Chapter 15: Architectures in Agile Projects

Agile Software Development

- Agility is the ability to both create and respond to change in order to profit in a turbulent business environment.

- Agile software development is an approach to software development that is people oriented, that enables people to respond effectively to change, and that results in the creation of working systems that meets the needs of its stakeholders.

History of Agile

- **Incremental** software development has been around since 1957.
- In 1974, E. A. Edmonds wrote a paper that introduced an **adaptive** software development process.
- The evolution of agile software development in the mid 1990s was a reaction to more heavyweight, document-driven methods (waterfall & RUP basically).
- In February 2001, 17 developers met in Snowbird, Utah, to discuss lightweight software development and published the Agile Manifesto. This signaled industry acceptance of agile philosophy.

Agile Manifesto

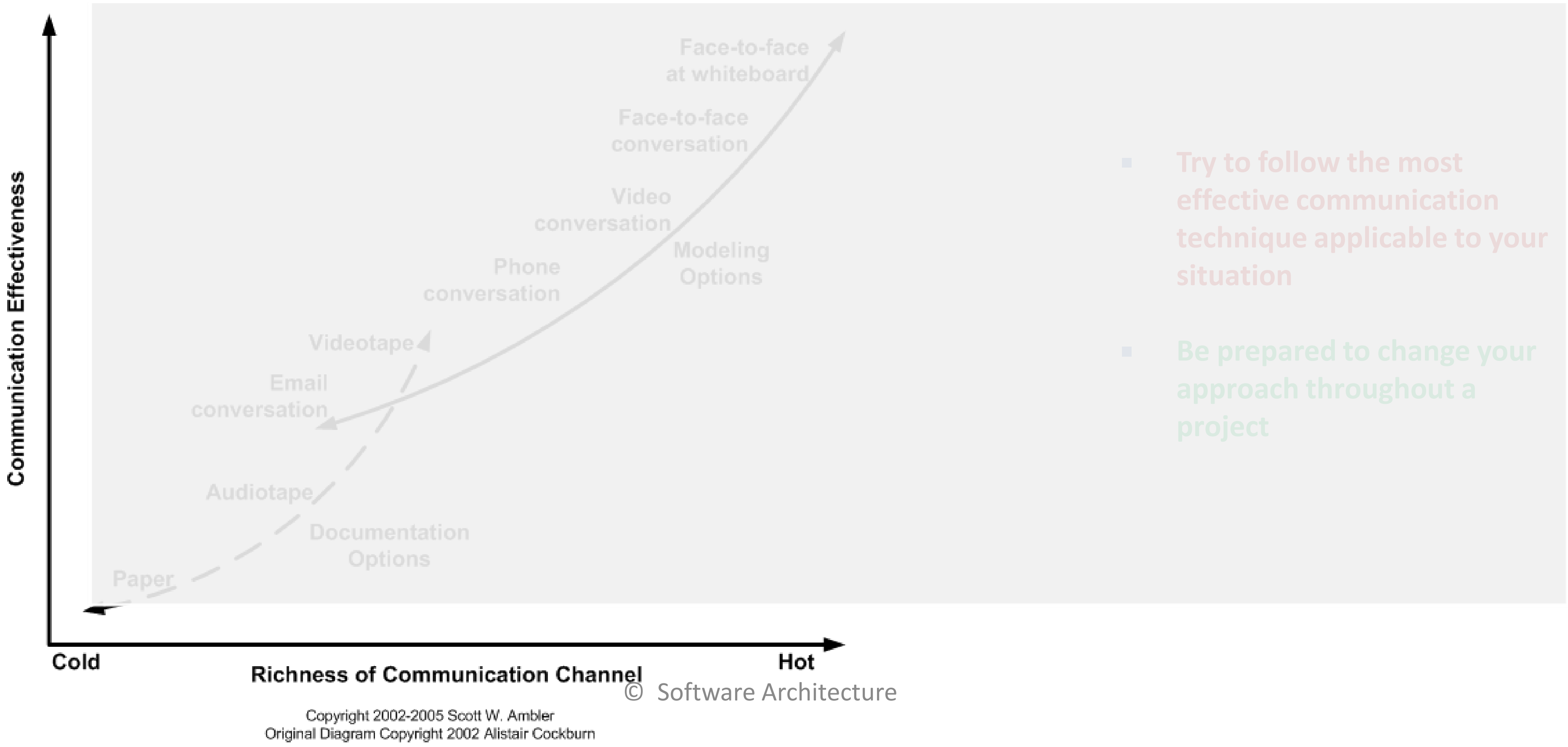
- **Individuals and interactions** over processes and tools
- **Working software** over comprehensive documentation
- **Customer collaboration** over contract negotiation
- **Responding to change** over following a plan

Twelve Agile Principles

1. Our highest priority is to satisfy the customer through early and continuous delivery of valuable software.
2. Welcome changing requirements **even late** in development. Agile processes harness change for the **customer's competitive advantage**.
3. Deliver working software frequently, from a couple of weeks to a couple of months, with a preference to the shorter timescale.
4. Business people and developers must work together daily throughout the project.
5. Build projects around *motivated individuals*. Give them the environment and support they need, and **trust them to get the job done**

Twelve Agile Principles

6. The most efficient and effective method of conveying information to and within a development team is face-to-face conversation.



- Try to follow the most effective communication technique applicable to your situation
- Be prepared to change your approach throughout a project

Twelve Agile Principles

6. The most efficient and effective method of conveying information to and within a development team is face-to-face conversation.

7. Working software is the primary measure of progress.

8. Agile processes **promote sustainable development.** The sponsors, developers, and users should be able to maintain a **constant pace** indefinitely.

9. Continuous attention to technical excellence and good design enhances Agility.

10. Simplicity–the art of maximizing the amount of work not done – is essential.

11. The best architectures, requirements, and designs emerge from self-organizing teams.

12. At regular intervals, the team reflects on how to become more effective, then tunes and adjusts its behavior accordingly.

Who should / should not use Agile?

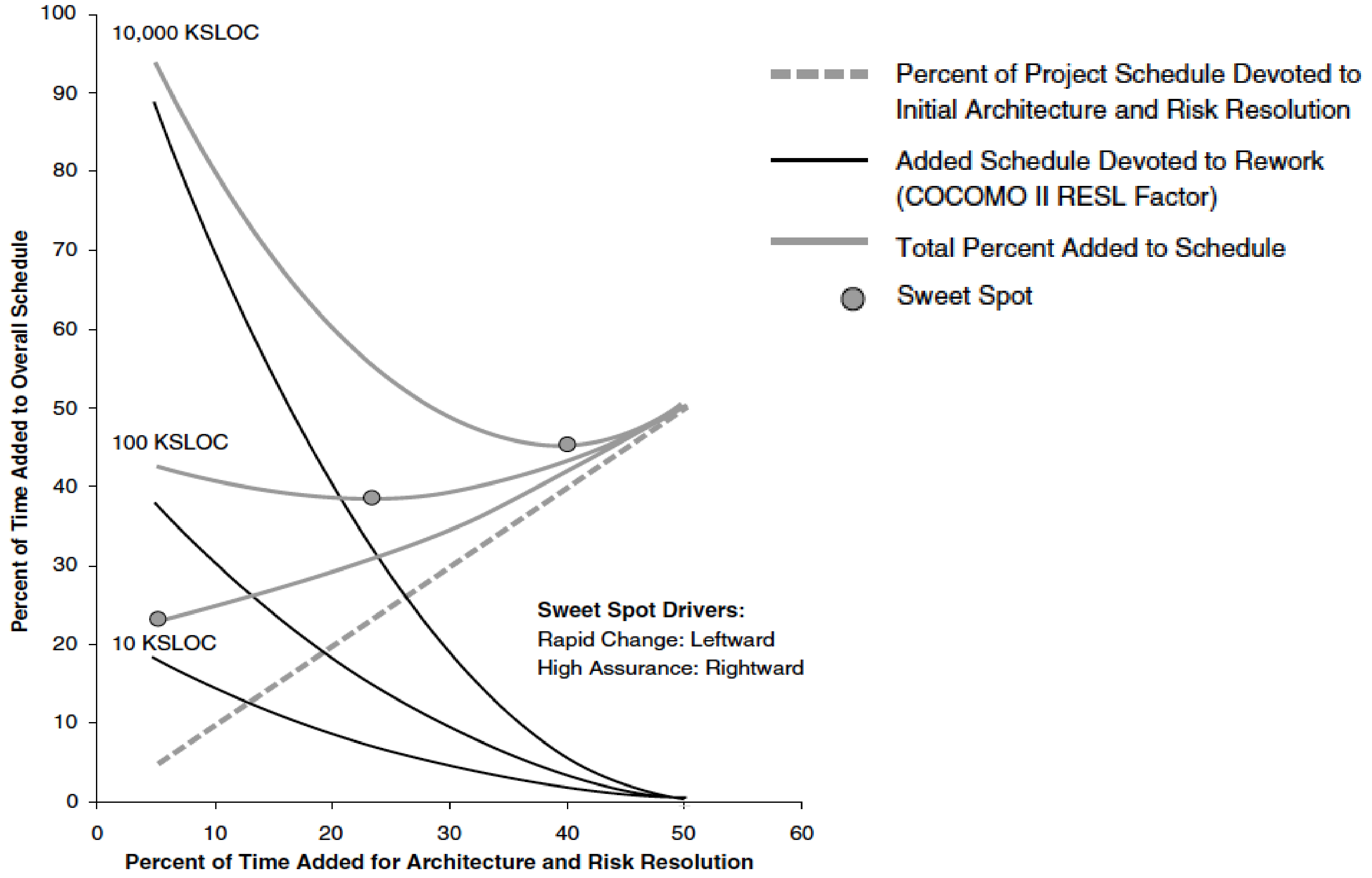
- Agile home ground:
  - Low criticality
  - Senior developers ??
  - Requirements change often
  - Small number of developers
  - Culture that thrives on chaos

- Plan-driven home ground:
  - High criticality
  - Junior developers ??
  - Requirements do not change often
  - Large number of developers
  - Culture that demands order

How Much Architecture?

- There are two activities that can add time to the project schedule:
  - **Up-front design work** on the architecture and **up-front** risk identification, planning, and resolution work
  - **Rework** due to fixing defects and addressing modification requests.
- Intuitively, these two trade off against each other.
- Boehm and Turner plotted these two values against each other for three hypothetical projects:
  - One project of 10 KSLOC
  - One project of 100 KSLOC
  - One project of 1,000 KSLOC

How Much Architecture?



How Much Architecture?

- These lines show that there is a sweet spot for each project.
  - For the 10KSLOC project, the sweet spot is at the far left. Devoting much time to up-front work is a waste for a small project.
  - For the 100 KSLOC project, the sweet spot is around 20 percent of the project schedule.
  - For the 1,000 KSLOC project, the sweet spot is around 40 percent of the project schedule.
- It is difficult to imagine how Agile principles alone can cope with this complexity if there is no architecture to guide

Agility and Documentation

- Write for the reader!
- If the reader doesn’t need it, don’t write it.
  - But remember that the reader may be a maintainer or other newcomer not yet on the project!

Agility and Architecture Evaluation

- Meeting stakeholders’ important concerns is a cornerstone of Agile philosophy.
- Our approach to architecture evaluation is exemplified by the **Architecture Tradeoff Analysis Method** (ATAM).
  - ATAM does not attempt to analyze all, or even most, of an architecture.
  - The focus is determined by a set of quality attribute scenarios that represent the most important of the concerns of the stakeholders.
  - It tailors a **lightweight** architecture evaluation.

An Example of Agile Architecting

- WebArrow **Web-conferencing system**
- To meet stakeholder needs, architect and developers found that they needed to think and work in two different modes at the same time:
  - *Top-down*—designing and analyzing **architectural structures** to meet the demanding requirements
  - *Bottom-up*—analyzing a wide array of implementation-specific and environment-specific constraints and solutions

Experiments to Make Tradeoffs

- To analyze architectural tradeoffs, the team adopted an **agile architecture** discipline combined with a **rigorous program of experiments**.
- The experiments answered questions such as:
  - Would moving to a distributed database from local flat files negatively impact **latency** for users?
  - What (if any) **scalability** improvement would result from using mod_perl versus standard Perl?
  - How many participants could be hosted by a single meeting server?
  - What was the correct ratio between database servers and meeting servers?

Guidelines for the Agile Architect

- Barry Boehm and colleagues have developed the **Incremental Commitment Model** to blend agility and architecture.
- This model is based upon the following principles:
  - Commitment and accountability of success-critical stakeholders
  - Stakeholder "**satisficing**" (meeting an acceptability threshold) based on negotiations and tradeoffs
  - **Incremental** growth of system definition and stakeholder commitment

Guidelines for the Agile Architect

- This model is based upon the following principles:
  - **Iterative** system development and definition
  - **Interleaved** system definition and development
    - allowing early fielding of core capabilities,
    - continual adaptation to change, and
    - timely growth of complex systems without waiting for every requirement and subsystem to be defined
  - Risk management—risk-driven **milestones**, which are key to synchronizing and stabilizing all of this concurrent activity

Agile Architect Advice – No.1

- If you are building **a large, complex system** with relatively **stable and well-understood requirements** and/or distributed development, doing a large amount of architecture work up front will pay off.

Agile Architect Advice – No.2

- On larger projects **with *unstable* requirements**, start by quickly designing a candidate architecture even if it leaves out many details.
  - Be prepared to change and elaborate this architecture as circumstances dictate
  - This early architecture will help in
    - guide development
    - early problem understanding and analysis
    - team coordinate
    - creation of coding templates …

Agile Architect Advice – No.3

- On smaller projects **with uncertain requirements**, at least try to get agreement on the major patterns to be employed.
- Don’t spend too much time on architecture design, documentation, or analysis up front.

Summary

- The Agile Manifesto and principles
- Agile processes were employed on small- to medium-sized projects, and seldom used for larger projects
- Large-scale successful projects need a blend of agile and architecture.
- **Agile architects** take a middle ground, proposing an initial architecture and running with that, until its technical debt becomes too great, at which point they need to refactor.
- Boehm and Turner found that projects have a “sweet spot” where up-front architecture planning pays off.

Chapter 16:
Architecture and Requirements

Architecturally Significant Requirement

- Architectures exist to build systems that satisfy requirements.
- But, to an architect, not all requirements are created equal.
- An **Architecturally Significant Requirement** (ASR) is a requirement that will have a profound effect on the architecture.
- How do we find those?

Approaches to Capture ASRs

- From Requirements Document
- By Interviewing Stakeholders
- By Understanding the Business Goals
- In Utility Tree

ASRs and Requirements Documents

- An obvious location to look for candidate ASRs is in the requirements documents
- Requirements should be in requirements documents!
- Unfortunately, this is not usually the case.

Don’t Get Your Hopes Up

- Many projects don’t create or maintain the detailed, high-quality requirements documents.
- Standard requirements pay more attention to functionality than quality attributes.
- The architecture **is driven by quality attribute requirements** rather than functionalities
- Most requirements specification does not affect the architecture.

Don’t Get Your Hopes Up

- Quality attributes are often captured poorly, e.g.
  - “The system shall be modular”
  - “The system shall exhibit high usability”
  - “The system shall meet users’ performance expectations”
- Much of what is useful to an architect is not in even the best requirements document
  - ASRs often derive from **business goals** in the development organization itself

Gathering ASRs from Stakeholders

- **Stakeholders** often have no idea what QAs they want in a system
  - if you insist on quantitative QA requirements, you're likely to get numbers that are arbitrary.
  - at least some of those requirements will be very difficult to satisfy.
- **Architects** often have very good ideas about what QAs are reasonable to provide.
- **Interviewing the stakeholders** is the surest way to learn what they know and need.

Gathering ASRs from Stakeholders

- **The results of stakeholder interviews** should include
  - a list of architectural drivers
  - a set of QA scenarios that the stakeholders (as a group) prioritized.
- This information can be used to:
  - refine system and software requirements
  - understand and clarify the system's architectural drivers
  - provide rationale for why the architect subsequently made certain design decisions
  - guide the development of prototypes and simulations
  - influence the order in which the architecture is developed.

Quality Attribute Workshop

- The QAW is a facilitated, stakeholder-focused method to generate, prioritize, and refine **quality attribute scenarios** before the software architecture is completed.

Quality Attribute Scenario: Example

- Our vehicle information system sends our current location to the traffic monitoring system.
- The traffic monitoring system combines our location with other information, overlays this information on a Google Map, and ***broadcasts*** it.
- Our location information is correctly included with a probability of 99.9%.

Quality Attribute Scenario: Example

- The developer wishes to change the user interface by modifying the code at design time. The modifications are made with no side effects within three hours.
  - **Stimulus** – Wishes to change UI
  - **Artifact** – Code
  - **Environment**: Design time
  - **Response** – Change made
  - **Response measure** – No side effects in three hours
  - **Source** - Developer

Quality Attribute Scenario: Example

- Users initiate transactions under normal operations. **The system** processes the transactions with an average latency of two seconds.
  - Stimulus: transaction arrivals
  - Source: users
  - Artifact: **the system**
  - Response: process the transactions
  - Response measure: average latency of two seconds
  - Environment: under normal operation

Quality Attribute Scenario: Example

- A disgruntled employee from a remote location attempts to modify the pay rate table during normal operations. The system maintains an audit trail and the correct data is restored within a day.
  - Stimulus: unauthorized attempts to modify the pay rate table
  - Stimulus source: a disgruntled employee
  - Artifact: the system with pay rate table
  - Environment: during normal operation
  - Response: maintains an audit trail
  - Response measure: correct data is restored within a day

Quality Attribute Scenario: Example

- The user downloads a new application and is using it productively after two minutes of experimentation.
  - Source: user
  - Stimulus: download a new application
  - Artifact: system
  - Environment: runtime
  - Response: user uses application productively
  - Response measure: within two minutes of experimentation

QAW Steps

- **Step 1: QAW Presentation and Introductions.**
  - QAW facilitators describe the motivation for the QAW and explain each step of the QAW.
- **Step 2: Business/Mission Presentation.**
  - The stakeholder representing the business concerns presents the system's business context, broad functional requirements, constraints, and known quality attribute requirements.
  - The quality attributes will be derived largely from the business/mission needs
- **Step 3: Architectural Plan Presentation.**
  - The architect will present the system architectural plans
  - This lets stakeholders know the current architectural thinking

QAW Steps

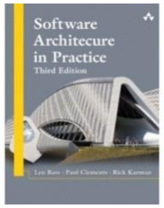
- **Step 4: Identification of Architectural Drivers.**
  - The facilitators will share **their list of key architectural drivers** assembled during Steps 2 and 3,
  - **Architectural drivers** includes overall requirements, business drivers, constraints, and quality attributes.
  - ask the stakeholders for clarifications, additions, deletions, and corrections, and **achieve a consensus** on the architectural drivers

QAW Steps

- **Step 5: Scenario Brainstorming.**
  - Each stakeholder expresses a scenario representing his or her concerns with respect to the system.
  - Facilitators ensure that each scenario has an explicit stimulus and response.
  - Make at least **one representative scenario** for each architectural driver listed in Step 4.

QAW Steps

- **Step 6: Scenario Consolidation.**
  – Similar scenarios are consolidated where reasonable.
- **Step 7: Scenario Prioritization.**
  – Allocating each stakeholder a number of votes equal to 30 percent of the total number of scenarios
  – Each stakeholder allocate their votes to scenario
- **Step 8: Scenario Refinement.**
  – The top scenarios are refined and elaborated.
  – Facilitators help the stakeholders put the scenarios in the six-part scenario form



ASRs from Business Goals

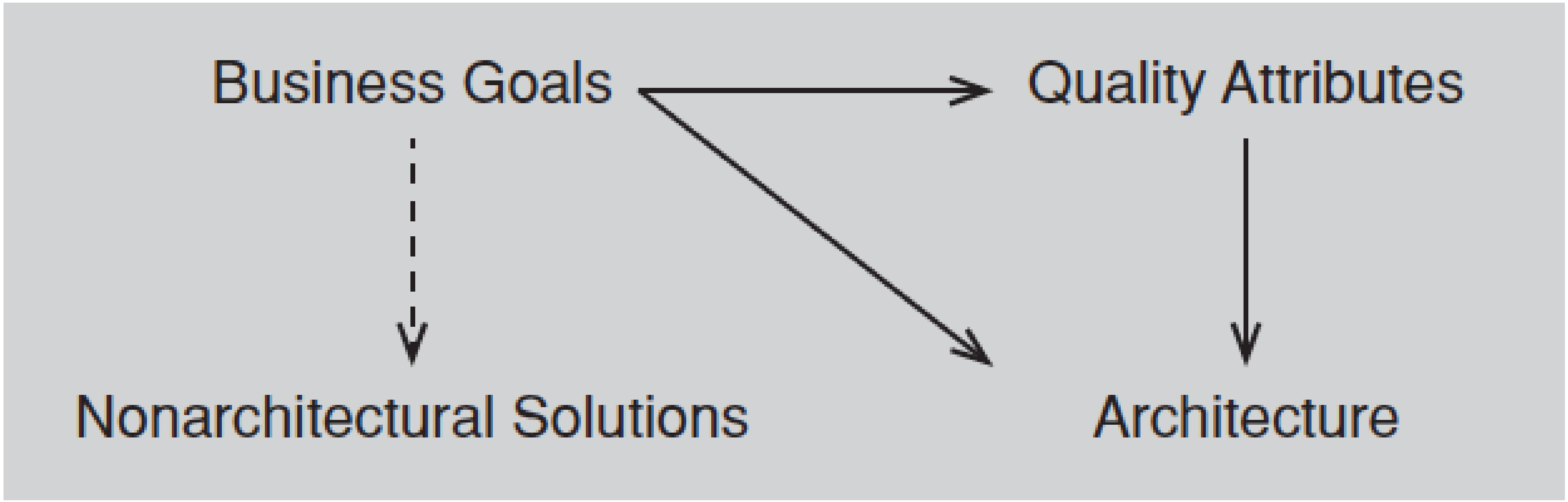
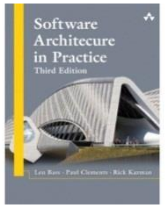
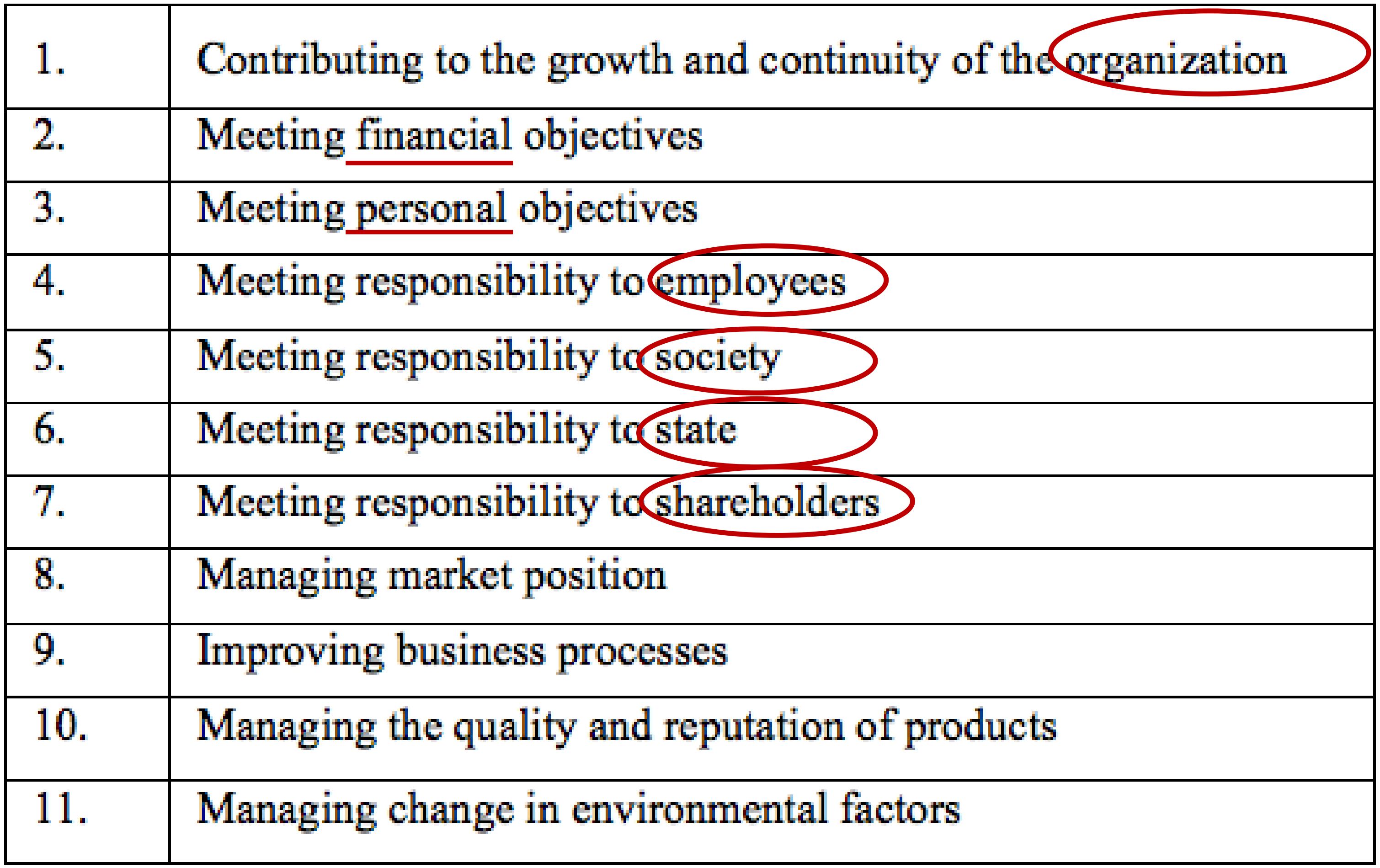


FIGURE 16.1 Some business goals may lead to quality attribute requirements (which lead to architectures), or lead directly to architectural decisions, or lead to nonarchitectural solutions.



Categories of Business Goals



1.	Contributing to the growth and continuity of the organization
2.	Meeting financial objectives
3.	Meeting personal objectives
4.	Meeting responsibility to employees
5.	Meeting responsibility to society
6.	Meeting responsibility to state
7.	Meeting responsibility to shareholders
8.	Managing market position
9.	Improving business processes
10.	Managing the quality and reputation of products
11.	Managing change in environmental factors

Expressing Business Goals

Business goal scenario, 7 parts

- *Goal-source*
  - The people providing the goal.
- *Goal-subject*
  - The stakeholders who own the goal and wish it to be true.
  - Each stakeholder might be an individual or the organization itself
- *Goal-object*
  - The entities to which the goal applies.
- *Environment*
  - The context for this goal
  - Environment may be social, legal, competitive, customer, and technological

Expressing Business Goals

Business goal scenario, 7 parts

- *Goal*
  - Any business goal articulated by the goal-source.
- *Goal-measure*
  - A testable measurement to determine if the goal has been achieved.
  - The goal-measure should state the time by which the goal should be achieved.
- *Pedigree and value*
  - The degree of confidence the person who stated the goal has in it
  - The goal’s value.

Expressing Business Goals

- For the system being developed, <goal-subject> desires that <goal-object> achieve <goal> in the context of <environment> and will be satisfied if <goal-measure>
- For MySys, the project manager has the goal that customer satisfaction will rise by 10 percent (as a result of the increased quality of MySys)

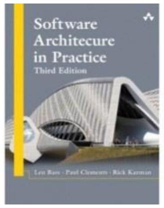
Capturing ASRs in a Utility Tree

An ASR must have the following characteristics:

- *A profound impact on the architecture*
  - Including this requirement will very likely result in a different architecture than if it were not included.
- *A high business or mission value*
  - If the architecture is going to satisfy this requirement it must be of high value to important stakeholders.

Utility Tree

- A way to record ASRs all in one place.
- Establishes priority of each ASR in terms of
  - Impact on architecture
  - Business or mission value
- ASRs are captured as scenarios.
- Root of tree is placeholder node called “Utility”.
- Second level of tree contains broad QA categories.
- Third level of tree refines those categories.
- Leaf nodes are the concrete quality attribute scenarios



Utility Tree Example (excerpt)

	Quality Attribute	Attribute Refinement	ASR
	Performance	Transaction response time	A user updates a patient's account in response to a change-of-address notification while the system is under peak load, and the transaction completes in less than 0.75 second. (H,M)
		Throughput	
	Usability	Proficiency training	
Utility		Normal operations	
	Configurability	User-defined changes	
	Maintainability	Routine changes	
		Upgrades to commercial components	

Key:
H=high
M=medium
L=low

Utility Tree: Next Steps

- ASRs that rate a (H,H) rating are the ones that deserve the most attention
  - A very large number of these might be a cause for concern: Is the system achievable?
- Stakeholders can review the utility tree to make sure their concerns are addressed.

Tying the Methods Together

- How should you employ requirements documents, stakeholder interviews, Quality Attribute Workshops, and utility trees together?
  - If important stakeholders have been overlooked in the requirements-gathering process, use interviews or a QAW.
  - Use a **quality attribute utility tree** as a repository for the scenarios produced by a **QAW**.

Summary

- Architectures are driven by Architecturally Significant Requirements (ASRs):
  - requirements that will have profound effects on the architecture.
- ASRs may be captured
  - from requirements documents,
  - by interviewing stakeholders, or
  - by conducting a Quality Attribute Workshop.
- Be mindful of the business goals of the organization.
  - Business goals can be expressed in a common, structured form and represented as scenarios.

Summary

- A useful representation of quality attribute requirements is in a utility tree.
- The utility tree helps to capture these requirements in a structured form.
- Scenarios are prioritized.
- This prioritized set defines your “marching orders” as an architect.

Chapter 17: Designing an Architecture

Design Strategy

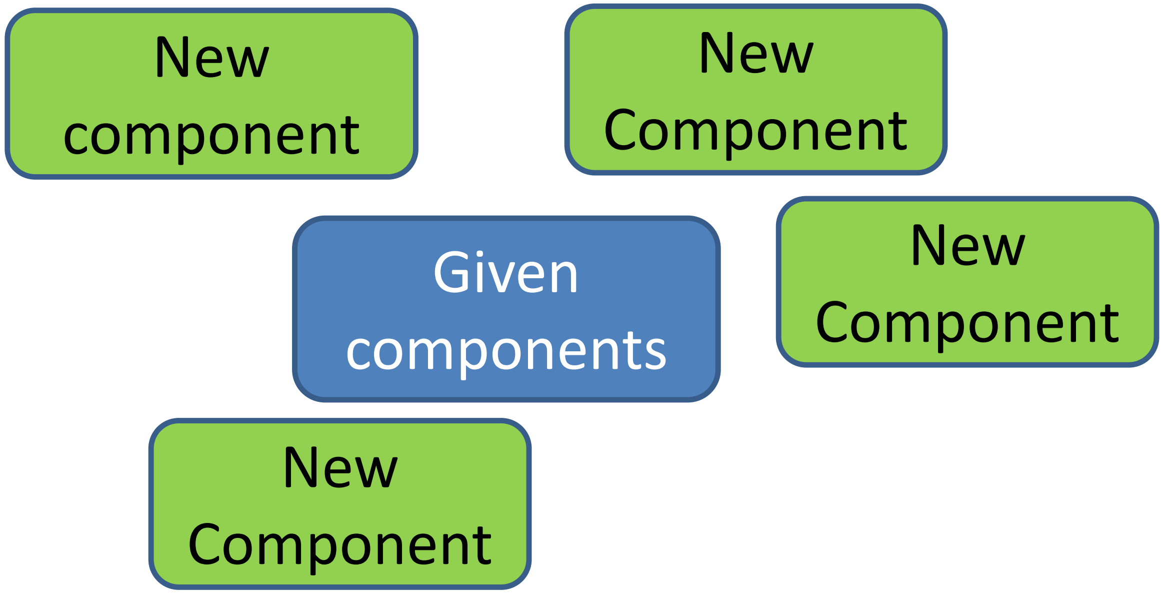
- Decomposition
- Designing to Architecturally Significant Requirements
- Generate and Test

Decomposition

- Architecture determines quality attributes
- **Important quality attributes** are characteristics of the *whole* system.
- Design begins with the whole system
  - The whole system is decomposed into parts
  - Each part may inherit all or part of the quality attribute requirements

Design Doesn’t Mean Green Field

- If you are given components to be used in the final design, then the **decomposition** must accommodate those components.



Designing to Architecturally Significant Requirements

- Remember architecturally significant requirements (ASRs)?
- These are the requirements that you must satisfy with the design
  - There are a small number of these
  - They are the most important (by definition)
- Two questions:
  - What happens to the other requirement?
  - Do I design for one ASR at a time or all at once?

What About Other Quality Requirements?

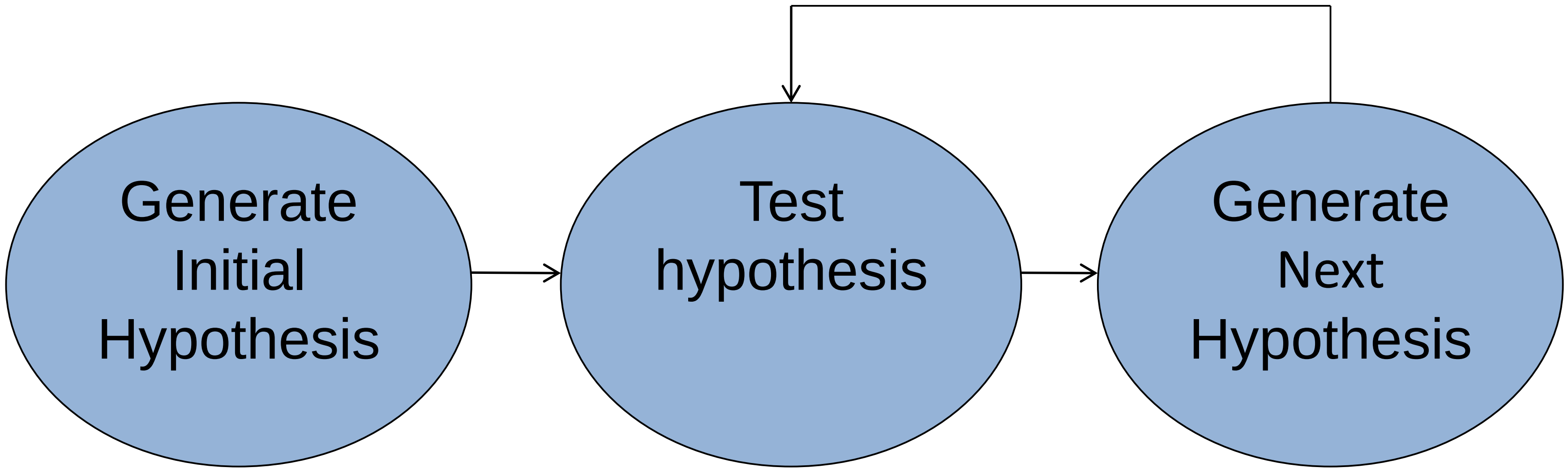
- If your design does not satisfy a particular non ASR quality requirement
  - **Adjust your design** so that the ASRs are still satisfied and so is this additional requirement or
  - **Weaken the additional requirement** so that it can be satisfied either by the current design or by a modification of the current design or
  - **Change the priorities** so that the one not satisfied becomes one of the ASRs or
  - **Declare** the additional requirement non-satisfiable in conjunction with the ASRs.

How Many ASRs Simultaneously?

- If you are inexperienced in design then design for the ASRs one at a time beginning with the most important.
- As you gain experience, you will be able to design for multiple ASRs simultaneously.

Generate and Test

- View the **current design as a hypothesis**.
- Ask whether the current design satisfies the requirements (**test**)
- If not, then **generate** a new hypothesis



Raises the Following Questions

- Where does initial hypothesis come from?
- How do I test a hypothesis?
- How do I generate the next hypothesis?
- When am I done?

Where Does the Initial Hypothesis Come From?

- **Existing systems**
  - Very few systems are completely constructed from the scratch
- **Frameworks**
  - A partial design that provides services that are common in particular domains, e.g., web applications, middleware
  - A design framework may constrain communication to be via a broker, or publish-subscribe system

Where Does the Initial Hypothesis Come From?

- **Less desirable** sources
  - Patterns and tactics
  - Design checklists
- Why “less desirable”?
  - The less desirable ones do not cover all of the requirements.
  - They typically omit many of the quality attribute requirements.

How Do I Test a Hypothesis?

- The **analysis technique** described previously
- **Design checklists** from quality attribute discussion.
- Architecturally significant requirements

- What is the output of the tests?
  - List of requirements – either responsibilities or quality – **not met** by current design.

How Do I Generate the Next Hypothesis?

- Add missing responsibilities.
- Use tactics to adjust quality attribute behavior of hypothesis.
  - The choice of tactics will depend on which quality attribute requirements are not met.
  - Be mindful of the side effects of a tactic.

When Am I Done?

- All ASRs are satisfied and/or…
- You run out of budget for design activity
  - In this case, use the best hypothesis so far and begin implementation
  - To relax or eliminate the requirement
  - To argue for more budges

The Attribute-Driven Design Method

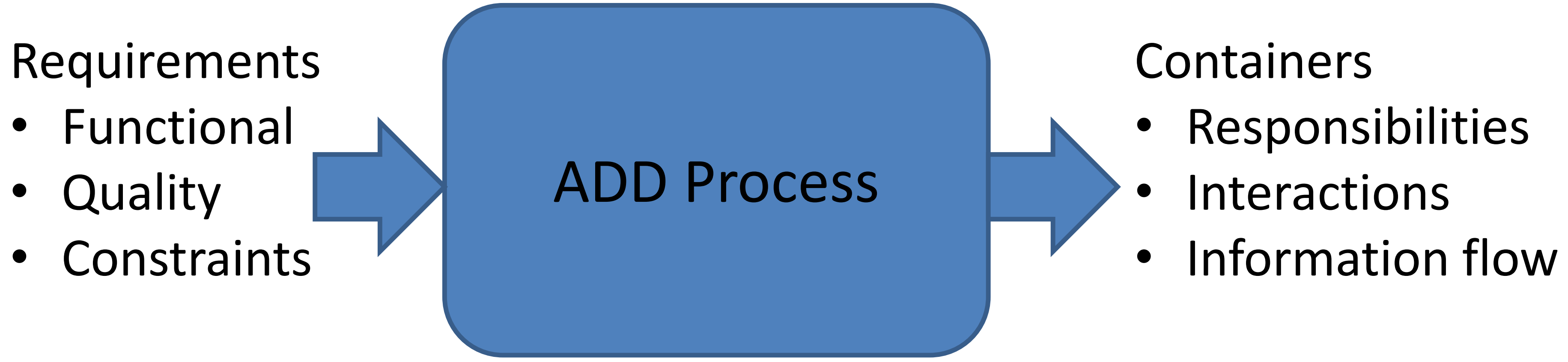
- An iterative method. At each iteration you
  - **Choose a part** of the system to design.
  - **Marshal** all the architecturally significant requirements for that part.
  - **Generate and test a design** for that part.
- ADD does not result in a complete design
  - Set of containers with responsibilities
  - Interactions and information flow among containers
- Does not produce an API for containers.

ADD Inputs

- Requirements
  - Functional, quality, constraints
- A context description
  - What are the boundary of the system being designed?
  - What are the external systems, devices, users and environment conditions with which the system being designed must interact?

ADD Outputs

- Architectural elements and their relationship
  - Responsibility of elements
  - Interactions
  - Information flow among the elements

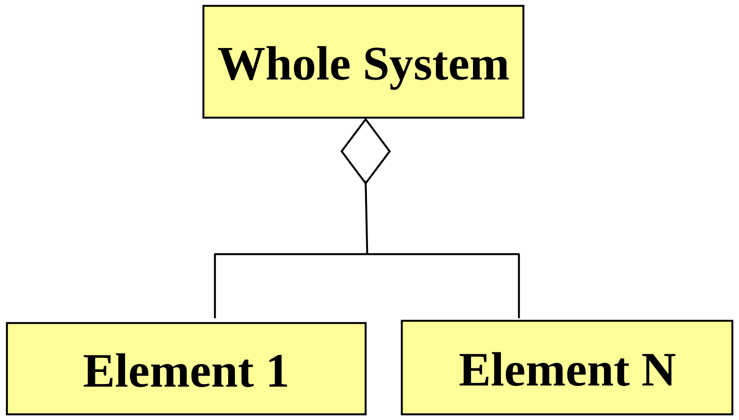


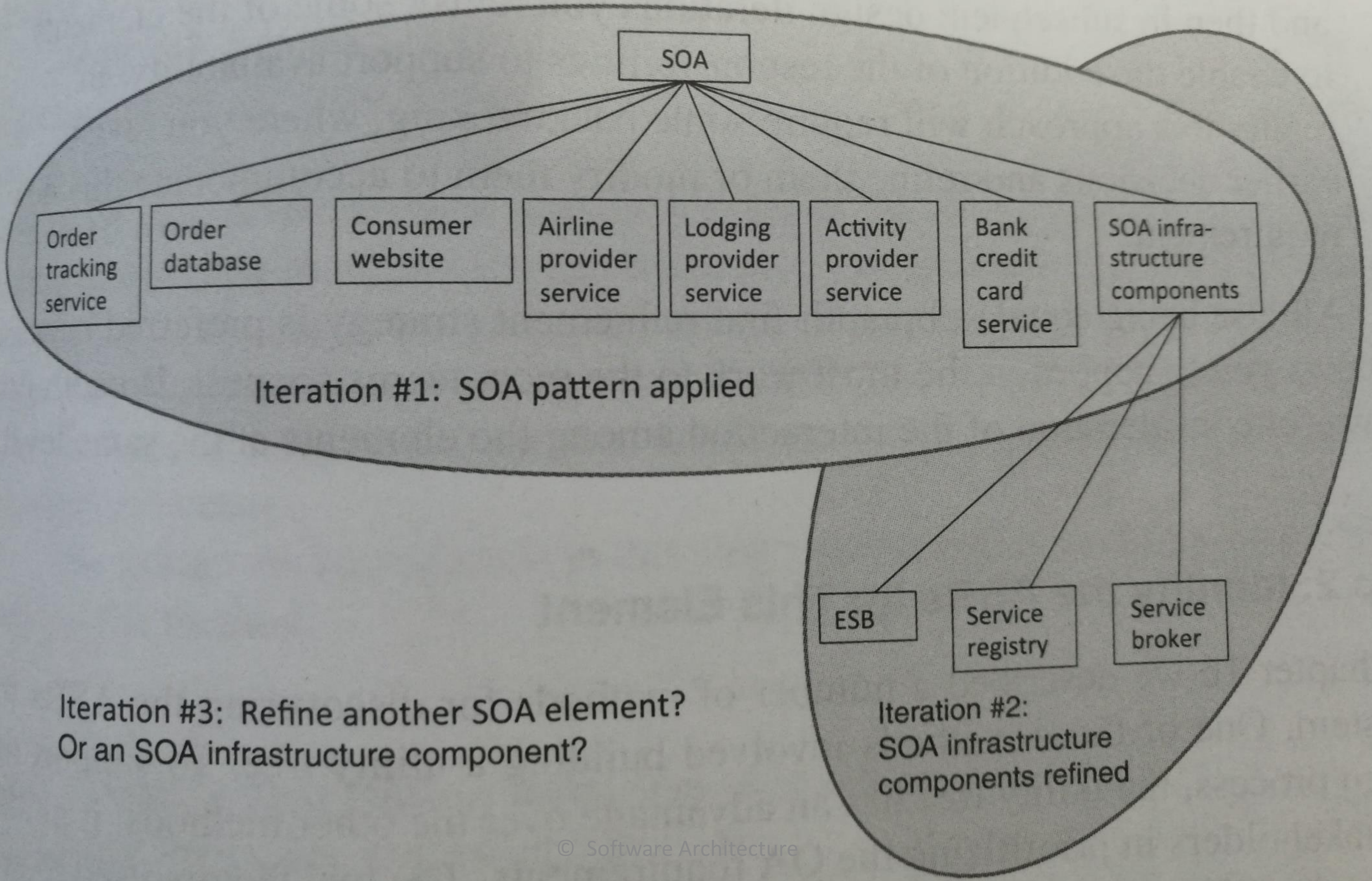
The Steps of ADD

1. Choose an element of the system to design.
2. Identify the ASRs for the chosen element.
3. Generate a design solution for the chosen element.
4. Inventory remaining requirements and select the input for the next iteration.
5. Repeat steps 1–4 until all the ASRs have been satisfied.

Step 1: Choose an Element of the System to Design

- For **green field designs**, the element chosen is usually the whole system.
- For **legacy designs**, the element is the portion to be added.
- After the first iteration:




SOA
Order tracking service
Order database
Consumer website
Airline provider service
Lodging provider service
Activity provider service
Bank credit card service
SOA infra-structure components
Iteration #1: SOA pattern applied
ESB
Service registry
Service broker
Iteration #2: SOA infrastructure components refined
Iteration #3: Refine another SOA element? Or an SOA infrastructure component?

Which Element Comes Next?

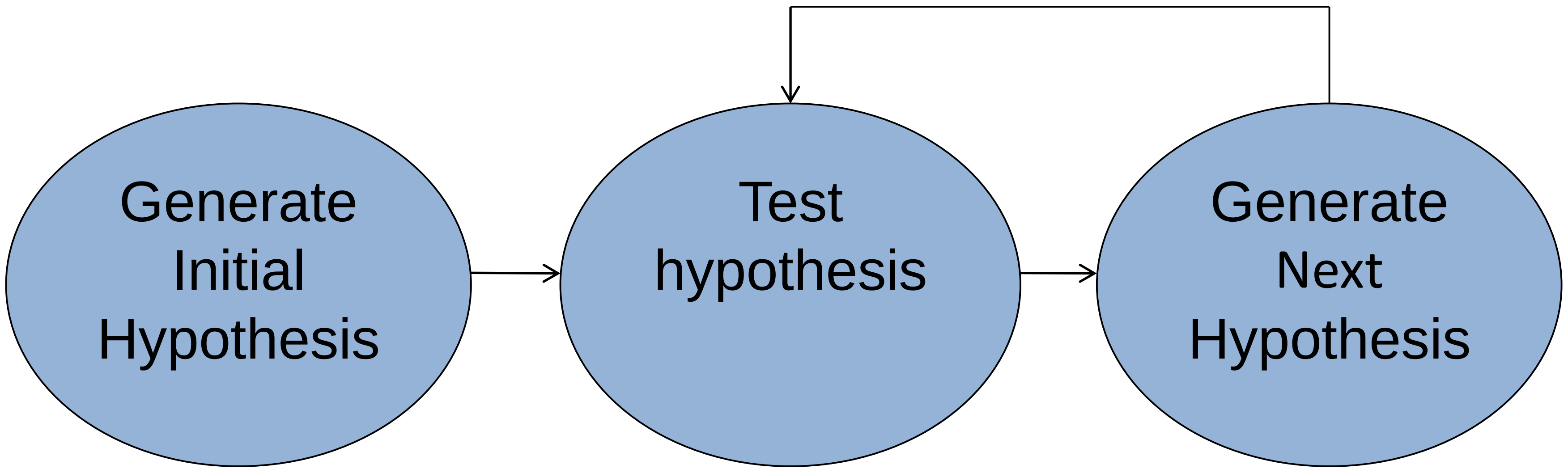
- Two basic refinement strategies:
  - Breadth first
  - Depth first
- Which one to choose?
- If using new technology => **depth** first: explore the implications of using that technology.
- If a team needs work => **depth** first: generate requirements for that team.
- Otherwise => breadth first.

Step 2: Identify the ASRs for the Chosen Element

- If the chosen element is the whole system, then use a utility tree (as described earlier).
- If the chosen element is further down the decomposition tree, then generate a utility tree from the requirements for that element.

Step 3: Generate a Design Solution for the Chosen Element

- Apply generate and test to the chosen element with its ASRs



Step 4: Select the Input for the Next Iteration

- For each **functional requirement**
  - Ensure that requirement has been **satisfied**.
  - If not, then add responsibilities to satisfy the requirement.
    - **Add** them to container with similar requirements
    - If no such container, may need to **create new one** or add to container with dissimilar responsibilities (coherence)
    - If container has too many requirements for a team, **split** it into two portions. Try to achieve loose coupling when splitting.

Quality Attribute Requirements

- If the quality attribute requirement has been **satisfied**, it does not need to be further considered.
- If the quality attribute requirement has not been satisfied then either
  - **Delegate** it to one of the child elements
  - **Split** it among the child elements
- If the quality attribute **cannot be satisfied**, see if it can be weakened. If it cannot be satisfied or weakened then it cannot be met.

Constraints

- Constraints are treated as quality attribute requirements have been treated.
  - Satisfied
  - Delegated
  - Split
  - Unsatisfiable

Repeat Steps 1–4 Until All ASRs are Satisfied

- At end of step 4, each child element will have associated with it a set of:
  - functional requirements,
  - quality attribute requirements, and
  - constraints.
- This sets up the child element for the next iteration of the method.

Summary

- Designing Strategies
  - Decomposition
  - Designing to Architecturally Significant Requirements
  - Generate and Test
- Attribute Driven Design
  - **Choose a part** of the system to design.
  - **Marshal** all the architecturally significant requirements for that part.
  - **Generate and test a design** for that part.

Chapter 19: Architecture, Implementation, and Testing

Architecture and Implementation

- Four techniques to keep the code and the architecture consistent:
  - Embedding the design in the code
  - Frameworks
  - Code templates
  - Keeping code and architecture consistent (i.e. avoiding “architecture erosion”)

Embedding the Design in the Code

- Architecture acts as a **blueprint** for implementation. This means
  - Implementers know what architectural structure they are implementing. E.g. layer, pub/sub, MVC, broker, …
  - They can document the architectural structure in the code as comments. Then anyone picking up the code will know some of the constraints.
  - Then tools can automatically relate the code and the architecture.

Frameworks

- A **framework** is a reusable set of libraries or classes organized around a particular theme.
- A programmer uses the services provided by a framework.
- Examples are
  - **Ruby on Rails** is based on MVC is designed for web applications. It offers the ability to gather information from the web server, talk to or query the database, and render templates.
  - **AUTOSAR** is designed for the computers inside of automobiles.

Code Templates

- **A code template** is collection of code within which the programmer provides application specific portions.
- For example :

  In the event that a failure is detected in a critical-application component, a switchover occurs as follows:

  1. A secondary copy, executing in parallel in background on a different processor, is promoted to the new primary.
  2. The new primary reconstitutes with the application's clients
  3. A new secondary is started to serve as a backup for the new primary.
  4. The newly started secondary announces itself to the new primary.

  If failure is detected within a secondary, a new one is started on some other processor.

```
initialize application/application protocols
ask for current state (image request)
Loop
Get_event
Case Event_Type is
-- "normal" (non-fault-tolerant-related) requests to
-- perform actions; only happens if this unit is the
-- current primary address space
when X => Process X
Send state data updates to other address spaces
when Y => Process Y
Send state data updates to other address spaces
...
when Terminate_Directive => clean up resources; terminate
        := true
when State_Data_Update => apply to state data
-- will only happen if this unit is a secondary address
-- space, receiving the update from the primary after it
-- has completed a "normal" action sending, receiving
-- state data
when Image_Request => send current state data to new
        address space
when State_Data_Image => Initialize state data
when Switch_Directive => notify service packages of
        change in rank
-- these are requests (that come in) after a PAS/SAS
-- switchover; they report services (that they had
-- requested from the old (failed) PAS) which this unit
-- (now the PAS) must complete. A, B, etc. are the names
-- of the clients.
when Recon_from_A => reconstitute A
when Recon_from_B => reconstitute B
...
when others => log error
end case
exit when terminate
end loop
```

Code Templates

- Process:
  – Use the code template for every critical component
  – Place application specific code in fixed places within the template.

Advantages of Code Templates

- Components with similar properties behave in a similar fashion.
- Template only needs to be **debugged once**.
- Complicated portions can be completed by skilled personnel and handed off to less skilled personnel.

Keeping Code and Architecture Consistent

- The implementation will drift away from the documented architecture.
- **Implementers** may make decisions that are not consistent either with each other or with the architecture.
- The architecture may not have foreseen all eventualities that come up.

Preventing Architecture Erosion

- Use tools to enforce architectural constraints.
  - can have architecture rules added that are enforced during a build
- Give more credence/freedom to remaining portion when creating code template
- Schedule documentation/code synchronization times.

Architecture and Testing

- Two Levels of Testing
- **Unit testing**: tests running on specific pieces of software.
  - Architecture defines the units that are to be tested. They are components or modules.
- What is needed to test:
  - **Responsibilities** for functional correctness
  - **Performance** through synthetic loads
  - **Availability** through fault injection.
  - **Modifiability** requirements can also be tested by assigning changes to test teams.

Two Levels of Testing

- **Integration Testing**: to test what happens when separate software units start to work together
- Integration test can test functionality, performance, availability, and security.
- **Security** can be tested by having the test executes various attack scenarios.
- Systems may degrade after being run for a long time if resources are not freed or a configuration is incorrectly specified.

Black-Box Testing

- **Black-Box Testing** treats the software as an "black box" without any knowledge about the internal design, structure, or implementation
- The tester's only source information are its requirements
- How does architecture help in black-box testing?
- It helps the tester understand what portions of the requirements related to the specified subsystem

White-Box Testing

- **White-Box testing** makes full use of internal structures, algorithms, and control and data flows of a unit of software
- Tests exercises all control paths of a unit
- It is often used in unit testing

Test Activities

- The architect should be actively involved in
  - **Test planning** is to allocate the resources, i.e., time, labor, technologies, tools, hardware or equipment
  - **Test development** is the procedure in which tests are written, test case are chosen, and test datasets are created.
    - Test driven development is a technique where the system is developed to satisfy a predetermined test.
  - **Test execution**. Testers apply the tests to the software and capture the record errors

Test Activities

- The architect should be actively involved in
  - **Test reporting and defect analysis**
  - **Test harness creation** includes the test engine and test script repository;
  - The primary purpose is to automate the testing process.

Summary

- Implementation
  - **Implementation activities** can embed architecture knowledge in the code
  - **Templates** can be used for critical sections that reoccur
  - **Architecture erosion** can be prevented through use of tools and management processes
- Testing
  - **Unit and integration tests** depend on architectural knowledge and a test harness.
  - The architect should be involved in **a wide variety of test activities**.

Chapter 21: Architecture Evaluation

Three Forms of Evaluation

- Evaluation by the designer within the design process.
- Evaluation by peers within the design process.
- Analysis by outsiders once the architecture has been designed.

Evaluation by the Designer

- Every time the designer makes a key design decision, the chosen alternatives should be evaluated.
- The "test" part of the "generate-and-test" approach
- How much analysis? Three factors include:
  - **The importance of the decision**.
  - **The number of potential alternatives**. More alternatives need more time in evaluating them.
  - **Good enough as opposed to perfect**. Do not spend more time on a decision than it is worth.

Peer Review

- Architectural designs can be peer reviewed, just as code can.
- A peer review can be carried out at any point of the design process where a candidate architecture, or at least a coherent reviewable part of one, exists.
- Allocate at least several hours and possibly half a day.

Peer Review Steps

1. The reviewers determine a number of quality attribute scenarios to drive the review.
2. The architect presents the portion of the architecture to be evaluated.
   - The purpose is to make the reviewers understand the architecture.
3. For each scenario, the designer walks through the architecture and explains how the scenario is satisfied.
4. Potential problems are captured.

Analysis by Outsiders

- “Outside” is relative; this may mean
  - outside the development project
  - outside the business unit where the project resides but within the same company
  - outside the company.
- Outsiders are chosen because they possess specialized knowledge or experience
- Managers tend to be more inclined to listen to problems uncovered by an outside team.
- Be used to evaluate complete architectures.

Contextual Factors for Evaluation

- What is the available artifact that describes the architecture
- Whether the result is public or private
- The number and skill of evaluators
- Which stakeholders will participate
- How the business goals are understood by the evaluators

The Architecture Tradeoff Analysis Method

- The Architecture Tradeoff Analysis Method (ATAM) has been used for over a decade to evaluate software architectures in domains ranging from automotive to financial to defense.
- The ATAM is designed so that evaluators need not be familiar with the architecture or its business goals, the system need not yet be constructed, and there may be a large number of stakeholders.

Participants in the ATAM

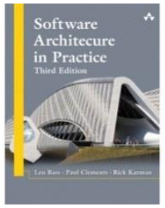
- The evaluation team
  - 3 to 5 people
  - Competent, unbiased outsiders
- Project decision makers
  - the project manager, the architect and the customer who bill for the development
- Architecture stakeholders
  - developers, testers, integrators, maintainers, performance engineers, users, builders of systems
  - to articulate the specific quality attribute goals
  - 12 to 15 stakeholders for a large enterprise-critical architecture

Primary Outputs of the ATAM

- A set of risks and nonrisks
  - A **risk** is defined as an architectural decision that may lead to undesirable consequences in light of quality attribute requirements.
  - A **nonrisk** is an architectural decision that is deemed safe
- A set of **risk themes**
  - examines the full set of risks to look for themes that identify system weaknesses in the architecture.
  - These risk themes will threaten the project’s business goals

Other Outputs of the ATAM

1. A concise presentation of the architecture
2. Articulation of the business goals.
3. Prioritized quality attribute requirements expressed as quality attribute scenarios.
4. Mapping of architectural decisions to quality requirements.
5. A set of identified sensitivity and tradeoff points: architectural decisions that have a marked effect on one or more quality attributes.



Phases of the ATAM

Phase	Activity	Participants	Typical duration
0	Partnership and preparation: Logistics, planning, stakeholder recruitment, team formation	Evaluation team leadership and key project decision-makers	Proceeds informally as required, perhaps over a few weeks
1	Evaluation: Steps 1-6	Evaluation team and project decision-makers	1-2 days followed by a hiatus of 2-3 weeks
2	Evaluation: Steps 7-9	Evaluation team, project decision makers, stakeholders	2 days
3	Follow-up: Report generation and delivery, process improvement	Evaluation team and evaluation client	1 week

Step 1: Present the ATAM

- The **evaluation leader** describes the ATAM steps in brief and the outputs of the evaluation

Step 2: Present Business Drivers

- The **project decision maker** (ideally the project manager or the system's customer) presents a system overview from a business perspective
- The presentation should describe
  - important functions
  - relevant technical, managerial, economic, or political constraints
  - business goals and context
  - major stakeholders
  - architectural drivers (architecturally significant requirements)

Step 3: Present the Architecture

- The **lead architect** makes a presentation describing the architecture
  - **technical constraints** such as operating system, hardware, or middleware prescribed for use, and other systems with which the system must interact.
  - **architectural approaches**, i.e., patterns & tactics used to meet the requirements
- Should convey the essence of the architecture and NOT go too deeply into the details

Step 4: Identify Architectural Approaches

- Understand its architectural approaches, especially patterns and tactics.
- The evaluation team catalogs the patterns and tactics that have been identified.
- The list is publicly captured and will serve as the basis for later analysis

Step 5: Generate Utility Tree

- The evaluation team works with the project decision makers to identify, prioritize, and refine the system's most important quality attribute goals
- The quality attribute goals are articulated in detail via a quality attribute utility tree.
- The scenarios are assigned a rank of importance (High, Medium, Low)

Step 6: Analyze Architectural Approaches

- The evaluation team examines the highest-ranked scenarios one at a time; the architect is asked to explain how the architecture supports each one
- Evaluation team members probe for the architectural approaches used to carry out the scenario
- the evaluation team documents the relevant architectural decisions and identifies and catalogs their risks, nonrisks, and tradeoff points. Examples:
  - **Risk**: The frequency of heartbeats affects the time in which the system can detect a failed component.
  - **Tradeoff**: The heartbeat frequency determines the time for detecting a fault. Higher frequency leads to better availability but consumes more processing time and communication bandwidth (potentially reducing performance).

Step 7: Brainstorm and Prioritize Scenarios

- The purpose of scenario brainstorming is to understand what system success means for stakeholders
- Once the scenarios have been collected, they are prioritized by voting.
- The list of prioritized scenarios is compared with those from the utility tree exercise.
  - If they agree, it indicates good alignment between the architect had in mind and what the stakeholders actually wanted.
  - If additional driving scenarios are discovered, this may itself be a risk. some disagreement between the stakeholders and the architect.

Step 8: Analyze Architectural Approaches

- In this step the evaluation team performs the same activities as in step 6

Step 9: Present Results

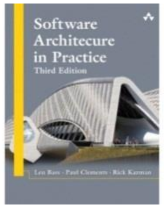
- The evaluation team group risks into risk themes, based on some systemic deficiency.
  - For example, a group of risks about the system's inability to function in the face of various hardware and/or software failures might lead to a risk theme about insufficient attention to providing high availability.
- For each risk theme, the evaluation team identifies which of the business drivers listed in step 2 are affected.

Step 9: Present Results

- The collected information from the evaluation is summarized and presented to stakeholders.
- The following outputs are presented:
  - The architectural approaches documented
  - The set of scenarios and their prioritization from the brainstorming
  - The utility tree
  - The risks discovered
  - The nonrisks documented
  - The sensitivity points and tradeoff points found
  - Risk themes and the business drivers threatened by each one

Lightweight Architectural Evaluation

- An ATAM is a substantial undertaking
  - It requires some 20 to 30 person-days of effort from an evaluation team, plus even more for the architect and stakeholders.
  - Investing this amount of time makes sense on a large and costly project
- A Lightweight Architecture Evaluation method for smaller, less risky projects.
  - May take place in a single day, or even a half-day meeting.
  - May be carried out entirely by members internal to the organization.
  - Of course this may not probe the architecture as deeply.



Typical Agenda: 4-6 Hours

Step	Time	Notes
1. Present the ATAM	0 hours	Participants already familiar with process.
2. Present business drivers	0.25 hours	The participants are expected to understand the system and its business goals and their priorities. A brief review ensures that these are fresh in everyone's mind and that there are no surprises.
3. Present architecture	0.5 hours	All participants are expected to be familiar with the system. A brief overview of the architecture, using at least module and C&C views, is presented. 1-2 scenarios are traced through these views.
4. Identify architectural approaches	0.25 hours	The architecture approaches for specific quality attribute concerns are identified by the architect. This may be done as a portion of step 3.
5. Generate QA utility tree	0.5- 1.5 hours	Scenarios might exist: part of previous evaluations, part of design, part of requirements elicitation. Put these in a tree. Or, a utility tree may already exist.
6. Analyze architectural approaches	2-3 hours	This step—mapping the highly ranked scenarios onto the architecture—consumes the bulk of the time and can be expanded or contracted as needed.
7. Brainstorm scenarios	0 hours	This step can be omitted as the assembled (internal) stakeholders are expected to contribute scenarios expressing their concerns in step 5.
8. Analyze architectural approaches	0 hours	This step is also omitted, since all analysis is done in step 6.
9. Present results	0.5 hours	At the end of an evaluation, the team reviews the existing and newly discovered risks, nonrisks, sensitivities, and tradeoffs and discusses whether any new risk themes have arisen.

Summary

- If a system is important enough for you, then that architecture should be evaluated.
- The **ATAM** is a comprehensive method for evaluating software architectures
- **Lightweight Architecture Evaluation**, based on the ATAM, provides an inexpensive architecture evaluation that can be carried out in several hours
- The number of evaluations and the extent of each evaluation may vary from project to project
  - A designer should perform an evaluation during the process of making an important decision
  - Lightweight evaluations can be performed several times during a project as a peer review exercise