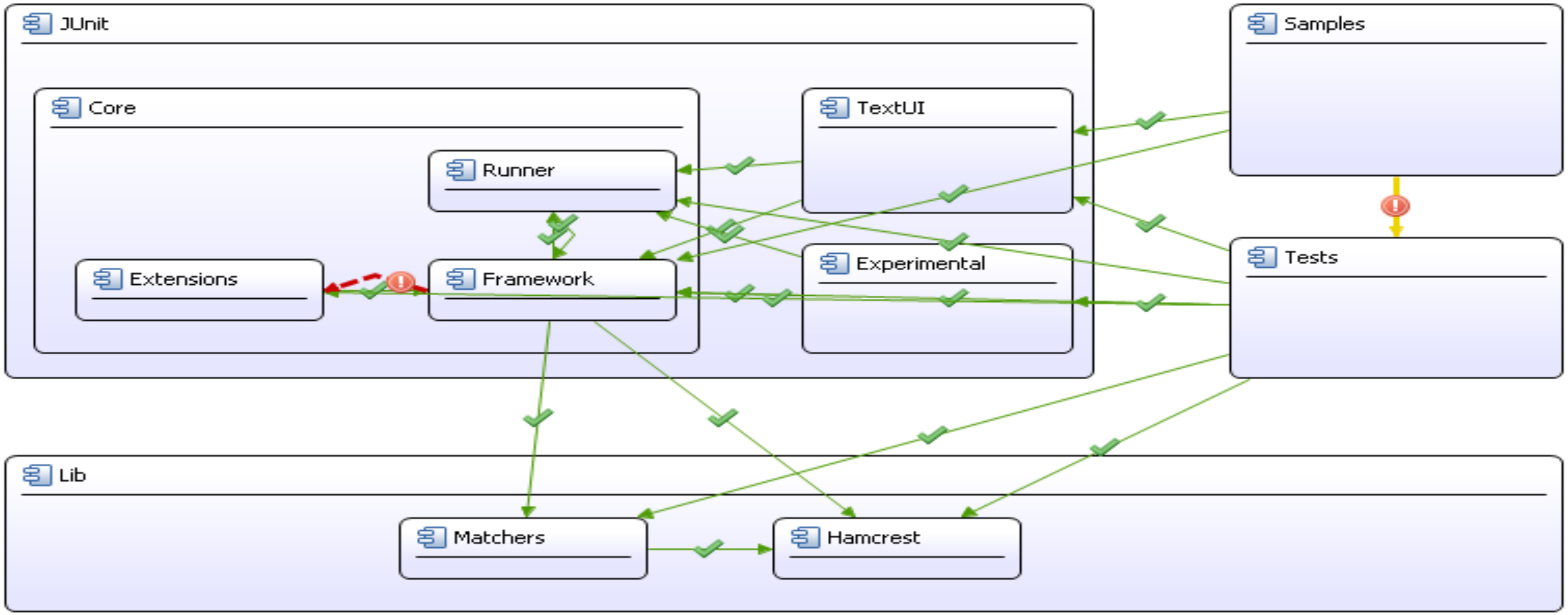
https://msdn.microsoft.com/en-us/library/hh205279.aspx

Code analysis...



http://www.sonarqube.org/

ConQAT ”architecture conformance analysis”



Scenario-based analysis methods

- SAAM (Software Architecture Analysis Method)
  - Concentrates especially to adaptability, portability and maintenance.
  - Developed at SEI (Software Engineering Institute, Carnegie-Mellon University)
  - Is based on evolution-time scenarios.
- ATAM (Architecture Trade-off Analysis Method)
  - Fits for all quality properties.
  - Developed at SEI.
  - Derived from SAAM.
- MPM (Maintenance Prediction Method)
  - Concentrates on maintainability.
  - Tries to find relatively accurate cost estimation for maintenance.
  - Developed by Jan Bosch
  - Is based on maintenance scenarios

ATAM

Architecture Tradeoff Analysis Method

ATAM data flow

http://www.sei.cmu.edu/architecture/tools/evaluate/atam.cfm

Basic concepts of ATAM

- **Scenario**: a test case of an architecture
- **Utility tree**: refining the quality requirements of the target systems towards scenarios.
- **Sensitivity point**: changes on this architecture decision may cause significant changes to and quality property.
- **Trade-off point**: architecture decision that affects several quality property in different directions
- **Risk**: architecture decision that may cause future problems from quality attribute's view
- **Non-risk**: architecture decision that may help fulfilling a quality property.

Tampereen yliopisto
Tampere University

Phases of ATAM (2 days)

0. Preparing
1. Present the ATAM
2. Present Business Drivers
3. Present Architecture
4. *Identify Architectural Approaches*
5. *Generate Quality Attribute Utility Tree*
6. *Analyse Architectural Approaches*
7. Brainstorm and Prioritise Scenarios
8. Analyse Architectural Approaches
9. Present Results

1. day (steps 1–6)

2. day (steps 7–9)

Tampereen yliopisto
Tampere University

Participants

- Stakeholders:
  - Architect
  - Administrator
  - Tester
  - Expert for standards
  - Security manager
  - Project manager
  - Product manager
  - Customer
  - End user
  - Application area expert
  - Maintenance
  - Marketing
  - Program developer
  - Hardware expert
  - Ancillary service manager

- 1. day
  - 3–5 persons. The architect and other persons that have been closely been involved in the application.
  - Evaluation group
- 2. Day
  - 5–10 persons. Representatives comprehensively from all stakeholders.
  - Evaluation group

ATAM process (day 1)

- Presenting ATAM
  - Phases of ATAM
  - Technologies of ATAM (scenarios, quality tree, etc.)
- Business view
  - Most important functionalities on user's point of view.
  - Business goals
  - Economical, political etc. restrictions.
- Presenting the architecture
  - Technical restrictions (operating system, software platforms, hardware, etc.)
  - External interfaces of the system.
  - Description of the architecture.

ATAM Day 1 continues

- Identify Architectural Approaches
  - The styles, patterns, and own solutions are identified and named.
  - It is explained how the given quality requirements are achieved by a the used approach.
- Generating the quality attribute utility tree and scenarios.
  - Quality requirements are refined by system-specific grouping
  - Each refined quality requirement is made concrete by a scenario.
  - Scenarios are prioritised by their importance and difficultness.
- Analysis of architectural approaches
  - Focus on the most important scenarios.
  - Question: Does this architecture make the scenario possible, and why?
  - Architecture is guilty until proved otherwise.
  - The intention is to find risks, safe approaches, sensitivity and trade-off points.

ATAM Day 2, Supplement

- Scenario brainstorm
  - All parties present scenarios from their points of views.
  - New scenarios are prioritised and added to the quality tree.
  - Old scenarios are confirmed.
- Re-analysis
  - The most important scenarios are checked against the architecture.
  - Identify possible new risks.

Scenarios and scenario styles

- Scenario makes quality requirement concrete using an example. Scenario is precise (test case, use case).
- Structure of a scenario: ***Stimulus*** **– environment – response**.
- Use case scenario: user's interaction with the system.
  - ***Remote user fetches database report using web interface*** **during the peak load and gets the report in 5s.**
- Evolving scenario: anticipating changes
  - ***New data server is added to the system*** **to decrease latency by 2.5s, the work is done in 1 person-week.**
- Explorative scenario: unexpected changes, loads, etc.
  - ***Half of the servers crash*** **during normal operating conditions; this does not affect the availability of the system.**
- Default environment: normal operating conditions.

Scenario example

3.8 Scenario 208

Scenario: Service person unplugs power supply of the group control, the system should shutdown gracefully and the backup controller should take control without loosing calls. This should be invisible to the user.
Quality Attribute: Availability
Environment: Service during normal operations
Stimulus: Group control is unplugged
Response: Backup controller takes the control without losing calls

Tampereen yliopisto
Tampere University

Analysed scenario (in assessment report)

- Typically 10–15 high-priority scenarios
- Architectural decisions relating to the scenario are identified and classified (e.g. T = trade-off point, R = risk, N = non-risk).
- Description: Architect’s report how the scenario is handled is documented.
- Argumentation: It is explained, how each decision is connected to the scenario.

3.8 Scenario 208

Scenario: Service person unplugs power supply of the group control, the system should shutdown gracefully and the backup controller should take control without loosing calls. This should be invisible to the user.
Quality Attribute: Availability
Environment: Service during normal operations
Stimulus: Group control is unplugged
Response: Backup controller takes the control without losing calls

#	Architectural Decisions	T	R	N
208.19	Synchronization in mode changes			N213
208.25	Lift owns its command			N214
208.7	Call owns its arguments			N215
208.29	Lift can act autonomously if group is down - lift owns the passenger			N216
208.37	Transaction based initialization			N217
208.30	Group controller can be duplicated	T201		

Description:
The master group controller dies. The commands which are sent to the lifts are owned by the lifts and stay there. The passenger does not see that the group goes
...
Argumentation:
208.19 Present when the new system is booting. Provides coherent state when starting the backup.
208.25 Makes possible to synchronize state to the backup controller and it is invisible to the customer, even during the change.

Example: utility tree

Quality

Quality properties	Refinements	Scenarios
Performance	Transaction handling	Handles 1000 service requests / s without user recognised delay. (H,M)
	Response time	User authentication < 1s. (H,M)
Adaptability	GUI changes	GUI to web-based in 1 month (M,H)
	Database changes	Database is changed to Oracle in 6 months (L,H)
Availability	Hardware failures	Reboot in 5min after broken disk on server. (L,H)
	Server crash	Reboot in 5min after authentication servers crashes. (M,M)
Security	Using credit cards	Credit card data are secure 99.999% (H,L)

Prioritising scenarios

- Usually two-part priority
  - How important (product manager, project manager)
  - How difficult to implement (architect)
- Three values: H (high), M (medium), L (low)
- Can be done by voting

Sensitivity point

- Sensitivity point = an architecture approach that is critical to reach a quality requirement
- Example: Using MVC style in GUI architecture is essential for portability of the system.

Trade-off point

- Trade-off point = sensitivity point that applies for several quality requirements (often in opposite ways).
- Example: Usage of XML as data format improves adaptability of the system but has negative effect on performance of the system.

Risk

- Risk = potentially problematic architecture approach that can weaken some quality property.
- Risk = approach/fact + quality ramification + argument
- Example: Criteria and rules to make middle layer components are unclear (approach or fact). This may cause replication of functionalities on different layers (argument), which weakens maintainability (quality ramification).

Tampereen yliopisto
Tampere University

Non-risk

- Non-risk = architecture approach that has (mostly) only good quality ramifications.
- Non-risk = assumption + approach + quality ramification + argument
- Example: Assuming that the components do not have to consider each other's space (assumption), the usage of the observer design pattern in the communication between the components (approach) improves the adaptability (quality ramification) because the components do not need to know about each other anything but recalls and registration interfaces (argument).

Reporting

- The most important results of ATAM:
  - Identifying the key architecture approaches.
  - Identifying the most essential use and development scenarios.
  - The quality attribute utility tree and scenarios: description of connection between quality requirements and architecture approaches.
  - Identifying the risks of the architecture.

Structure of report (example)

- 1. Introduction
- 2. Target System
  - 2.1 Description of the System
  - 2.2 Most Important Architectural Solutions
- 3. Analyzed Scenarios
  - 3.1 Maintainability
  - 3.2 Reliability
  - 3.3 Efficiency
  - 3.4 Usability
- 4. Analysis Overview
  - 4.1 General Observations
  - 4.2 Specific Issues
  - 4.3 About the Process
- 5. Conclusions
- References
- Appendix: Complete Scenario List

Scenarios in analysis report (example)

- Typically 10–15 high-priority scenarios
- Architectural decisions relating to the scenario are identified and classified (e.g. T = trade-off point, R = risk, N = non-risk).
- Description: Architect’s report how the scenario is handled is documented.
- Argumentation: It is explained, how each decision is connected to the scenario.

3.8 Scenario 208

Scenario: Service person unplugs power supply of the group control, the system should shutdown gracefully and the backup controller should take control without loosing calls. This should be invisible to the user.
Quality Attribute: Availability
Environment: Service during normal operations
Stimulus: Group control is unplugged
Response: Backup controller takes the control without losing calls

#	Architectural Decisions	T	R	N
208.19	Synchronization in mode changes			N213
208.25	Lift owns its command			N214
208.7	Call owns its arguments			N215
208.29	Lift can act autonomously if group is down - lift owns the passenger			N216
208.37	Transaction based initialization			N217
208.30	Group controller can be duplicated	T201		

Description:
The master group controller dies. The commands which are sent to the lifts are owned by the lifts and stay there. The passenger does not see that the group goes
…

Argumentation:
208.19 Present when the new system is booting. Provides coherent state when starting the backup.
208.25 Makes possible to synchronize state to the backup controller and it is invisible to the customer, even during the change.

Potential problems in ATAM / in similar methods?

- Big question: are the scenarios really sensible or useful, can the essential scenarios be selected (forecasting).
- Found risks vs. hidden ones
- Prioritising: are the right scenarios selected?
- ”Definite” benefit: collect together all stakeholders of the software.
  - Silent knowledge can be documented
  - A general understanding of the system is obtained
  - Worries and problems of different stakeholders are got out, and possibly get resources to take care of some most critical aspects.

Conclusions

- Finding architectural decisions and documenting them.
- Connecting quality properties to architecture approaches.
- Scenario-based, handling of scenarios
- ATAM:
  - http://www.sei.cmu.edu/architecture/tools/evaluate/atam.cfm
  - http://www.sei.cmu.edu/reports/00tr004.pdf

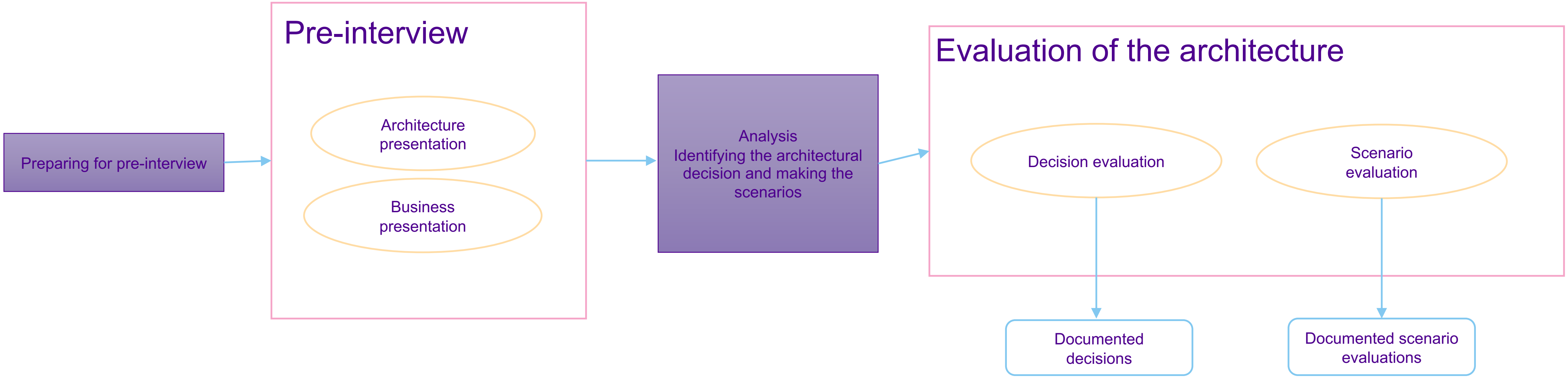
DCAR

Decision-centric architecture review method

Evaluating architectures

- Companies and ATAM is almost impossible to combine, only few companies can afford 6–10 persons for two-day evaluation.
- Members of the evaluation group make more preparations.

General overview of the process



DCAR

- Developed at TUT in co-operation with University of Groningen (RUG)
- TUT had experiences on ATAM evaluations.
- Groningen had experiences on management of architecture knowledge and documenting the decisions.
- Test evaluations in Finnish software companies and TUT.
- www.dcar-evaluation.com

Goals of DCAR

- Light and agile
- Incremental and iterative
- A broader consideration of the problem space.
  - Not only the quality properties, but other things, too
- The evaluation coverage is somehow estimable.
- Maintaining the strengths of ATAM method (increased communication, improved documentation etc.)

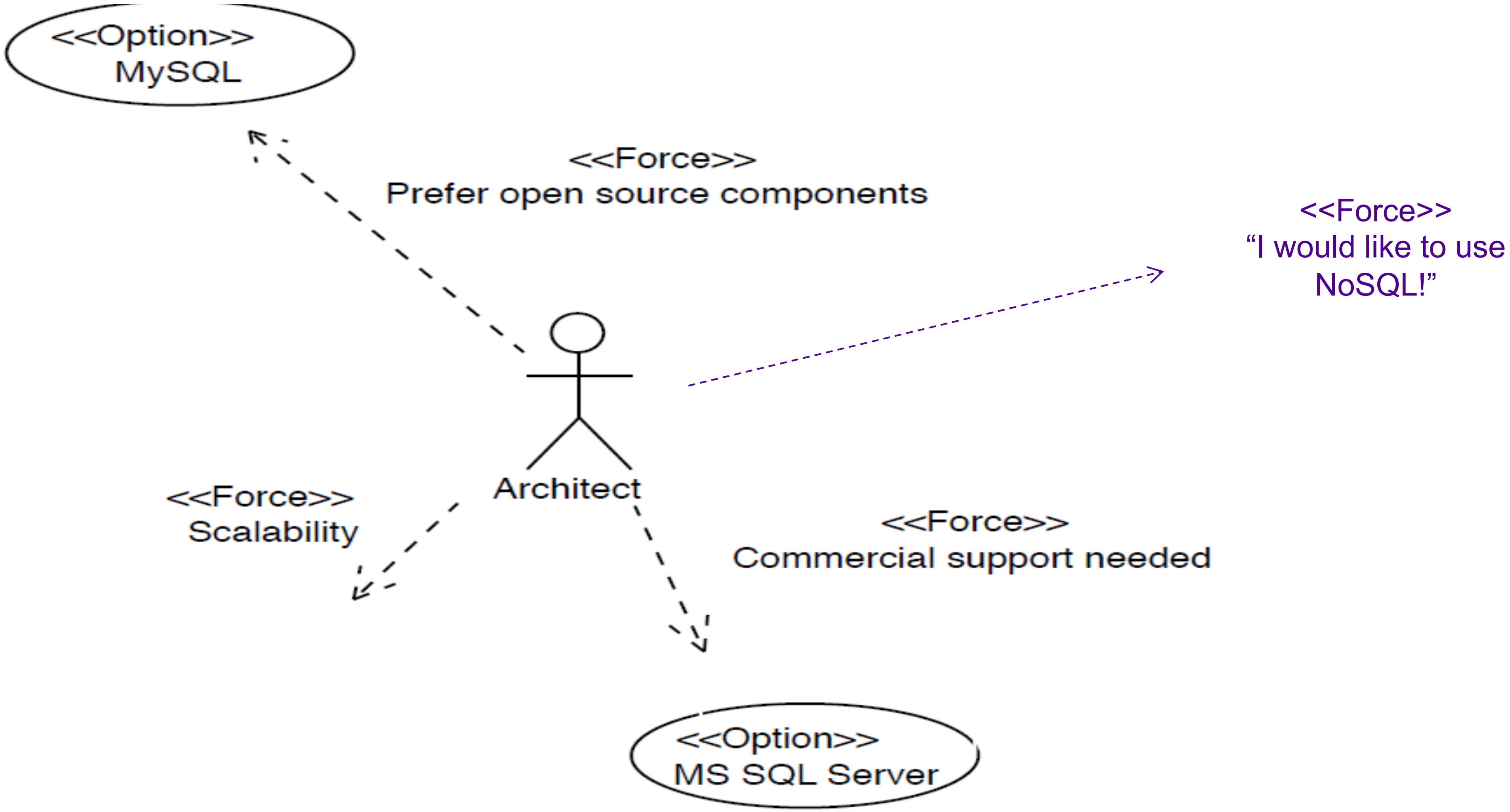
Participants

- Representatives of the company / project
  - Architect of the system
  - Project manager, product manager, …
  - Application developers
  - Experts of the application area
- Evaluation group
  - Head of evaluation
  - 2 scribes
    - Decision scribe
    - Forces scribe
  - Concern raisers

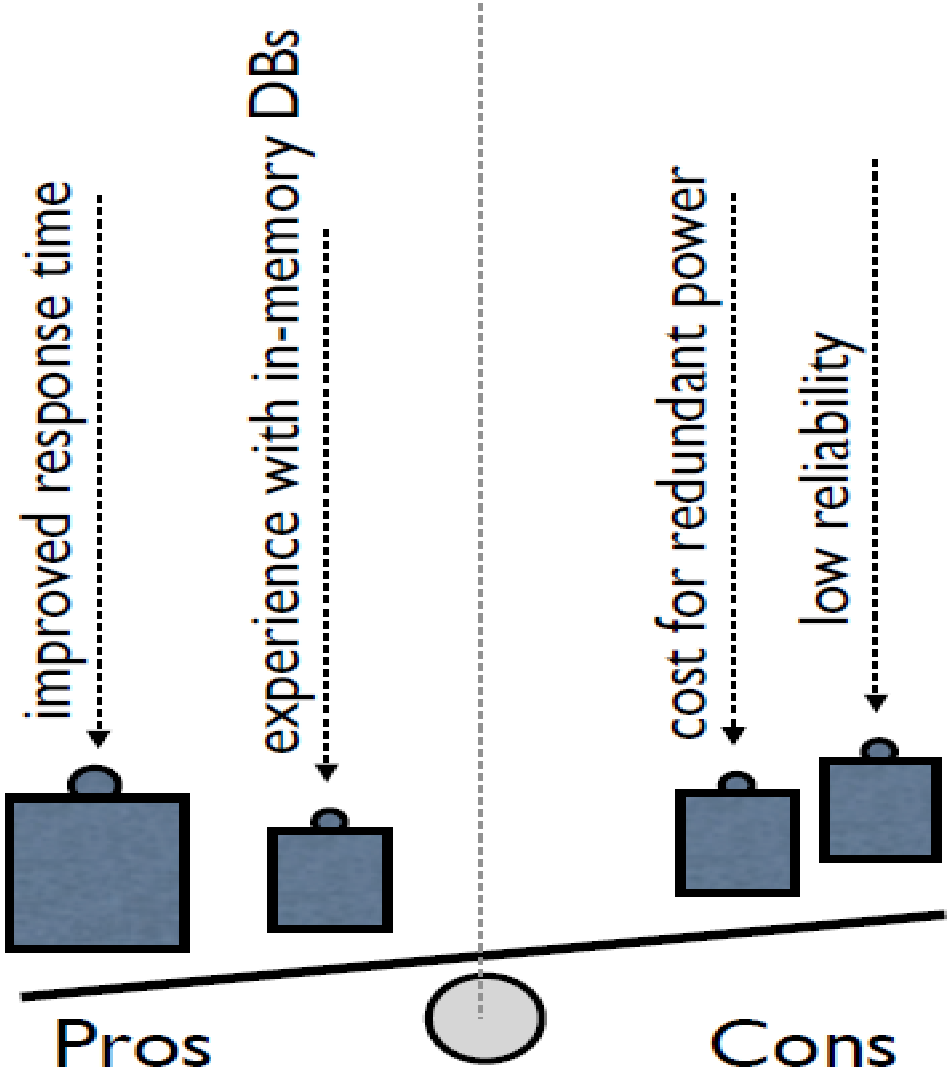
Mythical ”force” concept

- Different kinds of things affect on architectural decisions.
- Design is directed by quality requirements like performance, adaptability.
- Several limiting conditions affect on the decision: costs, time pressure, subcontracts, etc.
- Part of the affecting things are “tacit knowledge”, e.g. architect’s opinion, knowledge of the application area.
- All above are part of architectural knowledge, and should hence be documented.

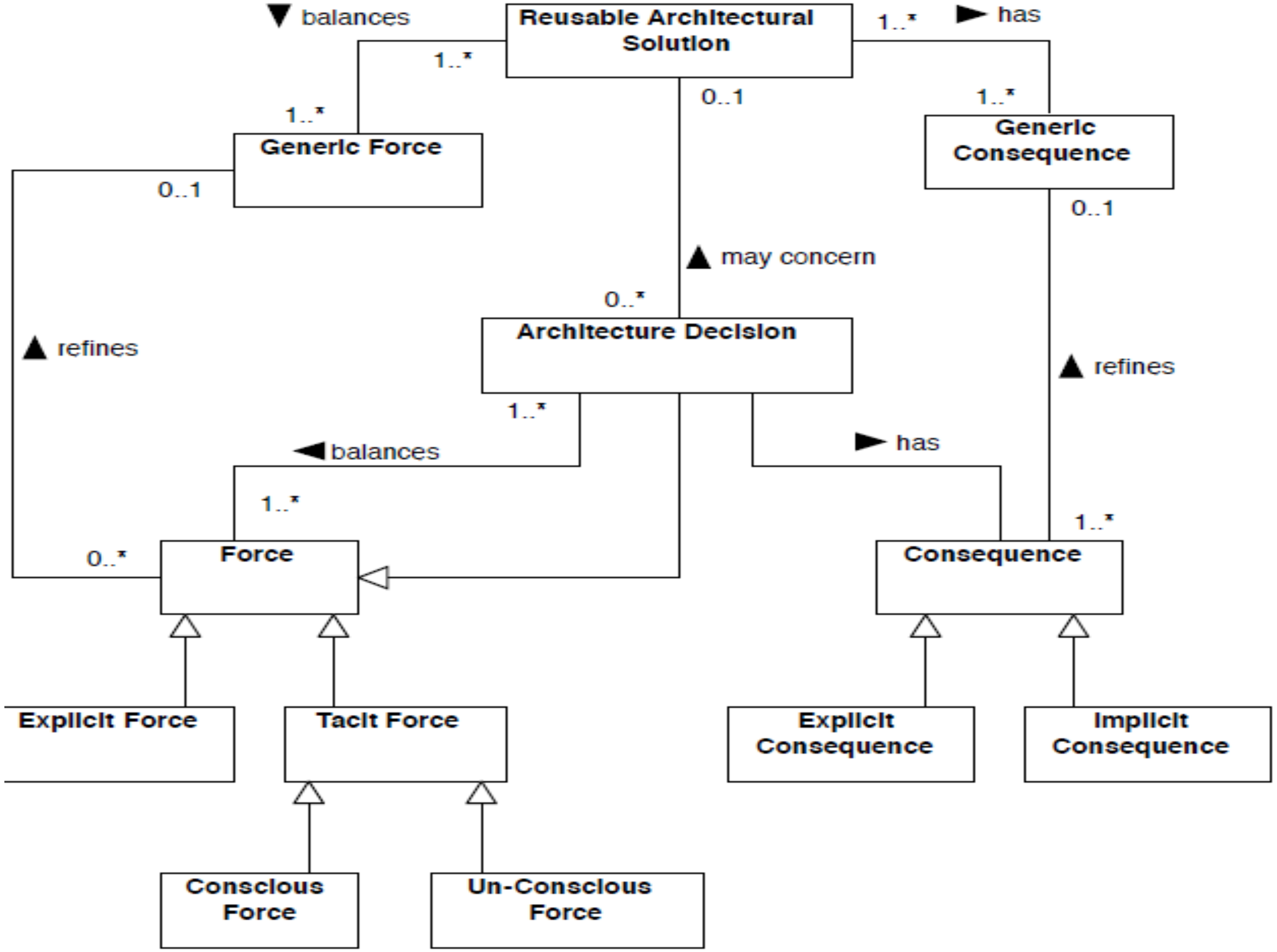
Force example



Forces (continues)



Meta model of forces



Force types: Explicit

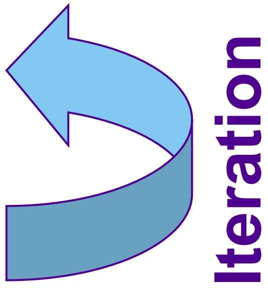
- Requirements
- Existing constraints
- Constraints for future decisions
- Technical risks
- General software engineering principles (e.g. high cohesion, low coupling)
- other decisions
- business goals (low price, quick time2market, innovation, …)
- business model
- business constraints (available licenses,…)
- company politics

Force types: tacit

- organization culture
- organization structure
- other decisions
- experience
- expertise
- intuition and bias
- the software development process
- impediments
- laws/regulations
- politics
- time pressure
- historical decisions

Phases of DCAR

1. Preparing the evaluation
2. Presenting DCAR method
3. Presentation of the application and business goals
4. Presenting the architecture
5. Reviewing and prioritising the decisions
6. Documenting the decisions
7. Analysing the decisions
8. Retrospection and reporting the results



Phase 1: Preparation

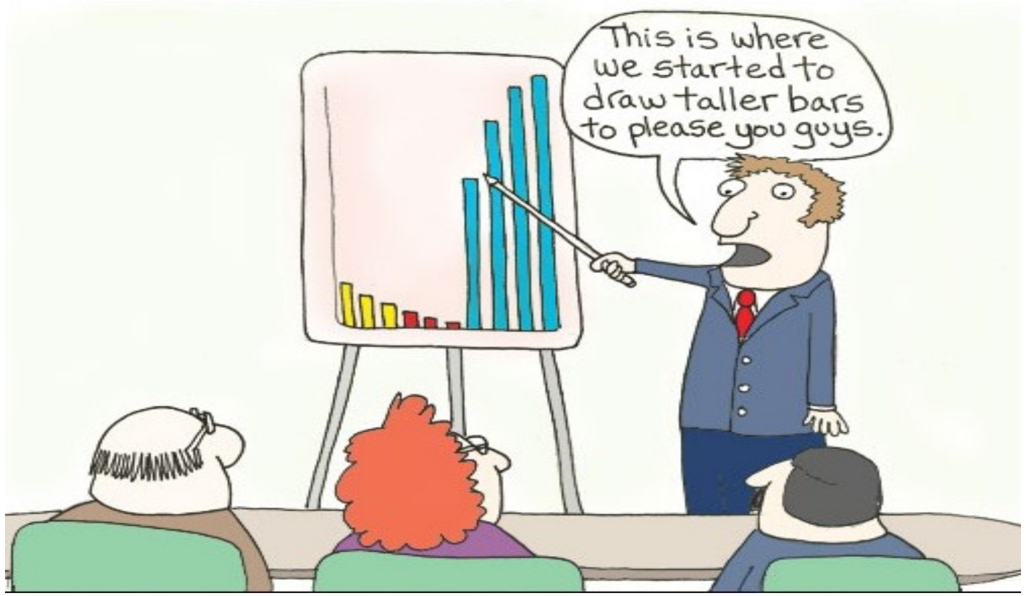
- Evaluators agree the evaluation day and location with the company
- Delimiting the evaluation (which system or which parts of the system are evaluated)
- The evaluation target? What is done with the results, which are the interesting points?
- Who will hold the presentations?
- The current state for the architectural documentations? Is more needed?
- Evaluators read the existing architectural documentation.
- Inspecting the presentations in advance.

Phase 2: DCAR presentation

- DCAR presentation in 15 minutes.
- The most important phases are repeated just before the execution phase.
- Presentation material of DCAR is given to the participants in advance so they can ask questions of unclear issues.

Tampereen yliopisto
Tampere University

Phase 3: Presentation of business goals



- The product owner or manager presents business goals and application area in 15 minutes.
- Evaluators intend to recognise ***forces*** that have affected the decisions (either consciously or unconsciously)

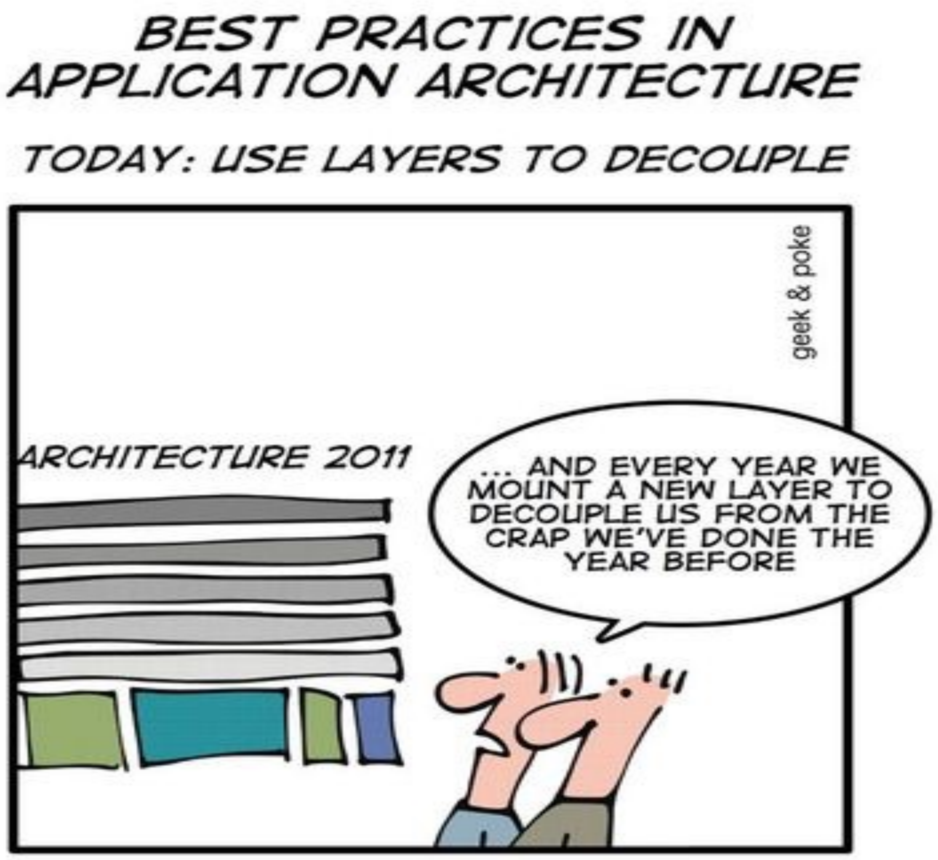
Task 1: Recognising the forces

- Listen the presentation and recognise the forces.

Tampereen yliopisto
Tampere University

Phase 4: Presenting the architecture

- 45 min presentation about the architecture by the architect
- Evaluators makes questions on (some) details
- Evaluators try to recognise architectural decisions.
- Forces are still collected.
- Evaluators make decision relationship view describing the relations between architectural decisions.



Task 2: Architectural decisions

- Recognise architectural decisions from the system and list them.

Phase 5: Review and prioritisation of decisions

- Evaluators show preliminary recognised architectural decisions for other participants.
- The list will be updated with participants
- The names of decisions will be specified, if needed.
- Prioritising decisions in two stages
  - 1. Stage: Initially, each stakeholder lists 3–5 decisions they want to be evaluated.
  - This creates a “short list”. Decisions that did not get votes, are not evaluated.
  - 2. Stage: Each stakeholder has 100 points to be shared to the short list decisions as they see appropriate.
  - Decision which has the most points is evaluated first.

Phase 6: Documenting the decisions

- Each stakeholder selects 1 – 3 decisions that are familiar to him and documents them.
- Evaluators help stakeholders to document the decisions.
- Generally, the evaluators give stakeholders an example decision to act as a model documentation.
- A part of the evaluation team finalises the graph illustrating the relations between decisions and unite their force-lists.

Documenting the decisions

Name	Redundancy of the controllers			
Problem	The application should run even if one of the redundant servers fail.			
Solution / description of decision	Solution goes here…. <Solution removed for confidentiality reasons>			
Considered alternative solutions	Both redundant server members could be active….			
Arguments in favour of decision	• Easier to implement •….			
Arguments against the decision	• Slower switchover • No possibility to offer more availability than current 99.99 % • ….			
Outcome				
Rationale for outcome				

Phase 7: Analysis of decisions

- Stakeholder documenting a decision will present it to the other members.
- Other stakeholders may ask questions and suggest refinements to the documentation.
- *Scribe* will write the refinement to the decision as needed.
- Evaluators make questions about the decision and try to find new arguments in favour of and against the decision (these are also included in the documentation).
- Evaluators use forces-list to invent new questions.
- After 10–15 minutes, the analysis and discusses is ended (the *police* will take care of timing). After that, the stakeholders vote on whether the decision is still valid or whether there are new facts that cause pondering of the decision again.
- Stakeholders vote simultaneously with their thumbs (thumb up, down or indifferent).

Phase 8: Retrospect and reporting

- Finally, there is a brief discussion on how the evaluation was going. How the operation of evaluators can be improved, etc.
- The result will be reported in writing as soon as possible.

End result – analysed decision

Name	Redundancy of the controllers			
Problem	The application should run even if one of the redundant servers fail.			
Solution / description of decision	Solution goes here…. Removed for confidentiality reasons..			
Considered alternative solutions	Both redundant server members could be active….			
Arguments in favour of decision	• Easier to implement • ….			
Arguments against the decision	• Slower switchover • No possibility to offer more availability than current 99.99 % • ….			
Outcome	Yellow	Yellow	Red	Green
Rationale for outcome	Rationale why yellow goes here..			

Force table

	Decision 1	Decision 2	Dec. 3	Dec. 4
long product lifecycles	-	-	-	+
Long update cycle in industry e.g. 10 years		-		+
connections to other systems needed (ERP, MES, CMMS)	+			+
Wireless devices & operator software	+		-	
Windows OS in all systems			++	

Evaluation report

Glossary
1. Introduction
1.1. Purpose and Scope
1.2. Review participants
1.2.1. Stakeholders
1.2.2. Review team
1.3. DCAR process description
1.4. Realization of the process
2. System overview
3. Architectural decisions
3.1. Decision relationship view
3.2. Prioritization of decisions
3.3. Detailed decision documentation
3.4. Traceability matrix for decisions forces and decisions
4. Potential risks, issues and indicators for technical debt
4.1. Risk X in detail
4.2. Risk Y in detail
5. Conclusions
References

Pros of DCAR

- Visibility, makes wrong decisions visible.
- Lightweight (tales 4 hours + lunch)
  - Even faster, if decisions are documented in advance
- Allows incremental work
- No waste
- End results directly utilised as part of architectural documentation.
- In addition, the benefits of ATAM

Cons of DCAR

- Architectural decision as concept is new in companies => not used widely.
- Examines the current state of the system. If the evaluators are not careful, some expected changes may be go unnoticed.
- Requires experienced evaluators.

Example schedule

- 09:45 - 10:00 Opening words, coffee
- 10:00 - 10:15 Presentation of DCAR method
- 10:15 - 10:30 Business presentation
- 10:30 - 11:15 Architecture presentation
- 11:15 - 11:30 Break
- 11:30 - 12:00 Decision overview & prioritization
- 12:00 - 12:45 Lunch
- 12:45 - 13:15 Decision documentation
- 13:15 - 14:00 Decision evaluation
- 14:00 - 14:15 Break
- 14:15 - 15:00 Decision evaluation
- 15:00 - 15:15 Feedback & retrospective