Software Architecture Design
Non-Functional Requirements

Matthew Dailey

Computer Science and Information Management
Asian Institute of Technology
Readings for these lecture notes:


Some material © Bass et al. (2003).
1 Introduction

2 Quality attributes
   - Quality attribute scenarios
   - System quality attributes
   - Business quality attributes
   - Architectural quality attributes

3 Achieving quality goals
   - Tactics for achieving system quality goals

4 Architectural evaluation
   - The ATAM evaluation method
   - Phases of an ATAM evaluation
Non-functional requirements are different from functional requirements in many ways.

Functional requirements:

- Describe **what** a system should do.
- Mostly come from the customer.
- Can be described by a use case model and set of formal “shall” statements.
Non-functional requirements:

- Are not related to individual use cases, but rather to system-wide attributes like performance.
- Can be complete show-stoppers if not met.
- Often conflict with each other, requiring tradeoffs.
- Are more architecture-dependent than functional requirements.
- Are often determined by the architect and stakeholders within the organization.
- Can be described in terms of standard quality attributes.
Outline

1 Introduction

2 Quality attributes
   - Quality attribute scenarios
   - System quality attributes
   - Business quality attributes
   - Architectural quality attributes

3 Achieving quality goals
   - Tactics for achieving system quality goals

4 Architectural evaluation
   - The ATAM evaluation method
   - Phases of an ATAM evaluation
Quality attributes

Usually, business considerations determine the qualities that must be accommodated in a system architecture.

Too often, functionality overrides maintainability, portability, scalability, and other factors determining the long-term success of a project.

Functionality and quality attributes are orthogonal, since a given functionality can be achieved by many different architectures.

Quality requirements depend on the system architecture more than on the functional requirements.
There are three main categories of quality attributes.

- **System qualities**: availability, modifiability, performance, security, testability, usability, others.
- **Business qualities**: time to market, cost and benefit, product lifetime, target market, rollout schedule, integration, others.
- **Architectural qualities**: conceptual integrity, correctness and completeness.

Believe it or not, these quality attribute definitions are standardized by the IEEE! See ISO 91.

We will look at the major system qualities in some detail.
In CMU’s attribute-driven architectural design method, we specify desired quality goals through quality attribute scenarios.

Attribute-driven design is compatible with UP, but the early architectural decomposition is driven more by the quality attribute scenarios than by the use case model.

A quality attribute scenario has six parts, shown in the schematic:
- **Source of stimulus**: the entity generating the stimulus. Could be an actor, an actuator, a sensor, and so on.
- **Stimulus**: a condition arriving at a system. Includes faults, stated intentions by actors, and so on.
- **Environment**: the conditions surrounding the stimulus. Might be normal operation, degraded operation, overload, and so on.
- **Artifact**: the part or parts of the system stimulated.
- **Response**: the response the system takes to the stimulus.
- **Response measure**: how the response can be measured and test.
We distinguish between general and concrete scenarios. Here is a general scenario for the availability system quality attribute:
The **availability** attribute is concerned with system failures. **Faults** are problems that are corrected or masked by the system. **Failures** are uncorrected errors that are user-visible. Formally, availability is written

\[ \alpha = \frac{\text{mean time to failure}}{\text{mean time to failure} + \text{mean time to repair}} \]

An example of a concrete availability scenario:
## General availability scenario:

<table>
<thead>
<tr>
<th>Portion of Scenario</th>
<th>Possible Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>Source</td>
<td>Internal to the system; external to the system</td>
</tr>
<tr>
<td>Stimulus</td>
<td>Fault: omission, crash, timing, response</td>
</tr>
<tr>
<td>Artifact</td>
<td>System’s processors, communications channels, persistent storage, processes</td>
</tr>
<tr>
<td>Environment</td>
<td>Normal operation; degraded mode (i.e., fewer features, a fall back solution)</td>
</tr>
<tr>
<td>Portion of Scenario</td>
<td>Possible Values</td>
</tr>
<tr>
<td>--------------------</td>
<td>----------------</td>
</tr>
<tr>
<td>Response</td>
<td>System should detect event and do one or more of:</td>
</tr>
<tr>
<td></td>
<td>- record it</td>
</tr>
<tr>
<td></td>
<td>- notify the user and other systems as appropriate</td>
</tr>
<tr>
<td></td>
<td>- disable sources of events that cause fault or failure according to defined rules</td>
</tr>
<tr>
<td></td>
<td>- be unavailable for a prespecified interval, where interval depends on criticality of system</td>
</tr>
<tr>
<td></td>
<td>- continue to operate in normal or degraded mode</td>
</tr>
</tbody>
</table>

| Response Measure    | Time interval when the system must be available; Availability time; Time interval in which system can be in degraded mode; Repair time |
The **modifiability** quality is concerned with **what** can change, **when** are changes made, and **who** makes the changes.

Example concrete modifiability scenario:
General modifiability scenario:

<table>
<thead>
<tr>
<th>Portion of Scenario</th>
<th>Possible Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>Source</td>
<td>End user, developer, system administrator</td>
</tr>
<tr>
<td>Stimulus</td>
<td>Wishes to add/delete/modify/vary functionality, quality attribute, capacity</td>
</tr>
<tr>
<td>Artifact</td>
<td>System user interface, platform, environment; system that interoperates with target system</td>
</tr>
<tr>
<td>Environment</td>
<td>At runtime, compile time, build time, design time</td>
</tr>
<tr>
<td>Response</td>
<td>Locates places in architecture to be modified; makes modification without affecting other functionality; tests modification; deploys modification</td>
</tr>
<tr>
<td>Response Measure</td>
<td>Cost in terms of number of elements affected, effort, money; extent to which this affects other functions or quality attributes</td>
</tr>
</tbody>
</table>
The performance quality is concerned with response times and similar measures for various events.

Example concrete performance attribute scenario:
## Quality attributes

### System quality attributes

General **performance** scenario:

<table>
<thead>
<tr>
<th>Portion of Scenario</th>
<th>Possible Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>Source</td>
<td>One of a number of independent sources, possibly from within system</td>
</tr>
<tr>
<td>Stimulus</td>
<td>Periodic events arrive; sporadic events arrive; stochastic events arrive</td>
</tr>
<tr>
<td>Artifact</td>
<td>System</td>
</tr>
<tr>
<td>Environment</td>
<td>Normal mode; overload mode</td>
</tr>
<tr>
<td>Response</td>
<td>Processes stimuli; changes level of service</td>
</tr>
<tr>
<td>Response Measure</td>
<td>Latency, deadline, throughput, jitter, miss rate, data loss</td>
</tr>
</tbody>
</table>
The **security** quality attribute is concerned with:

- Nonrepudiation
- Confidentiality
- Integrity
- Assurance or authenticity
- Availability (no denial of service)
- Auditing

Example concrete security scenario:
### General security scenario:

<table>
<thead>
<tr>
<th>Portion of Scenario</th>
<th>Possible Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>Source</td>
<td>Individual or system that is correctly identified, identified incorrectly, of unknown identity who is internal/external, authorized/not authorized with access to limited resources, vast resources</td>
</tr>
<tr>
<td>Stimulus</td>
<td>Tries to display data, change/delete data, access system services, reduce availability to system services</td>
</tr>
<tr>
<td>Artifact</td>
<td>System services; data within system</td>
</tr>
<tr>
<td>Environment</td>
<td>Either online or offline, connected or disconnected, firewalled or open</td>
</tr>
<tr>
<td>Portion of Scenario</td>
<td>Possible Values</td>
</tr>
<tr>
<td>--------------------</td>
<td>----------------</td>
</tr>
<tr>
<td>Response</td>
<td>Authenticates user; hides identity of the user; blocks access to data and/or services; allows access to data and/or services; grants or withdraws permission to access data and/or services; records access/modifications or attempts to access/modify data/services by identity; stores data in an unreadable format; recognizes an unexplainable high demand for services, and informs a user or another system, and restricts availability of services</td>
</tr>
<tr>
<td>Portion of Scenario</td>
<td>Possible Values</td>
</tr>
<tr>
<td>---------------------</td>
<td>-----------------</td>
</tr>
<tr>
<td>Response Measure</td>
<td>Time/effort/resources required to circumvent security measures with probability of success; probability of detecting attack; probability of identifying individual responsible for attack or access/modification of data and/or services; percentage of services still available under denial-of-services attack; restore data/services; extent to which data/services damaged and/or legitimate access denied</td>
</tr>
</tbody>
</table>
The **testability** attribute is concerned with detecting failure modes. Typically, 40% of the cost of a large project is spent on testing.

This means architectural support for testing that reduces test cost is time well spent. We need to **control** the internal state of and inputs to each unit, then **observe** the corresponding output of that unit.

Example concrete testability attribute scenario:
### General testability scenario:

<table>
<thead>
<tr>
<th>Portion of Scenario</th>
<th>Possible Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>Source</td>
<td>Unit developer; Increment integrator; System verifier; Client acceptance tester; System user</td>
</tr>
<tr>
<td>Stimulus</td>
<td>Analysis, architecture, design, class, subsystem integration completed; system delivered</td>
</tr>
<tr>
<td>Artifact</td>
<td>Piece of design, piece of code, complete application</td>
</tr>
<tr>
<td>Environment</td>
<td>At design time, at development time, at compile time, at deployment time</td>
</tr>
<tr>
<td>Portion of Scenario</td>
<td>Possible Values</td>
</tr>
<tr>
<td>---------------------</td>
<td>--------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>Response</td>
<td>Provides access to state values; provides computed values; prepares test environment</td>
</tr>
<tr>
<td>Response Measure</td>
<td>Percent executable statements executed; Probability of failure if fault exists; Time to perform tests; Length of longest dependency chain in a test Length of time to prepare test environment</td>
</tr>
</tbody>
</table>
The **usability** quality attribute is concerned with

- How easy it is to learn the features of the system
- How efficiently the user can use the system
- How well the system handles user errors
- How well the system adapts to user needs
- To what degree the system gives the user confidence in the correctness of its actions.

Usability scenarios need to be complete before the architecture is designed.

Example concrete usability scenario:
General **usability** scenario:

<table>
<thead>
<tr>
<th>Portion of Scenario</th>
<th>Possible Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>Source</td>
<td>End user</td>
</tr>
<tr>
<td>Stimulus</td>
<td>Wants to learn system features; use system efficiently; minimize impact of errors; adapt system; feel comfortable</td>
</tr>
<tr>
<td>Artifact</td>
<td>System</td>
</tr>
<tr>
<td>Environment</td>
<td>At runtime or configure time</td>
</tr>
</tbody>
</table>
## Quality attributes
### System quality attributes

<table>
<thead>
<tr>
<th>Portion of Scenario</th>
<th>Possible Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>Response</td>
<td>System provides one or more of:</td>
</tr>
<tr>
<td></td>
<td>• To support <em>learn system features</em>: help system is sensitive to context; interface is familiar to user; interface is usable in an unfamiliar context</td>
</tr>
<tr>
<td></td>
<td>• To support <em>use system efficiently</em>: aggregation of data and/or commands; re-use of already entered data and/or commands; support for efficient navigation within a screen; distinct views with consistent operations; comprehensive searching; multiple simultaneous activities</td>
</tr>
<tr>
<td>Portion of Scenario</td>
<td>Possible Values</td>
</tr>
<tr>
<td>--------------------</td>
<td>----------------</td>
</tr>
<tr>
<td>Response</td>
<td>To <em>minimize impact of errors</em>: undo, cancel, recover from system failure, recognize and correct user error, retrieve forgotten password, verify system resources</td>
</tr>
<tr>
<td></td>
<td>To <em>adapt system</em>: customizability; internationalization</td>
</tr>
<tr>
<td></td>
<td>To <em>feel comfortable</em>: display system state; work at the user’s pace</td>
</tr>
<tr>
<td>Response Measure</td>
<td>Task time, number of errors, number of problems solved, user satisfaction, gain of user knowledge, ratio of successful operations to total operations, amount of time/data lost</td>
</tr>
</tbody>
</table>
Some situations might warrant additional quality attributes:

- Scalability
- Portability
- And so on...

The process is the same: we create a general scenario then create a set of specific scenarios for the effort at hand.
Business quality requirements analysis follows the same scenario-based approach, with a new set of qualities:

- **Time to market**: architectural reuse affects development time.
- **Cost and benefit**: in-house architectural expertise is cheaper than outside expertise.
- **Projected lifetime of the system**: long-lived systems require architectures that are modifiable and scalable.
- **Targeted market**: architecture affects what platforms will be compatible and incompatible with the system.

- **Rollout schedule**: if functionality is planned to increase over time, the architecture needs to be customizable and flexible.

- **Integration with legacy systems**: the architecture of the legacy system being integrated will influence the overall system’s architecture.
Architectural quality attributes are also similar to system quality attributes, but concerned with aspects of the architecture itself.

- **Conceptual integrity** is the underlying vision or theme unifying the components and their interactions. The architecture should do similar things in similar ways.

- **Correctness and completeness** is concerned with checking the architecture for errors and omissions.

- **Buildability** is concerned with the organization’s capabilities to actually construct the architecture in question.
1. Introduction

2. Quality attributes
   - Quality attribute scenarios
   - System quality attributes
   - Business quality attributes
   - Architectural quality attributes

3. Achieving quality goals
   - Tactics for achieving system quality goals

4. Architectural evaluation
   - The ATAM evaluation method
   - Phases of an ATAM evaluation
Achieving quality goals

Now that we know what quality attributes are, how do we design an architecture to meet the quality attribute requirements?

For each quality, we need to find or formulate tactics to achieve that quality.

A tactic is a design decision that influences the control of a quality attribute response.

A collection of tactics is an architectural strategy.

Architects usually choose architectural patterns to realize some tactic. But patterns inevitably implement multiple tactics. This can make architectural analysis more difficult.
Achieving quality goals

Example: redundancy is a tactic for achieving availability.

- **Tactics can refine other tactics**: redundancy for a database system might have some shared and some different aspects from redundancy of computation in an embedded system. The tactics for a particular quality attribute can be organized in a hierarchy.

- **Patterns package tactics**: an architectural pattern might compose several concrete tactics to achieve a quality. A pattern for availability would package a redundancy tactic with fault detection and fault prevention tactics.

Here we look at availability tactics in some detail and give an overview of tactics for other system quality attributes.
To achieve availability, we must prevent faults from becoming failures.

Availability tactics can be broken down into three categories:

- Fault detection
- Fault recovery
- Fault prevention
Tactics for fault detection:

- **ping/echo**: send the component a ping and wait for an echo. If not received, notify the fault correction system.
- **heartbeat**: have the component emit a heartbeat periodically, and have another component listen. If not received, notify the fault correction system.
- **exceptions**: trigger an exception handler when a fault occurs.
Tactics for fault recovery:

- **voting**: have redundant processors and have the processes vote for the answer. If one is different, fail it. No downtime.
- **active redundancy**: all redundant components respond to all events. No downtime.
- **passive redundancy**: only the master responds to events, but the backup’s state is updated so it can take over whenever necessary. Downtime is seconds.
- **spare**: unused platforms are configured to replace any of many different components on failure. Downtime is minutes.
- **shadow operation**: have a previously failed component mimic the backup for a while to verify correct operation.
- **state resynchronization**: when a failed component is returned to service, its state needs to be resynchronized with the backup.
- **checkpoint/rollback**: record consistent states, and when a fault occurs, restore to the checkpoint.
Tactics for **fault prevention**:

- **removal from surface**: take a component down periodically to prevent anticipated failures, e.g. reboots to prevent memory leaks from leading to failure.

- **transactions**: bundle sequential steps into chunks that can be undone all at once in case of a fault on an intermediate step.

- **process monitor**: monitor for faulting processes, and kill and restart when a fault is detected.
Achieving quality goals
Tactics for achieving system quality goals

- **Availability**
  - Fault Detection
    - Ping/Echo
    - Heartbeat
    - Exception
  - Recovery-Preparation and Repair
    - Voting
    - Active Redundancy
    - Passive Redundancy
    - Spare
  - Recovery-Reintroduction
    - Shadow
    - State Resynchronization
    - Rollback
  - Prevention
    - Removal from Service
    - Transactions
    - Process Monitor

Fault \rightarrow Availability \rightarrow Fault Masked or Repair Made
Tactics for **modifiability**: 

![Diagram showing tactics to control modifiability](image-url)
Achieving quality goals
Tactics for achieving system quality goals

- Modifiability
  - Localize Changes
    - Semantic Coherence
    - Anticipate Expected Changes
    - Generalize Module
    - Limit Possible Options
    - Abstract Common Services
  - Prevention of Ripple Effect
    - Hide Information
    - Maintain Existing Interface
    - Restrict Communication Paths
    - Use an Intermediary
  - Defer Binding Time
    - Runtime Registration
    - Configuration Files
    - Polymorphism
    - Component Replacement
    - Adherence to Defined Protocols

Changes Arrive
Changes Made, Tested, and Deployed Within Time and Budget
Achieving quality goals
Tactics for achieving system quality goals

Tactics for performance:
Achieving quality goals

Tactics for achieving system quality goals

- **Performance**
  - **Resource Demand**
    - Increase Computation Efficiency
    - Reduce Computational Overhead
    - Manage Event Rate
    - Control Frequency of Sampling
  - **Resource Management**
    - Introduce Concurrency
    - Maintain Multiple Copies
    - Increase Available Resources
  - **Resource Arbitration**
    - Scheduling Policy

Events Arrive → Response Generated Within Time Constraints
Tactics for security:
Achieving quality goals

Tactics for achieving system quality goals
Tactics for testability:
Achieving quality goals
Tactics for achieving system quality goals

Completion of an Increment

Testability

Manage Input/Output
- Record/Playback
- Separate Interface from Implementation
- Specialized Access Routines/Interfaces

Internal Monitoring
- Built-in Monitors

Faults Detected
Tactics for **usability**: 

```
User Request  →  Tactics to Control Usability  →  User Given Appropriate Feedback and Assistance
```
Achieving quality goals
Tactics for achieving system quality goals

![Diagram showing tactics for achieving system quality goals]

- User Request
  - Separate User Interface
    - Cancel
    - Undo
    - Aggregate
  - Support User Initiative
  - Support System Initiative
    - User Model
    - System Model
    - Task Model
  - User Given
    - Appropriate Feedback
    - Assistance
Outline

1. Introduction

2. Quality attributes
   - Quality attribute scenarios
   - System quality attributes
   - Business quality attributes
   - Architectural quality attributes

3. Achieving quality goals
   - Tactics for achieving system quality goals

4. Architectural evaluation
   - The ATAM evaluation method
   - Phases of an ATAM evaluation
Architectural evaluation

Architecture evaluation aims to improve and find deficiencies in an architecture as early as possible in the development process.

Modifying the architecture during the architectural design phase is cheap; modifying the architecture later is costly.

AT&T, reporting on experience from 300 full-scale architecture reviews, estimated a cost savings of 10% on the entire project cost.

Here we provide a brief overview of the Architectural Tradeoff Analysis Method (ATAM) from the SEI.

ATAM is a questioning technique. Quality attribute scenarios are used to come up with probing questions about how the architecture responds to situations.
Before you start an evaluation, you need

- Clearly articulated goals and requirements for the architecture
- Controlled scope (a small number of explicit goals)
- Cost-effectiveness (it might not be necessary to do full-scale reviews for small projects)
- Ensure the key personnel (representatives of each group of stakeholder) are available
- Have a competent evaluation team, ideally separate from the architects and developers
- Managed expectations

The results of the review need to be circulated to all stakeholders afterwards in draft form, with a ranked list of potential issues found.
The goal of ATAM is to come up with a list of risks that the architecture will not meet business goals.

The method is a formal, with specific roles for groups of participants:

- **The evaluation team** is a group external to the project. The team might be consultants or other members of the organization. Typically 3–5 members.

- **The project decision makers** speak for the development project and have the authority to mandate changes to it. Most importantly, the architect must be present. 2 or more members.

- **The architectural stakeholders** with a vested interest in the architecture performing correctly. 12–15 members.
The outputs of an ATAM evaluation are:

- A concise presentation of the architecture.
- Articulation of the business goals.
- A set of quality requirements in the form of quality scenarios.
- Mapping of architectural decisions to quality requirements. This specifies how the architecture supports each of the quality scenarios.
- A set of identified sensitivity and tradeoff points. Architecture decisions that affect qualities are identified along with how they cause tradeoff (e.g. performance vs. reliability).
- A set of risks and nonrisks. This will be the basis of a risk mitigation plan.
- A set of risk themes. The systemic weakness that explain the risks.

The intangible outputs are a sense of community among the stakeholders, open communication, and better understanding of the issues.
An ATAM evaluation may take place over a one-month period.

Phase I only involves the evaluation team and the project decision makers; Phase II includes the stakeholders.

In Phase I:

- The evaluation ground rules are presented.
- The business drivers (including quality attribute goals) are presented.
- The architecture itself is presented.
- The architectural approaches are identified
- The quality attribute goals are prioritized and refined into specific scenarios
- The architectural approaches are analyzed to determine how they satisfy the refined quality goals.
After Phase I, there is a detailed set of quality requirements with the architectural decision tradeoffs that affect those requirements. In Phase II,

- The stakeholders help to prioritize the existing scenarios and propose new ones.
- The architectural tradeoffs are again analyzed, now in consultation with the stakeholders.
- The results of the review are presented to all.

During Phase II, in addition to brainstorming on quality scenarios, the stakeholders provide the in-depth knowledge of whether the proposed architectural tactics will actually meet their expectations.
Bass et al. report that

- Technical participants in ATAM are typically amazed at how many risks can be found in a short time.
- Managers appreciate the opportunity to see precisely how technical issues threaten achievement of their business goals.