Architecture and Architecturally Significant Requirements

Andrew Kotov

Software Engineering Institute
Carnegie Mellon University
Pittsburgh, PA 15213

Document Markings

This material is based upon work funded and supported by the Department of Health and Human Services (HHS) under Contract No. FA8702-15-D-0002 with Carnegie Mellon University for the operation of the Software Engineering Institute, a federally funded research and development center sponsored by the United States Department of Defense.

DM21-0680

Agenda

- Definition and importance of architecture
- Architectural drivers, quality attribute scenarios

What is software architecture?

Software Engineering Institute
Carnegie Mellon University
Pittsburgh, PA 15213

Software Architecture

What is software architecture? The software architecture of a system is *the set of structures* needed to reason about the system, which comprises software elements, relations among them, and properties of both.

- Every system has an architecture!
- Architecture is an abstraction of a system
- Architecture defines the system elements and how they interact via interface
- Architecture doesn't include implementation details of the elements
- Properties of components are assumptions that one component can make about another (provided/required services, performance, how it handles faults or consumes resources, etc.)

Some Implications of Our Definition

Every system has an architecture.

- Every system is composed of elements, and there are relationships among them.
- In the simplest case, a system is composed of a single element, related only to itself.

Just having an architecture is different from having an architecture that is *known* to everyone:

- Is the "real" architecture the same as the specification?
- What is the rationale for architectural decisions?

If you don't explicitly develop an architecture, you will get one anyway—and you might not like what you get!

More Implications of Our Definition

Box-and-line drawings alone are *not* architectures: they are just a starting point.

- You might imagine the behavior of a box or element labeled "database"
  or "executive."
- You need to add specifications and properties to the elements and relationships.

Finally, the definition of architecture is indifferent as to whether the architecture of a system is a good one or a bad one.

- **A good architecture is one that allows a system to meet its functional, quality attribute, and lifecycle requirements.**

Role of Software Architecture

If the only criterion for software was to get the right answer, we would not need architectures—unstructured, monolithic systems would suffice.

But other things also matter, such as

- modifiability
- time of development
- performance
- coordination of work teams

Quality attributes such as these are largely dependent on architectural decisions.

- All design involves tradeoffs among quality attributes.
- The earlier we reason about tradeoffs, the better.

Why Is Software Architecture Important?

Architecture is important for three primary reasons:

1. It provides a vehicle for communication among stakeholders.

2. It is the manifestation of the most important design decisions about a system.

3. It is a transferable, reusable abstraction of a system.

Vehicle for Communication

Architecture provides a common frame of reference in which competing interests can be exposed and negotiated. These interests include

- negotiating requirements with users
- keeping the customer informed of progress and cost
- implementing management decisions and allocations

Architects and implementers use the architecture to guide development.

- Doing so supports architectural analysis.

Most Important Design Decisions – 1

Architecture defines *constraints on implementation*:

- The implementation must conform to prescribed design decisions such as those regarding
  - elements
  - interactions
  - behaviors
  - responsibilities
- The implementation must conform to resource allocation decisions such as those regarding
  - scheduling priorities and time budgets
  - shared data and repositories
  - queuing strategies

Architectures are both prescriptive and descriptive.

Most Important Design Decisions – 2

Architecture dictates the *structure of the organization*.

- Architecture represents the highest level decomposition of a system and is used as a basis for
  - partitioning and assigning the work to be performed
  - formulating plans, schedules, and budgets
  - establishing communication channels among teams
  - establishing plans, procedures, and artifacts for configuration management, testing, integration, deployment, and maintenance

For managerial and business reasons, once established, an architecture becomes very difficult to change.

Most Important Design Decisions – 3

Architecture permits/precludes the *achievement of a system's desired quality attributes*. The strategies for achieving them are architectural.

If you desire…	you need to pay attention to…
high performance	minimizing the frequency and volume of inter-element communication
modifiability	limiting interactions between elements
security	managing and protecting inter-element communication
reusability	minimizing inter-element dependencies
subsetability	controlling the dependencies between subsets and, in particular, avoiding circular dependencies
availability	the properties and behaviors that elements must have and the mechanisms you will employ to address fault detection, fault prevention, and fault recovery
and so forth	…

Most Important Design Decisions – 4

Architecture allows us to predict *system quality attributes* without waiting until the system is developed or deployed.

- Since architecture influences quality attributes in known ways, it follows that we can use architecture to *predict* how quality attributes may be achieved.
- We can analyze an architecture to evaluate how well it meets its quality attributes requirements.
  - These analysis techniques may be heuristic (e.g., back-of-the-envelope calculations, experience-based analogy) and inexpensive.
  - They may be precise (e.g., prototypes, simulations, instrumentation) and expensive.
  - Or they may fall in between (e.g., scenario-based evaluation).

Most Important Design Decisions – 5

Architecture helps us reason about and manage *changes to a system* during its lifetime.

All systems accumulate technical debt over their lifetimes. When this debt is attributable to architectural degradation, we call it *architecture debt*.

Fortunately, by analyzing an architecture we can monitor and manage architectural debt.

Typically, refactoring is used to pay down architecture debt. When to refactor is a decision that includes both technical and business considerations.

Most Important Design Decisions – 6

Once an architecture has been defined, it can be analyzed and prototyped as a skeletal system. Doing so aids the development process in three ways:

1. The architecture can be implemented as a skeletal framework into which elements can be “plugged.”
2. Risky elements of the system can be identified via the architecture and mitigated with targeted prototypes.
3. The system is executable early in the product’s lifecycle. The fidelity
   of the system increases as prototyped parts are replaced by completed elements.

Most Important Design Decisions – 7

Architecture enables more accurate *cost and schedule estimates*, *project planning, and tracking*:

- The more knowledge we have about the scope and structure of a system, the better our estimates will be.
- Teams assigned to individual architectural elements can provide more accurate estimates.
- Project managers can roll up estimates and resolve dependencies and conflicts.

Transferable, Reusable Abstraction – 1

Software architecture constitutes a *model that is transferable across similar systems*.

Software architecture can serve as the basis of a strategic reuse agenda that includes the reuse of

- requirements
- development-support artifacts (templates, tools, etc.)
- code
- components
- experience
- standards

Transferable, Reusable Abstraction – 2

Architecture supports building systems using *large, independently developed components*.

- Architecture-based development focuses on composing elements rather than programming them.
- Composition is possible because the architecture defines which elements can be incorporated into the system and how they are constrained.
- The focus on composition provides for component interchangeability.
- Interchangeability is key to allowing third-party software elements, subsystems, and communication interfaces to be used as architectural elements.

Architecturally Significant Requirements

Software Engineering Institute
Carnegie Mellon University
Pittsburgh, PA 15213

Where Do Architectures Come From?

Software architecture is based on much more than requirements specifications.

It is the result of many different technical, business, and social influences.

Its existence, in turn, influences the technical, business, and social environments that subsequently affect future architectures.

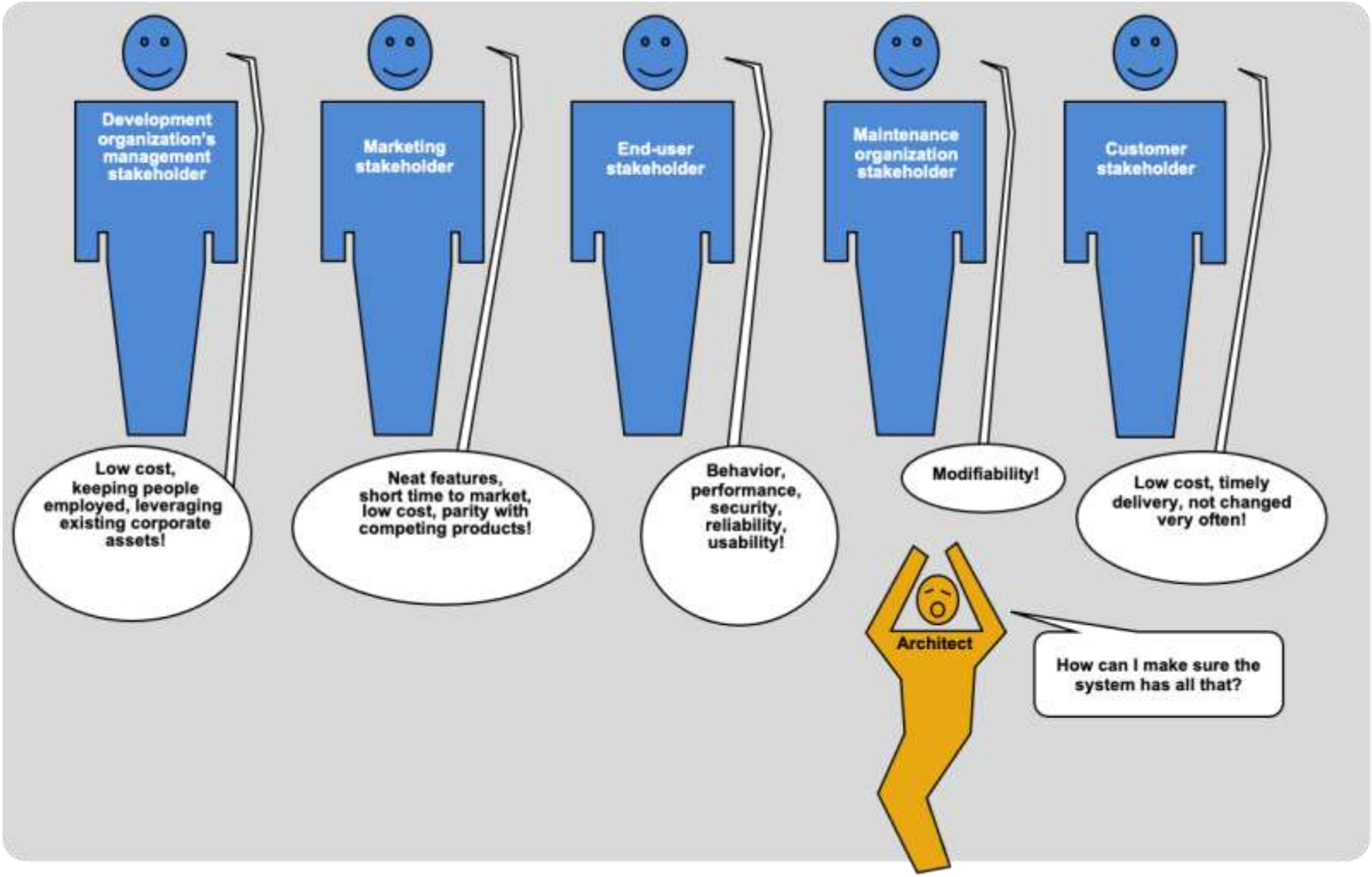
Architects need to know and understand the nature, source, and priority of these influences as early in the process as possible.

Factors Influencing Architectures

Architectures are influenced by

- system stakeholders
- the development organization's business environment
- the technical environment
- the architect's professional background and experience

Concerns of System Stakeholders



Architecture Influence Cycle (AIC)

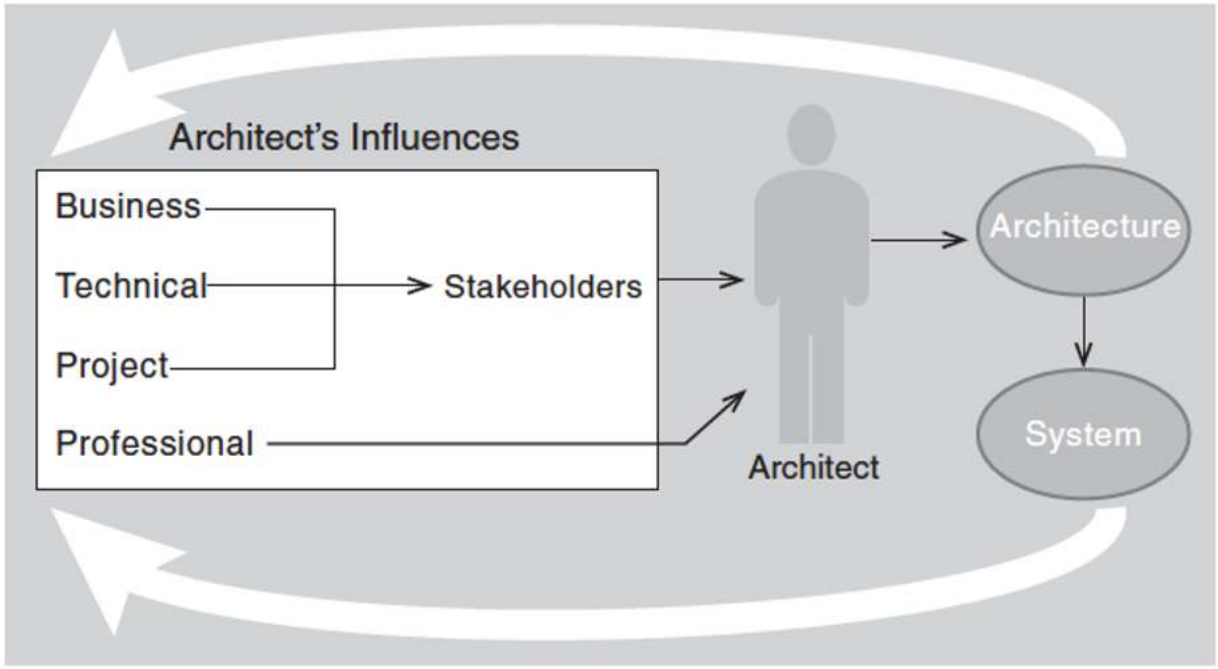


FIGURE 3.5 Architecture Influence Cycle

Architectural Drivers

Architectural drivers are the combination of

- design purpose,
- quality attribute requirements,
- primary functional requirements,
- architectural concerns, and
- constraints

that shape an architecture.

And these are driven, in turn, by business goals.

Design Purpose

Before you can begin you need to be clear about *why* you are designing.

Your objectives will change what and how you design; some examples include

- part of a project proposal (e.g., pre-sales)
- part of creating an exploratory prototype
- during development: greenfield, refactoring, refresh, …

Quality Attributes or Non-Functional Requirements (NFRs)

Quality attributes are properties of work products or goods by which stakeholders judge their quality.

Some examples of quality attributes by which stakeholders judge the quality of software systems are

- performance
- security
- modifiability
- reliability
- usability
- calibratability
- availability
- adaptability
- throughput
- configurability
- subsetability
- reusability

Quality Attribute Requirements

Quality attribute (QA) requirements have the most profound effect on shaping the architecture.

Why is this?

Quality Attribute Requirements

If a functional requirement is “When the user presses the green button, the Options dialog appears”…

- a performance QA annotation might describe how quickly the dialog will appear;
- an availability QA annotation might describe how often this function will fail, and how quickly it will be repaired;
- a usability QA annotation might describe how easy it is to learn this function.

Functional Requirements

The way the system is structured normally does not inhibit the satisfaction of functional requirements.

- Functionality and quality are *orthogonal* concerns.

When designing the architecture, it is obviously important to ensure that the chosen design elements can satisfy the functional requirements.

Functional requirements are often documented as use cases or user stories.

Primary Functional Requirements

Primary use cases

- are critical to the achievement of business goals
- are associated with an important QA scenario
- may imply a high level of technical difficulty
- exercise many architectural elements
- represent a “family” of use cases

Usually only 10-20% of the use cases are primary.

Functionality and Architecture

Functionality is the ability of a system to do the work it was intended
to do.

- Functionality often has associated quality attribute requirements (e.g., a function is required to have a certain level of availability, reliability, and performance).
- We can achieve functional requirements and yet fail to meet their associated quality attribute requirements.
- Functionality can be achieved using many different architectures.
- Achieving quality attribute requirements can be achieved only through judicious choice of architectures.

Architectural Concerns - 1

Architectural concerns are design decisions that should be made whether they are expressed as requirements or not.

We divide them into four categories:

- general concerns
- specific concerns
- internal requirements
- issues

Architectural Concerns - 2

General concerns: “broad” issues that every architect deals with in creating an architecture.

Specific concerns: system-internal issues that an architect must address

Internal requirements: These are derived requirements that are typically not specified in requirement documents.

Issues: These result from analysis activities, such as a design review, so they may not be present initially.

Constraints

Constraints limit the range of possibilities when making design decisions.

- In some cases they are decisions about which you have zero choice.

Before commencing design, identify and justify constraints.

- Technical constraints
  - Use of a legacy database
  - Compliance with a vendor's interface
  - Corporate or industry technical standards
- Other constraints
  - Development team only familiar with Java
  - Obey Sarbanes-Oxley
  - Ready in time for April 15th

Capturing System Requirements

Question: What do we need to define the software architecture?

- **Functional requirements**. They define what the application must do to behave properly.
- **Quality attributes or NFRs**. Quality attributes serve as qualifications of functional requirements or the overall application. They are also called non-functional requirements (NFRs). The can qualify how fast an application operation should be performed or define its service-level agreement.
- **Constraints**. These reflect design decisions that have already been made and cannot be changed. A choice of a specific migration platform, such as ACS, is an example.

Capturing Architecturally Significant Requirements

Not all requirements are created equal for architectural purposes.

Architecturally significant requirement (ASR):

- A profound impact on the architecture – this requirement will likely result in a different architecture than if it were not included
- A high business value – if the architecture is going to satisfy this requirement, it must be of high value to important stakeholders

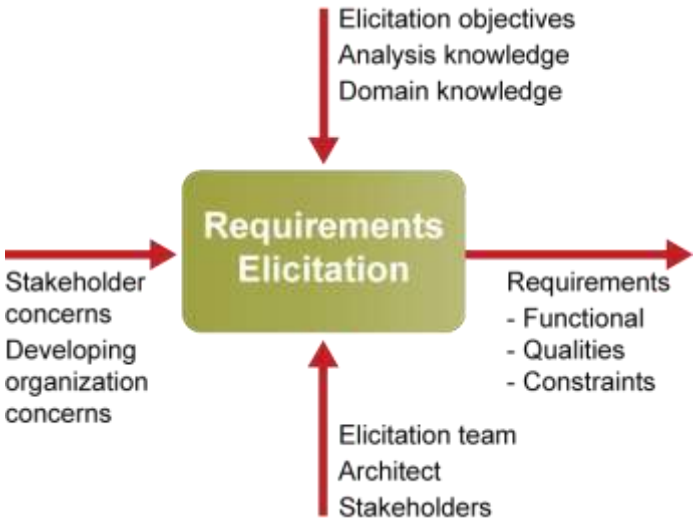
Architecture and Design Decisions

The architecture of a system is the result of design decisions. Design decisions influence achievement of desired qualities.

If you desire…	**you need to pay attention to…**
High performance	minimizing the frequency and volume of inter-element communication
Modifiability/Flexibility	limiting interactions between elements
Security	managing and protecting inter-element communication
Reusability	minimizing inter-element dependencies
Subsetability	controlling the dependencies between subsets and, in particular, avoiding circular dependencies
Availability	the properties and behaviors that elements must have and the mechanisms you will employ to address fault detection, fault prevention, and fault recovery
And so forth	…

Eliciting the Target System Qualities

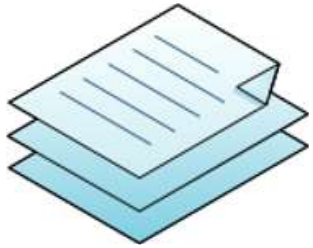
Requirements elicitation methods elicit quality attribute requirements.



For more information, see Bass, L., Bergey, J., Clements, P., Merson, P., Ozkaya, I., Sangwan, R. *A Comparison of Requirements Specification Methods from a Software Architecture Perspective* (CMU/SEI-2006-TR-013). Software Engineering Institute, Carnegie Mellon University, 2006.

Stakeholders and Quality Attributes

Lead To



Stakeholder Concerns

Quality Attribute Requirements

"Increase market share"	Modifiability, Usability
"Maintain a quality reputation"	Performance, Usability, Availability
"Introduce new capabilities seamlessly"	Performance, Availability, Modifiability
"Provide a programmer-friendly framework"	Modifiability
"Integrate with other systems easily"	Interoperability, Portability, Modifiability

Quality Attribute Data from SEI Architecture Evaluations: Top 20 QA Concerns[1]

1. Modifiability: Reduce coupling
2. Performance: Latency
3. Interoperability: Upgrade and integrate with other system components
4. Modifiability: Designing for portability
5. Usability: Ease of operation
6. Availability: Detect faults
7. Interoperability: Ease of interfacing with other systems
8. Modifiability: Designing for extensibility
9. Availability: Recover from faults
10. Performance: Resource management
11. Deployability: Minimize build, test, release duration
12. Modifiability: Reusability
13. Availability: Prevent faults
14. Scalability: Increased processing demands
15. Security: Authorization
16. Interoperability: Resource and data sharing
17. Security: Resist attack
18. Deployability: Configuration and/or dependency management
19. Modifiability: Configurability/composability
20. Deployability: Backward compatibility and/or rollback strategy

[1] Bellomo, S.; Gorton, I.; & Kazman, R. "Insights from 15 Years of ATAM Data: Towards Agile Architecture", *IEEE Software*, September/October, 2015, 32:5, 38-45.

Quality Attribute Data from SEI Architecture Evaluations: QA Concerns Grouped by QA[1]

Modifiability

- Designing for portability
- Designing for extensibility
- Reusability
- Configurability/composability
- Increase cohesion
- Add or modify functionality

Performance

- Latency
- Resource management
- Throughput
- Performance monitoring
- Initialization
- Accuracy

Interoperability

- Upgrade and integrate with other system components
- Ease of interfacing with other systems or components
- Resource and data sharing
- Data integrity
- Compliance with standards/protocols

Availability

- Detect faults
- Recover from faults
- Prevent faults
- Transaction auditing and logging
- Graceful degradation

[1] Bellomo, S.; Gorton, I.; & Kazman, R. "Insights from 15 Years of ATAM Data: Towards Agile Architecture", *IEEE Software*, September/October, 2015, 32:5, 38-45.

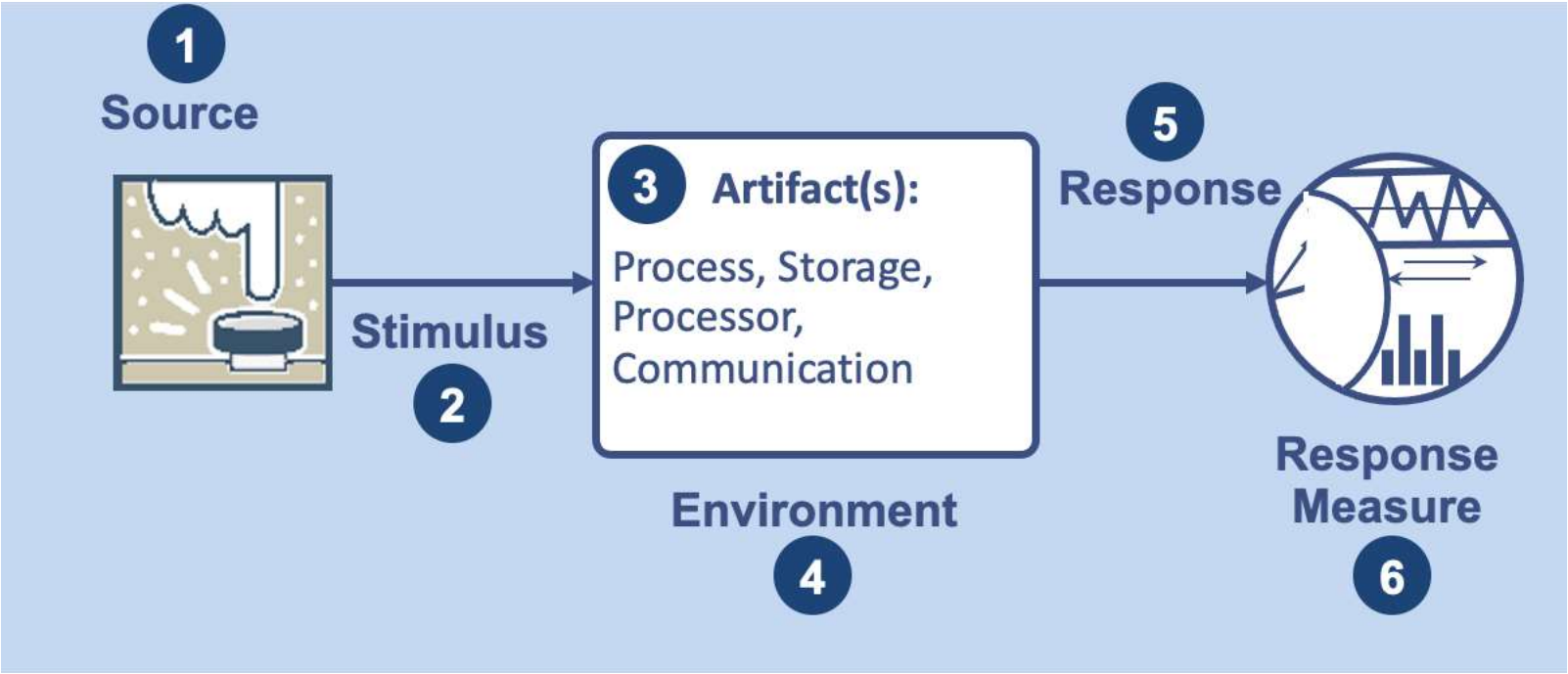
Describing Quality Attributes

Quality attribute names by themselves are not enough.

- Quality attribute requirements are often non-operational.
  - For example, it is meaningless to say that the system shall be "modifiable." Every system is modifiable with respect to some set of changes and not modifiable with respect to some other set of changes.
- Heated debates often revolve around the quality attribute to which a particular system behavior belongs.
  - For example, system failure is an aspect of availability, security, and usability.
- The vocabulary describing quality attributes varies widely.

Parts of a Quality Attribute Scenario

We specify the most important quality attribute requirements as quality attribute scenarios, using a 6-part structure:



Quality Attribute Example Scenario

An unanticipated external message is received by a process during normal operation. The process informs the operator of the message's receipt, and the system continues to operate with no downtime.

Source	External to the system
Stimulus	Unanticipated message
Artifact(s)	Process
Environment	Normal operation
Response	Inform operator; continue to operate
Response Measure	No downtime

Prioritizing QA Scenarios

Before commencing design, prioritize quality attribute scenarios.

- Typically only the most important scenarios can be considered early in architectural design.
- Choose the top 5-7 scenarios in the initial design round.

If a Quality Attribute Workshop was performed, the scenarios will already be prioritized.

Or you could create a utility tree, where scenarios are prioritized across two dimensions:

- importance to the success of the system, ranked by the customer (H, M, L)
- degree of technical risk, ranked by the architect (H, M, L)

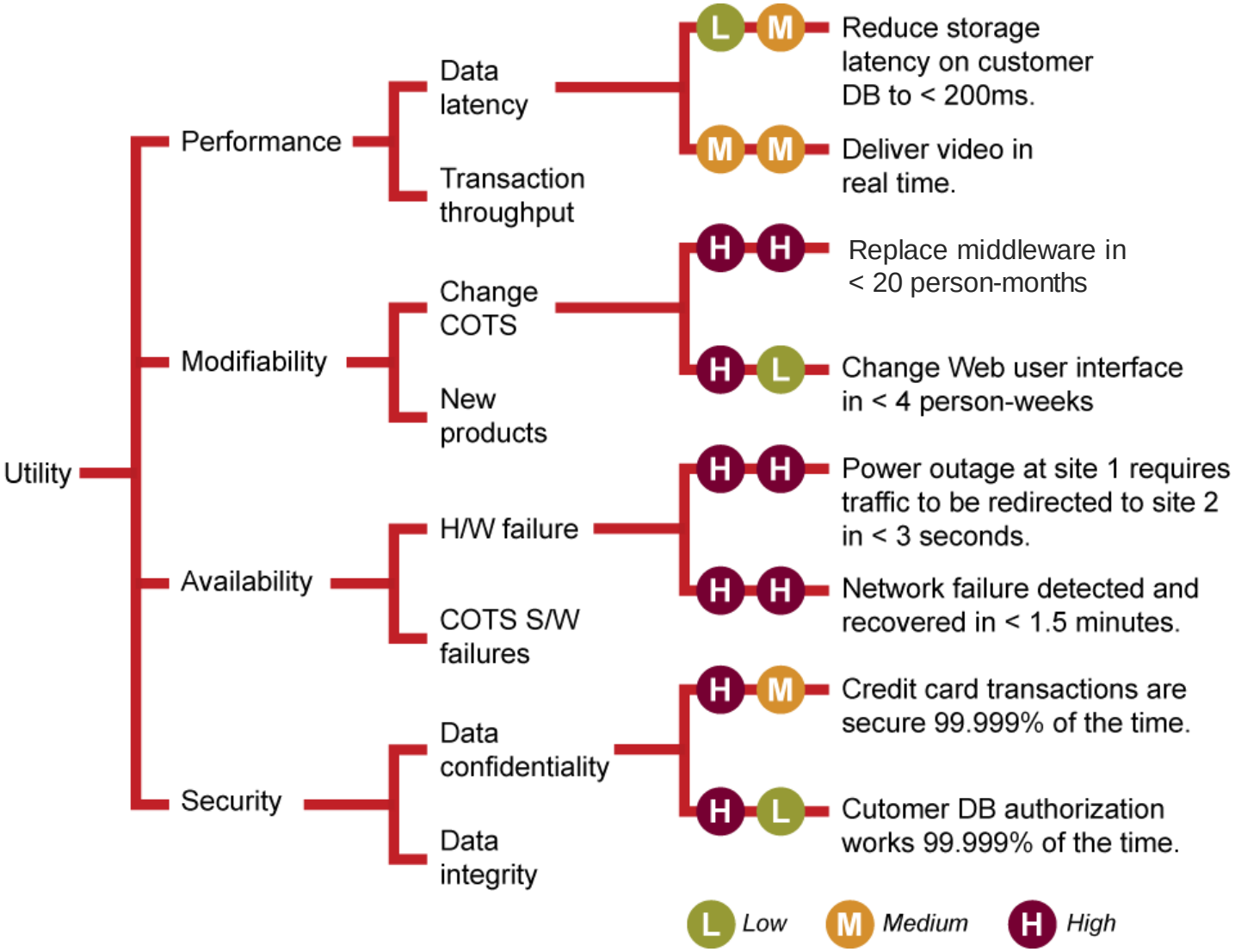
What Are Quality Attribute Utility Trees?

You can identify, prioritize, and refine the most important quality attribute goals by building a utility tree.

- A utility tree is a top-down vehicle for characterizing the “driving” attribute-specific requirements.
- The highest level nodes are typically quality attributes such as performance, modifiability, security, availability, and so forth.
- Scenarios are the leaves of the utility tree.

The utility tree is a characterization and a prioritization of specific quality attribute requirements.

Example of Quality Attribute Utility Tree



How Scenarios Are Used

Scenarios are used to

- represent stakeholders' interests
- understand quality attribute requirements

Scenarios should cover a range of

- anticipated uses of the system (use case scenarios)
- anticipated changes to the system (growth scenarios)
- unanticipated stresses on the system (exploratory scenarios)

Scenarios are linked to business goals, for traceability.

A good scenario clearly states the stimulus and the responses of interest.

Examples of Scenarios

Use case scenario

- A remote user requests a database report via the Web during a peak period and receives it within 5 seconds.

Growth scenario

- During maintenance, add an additional data server within 1 person-week.

Exploratory scenario

- Half of the servers go down during normal operation without affecting the overall system availability.

Scenarios should be as specific as possible.

Stimulus, Environment, Response

Use case scenario

- The remote user requests a database report via the Web during a peak period and receives it within 5 seconds.

Growth scenario

- During maintenance, add an additional new data server within 1 person-week.

Exploratory scenario

- Half of the servers go down during normal operation without affecting the overall system availability.

Quality Attribute Requirements – Elicitation Approaches

Goal: Broad coverage through stakeholder engagement and representation

Approaches:

- Quality Attribute Workshop – Original method, synchronous in-person collaborative working meeting
- Virtual QAW – Synchronous working telemeeting
- Interviews – Variation to avoid holding a single event, can by in-person and/or telemeeting
- Seeded Crowdsourcing – Create initial set of scenarios based on experience or interviews, open to broader asynchronous contributions

Quality Attribute Workshop (QAW)

The QAW is a facilitated method that engages system stakeholders early in the lifecycle to discover the driving quality attribute requirements of a software-reliant system.

Key points about the QAW are that it is

- system-centric
- stakeholder focused
- held before a major software architecture design exercise
- scenario based

Questions