SEI Software Architecture Principles and Practices Overview Training

Andrew Kotov and John Klein

Software Engineering Institute
Carnegie Mellon University
Pittsburgh, PA 15213

This material is based upon work funded and supported by the Department of Defense 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.

DM21-0242

Rules of Engagement

We will be very busy over the next three days. To complete everything and get the most from the course, we will need to follow some rules of engagement:

- Your participation is essential.
- Feel free to ask questions at any time.
- Discussion is good, but we may need to cut some discussions short in the interest of time.
- Please try to limit side discussions during the lectures.
- Please turn off your cell phone ringers and computers.
- Let's start on time.
- Participants must be present for all sessions to earn a course completion certificate.

Agenda

Day 1:

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

Day 2:

- Architecture Documentation: Views – Structure and Behavior, Principles of Sound Documentation, Architecture Decision Records
- Architecture-centric Engineering

Day 3:

- Architecture analysis
  - Evaluation approaches, lightweight evaluation
- Architecture design
  - Design process, Attribute-Driven Design

Logistics

Tuesday 16 Mar. 2021 → Thursday 18 Mar. 2021

- Start each day at 11:30 a.m. EDT
- End each day by 3:00 p.m. EDT
- Break every hour for 10 minutes

Zoom teleconference:

- https://sei.zoomgov.com/j/1605424743?pwd=QXAwdytsd0NUck9ieDREb2RYR3lYUT09&from=addon

  Meeting ID: 160 542 4743
  Passcode: 099303 One tap mobile
  +16692545252,,1605424743#,,,,*099303# US (San Jose)
  +16468287666,,1605424743#,,,,*099303# US (New York)

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.

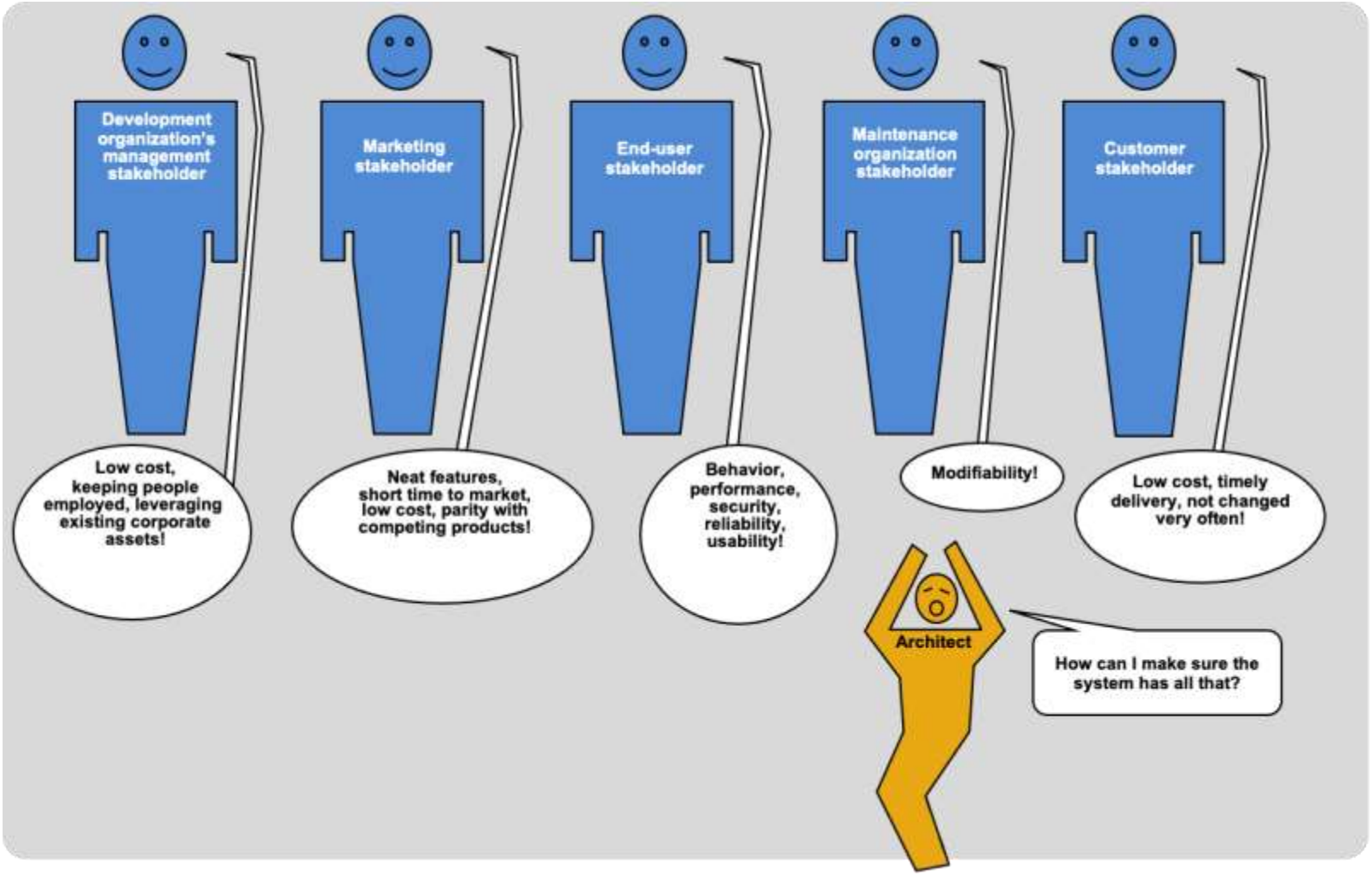
Architects need to know and understand the nature, source, and priority of these influences as early in the process as possible.

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)

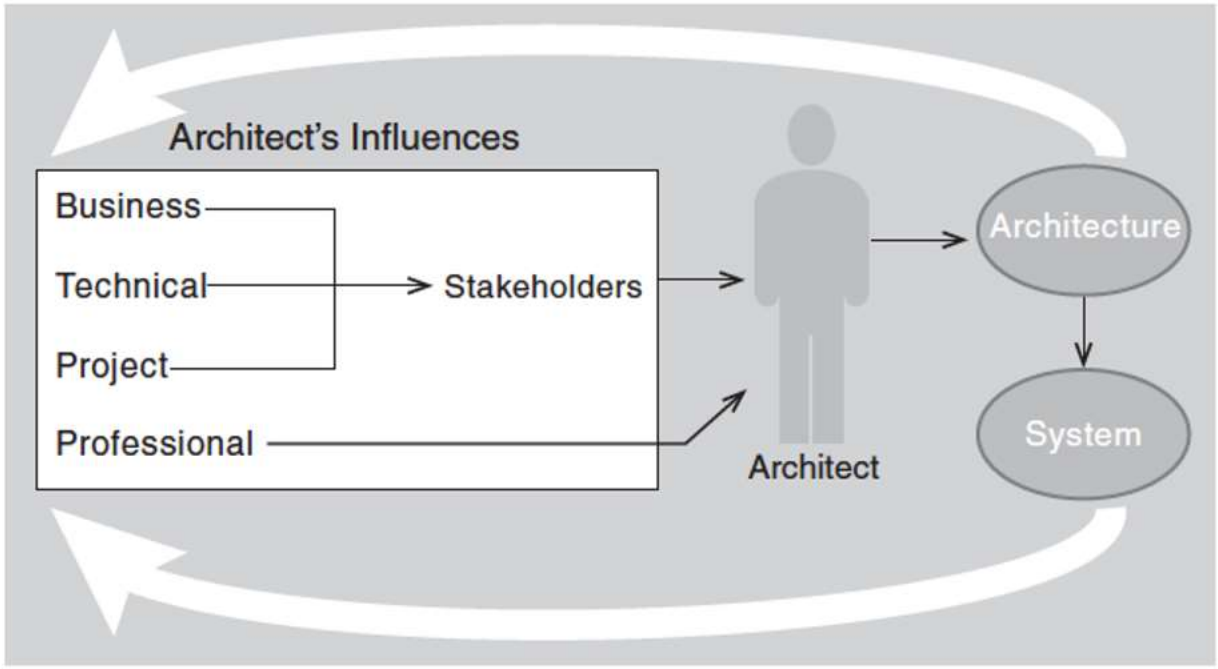


FIGURE 3.5 Architecture Influence Cycle

Source: Bass et al. *Software Architecture in Practice*. Addison-Wesley, 2012.

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

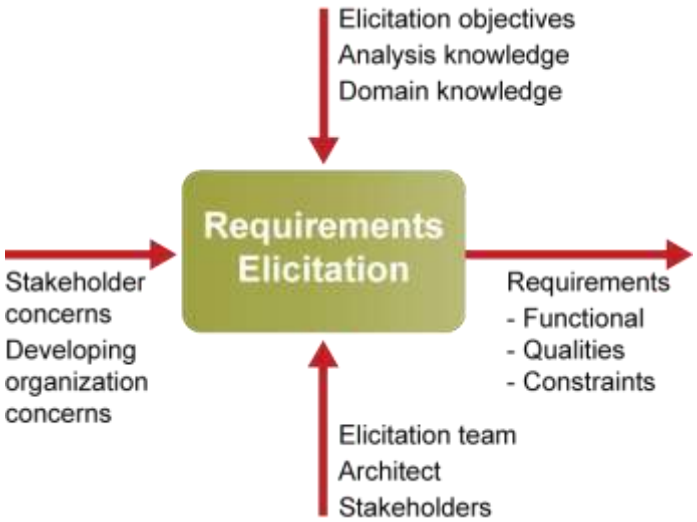
Architecture and Design Decisions

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

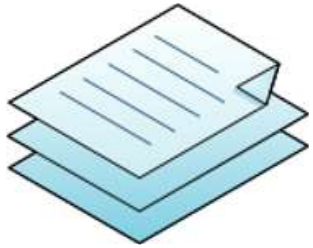
Requirements elicitation methods elicit quality attribute requirements.



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.

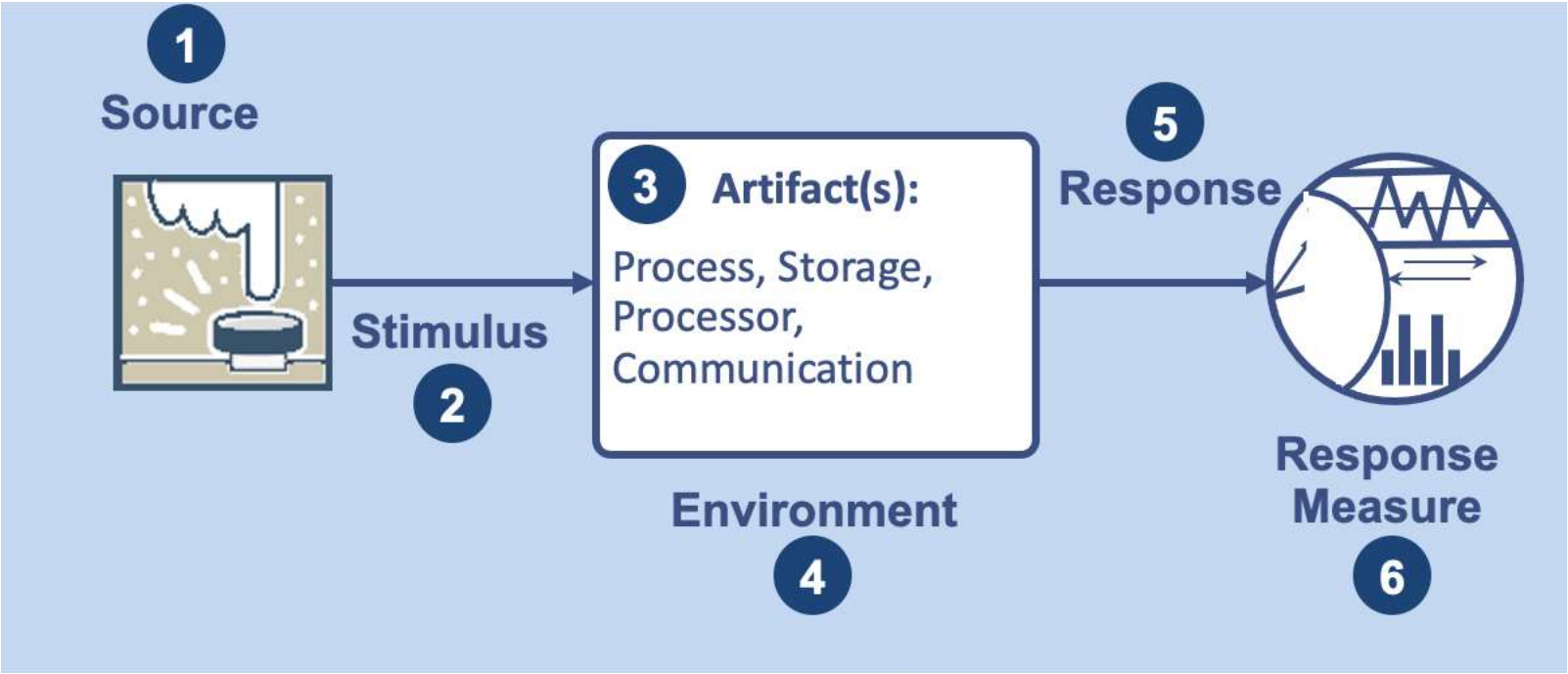
Describing Quality Attributes

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)

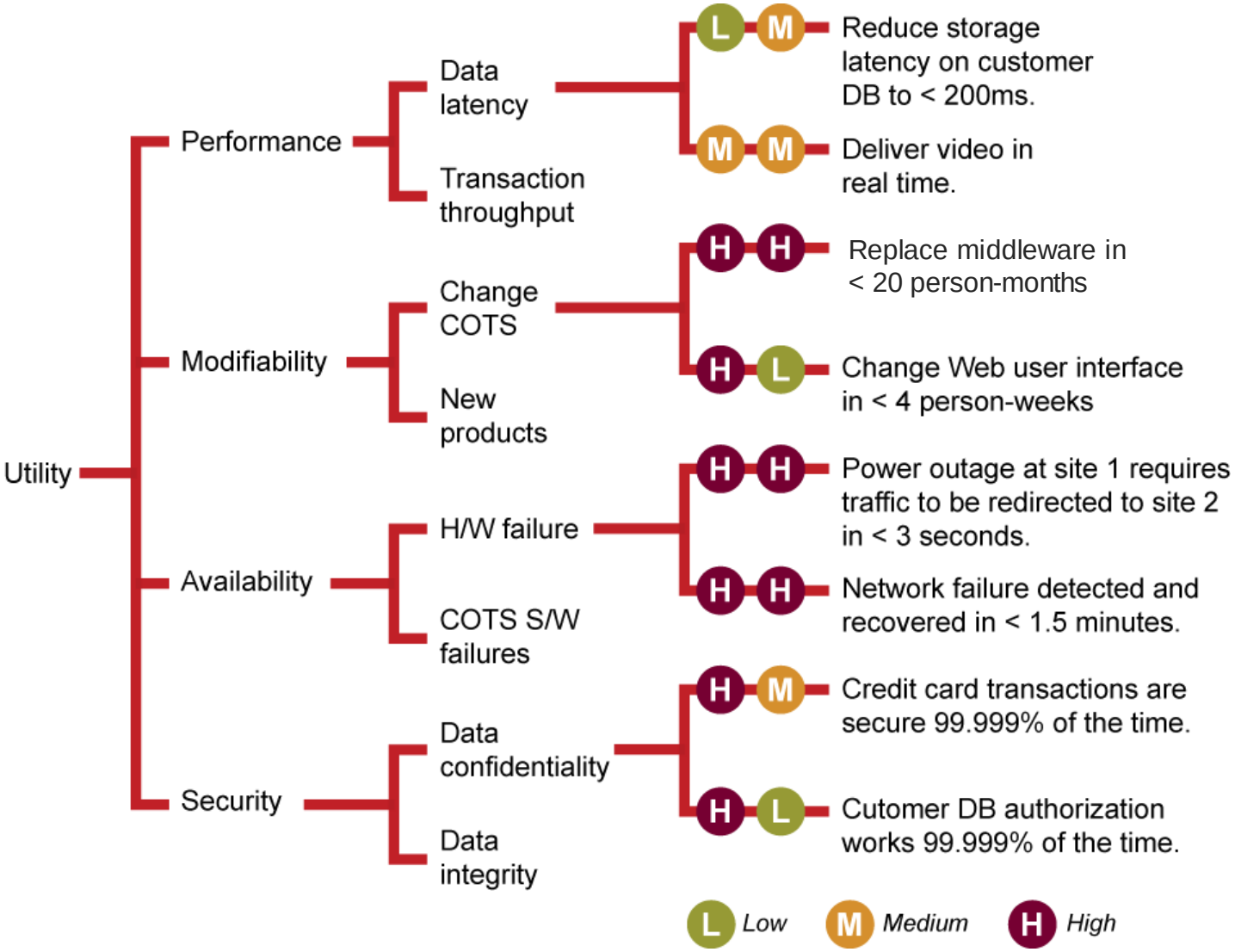
What Are Quality Attribute Utility Trees?

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

Exercise – Quality Attribute Scenarios for DIP System

Stimulus, Environment, Response

Start of Day 2

Software Engineering Institute
Carnegie Mellon University
Pittsburgh, PA 15213

Agenda

Day 1:

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

Day 2:

- Architecture Documentation: Views – Structure and Behavior, Principles of Sound Documentation, Architecture Decision Records
- Architecture-centric Engineering

Day 3:

- Architecture analysis
  - Evaluation approaches, lightweight evaluation
- Architecture design
  - Design process, Attribute-Driven Design

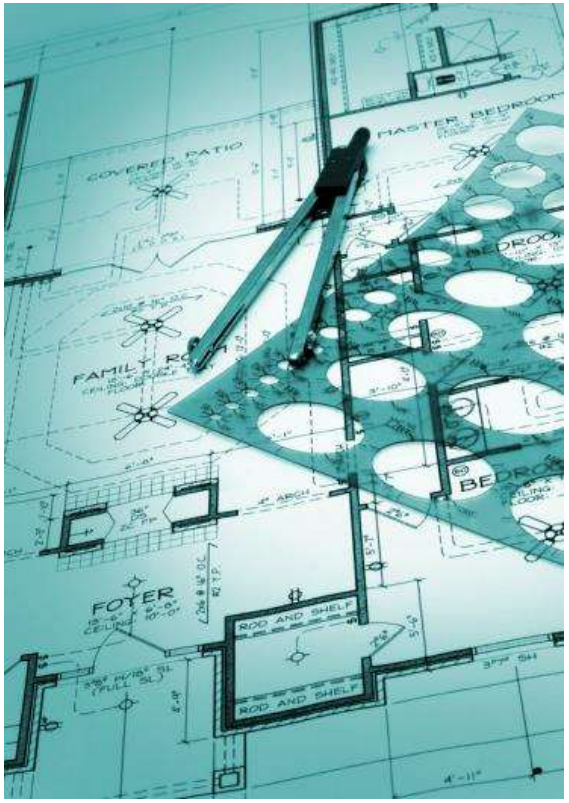
Structures, Views, Documentation

Software Engineering Institute
Carnegie Mellon University
Pittsburgh, PA 15213

Why Document an Architecture?

Architecture documentation has three fundamental uses.

- **Education**, introducing people to the system: new members of the team, external analysts, the customer, or even a new architect.
- **Communication** vehicle among stakeholders and to/from the architect.
- **Analysis**, especially for the quality attributes that the architecture is designed to provide.



Stakeholders for Documentation

Who are the stakeholders for architecture documentation?

What do they need to know?

A Business Case for Architecture Documentation

Benefit of documentation =

1. *Project activities will be less costly with high-quality, up-to-date documentation than they would otherwise.*

$$\sum_{\text{over all activities A}} \left(\begin{array}{l}\textbf{Cost of performing A } \textbf{\textit{without}} \\ \textbf{architecture documentation}\end{array} - \begin{array}{l}\textbf{Cost of performing A } \textbf{\textit{with}} \\ \textbf{architecture documentation}\end{array} \right)$$

2. *The effort saved from architecture documentation should outweigh the cost to create it.*

Benefit > Cost to produce/maintain the documentation.

Architectural Structures

Modern software systems are too complex to grasp all at once. At any moment, we restrict our attention to a small number of a software system's structures.

To communicate meaningfully about an architecture, we must make clear which structure or structures we are discussing.

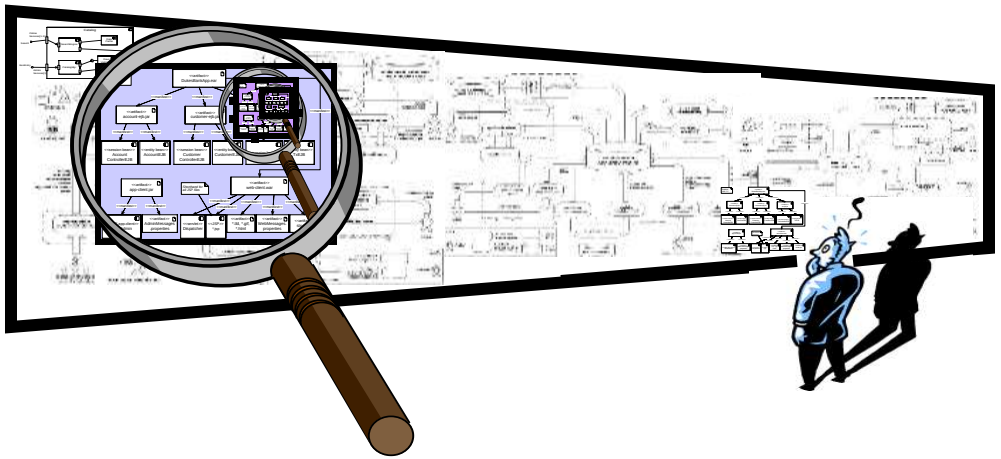
Architectural Structure Types

Architectural structures for software systems can be divided into three types:

1. **Module structures** – consisting of elements that are units of implementation
called modules and the relationships among them
2. **Component-and-connector structures** – consisting of runtime components
(units of computation) and the connectors (communication paths) between them
3. **Allocation structures** – consisting of software elements and their relationships to elements in external environments in which the software is created and executed

Documenting Software Architecture Using Views

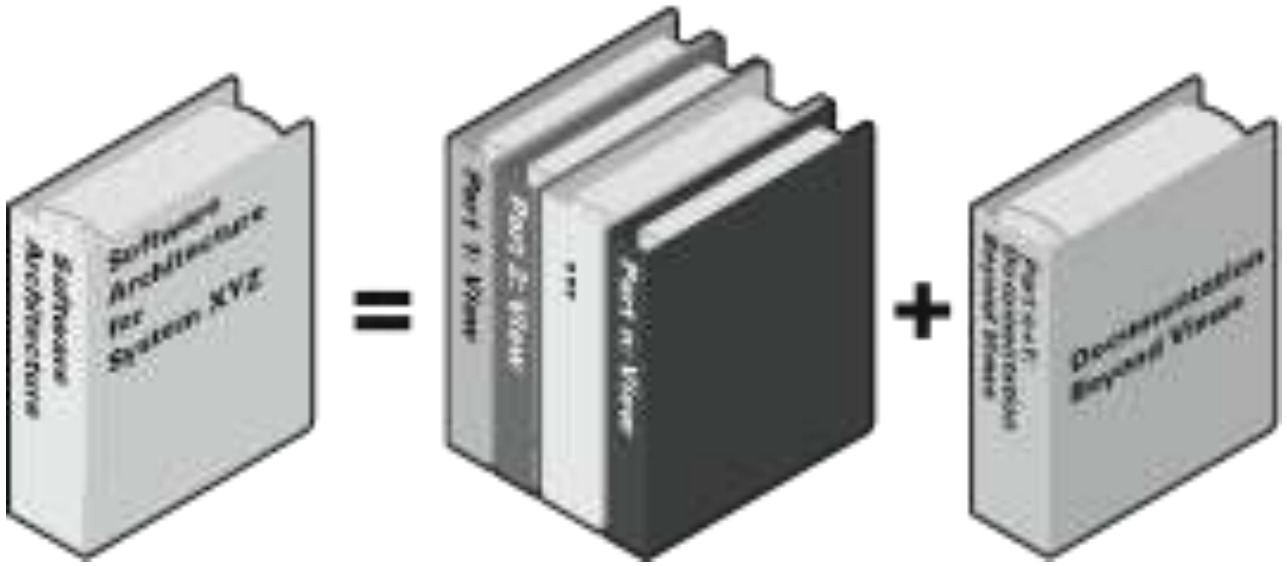
- Software architecture needs to be communicated to the team and key stakeholders. It must be documented!
- A single diagram that showed the entire architecture would contain too much information to present all at once.
- To break an architecture into digestible chunks, we create diagrams that we refer to as views.
- These views illustrate structures in the architecture.



View-Based Documentation

Views give us our basic principle of architecture documentation:

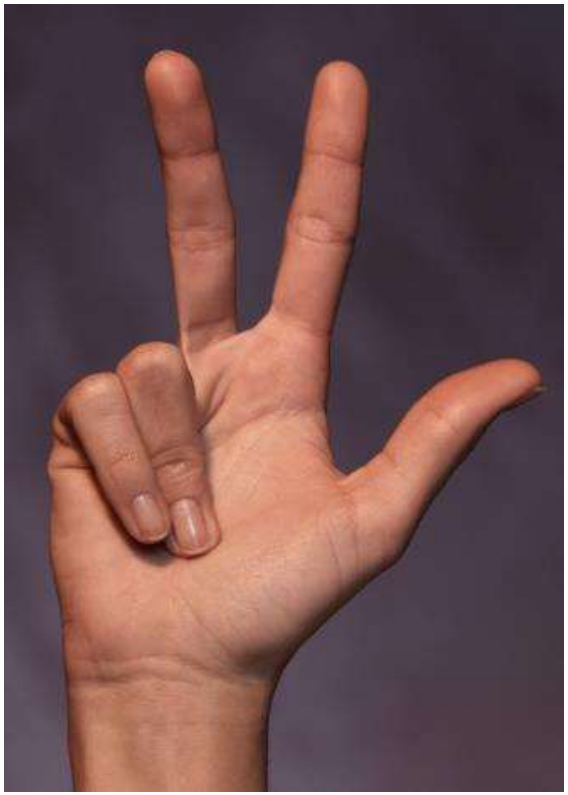
> Documenting a software architecture is a matter of documenting the relevant views and then adding information that applies to more than one view.



Recognizing Types of Views - 1

An architect must consider the system in three ways:

1. How is it structured as a set of implementation units?
2. How is it structured as a set of elements having runtime behaviors and interactions?
3. How does it relate to non-software structures in its environment?

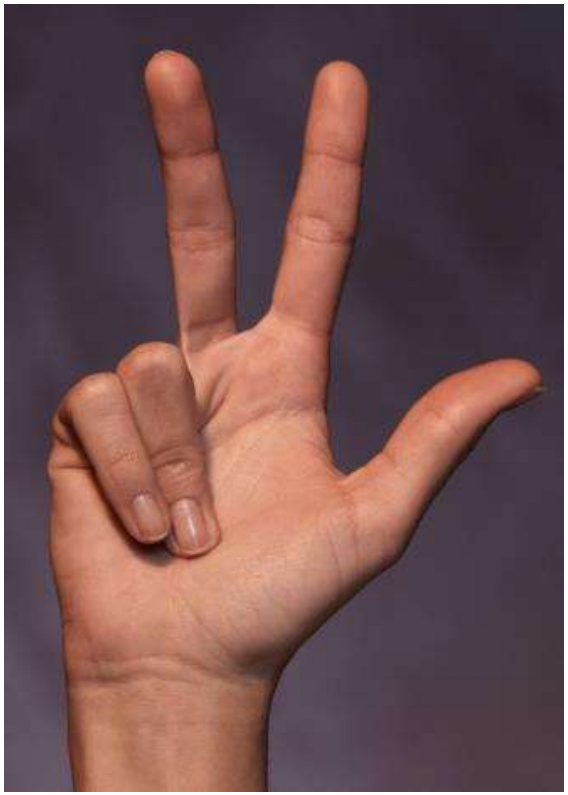


Recognizing Types of Views - 2

Different types of views show different types of information:

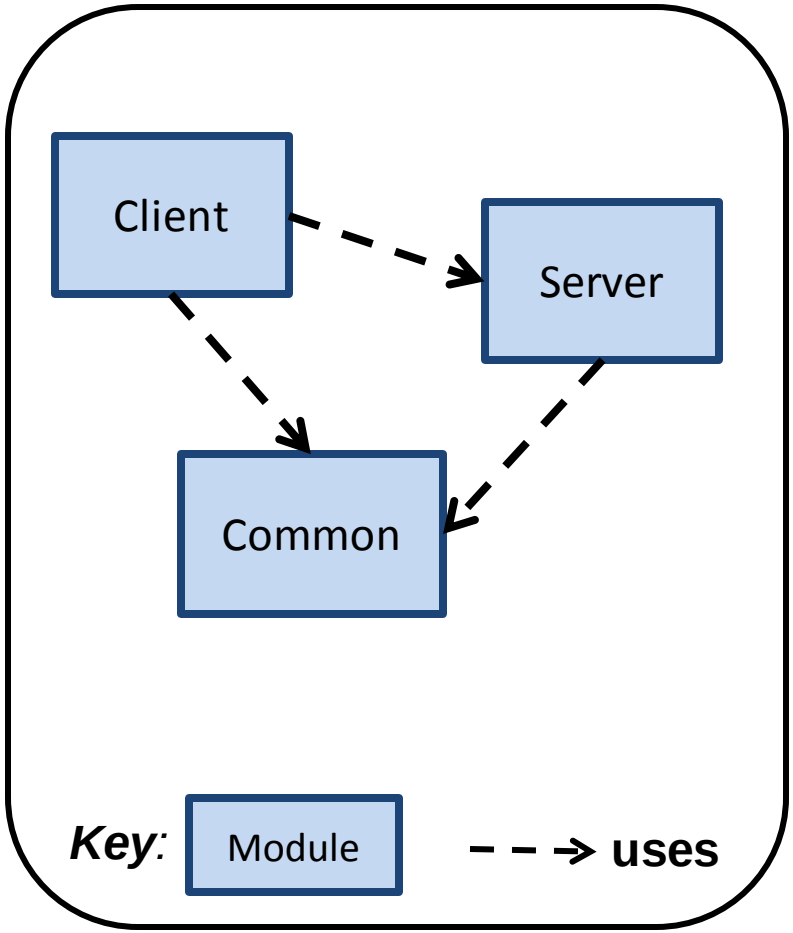
- Module views show how the system is structured as a set of implementation units.
- Component-and-connector views show how the system is structured as a set of elements with runtime behaviors and interactions.
- Allocation views show how the system relates to non-software structures in its environment.

Every view contains information from at least one of these categories.

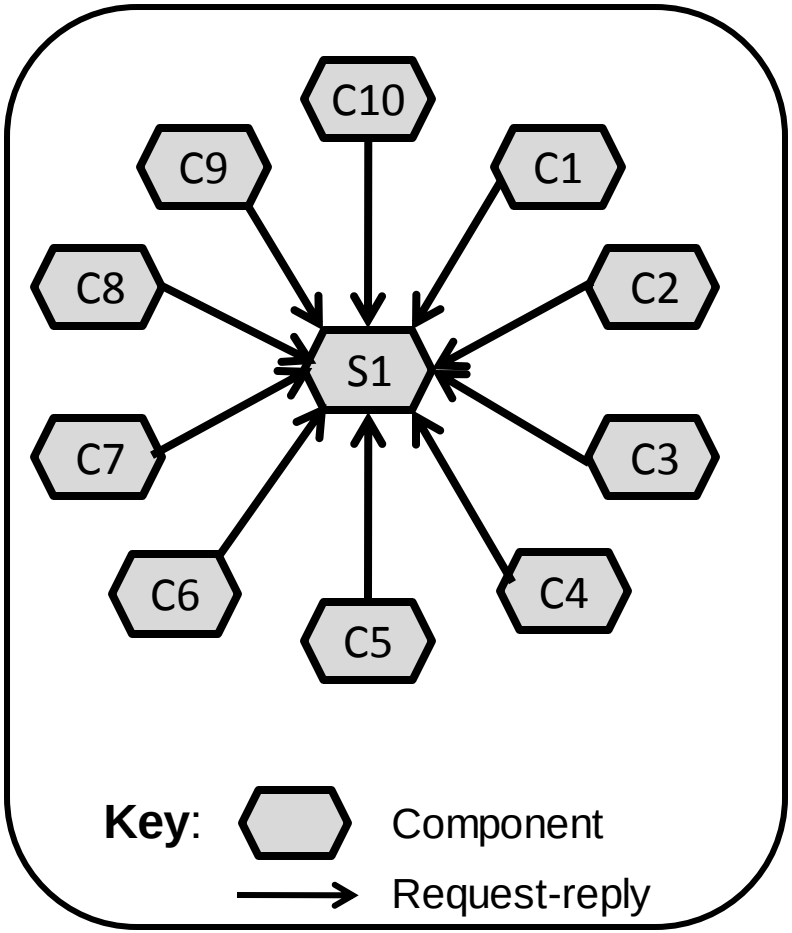


Modules vs. Components: A simple example

This client-server system has 3 modules and 11 components.



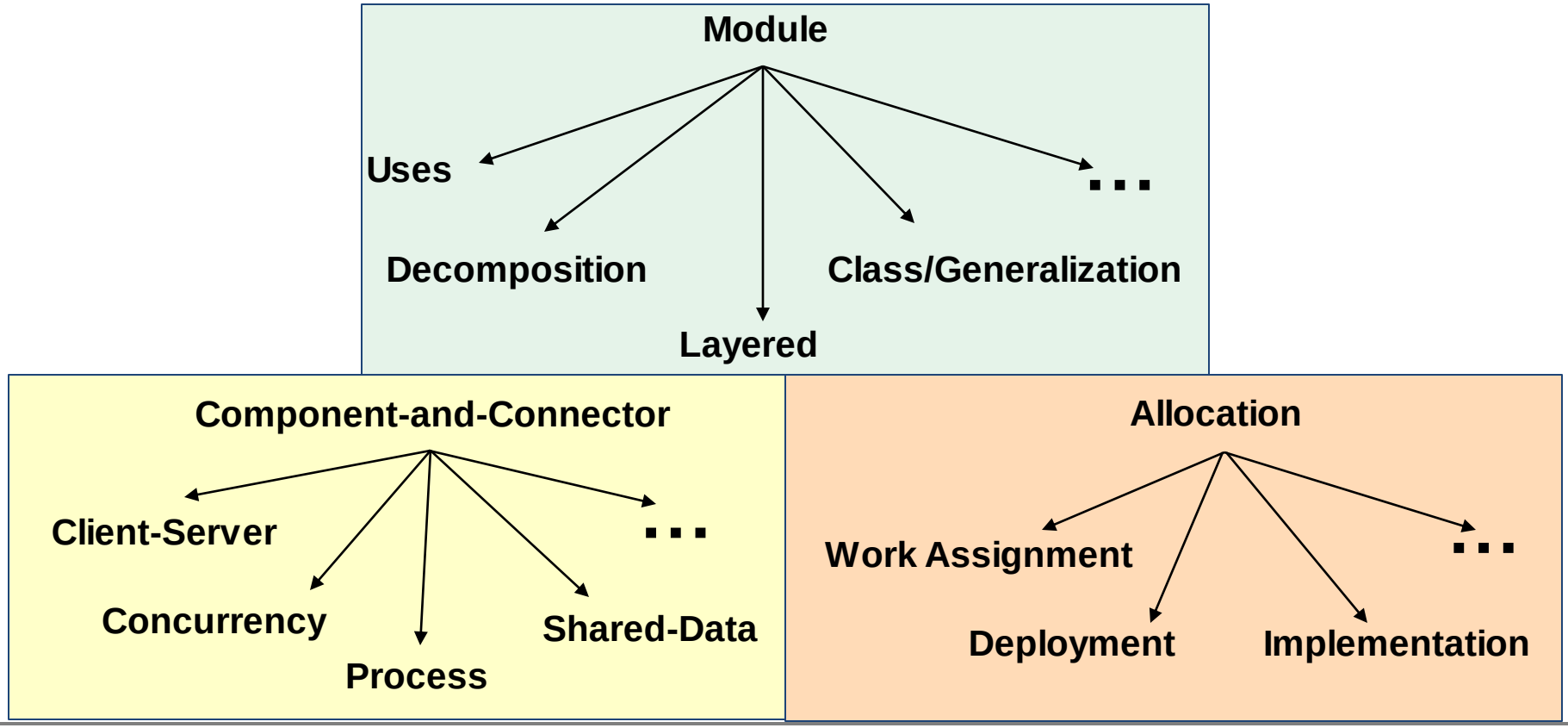
Module uses view



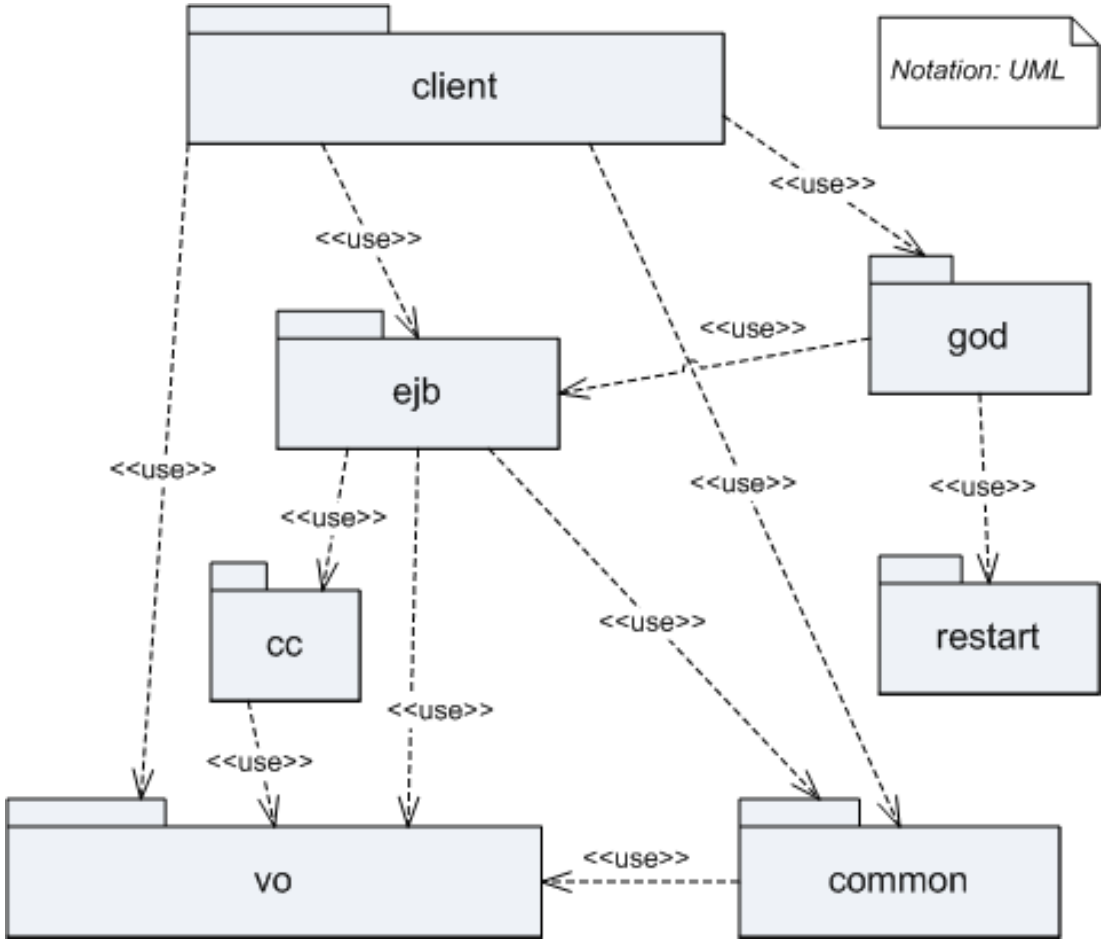
Client-server view

Standard View Types for Documenting Architecture

With a clear understanding of requirements, architects then select structures that promote those qualities. There are many to choose from!



Module View Example
Uses Style: UML

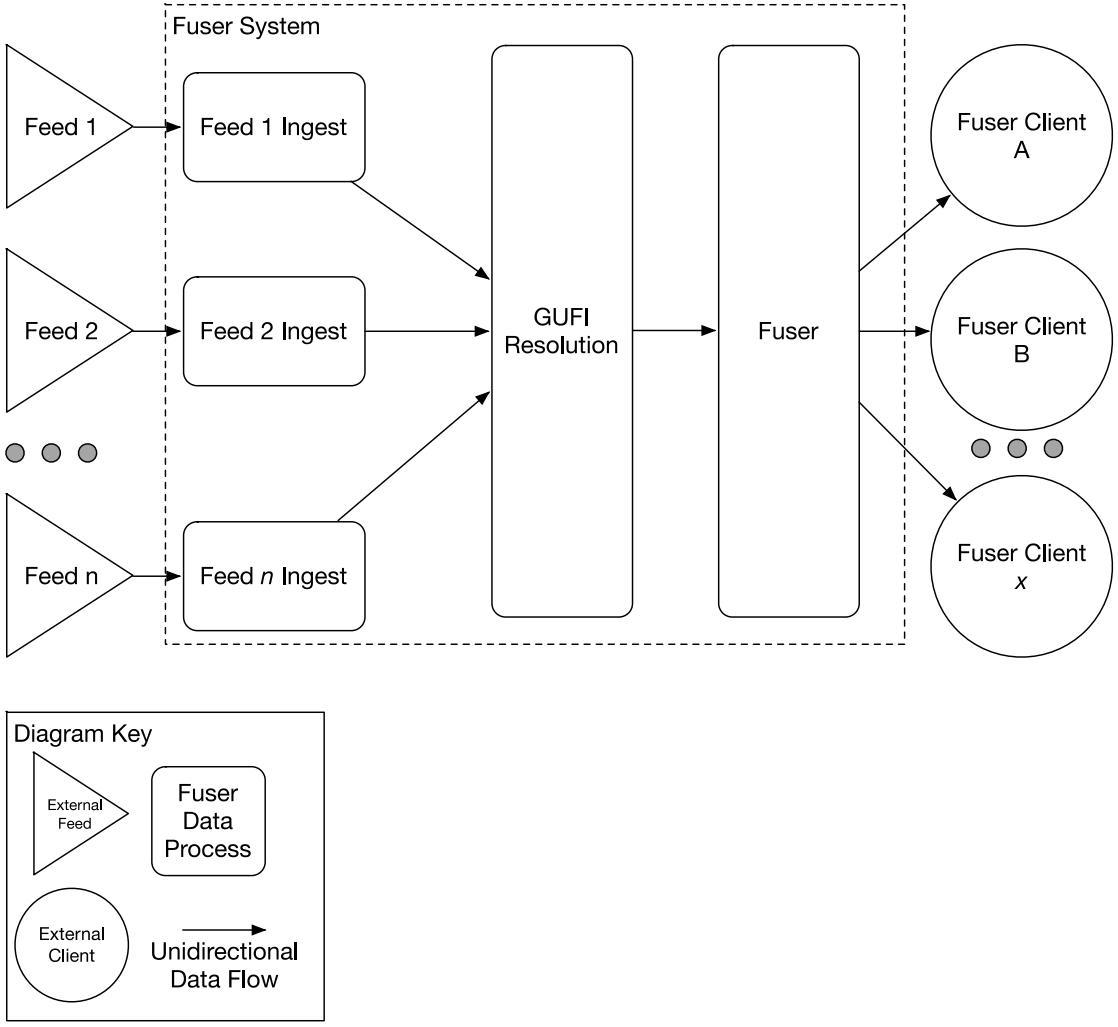


Module View Example – Extract from Fuser
Uses Style: Outline

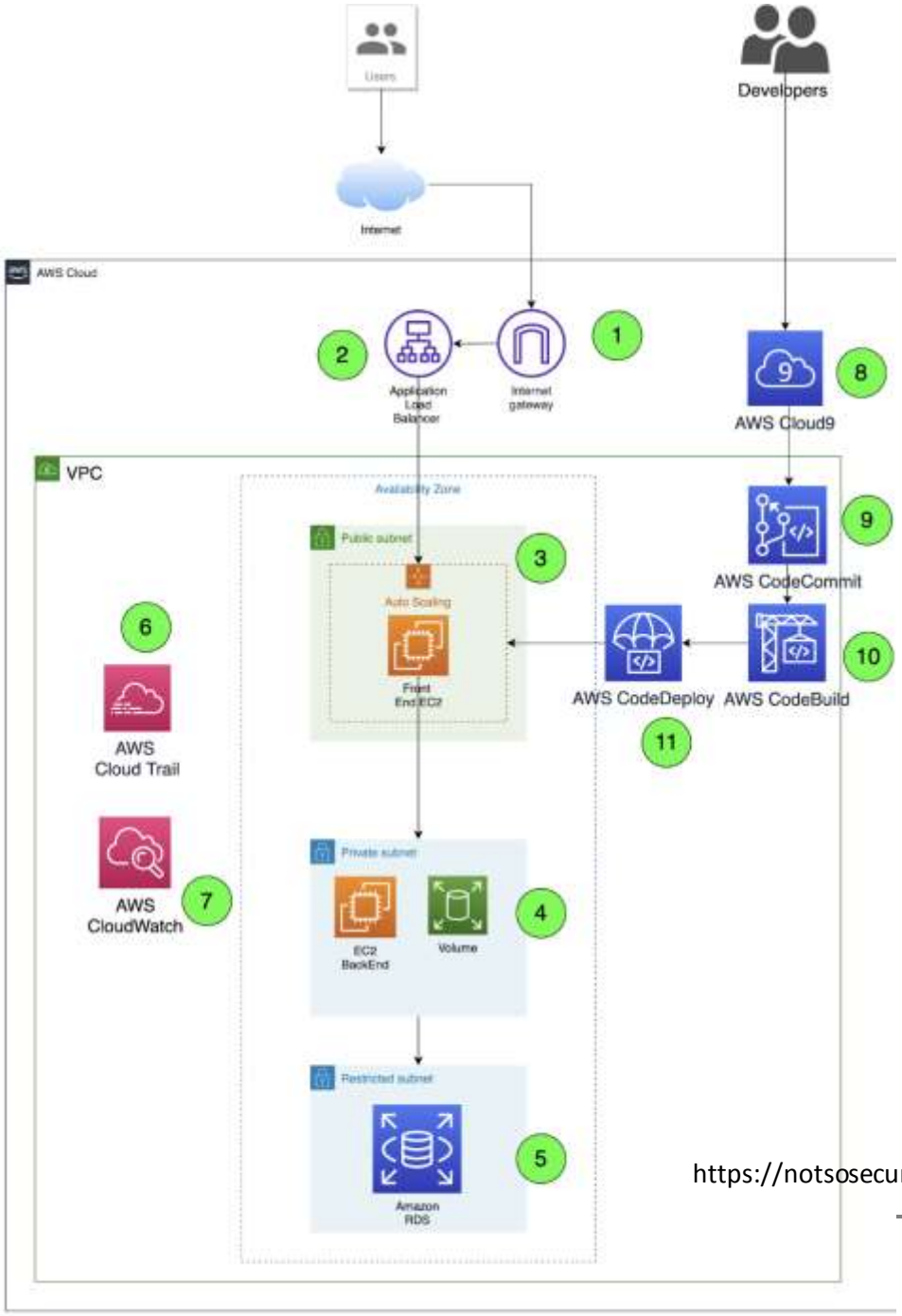
fuser-parent
+ com.codahale.metrics
+ com.google.gson
+ com.matm.actypelib
+ com.mosaicatm.adaptation
+ com.mosaicatm.adsb.data
+ com.mosaicatm.adsbplugin
+ com.mosaicatm.aircraftmanagement
+ com.mosaicatm.aodbplugin.matm
+ com.mosaicatm.aptcode
+ com.mosaicatm.container.io
+ com.mosaicatm.data.flightstate
+ com.mosaicatm.faa.util
+ com.mosaicatm.flighthubplugin.matm
+ com.mosaicatm.flightmanagement
+ com.mosaicatm.fmccommon
+ com.mosaicatm.fmcplugin
+ com.mosaicatm.guficlient
+ com.mosaicatm.lib.camel
+ com.mosaicatm.lib.coord
+ com.mosaicatm.lib.database
+ com.mosaicatm.lib.jaxb
+ com.mosaicatm.lib.messaging
+ com.mosaicatm.lib.playback
+ com.mosaicatm.lib.spring
+ com.mosaicatm.lib.text
+ com.mosaicatm.lib.time
+ com.mosaicatm.lib.util
+ com.mosaicatm.matmdata.aircraft
+ com.mosaicatm.matmdata.aircraftcomposite
+ com.mosaicatm.matmdata.airline
+ com.mosaicatm.matmdata.common
+ com.mosaicatm.matmdata.envelope
+ com.mosaicatm.matmdata.flight
+ com.mosaicatm.matmdata.fusersurveillance
+ com.mosaicatm.matmdata.heartbeat
+ com.mosaicatm.matmdata.position
+ com.mosaicatm.matmdata.positionenvelope
+ com.mosaicatm.matmdata.sector
+ com.mosaicatm.matmdata.util
+ com.mosaicatm.matmplugin.matm
+ com.mosaicatm.performancemonitor.common
+ com.mosaicatm.rmasplugin.matm
+ com.mosaicatm.rollingfile
+ com.mosaicatm.sector.geometry
+ com.mosaicatm.sfdps.data.transfer;
+ com.mosaicatm.sfdps.data.transfer
+ com.mosaicatm.sfdpsplugin.matm
+ com.mosaicatm.smes.transfer
+ com.mosaicatm.smesplugin.matm
+ com.mosaicatm.surveillanceplugin.matm
+ com.mosaicatm.tfm.thick.flight.mtfms.data
+ com.mosaicatm.tfmplugin.matm
+ com.mosaicatm.tfmplugin
+ com.mosaicatm.tfmtfdmplugin
+ com.mosaicatm.tma.common
+ com.mosaicatm.tmaplugin
+ com.mosaicatm.ttp.util
+ com.mosaicatm.ttpplugin
+ com.mosiacatm.ifile
+ org.apache.commons.logging

asdex-parent
+ com.mosaicatm.aptcode
+ com.mosaicatm.gufi.plugin
+ com.mosaicatm.guficlient
+ com.mosaicatm.gufiservice
+ com.mosaicatm.lib.bulk2
+ com.mosaicatm.lib.camel.jms
+ com.mosaicatm.lib.fitnesse
+ com.mosaicatm.lib.playback
+ com.mosaicatm.lib.text
+ com.mosaicatm.lib.wrap
+ com.mosaicatm.lib.xml
+ com.mosaicatm.rollingfile

Component-and-Connector View: Pipe-and-Filter Style



Deployment View



1. **Internet Gateway**: Allows communication between Virtual Private Cloud (VPC) and the internet.
2. **Application Load Balancer (ALB):** ALB manages traffic within the implementation and from external connections.
3. **Front-end Server (EC2):** Is placed in an auto-scaling group where the front-end web server is deployed on EC2 instances and serves as an application user interface (UI).
4. **Back-end Server**: Back-end web services are placed in a private subnet, deployed on EC2 instances.
5. **Amazon RDS (Database):** Application database is kept in a restricted subnet that interacts with back-end services.
6. **AWS CloudTrail:** Provides compliance, governance, operational and risk auditing of the AWS accounts.

https://notsosecure.com/security-architecture-review-of-a-cloud-native-environment/

Seven Principles of Sound Documentation

Certain principles apply to all documentation, not just that for software architectures.

Avoid Ambiguity - 1

Documentation is for communicating information and ideas.
If the reader misunderstands because of ambiguities,
the documentation has failed.

Even “simple” concepts can confuse. Here, what does the arrow mean?

- C1 calls C2
- Data flows from C1 to C2
- C1 instantiates C2
- C1 sends a message to C2
- C1 is a subtype of C2
- C2 is a data repository and C1 is writing data to C2
- C1 is a repository and C2 is reading data from C1

Avoid Ambiguity - 2

Precisely defined notations/languages help avoid ambiguity.

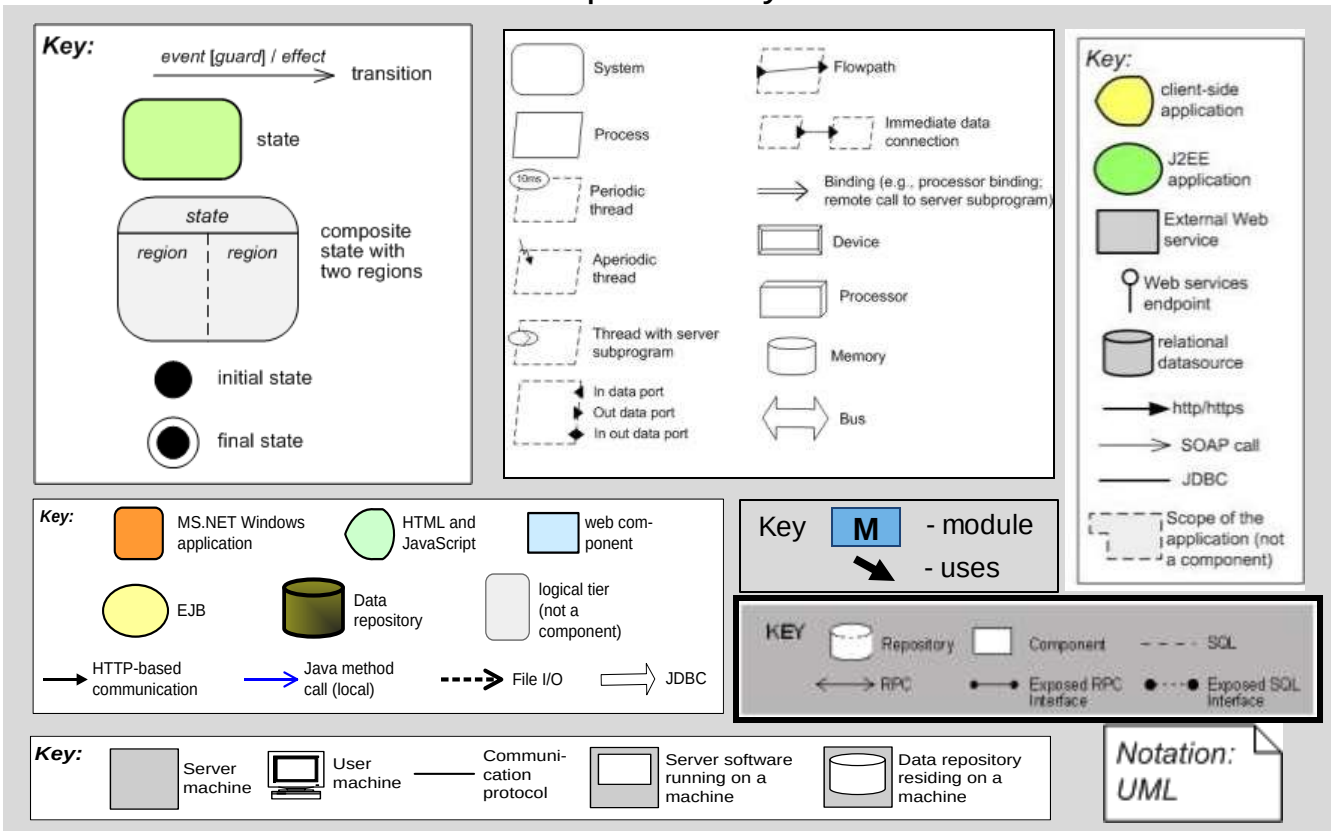
If your documentation uses a graphical language, always include a key!

- It can point to the language's formal definition
- It can give the meaning of each symbol. (Don't forget the lines!) If color or position is significant, indicate how.

Be sure to make the key meaningful: don't just say "element" and "relation." Different element and relation types should have different symbols.

Avoid Ambiguity - 3

Examples of keys

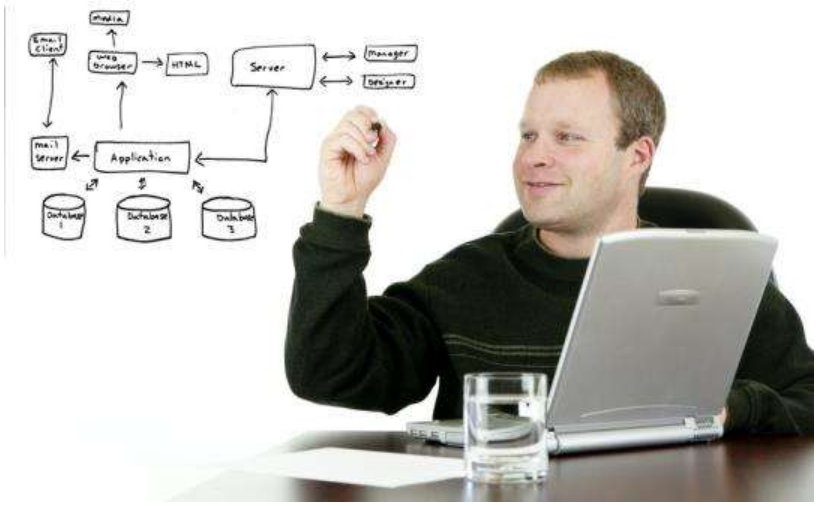


Avoid Ambiguity - 4

Box-and-line diagrams are a common form of architectural notation.

But what do they mean?

If you use a box-and-line diagram, always define precisely what the boxes and lines mean.



If you see an ambiguous box-and-line diagram, ask the owner what it means. (The result is often entertaining!)

Use a Standard Organization

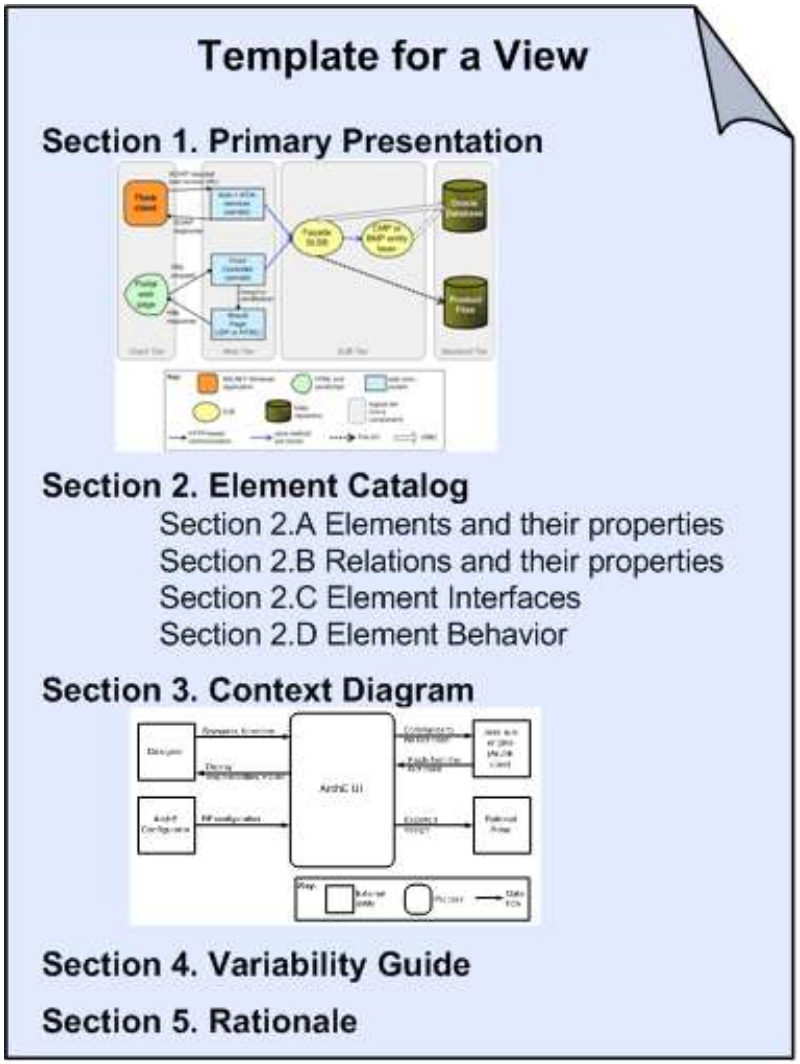
Establish it, make sure that your documents follow it, and make sure that readers know what it is.

A standard organization

- helps the reader navigate and find information
- tells the writer what to document, where it belongs,
- helps plan the work, and measure the work left to be done
- lets the writer record information as soon as it is known, in whatever order it is discovered
- embodies completeness rules and helps with validation

Examples:

- SEI Views and Beyond
- ISO-42010
- Others, e.g., arc42.org, Simon Brown's *Visualize* book, …



What Is the “Right” Set of Views?

Unlike approaches that prescribe a fixed set of views, Views and Beyond is a more general approach:

Choose the best views for each situation.

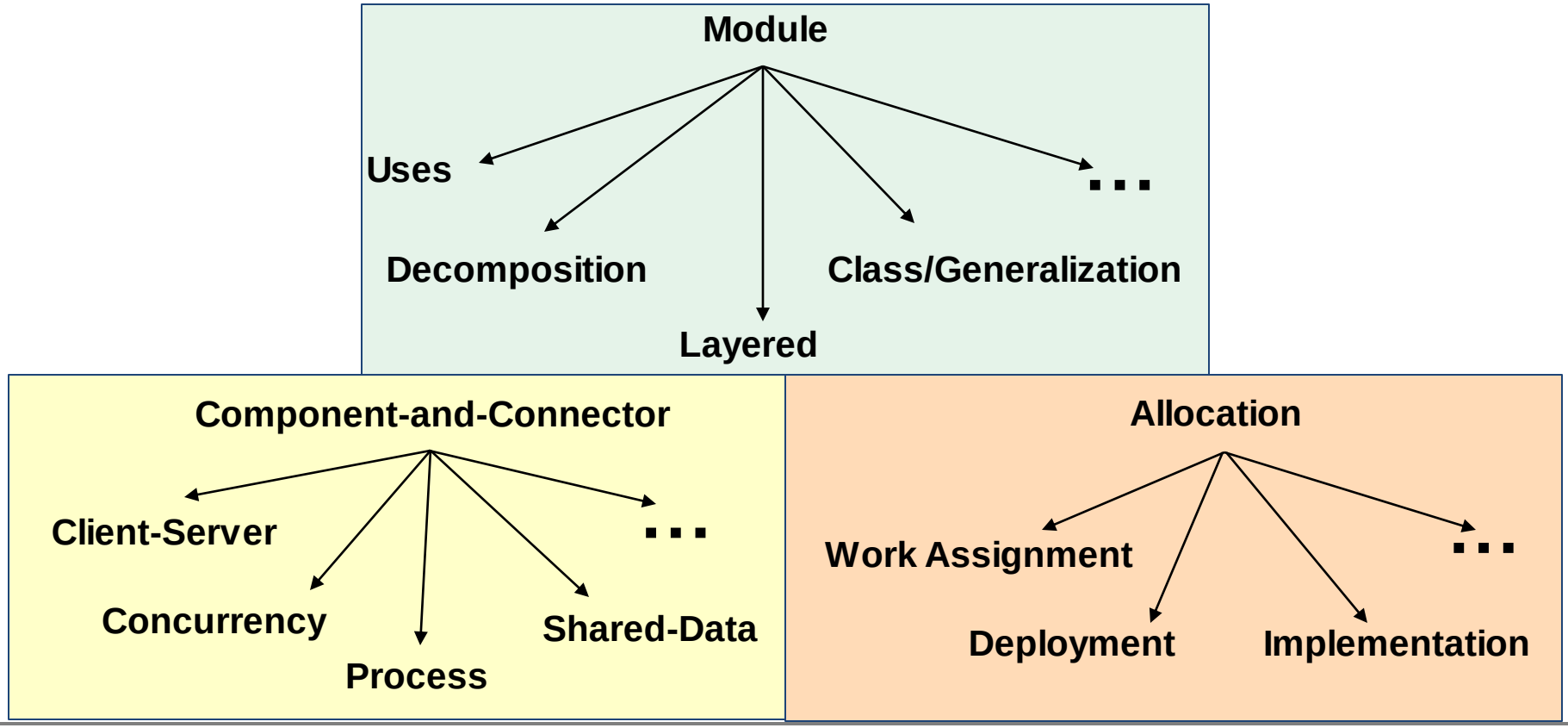
ISO/IEC 420101 concurs – It prescribes creating your own views to serve specific stakeholder concerns.

Which views are “right” depends on

- the structures that are inherent in the software
- who the stakeholders are and how they will use the documentation

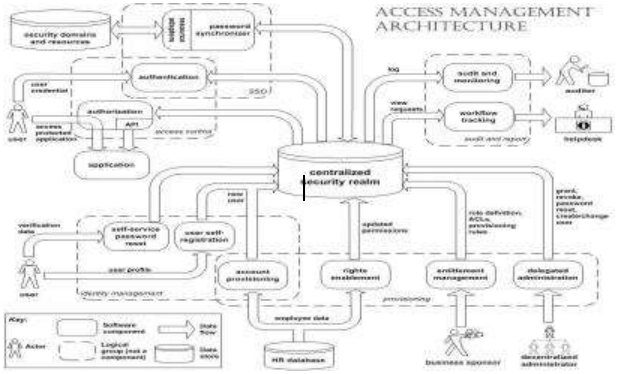
Standard View Types for Documenting Architecture

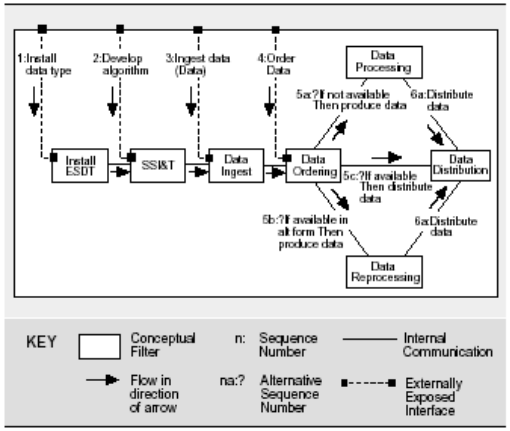
With a clear understanding of requirements, architects then select structures that promote those qualities. There are many to choose from!



Behavior: Beyond Structure

Structural diagrams show all the potential interactions among software elements.





Behavioral diagrams describe specific patterns of interaction—the system's response to stimuli.

Two Classes of Languages

Trace-oriented languages

- describe how the system reacts when a specific stimulus arrives and the system is in a specific state
- are easy to use because of their narrow focus
- do not completely capture behavior *unless* you collect all possible traces

Comprehensive languages

- show the complete behavior of a system
- are usually state based (e.g., statecharts)
- can be used to express all traces
- support the documentation of alternatives

Both can show the behavior of the whole system, parts of the system, or individual elements.

Traces can correspond to use cases, so behavioral documentation can show satisfaction of requirements.

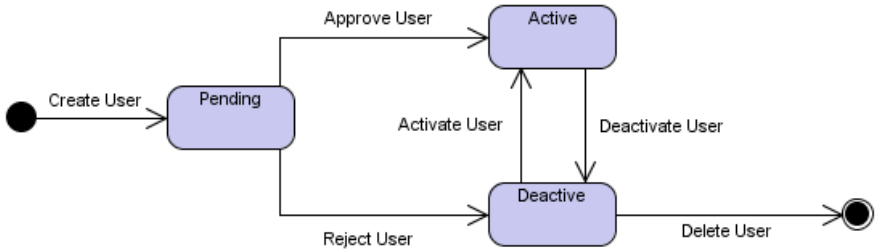
Which Is Which?

Trace-Oriented

- use cases*
- communication diagrams*
- sequence diagrams*
- message sequence charts
- activity diagrams*
- timing diagrams*
- Business Process Execution Language (BPEL) and Business Process Modeling Notation (BPMN)
- …

Comprehensive

- State machine diagrams*
- SDL diagrams
- Z specifications
- some ADLs
- CSP (communicating sequential processes)
- …



***** available in UML

Documenting Rationale: Architectural Decisions

Developing a complex software architecture involves making hundreds or thousands of big and small decisions.

The results of these decisions are reflected in the views that document the architecture—the structures with their elements and relations and properties, and the interfaces and behavior of those elements.

Rationale is the explanation of the reasoning that lies behind an architectural decision.

Architecture Decision Record (ADR) Template - 1

1. Issue. State the architectural design issue being addressed.
2. Decision. State the solution chosen among the alternatives that the architect evaluated.
3. Status. State the status of the decision, such as pending, decided, or approved. (This is not the status of implementing the decision.)
4. Group. Name a containing group. Grouping allows for filtering based on the technical stakeholder interests. Examples: “integration,” “presentation,” “data,” etc.
5. Assumptions. Describe the key assumptions under which a decision was made:
   - About the environment. Examples: accepted technology standards, an enterprise architecture, commonly employed patterns, team size and skill set available, cost and schedule, etc.
   - About need. Example: “The system will only be used in the USA” (justifies why the design has no support for internationalization)

Architecture Decision Record (ADR) Template - 2

6. Alternatives. List alternatives (that is, options or positions) considered. Explain with sufficient detail to judge their suitability. Listing alternatives espoused by others helps them know that their opinions were heard.
7. Argument. Outline why a position was selected. This can include items such as implementation cost, total cost of ownership, time to market, and availability of required development resources.
8. Implications. Describe the decision's implications. E.g., it may
   - Introduce a need to make other decisions
   - Create new requirements or modify existing requirements
   - Pose additional constraints to the environment
   - Require renegotiation of scope or schedule

Architecture Decision Record (ADR) Template - 3

9. Related Decisions. List decisions related to this one. Useful relations among decisions include causality (which decisions caused other ones), structure (showing decisions' parents or children), or temporality (which decisions came before or after others).
10. Related Requirements. Map decisions to objectives or requirements, to show accountability.
11. Affected Artifacts. List the architecture elements and/or relations affected by this decision. You might also include external artifacts upstream and downstream of the architecture, as well as management artifacts such as budgets and schedules.
12. Notes. Capture notes and issues that are discussed during the decision process.

As in all cases, choose the parts of this template appropriate for your situation.

Exercise – View selection for DIP

Who is going to use the architecture documentation? For what purpose? What types of views would be beneficial?

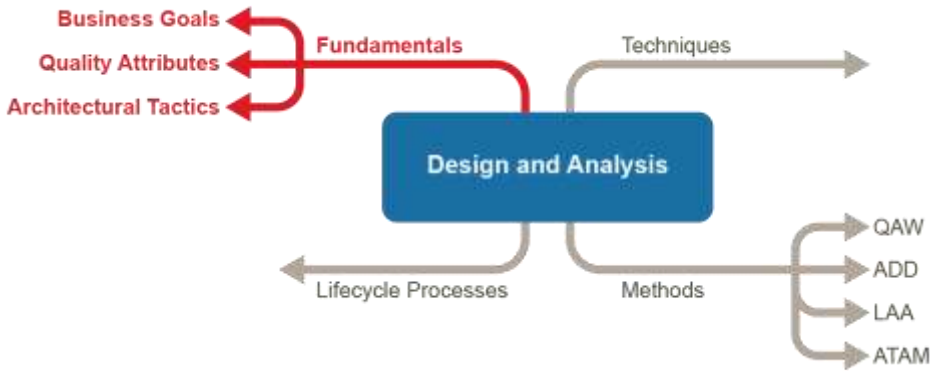
Architecture design and analysis

Software Engineering Institute
Carnegie Mellon University
Pittsburgh, PA 15213

Architecture-Centric Approach

The SEI architecture-centric approach can add value, building on the fundamental concepts of

- *business goals*: Systems are built to satisfy business goals. Business goals determine requirements.
- *quality attributes*: Quality attribute requirements exert the strongest influence on architectural design.
- *architectural design primitives*: Architectural tactics and patterns are the basic building blocks of design used by practitioners.



What Is Design and Analysis?

Design is making the decisions that lead to the creation of architecture.

- Which design decisions will lead to a software architecture that successfully addresses the desired system qualities?

Analysis ensures that the architecture used is the right one.

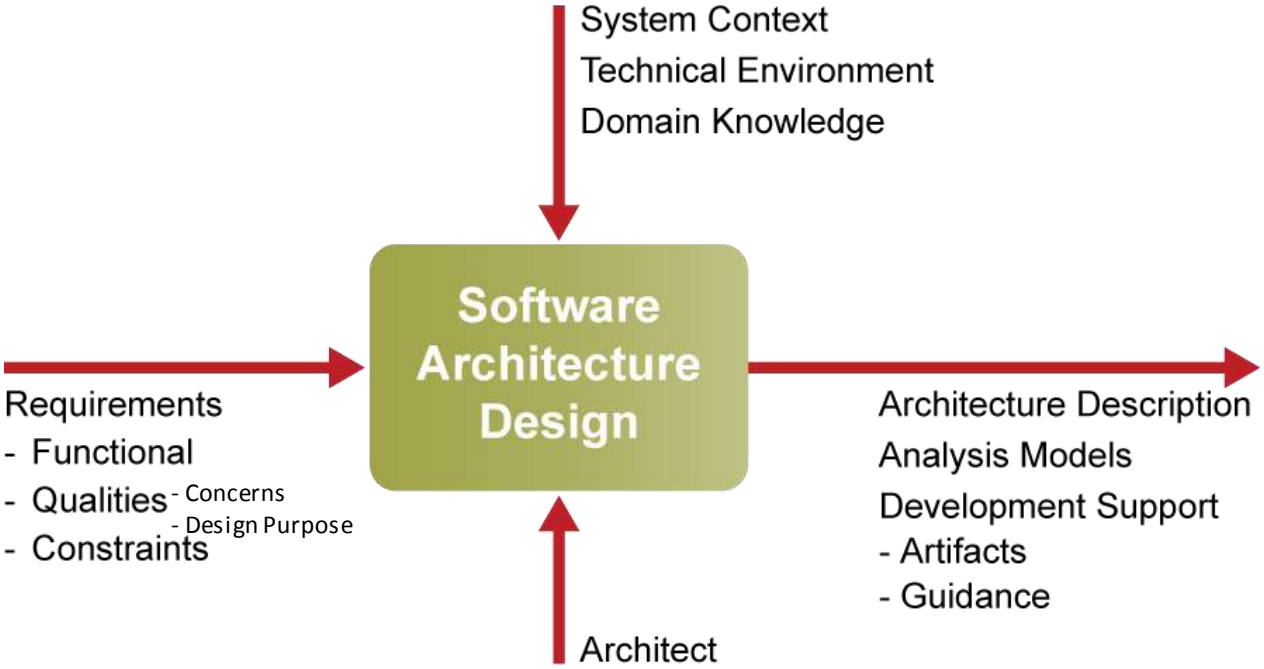
- How do you know if a given software architecture is deficient or at risk relative to its target system qualities?

Implications for Software Architecture Design and Analysis

The degree to which a system meets its quality attribute requirements is dependent on architectural decisions.

- A change in structure improving one quality often affects the other qualities.
- Architecture is critical to the realization of quality attributes. These quality attributes should be designed into the architecture.
- Architecture can only permit, not guarantee, any quality attribute.

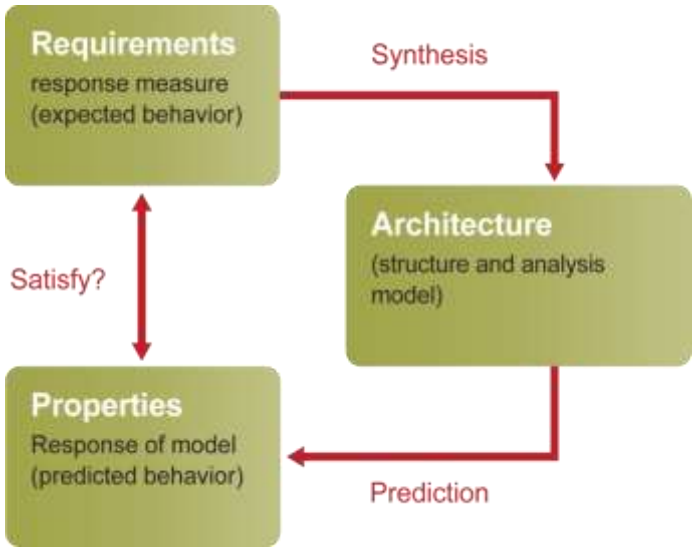
Use of Analysis in Design Methods



For More Information: Hofmeister, C.; Kruchten, P.; Nord, R.L.; Obbink, H.; Ran, A.; America, P. "A General Model of Software Architecture Design Derived from Five Industrial Approaches." *Journal of Systems and Software*, 2007.

Architectural Analysis and Design

Architectural analysis and design are tightly coupled.

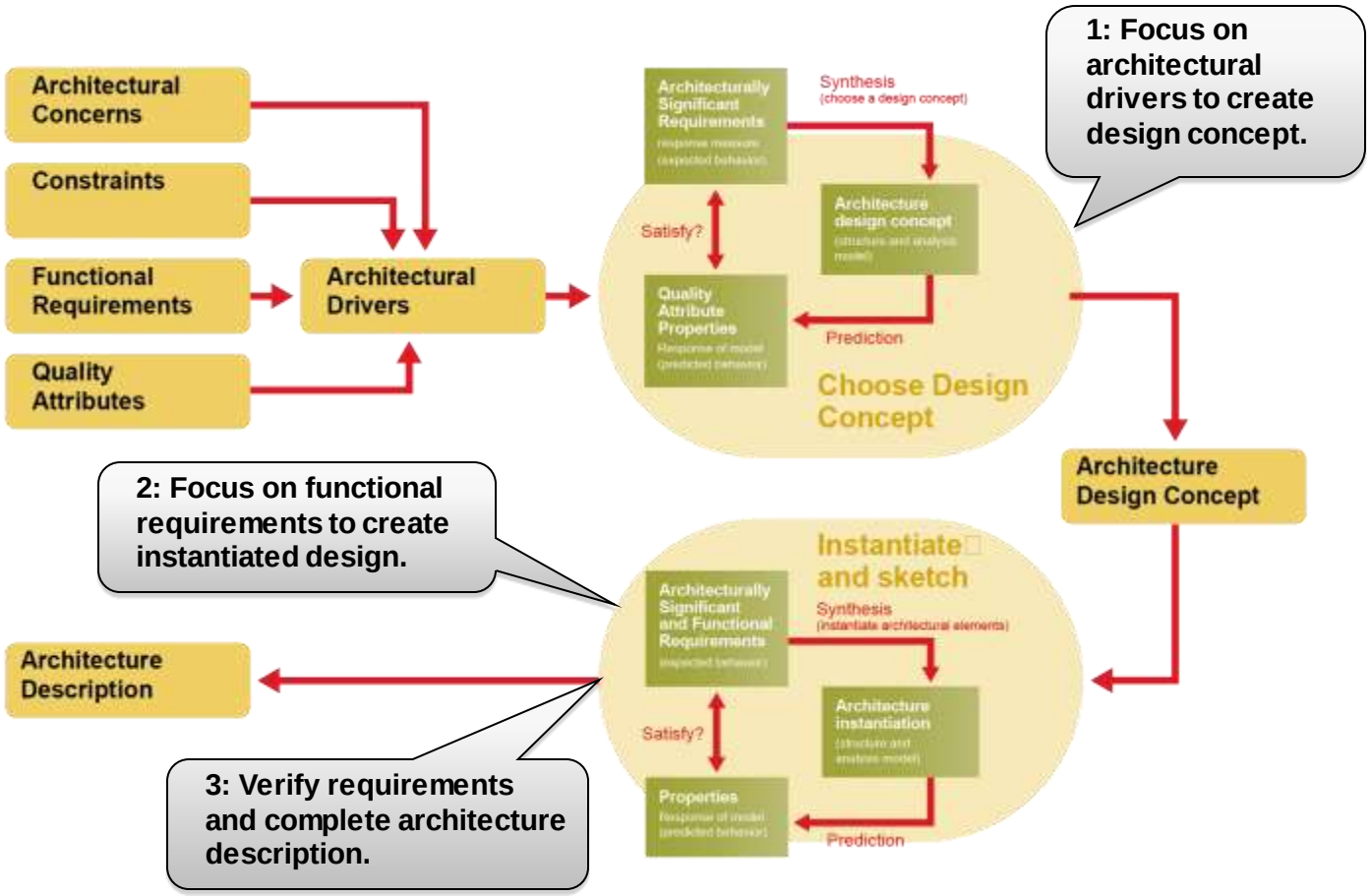


The word *analysis* is defined by Merriam-Webster as

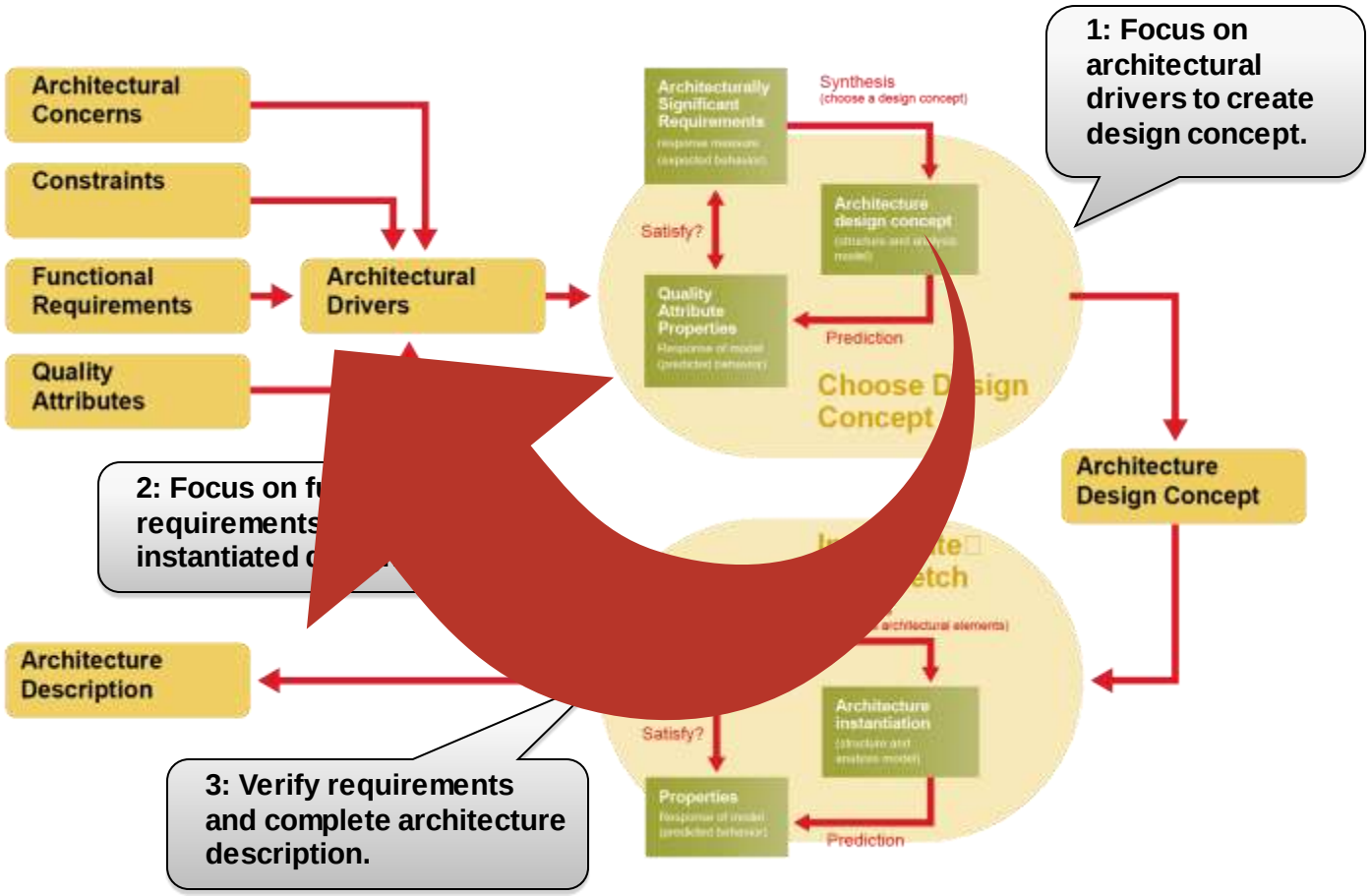
- the careful study of something to learn about its parts, what they do, and how they are related to each other
- an explanation of the nature and meaning of something

Both of these definitions apply to architecture analysis.

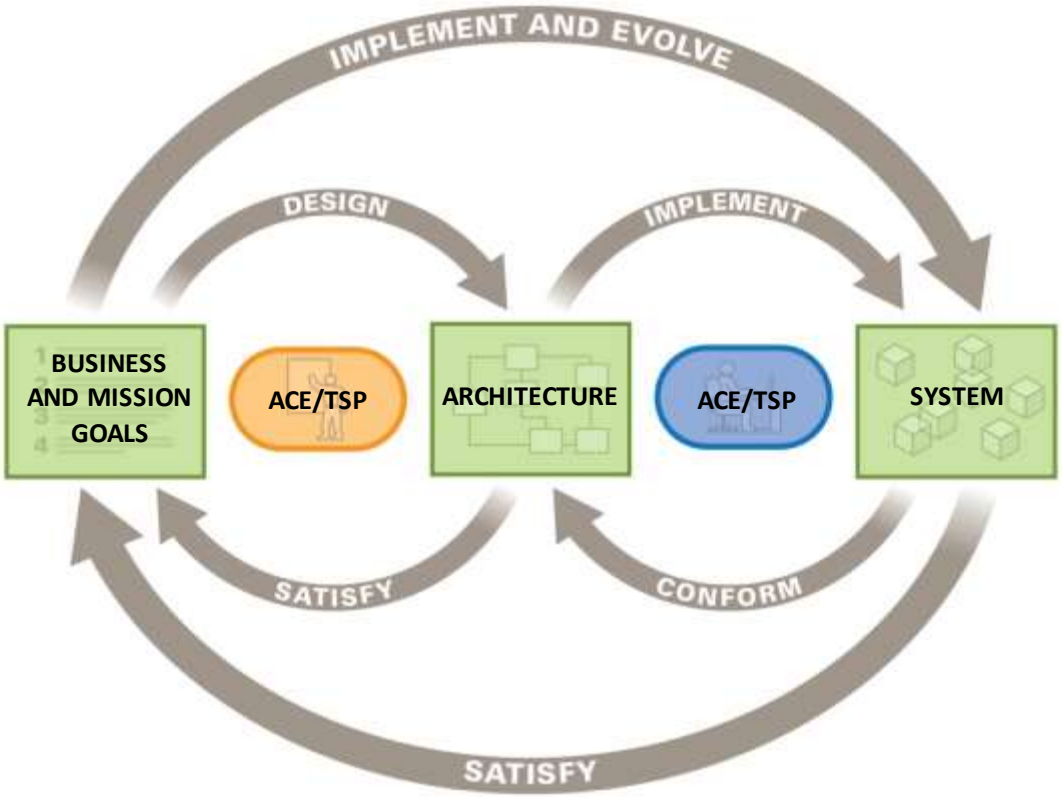
Analysis and Documentation as a Portion of Design



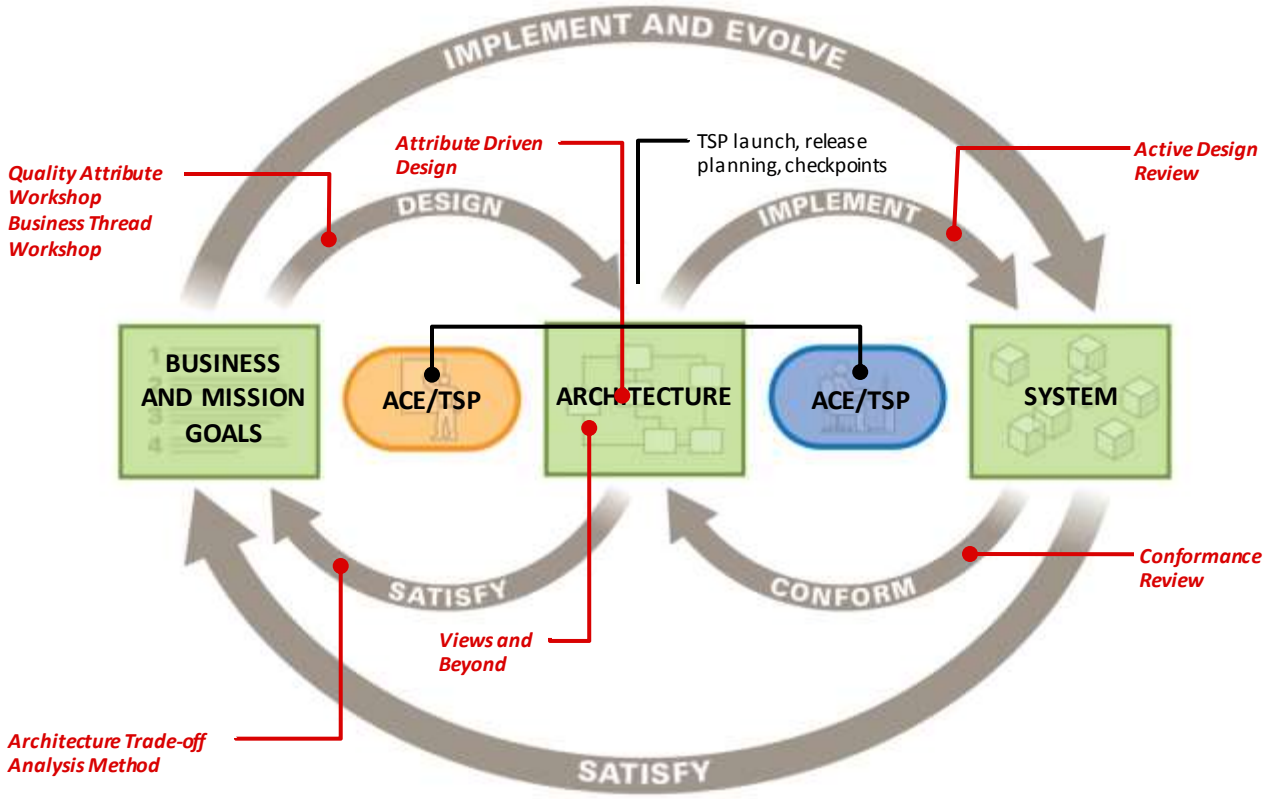
Analysis and Documentation as a Portion of Design – Iterative Process



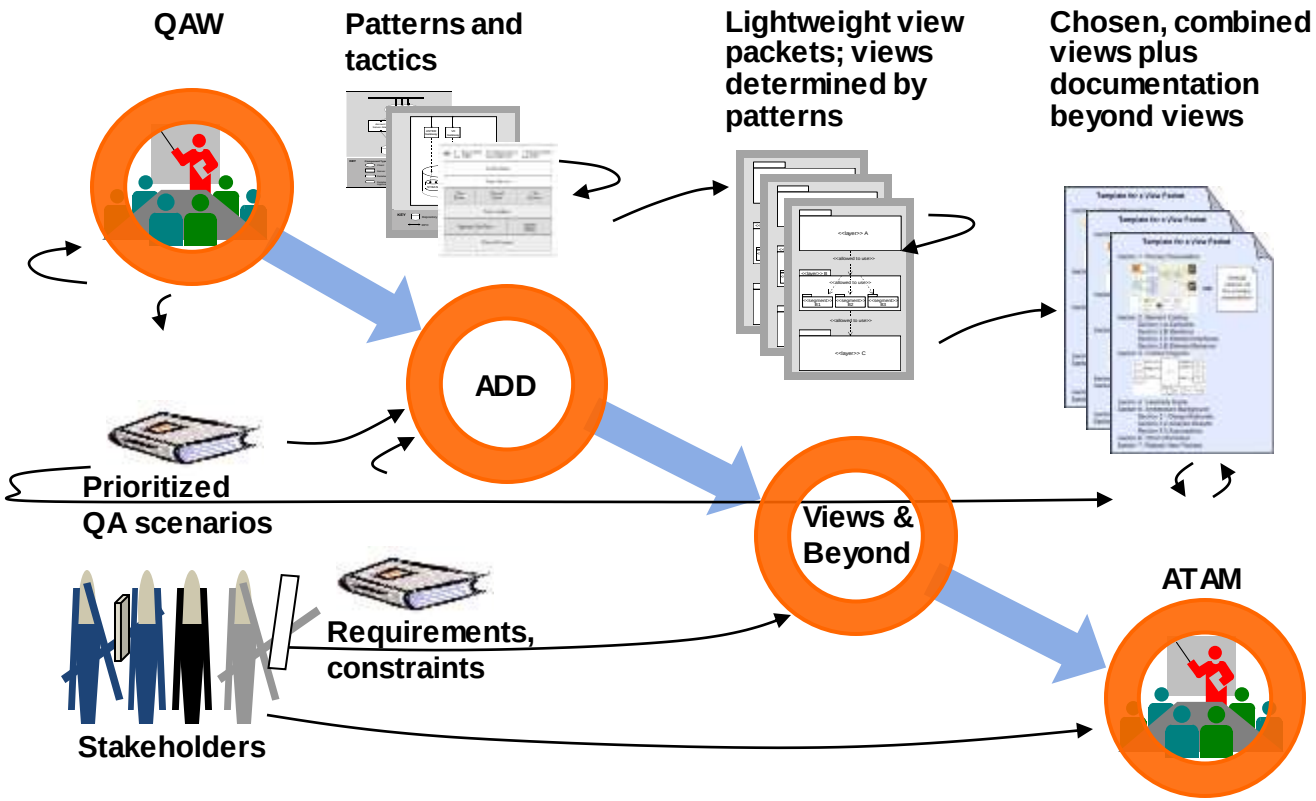
Architecture Centric Engineering



Architecture Centric Engineering - Methods



QAW, ADD, V&B, and ATAM Together



Start of Day 3

Software Engineering Institute
Carnegie Mellon University
Pittsburgh, PA 15213

Agenda

Day 1:

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

Day 2:

- Architecture Documentation: Views – Structure and Behavior, Principles of Sound Documentation, Architecture Decision Records
- Architecture-centric Engineering

Day 3:

- Architecture analysis
  - Evaluation approaches, lightweight evaluation
- Architecture design
  - Design process, Attribute-Driven Design

Architecture Analysis

Software Engineering Institute
Carnegie Mellon University
Pittsburgh, PA 15213

What Is Analysis?

Architectural analysis is the separation of a software architecture into its constituent parts for individual study.

Analysis ensures that the architecture used is the right one.

- How do you know if a given software architecture is deficient or at risk relative to its target system qualities?

Why Analyze?

We analyze because we can.

We analyze because it is a prudent way of informing decisions and managing risk.

Analysis is the key to evaluation.

Analyze Early and Often

The decisions that you make when designing an architecture are critical to achieve your quality attribute goals.

And the cost associated with correcting them at a later time can be significant.

Thus, it is important to perform analysis *during* the design process, so problems can be identified and corrected quickly.

If you have followed the ADD process, you should be able to perform analysis (by yourself or with peers) using the sketches and views that you produced.

Analysis Techniques: Cost and Confidence

In general there is a correlation between the cost of the technique and our resulting confidence in its analysis results.

Lifecycle Stage	Form of Analysis	Cost	Confidence
Requirements	Experience-based analogy	Low	Low-High
Requirements	Back-of-the-envelope analysis	Low	Low-Medium
Architecture	Thought experiment/reflective questions	Low	Low-Medium
Architecture	Checklist-based analysis	Low	Medium
Architecture	Tactics-based analysis	Low	Medium
Architecture	Scenario-based analysis	Low-Medium	Medium
Architecture	Analytic model	Low-Medium	Medium
Architecture	Simulation	Medium	Medium
Architecture	Prototype	Medium	Medium-High
Implementation	Experiment	Medium-High	Medium-High
Fielded System	Instrumentation	Medium-High	High

Techniques for Analysis

- Experience-based analogy
- Back-of-the-envelope analysis
- Thought experiments
- Reflective questions
- Tactics-based questionnaires
- Checklists
- Scenario-based analysis
- Analytic models
- Simulations
- Prototypes
- Instrumentation of fielded systems

Lightweight Analysis Techniques

Early in the lifecycle, some lightweight techniques have proven to be useful and relatively low-cost:

- Reflective question-based analysis
- Tactics-based analysis
- Checklist-based analysis
- Scenario-based analysis

Examples:

- Maintainability: https://resources.sei.cmu.edu/library/asset-view.cfm?assetid=650480
- Integrability: https://resources.sei.cmu.edu/library/asset-view.cfm?assetid=637375
- *Software Architecture in Practice*, 3rd. Edition

Reflective Questions

The practice of asking (and answering) reflective questions can augment analysis and design processes.

We think differently when we are problem-solving and when we are reflecting.

For this reason, researchers have advocated a distinct “reflection” activity that challenges the decisions made, and that challenges us to examine our biases.

Reflective Question Examples

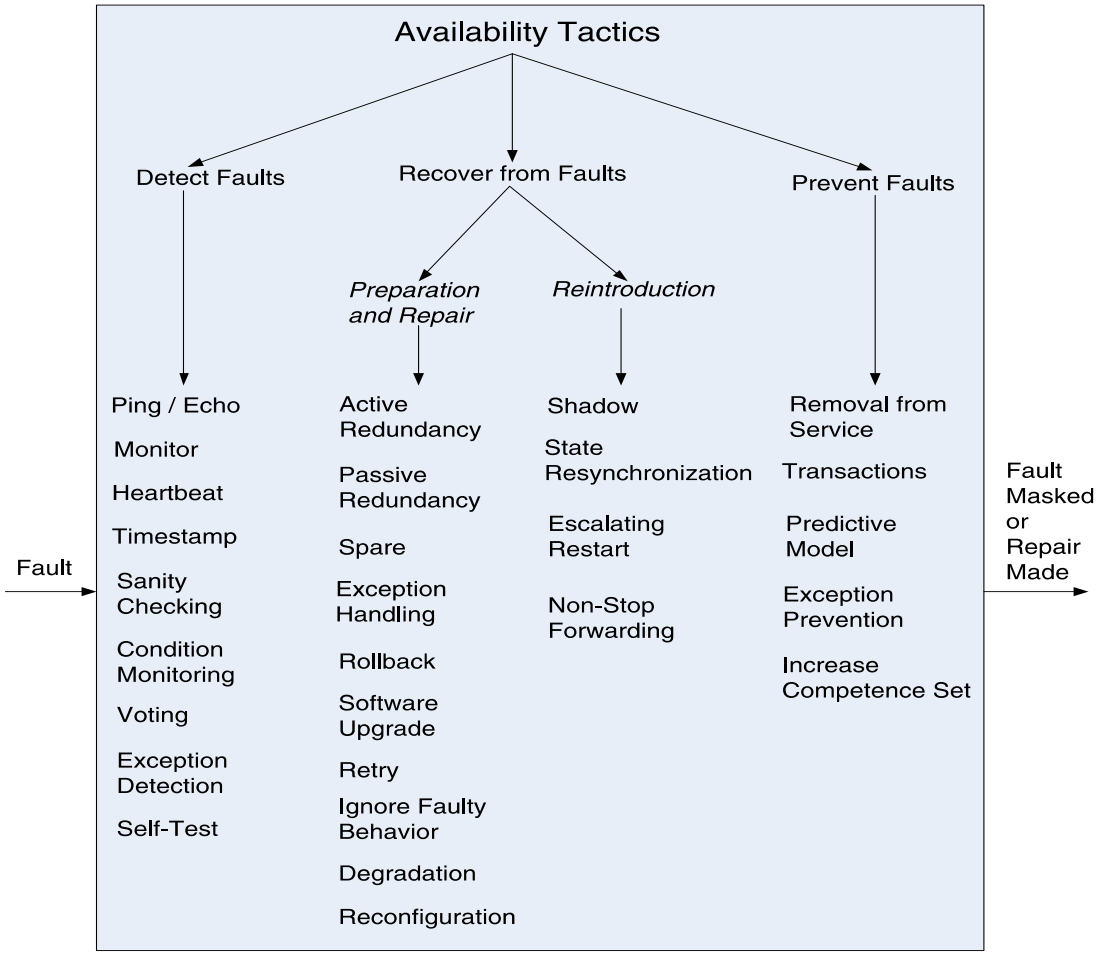
- What assumptions are made? Do the assumptions affect the design problem? Do the assumptions affect the solution option? Is an assumption acceptable in a decision?
- What are the risks that certain events would happen? How do the risks cause design problems? How do the risks affect the viability of a solution? Is the risk of a decision acceptable? What can be done to mitigate the risks?
- What are the constraints imposed by the contexts? How do the constraints cause design problems? How do the constraints limit the solution options? Can any constraints be relaxed when making a decision?
- What are the contexts and the requirements of this system? What does this context mean? What are the design problems? Which are the important problems that need to be solved? What does this problem mean? What potential solutions can solve this problem? Are there other problems to follow up in this decision?

Tactics-Based Questionnaires

We can employ tactics as an a guide to analysis. By turning every tactic into a question, we create a set of QA-specific questionnaires. These are employed as follows:

1. *The reviewers determine a number of quality attributes to drive the review.* These quality attributes will determine the selection of tactics-based questionnaires to use.
2. *The architect presents the portion of the architecture to be evaluated.* The reviewers individually ensure that they understand the architecture. Questions at this point are just for understanding.
3. *For each question from the questionnaire, the designer walks through the architecture and explains whether and how the tactic is addressed.* The reviewers ask questions to determine how the tactic is employed, the extent to which it is employed, and how it is realized.
4. *Potential problems are captured.* Real problems must be fixed, or a decision must be explicitly made to accept the risks.

Example: Availability



Tactics-Based Questions for Availability

- Does the system use **ping/echo** to detect a failure of a component or connection, or network congestion?
- Does the system use a component to **monitor** the state of health of other parts of the system? A system monitor can detect failure or congestion in the network or other shared resources, such as from a denial-of-service attack.
- Does the system use a **heartbeat**—a periodic message exchange between a system monitor and a process—to detect a failure of a component or connection, or network congestion?
- Does the system use a **timestamp** to detect incorrect sequences of events in distributed systems?
- Does the system employ **rollback**, so that it can revert to a previously saved good state (the "rollback line") in the event of a fault?

Checklists

Architecture design is a systematic approach to making design decisions.

We categorize the design decisions that an architect needs to make as follows:

1. Allocation of responsibilities
2. Coordination model
3. Data model
4. Management of resources
5. Mapping among architectural elements
6. Binding time decisions
7. Choice of technology

Each of these categories applies to every quality attribute.

Example Design Checklist for Availability: Allocation of Responsibilities

Determine the system responsibilities that need to be highly available.

Ensure that additional responsibilities have been allocated to detect an omission, a crash, incorrect timing, or an incorrect response.

Ensure that there are responsibilities to

- log the fault
- notify appropriate entities (people or systems)
- disable source of events causing the fault
- be temporarily unavailable
- fix of mask the fault/failure
- operate in a degraded mode

Example Design Checklist for Availability: Coordination Model

Determine the system responsibilities that need to be highly available. With respect to those responsibilities,

- ensure that coordination mechanisms can detect an omission, a crash, incorrect timing, or an incorrect response. Consider, for example, whether guaranteed delivery is necessary. Will the coordination work under degraded communication?
- ensure that coordination mechanisms enable the logging of the fault, notification of appropriate entities, disabling of the source of the events causing the fault, fixing or masking the fault, or operating in a degraded mode
- ensure that the coordination model supports the replacement of the artifacts (processors, communications channels, persistent storage, and processes). For example, does replacement of a server allow the system to continue to operate?
- determine if the coordination will work under conditions of degraded communication, at startup/shutdown, in repair mode, or under overloaded operation. For example, how much lost information can the coordination model withstand and with what consequences?

Scenario-Based Analysis

In scenario-based analysis, QA scenarios drive the analysis by guiding the analysts to examine whether and how the scenario's response goal can be satisfied.

The ATAM is the best known scenario-based analysis method.

The ATAM can be run as a "milestone" evaluation, or lightweight versions of the ATAM can be employed during the design process.

In the ATAM we

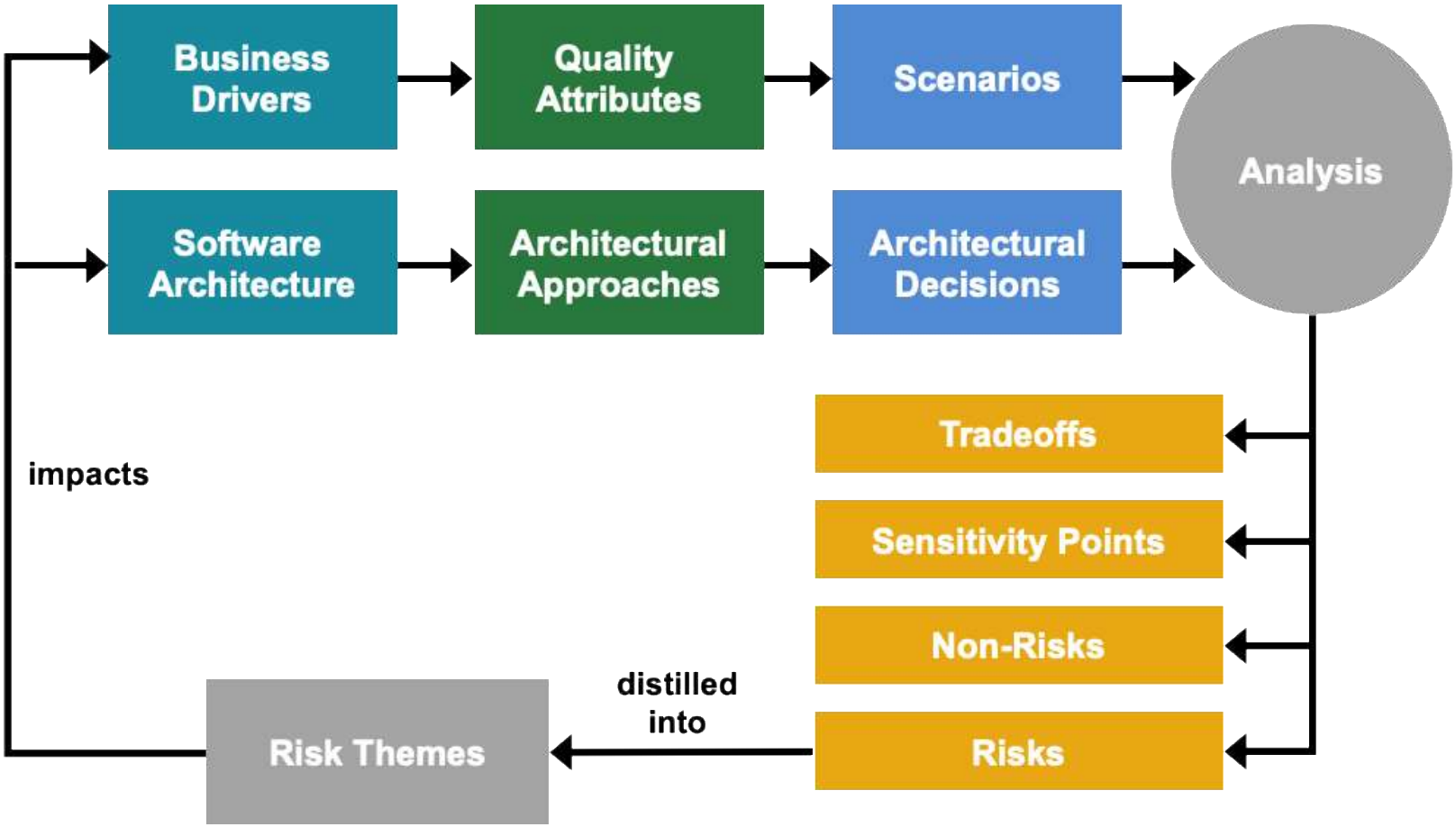
- precisely capture architectural requirements as 6-part scenarios
- prioritize those scenarios
- map the highest-priority scenarios onto representations of the architecture to understand their consequences

The Architecture Tradeoff Analysis Method (ATAM)

The purpose of the ATAM is to assess the consequences of architectural decisions in light of quality attribute requirements and business goals.

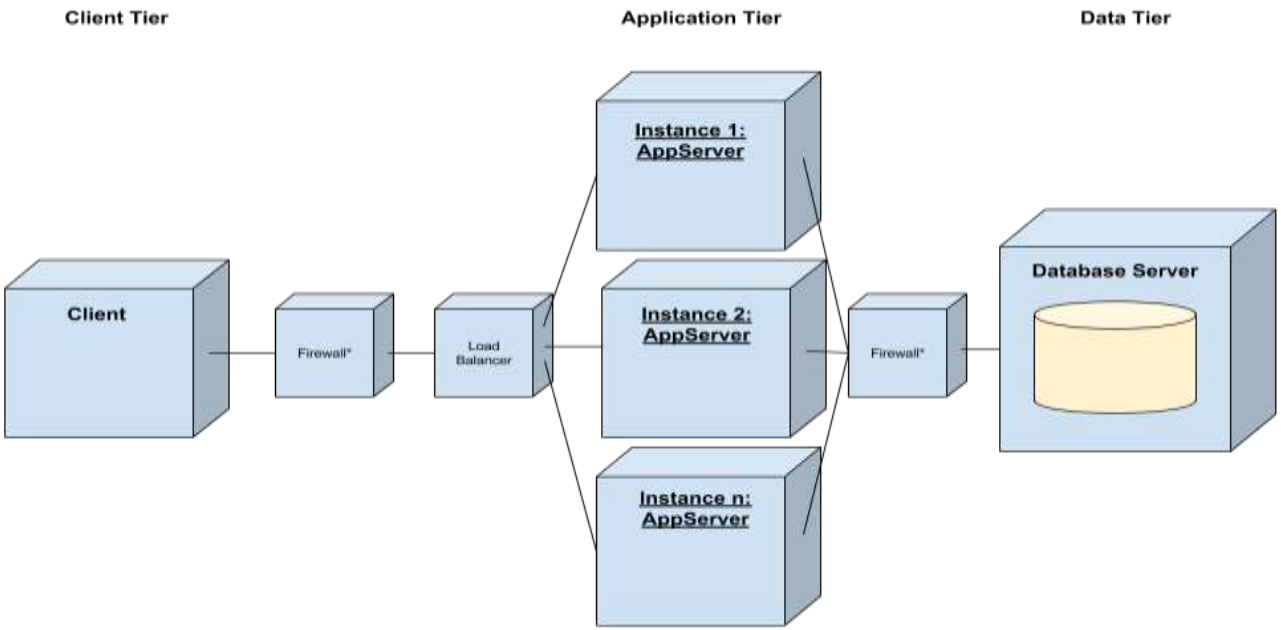
The ATAM helps stakeholders ask the right questions to discover potentially problematic architectural decisions.

Conceptual Flow of the ATAM



Example Scenario Walkthrough

An external client queries the travel database during normal operations and receives a result in < 2 seconds.



Continuous Architecture Evaluation

ATAM (Architecture Tradeoff Analysis Method) evaluations are substantial undertakings

- 20 to 30 person-days of effort from evaluation team
- Only makes sense for large/expensive projects where architecture mistakes are unacceptable

For small, less risky projects, use Lightweight Architecture Evaluation (1/2 to 1 day meeting or less)

- Participants are fewer and internal to organization, so process & technologies are familiar to all

Lightweight Architecture Evaluation

1: Present Business Drivers	0.25 hrs	Expected to understand, quick refresher
2: Present Architecture	0.5 hrs	Brief overview with 1 to 2 scenarios
3: Identify Architectural Approaches	0.25 hrs	Create architecture approaches for specific quality attribute concerns
4: Generate Quality Attribute Utility Tree	0.5-1.5 hrs	Create or update scenarios
5: Analyze Architectural Approaches	2-3 hrs	Map scenarios to architecture
6: Present Results	0.5 hrs	Review and update risks & tradeoffs

Steps 1 – 5 are done offline ahead of time by the development team. Evaluators review ahead of design review meeting, and the 2 - 3 hour period is used for discussion of architectural approaches and Step 6.

Summary

Functional requirements are important, but quality attributes define how your system is measured by stakeholders (e.g., fast, secure, reliable)

Architectures have significant impact on achievement of quality attribute requirements

ATAM is an established architectural evaluation method that uses quality attribute scenarios as measuring sticks against which to evaluate design decisions

ATAM is a foundation for incremental design reviews

The main output of design reviews is a set of design risks

Design risks with long-term negative consequences that accumulate are referred to as technical debt

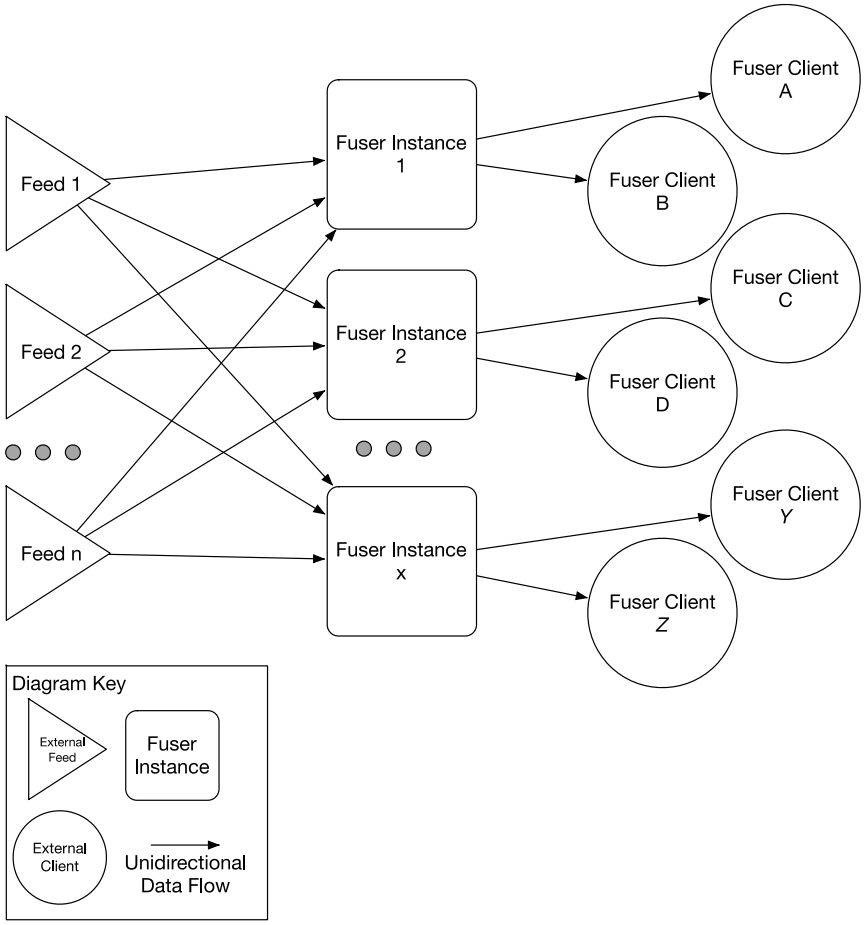
Technical debt with non-local or enterprise impact is particularly important to watch and pay down

Exercise

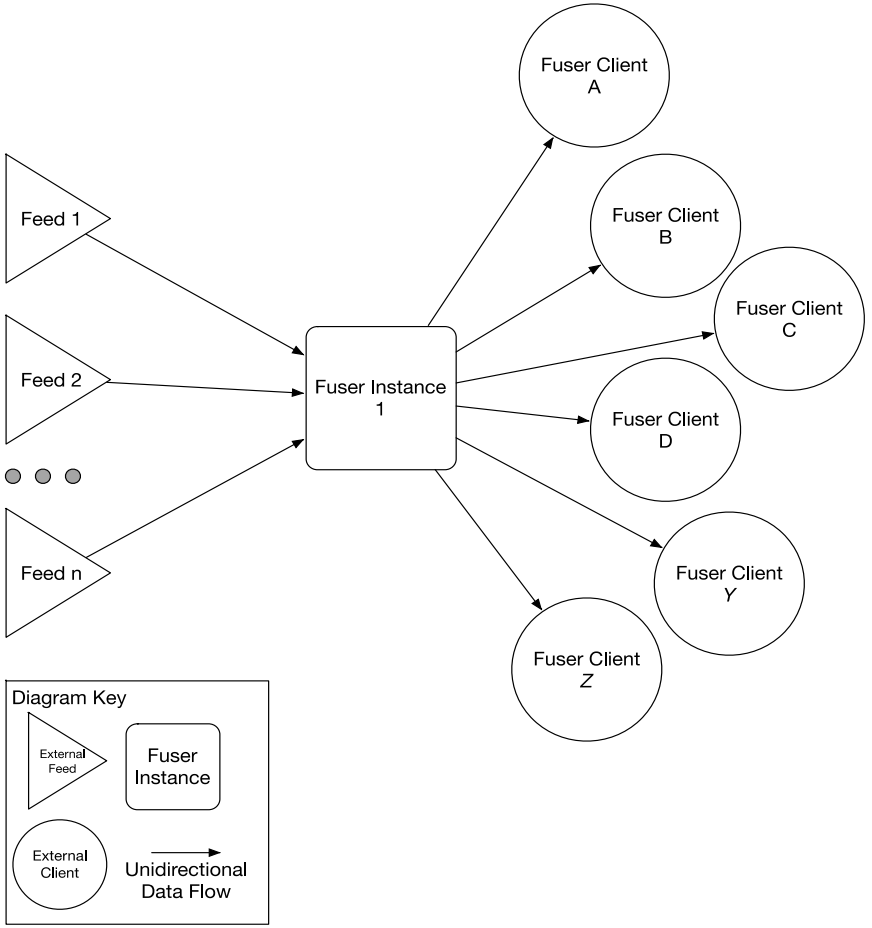
Fuser Architecture Analysis

Fuser Deployment Views (Primary Presentation)

Alternative #1



Alternative #1



Tradeoffs

Concern	Alt. #1	Alt. #2	Scenario
Fuser rule maintainability			
Fuser core message processing throughput			
Client connection workload impact on Fuser			
Operations complexity			
Data uniformity to clients			
Feed customizability for clients			
Cloud data egress costs			

+ Alternative helps address this concern
- Alternative is less helpful is addressing this concern

Architecture Design

Software Engineering Institute
Carnegie Mellon University
Pittsburgh, PA 15213

What Is Design?

Design is making the decisions that lead to the creation of architecture.

We make decisions to achieve goals and satisfy requirements and constraints.

Why does a yurt look like a yurt, which is different from an igloo or a chalet or a longhouse?

The architectures of these styles of houses have evolved, over the centuries, to reflect their unique sets of goals, requirements and constraints.

Is Design Hard?

Yes and no.

Novel design is hard.

But the vast majority of design is not novel.

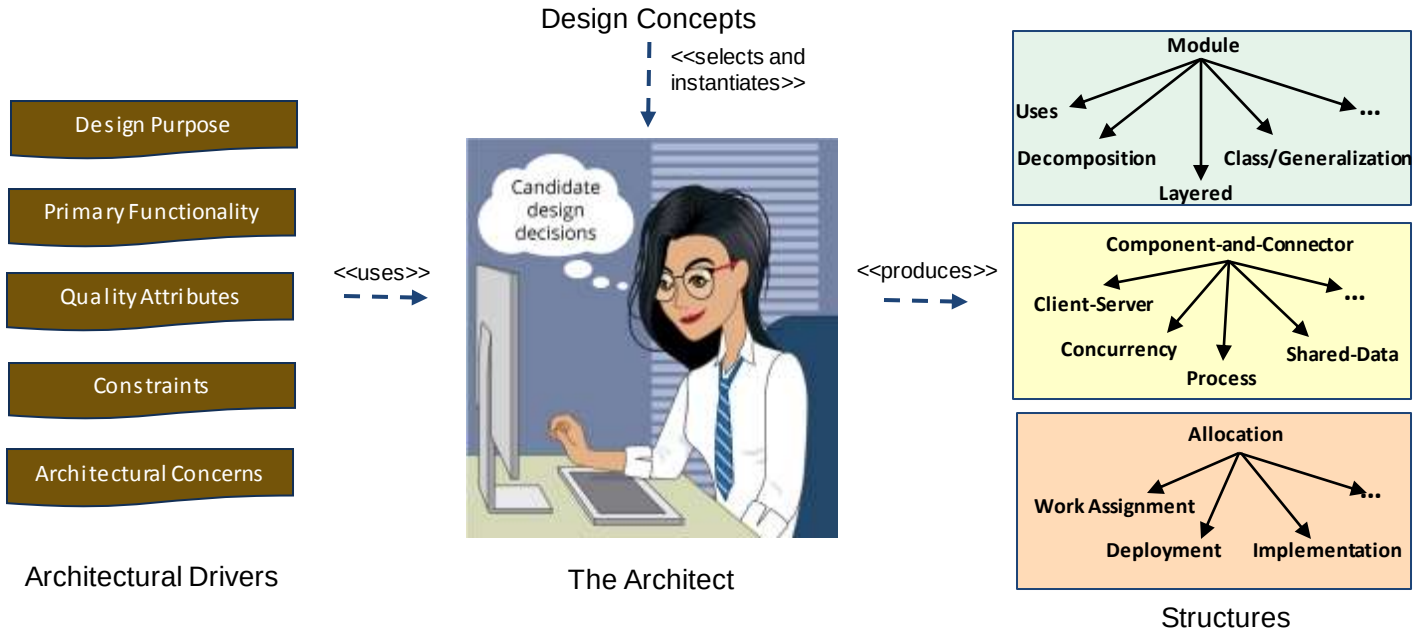
There are ample proven designs and design fragments, that we call *design concepts*, that can be reused and combined to reliably achieve your goals for most designs.

Traditional American House

Frank Lloyd Wright's Fallingwater

Architectural Design

In architectural design, we make decisions to transform our design purpose, quality attributes, functionality, constraints, and concerns—the *drivers*—into structures.



What Makes a Decision “Architectural”?

A decision is architectural if it has non-local consequences *and* those consequences matter to the achievement of an architectural driver.

No decision is, therefore, inherently architectural or non-architectural.

Consider the choice of a buffering strategy.

Is this architectural or not?

Architectural Drivers

As mentioned earlier, architectural drivers consist of

- design purpose
- quality attributes
- primary functionality
- architectural concerns
- constraints

These are critical to the success of the system and, as such, they *drive* the architecture.

Architectural drivers need to be baselined and managed throughout the architecture influence cycle.

Design Concepts

We have identified six broad, reusable categories of design concepts that aid an architect:

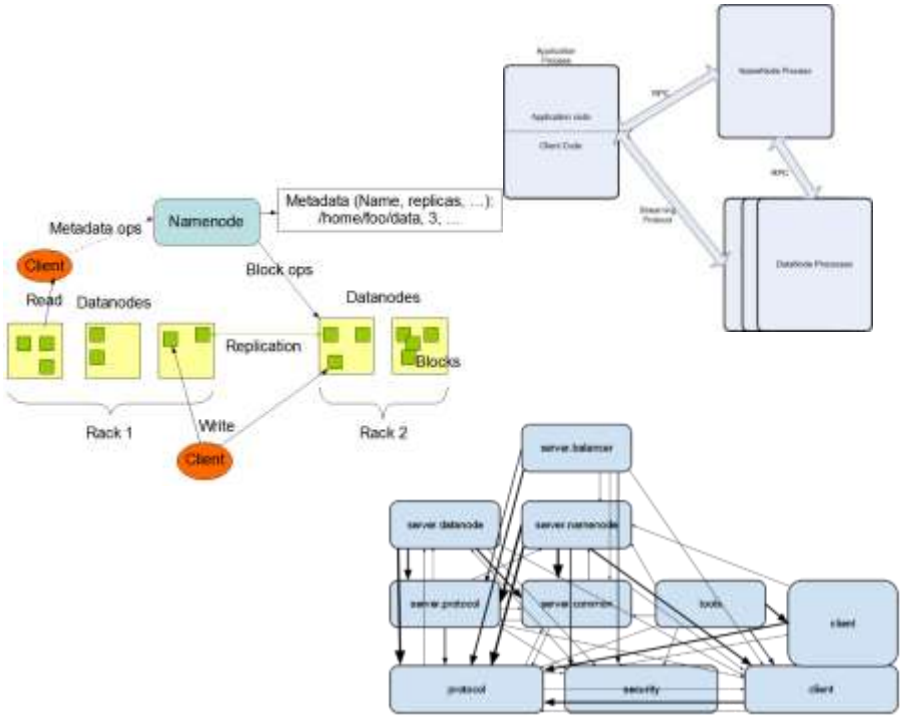
- design principles
- reference architectures
- externally developed components
- deployment patterns
- architectural design patterns
- tactics

The Output of the Design Process

The output of the design process is a set of architectural *structures* resulting from design decisions.

These structures will guide

- analysis and construction
- the education of a new project member
- cost and schedule estimation
- team formation
- risk analysis and mitigation
- (and of course) implementation



Design Principles

Design principles are basic tenets that guide us toward good designs.

There are general design principles, e.g.:

- Information hiding—hide data structures, hide details, hide variations
- Low coupling, high cohesion

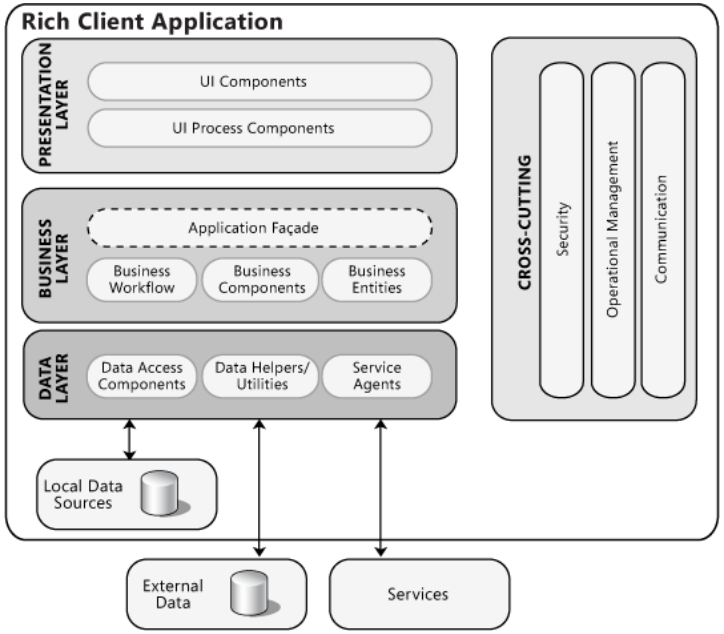
There are more specific design principles. For example, the SOLID principles aid in designing modifiable, extensible OO-based architectures.

Design Principles

SOLID principles:

- **S**ingle Responsibility Principle: There should be only one reason for a class to change.
- **O**pen/Closed Principle: Classes and methods should be open for extension but closed for modification.
- **L**iskov Substitution Principle: Every function or method that expects an object parameter of class A must be able to accept a subclass of A as well, without knowing it.
- **I**nterface Segregation Principle: Classes should not be forced to depend on interfaces that they do not use.
- **D**ependency Inversion Principle: High-level classes should not depend on low-level classes. Both should depend on abstractions.

Reference Architectures



Source: Microsoft Patterns and Practices Team. *Microsoft Application Architecture Guide*, 2nd ed. Microsoft Press. 2009. https://msdn.microsoft.com/en-us/library/ff650706.aspx

Used with permission from Microsoft.

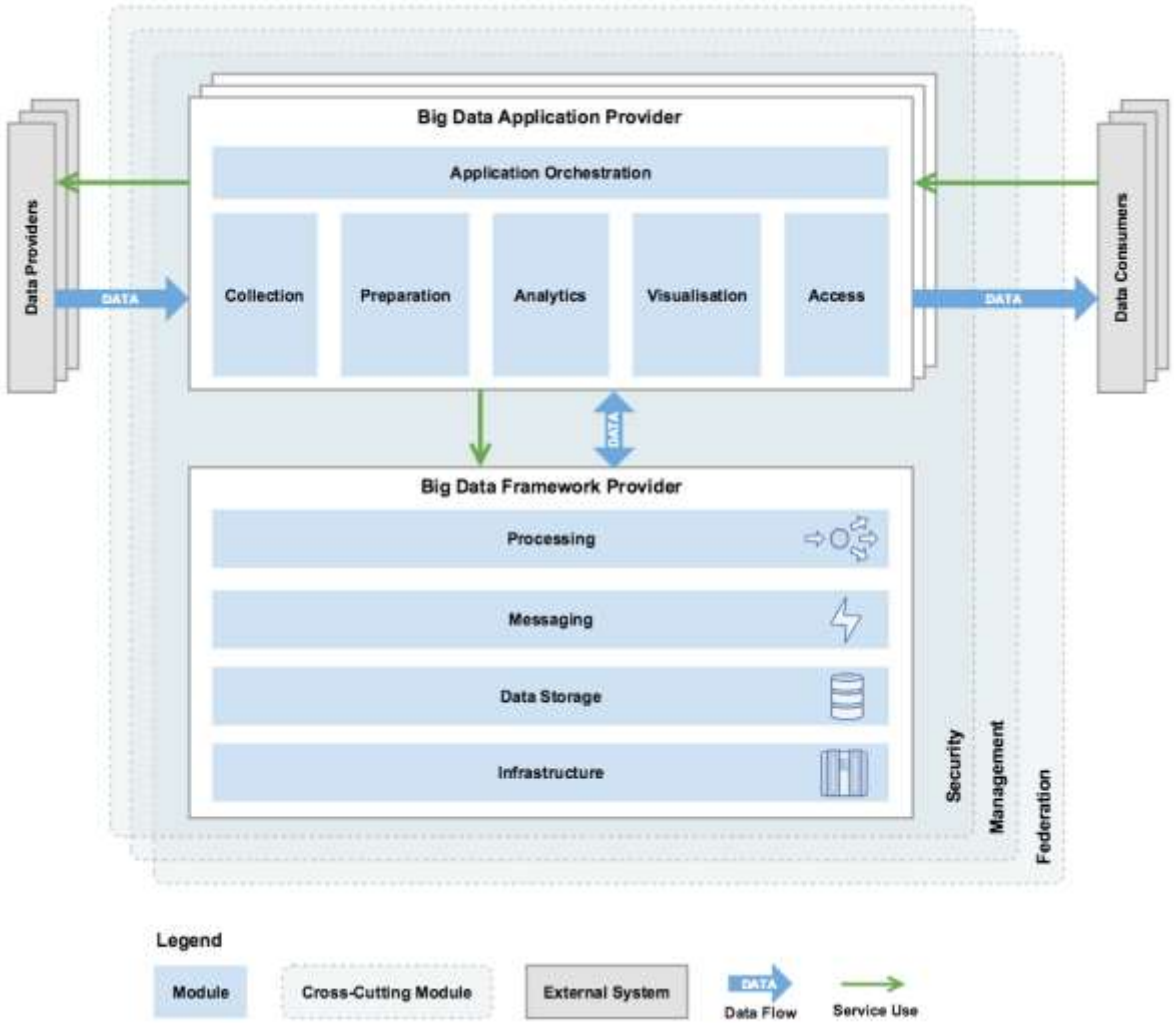
Reference architectures are archetypes that provide an overall logical structure for specific types of applications, e.g.,

- Web application
- mobile application
- lambda architecture

They typically employ and combine patterns.

They aid in planning and reasoning.

Reference Architecture for Big Data Systems



J. Klein, R. Buglak, D. Blockow, et al., "A Reference Architecture for Big Data Systems in the National Security Domain," in *Proc. 2nd Int. Workshop on BIG Data Software Eng. (BIGDSE'16)*, Austin, TX, USA, 2016, pp. 51-57. URL: https://resources.sei.cmu.edu/library/asset- view.cfm?assetid=454876

Externally Developed Components

Products: A product (or software package) refers to a self-contained functional piece of software that can be integrated into the system that is being designed and that requires only minor configuration or coding. Example: MySQL

Application frameworks: An application framework (or just framework) is a reusable software element, constructed out of patterns and tactics, that provides generic functionality addressing recurring domain and quality attribute concerns across a broad range of applications. Example: Hibernate

Technology families: A technology family represents a group of specific technologies with common functional purposes. Example: ORM

Platforms: A platform provides a complete infrastructure upon which to build and execute applications. Example: Google Cloud

Architectural Design Decisions

In the early stages of design, decisions are focused on the biggest, most critical choices that will have substantial downstream consequences: reference architectures, major technologies (such as frameworks), and patterns.

Once a design decision has been made, you should think about how you will *document* that decision.

Documenting During Design

As you instantiate design concepts you will typically create *sketches*. These are initial documentation for your architecture.

- Capture them and flesh them out later.
- If you use an informal notation, be consistent.
- Develop a discipline of writing down the responsibilities that you allocate to elements.
- Writing it down ensures you won't have to remember it later.

Tracking Progress

When designing there are three key questions to answer:

- How much design do we need to do?
- How much design has been done so far?
- Are we finished?

Agile practices such as the use of backlogs and kanban boards can help you track design progress and answer these questions.

Design Backlog

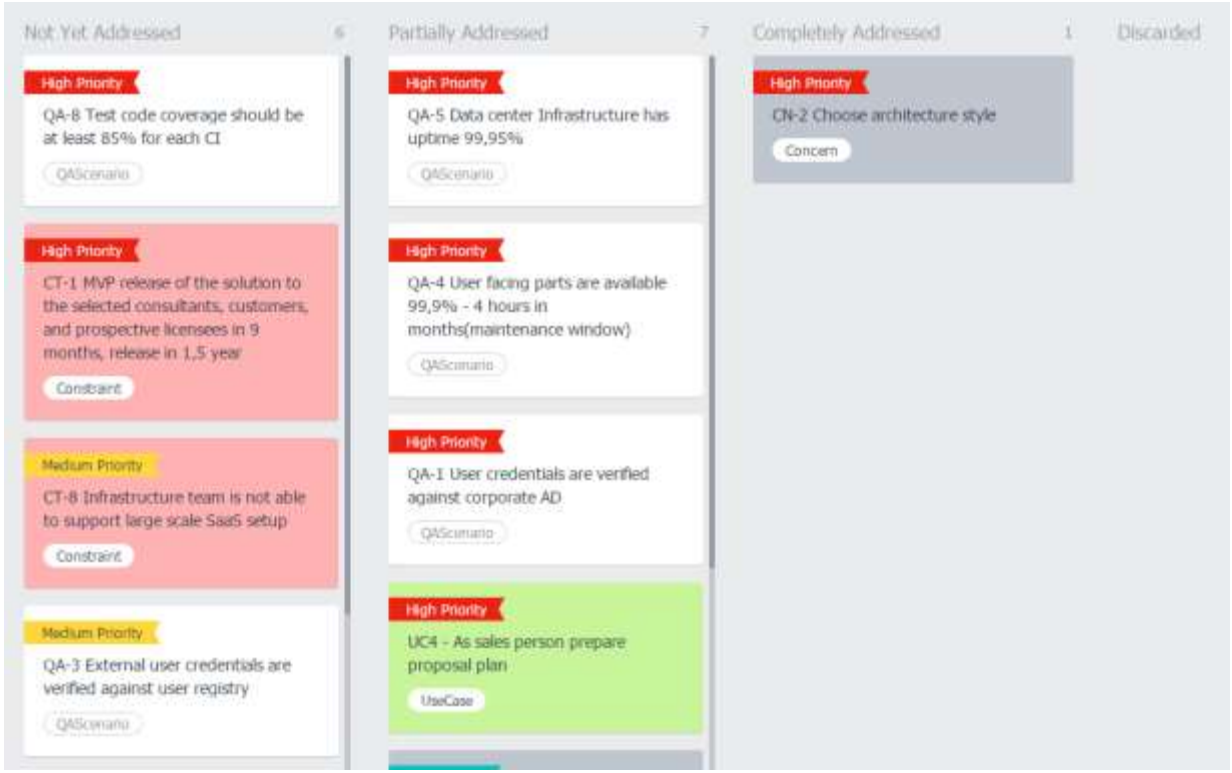
You should create a list of the actions that still need to be performed as part of the architecture design process.

Initially, populate the design backlog with your drivers, but other activities such as the following can be included:

- Creation of a prototype to test a particular technology or to address a specific quality attribute risk
- Exploration and understanding of existing assets (possibly requiring reverse engineering)
- Issues uncovered in a review of the design (recall that we analyze as we are designing)
- Review of a partial design that was performed on a previous iteration

Using a Design Kanban Board

One possible tool for tracking progress is a Kanban board.



Termination Criteria

The design process continues across several iterations:

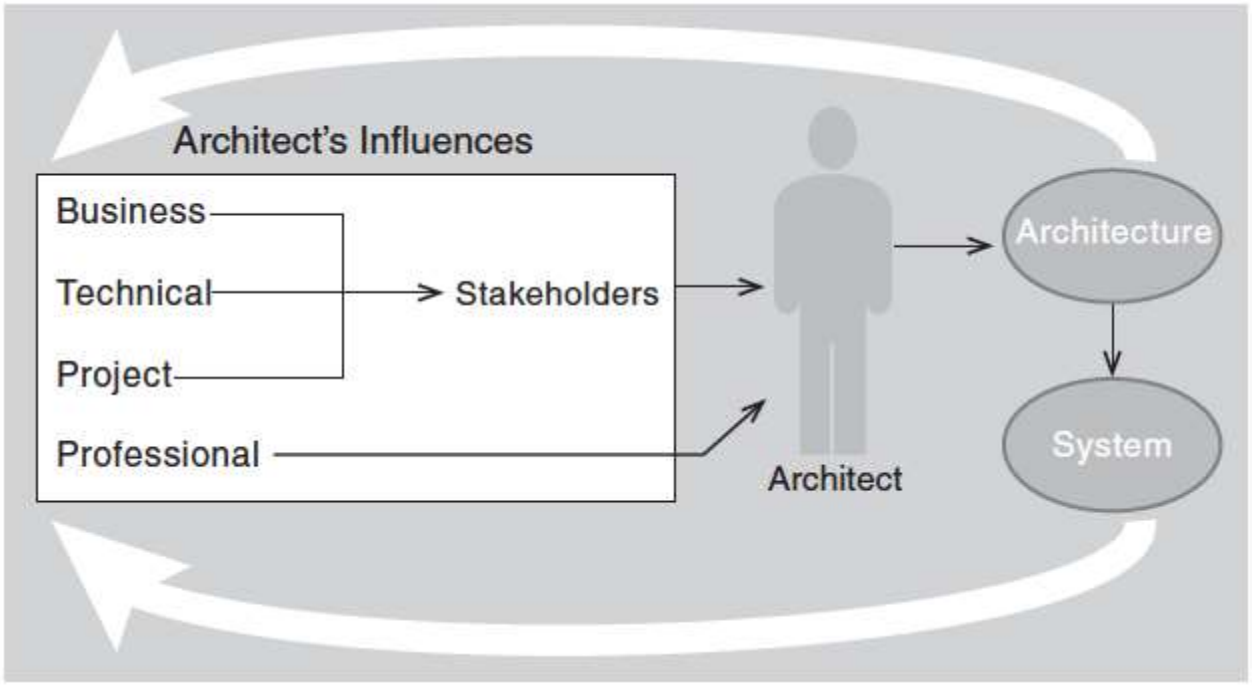
- until the most important technical risks have been mitigated; or
- until design decisions have been made for all of the driving architectural requirements; or
- until the time allotted for architecture design is consumed (not very desirable!).

Attribute-Driven Design (ADD) Method

The ADD method is an approach to defining software architectures by basing the design process on the architecture's quality attribute requirements.

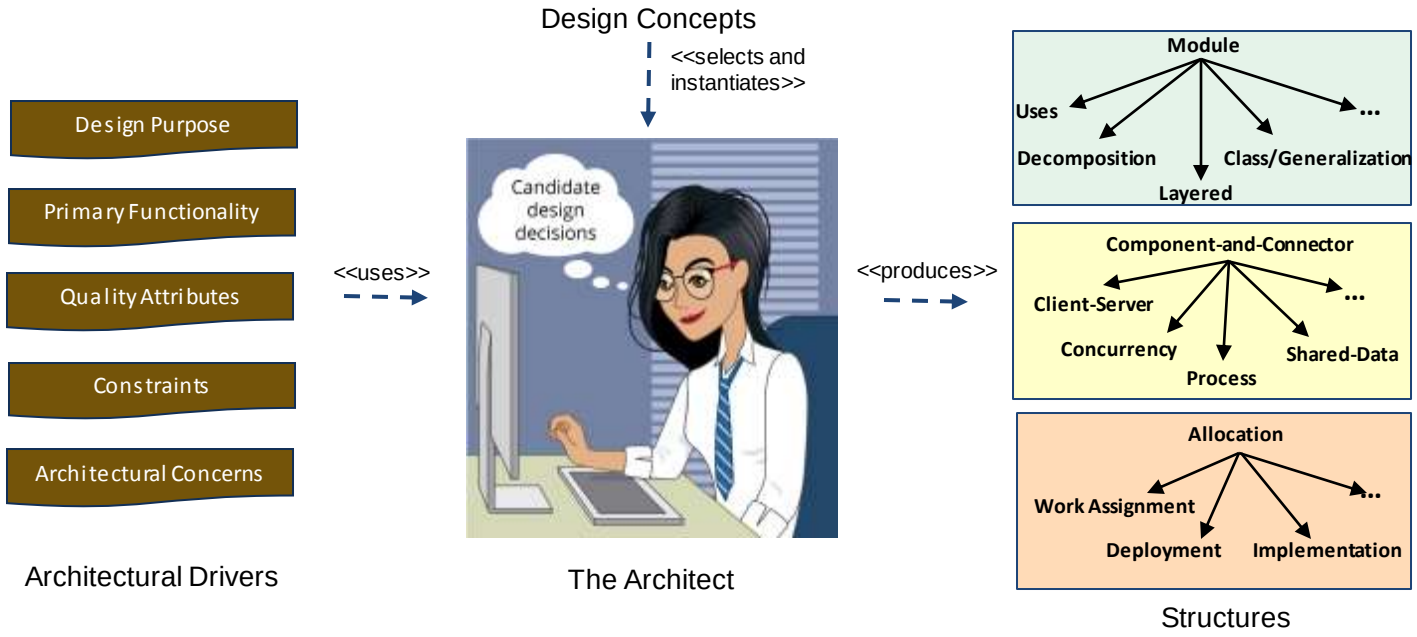
It follows a recursive decomposition process where, at each stage in the decomposition, tactics, patterns, and technologies are chosen to satisfy a set of architectural drivers.

Recall: The Architecture Influence Cycle (AIC)



Design Is a Transformation

In architectural design, we make decisions to transform our design purpose, quality attributes, functionality, constraints, and concerns—the *drivers*—into structures.



On the Need for an Architecture Design Method

Architecture design is notoriously difficult to master

- Design can (and should) be performed in a systematic way.
- Design decisions should be justified.

The architect is accountable for design decisions

- Design decisions should be recorded.

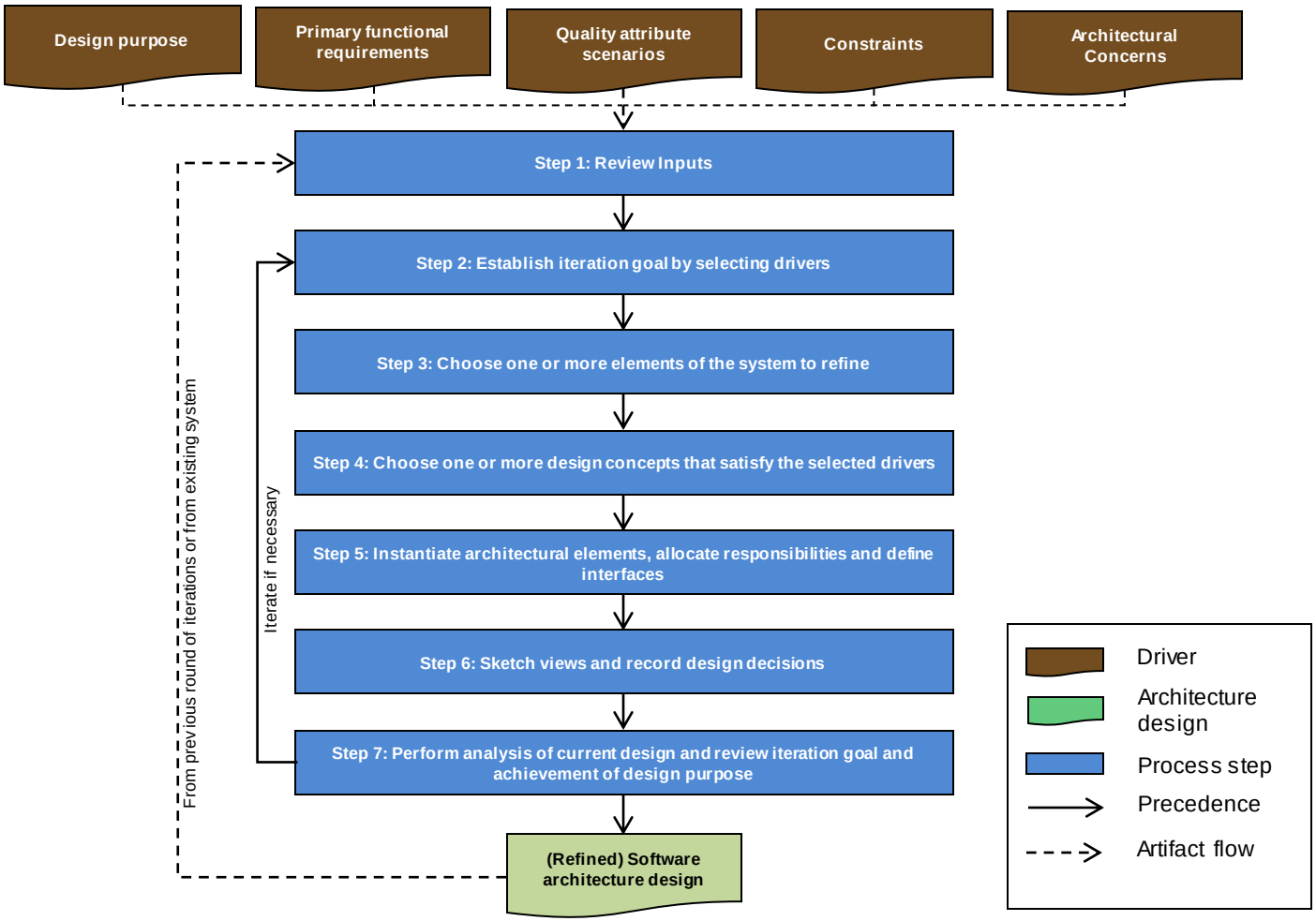
Otherwise, architecture design may end up being a mystic activity performed by gurus.

To make any activity more predictable and repeatable, we need methods.

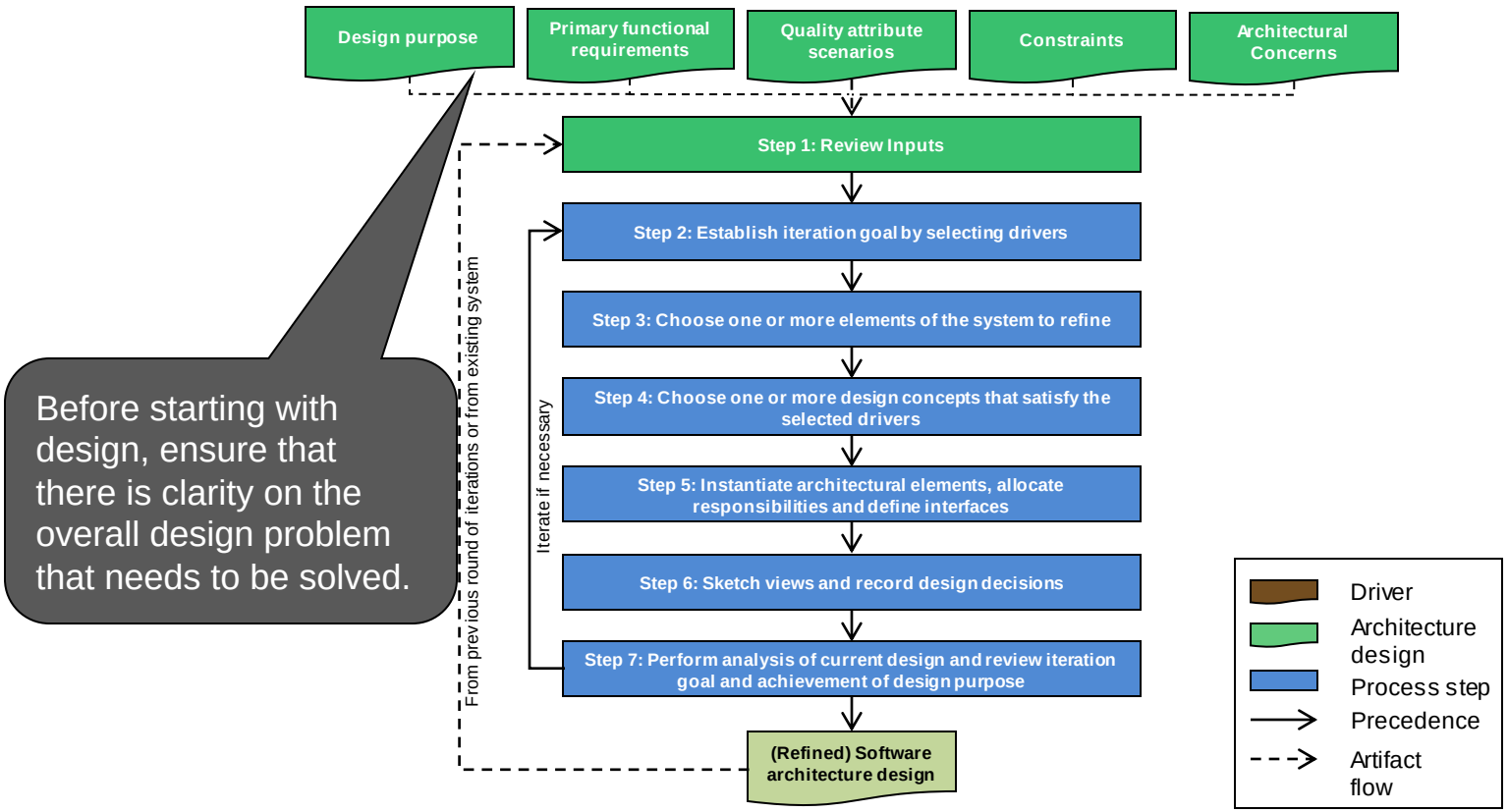
ADD 3.0 (henceforth just ADD) is the result of more than a decade of evolution and practice.

Let's look at its steps in detail…

ADD



ADD Step 1



Recall: Prioritizing QA Scenarios

Before commencing design, prioritize quality attribute scenarios.

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

If a QAW was performed, the scenarios will already be prioritized.

One could also use the ATAM technique where scenarios are prioritized across two dimensions.

- Importance (high, medium, or low) to the success of the system, ranked by the customer.
- Degree of technical risk (high, medium, or low), ranked by the architect.

Recall: 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, however, it is important to ensure that the structure contains the necessary elements to satisfy functional requirements.

Recall: Prioritizing Functional Requirements

Primary use cases

- are critical to achieve business goals
- are associated with an important quality attribute 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.

Recall: Constraints

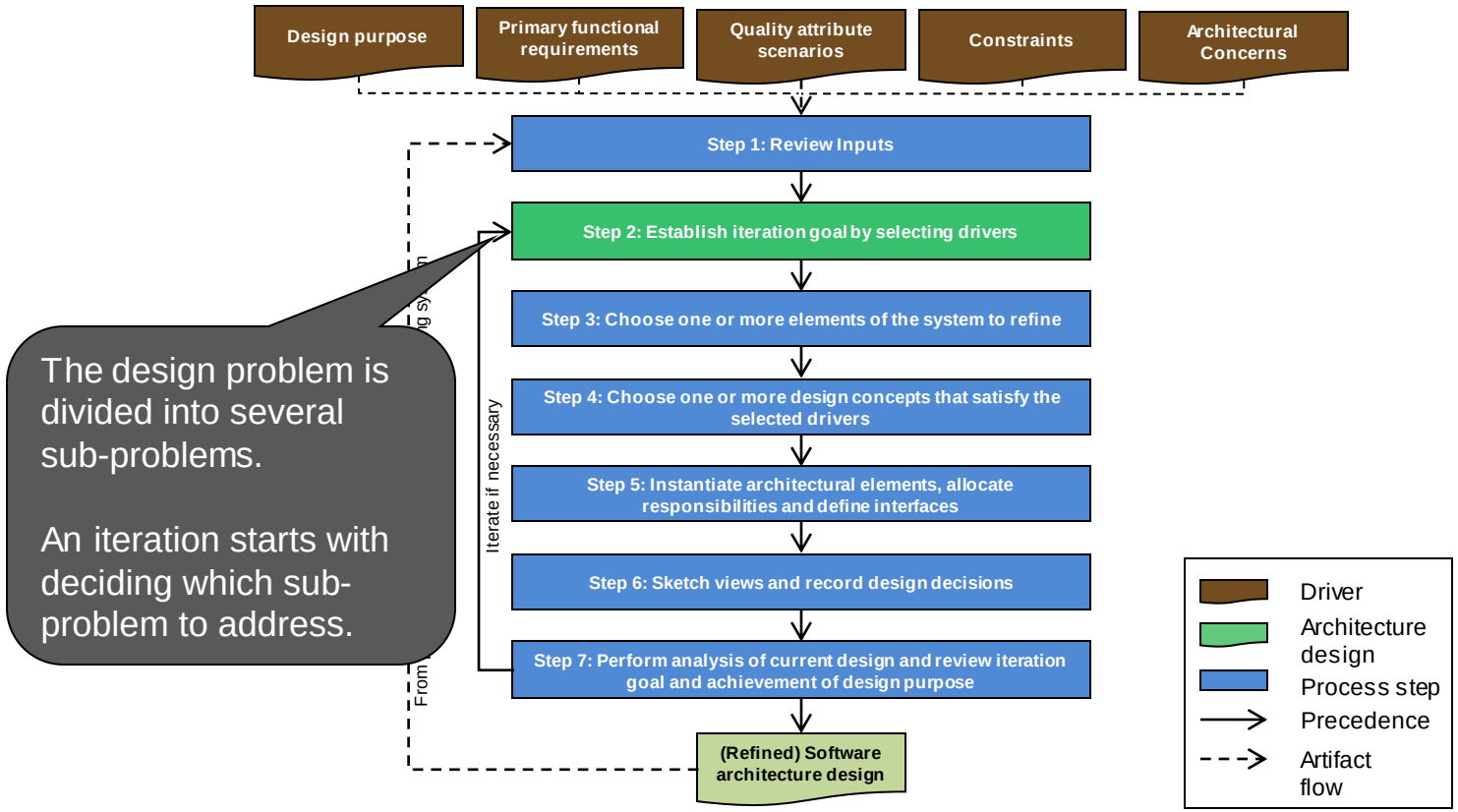
Constraints limit the range of possibilities when making design decisions.

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

Before commencing design, identify, justify, and (ideally) prioritize constraints.

- Technical constraints
  - Use of a legacy database
  - Use of Eclipse IDE
- Other constraints
  - Development team only familiar with Java
  - Obey Sarbanes-Oxley
  - Ready in time for April 15th

ADD Step 2



Recall: Scenario Prioritization

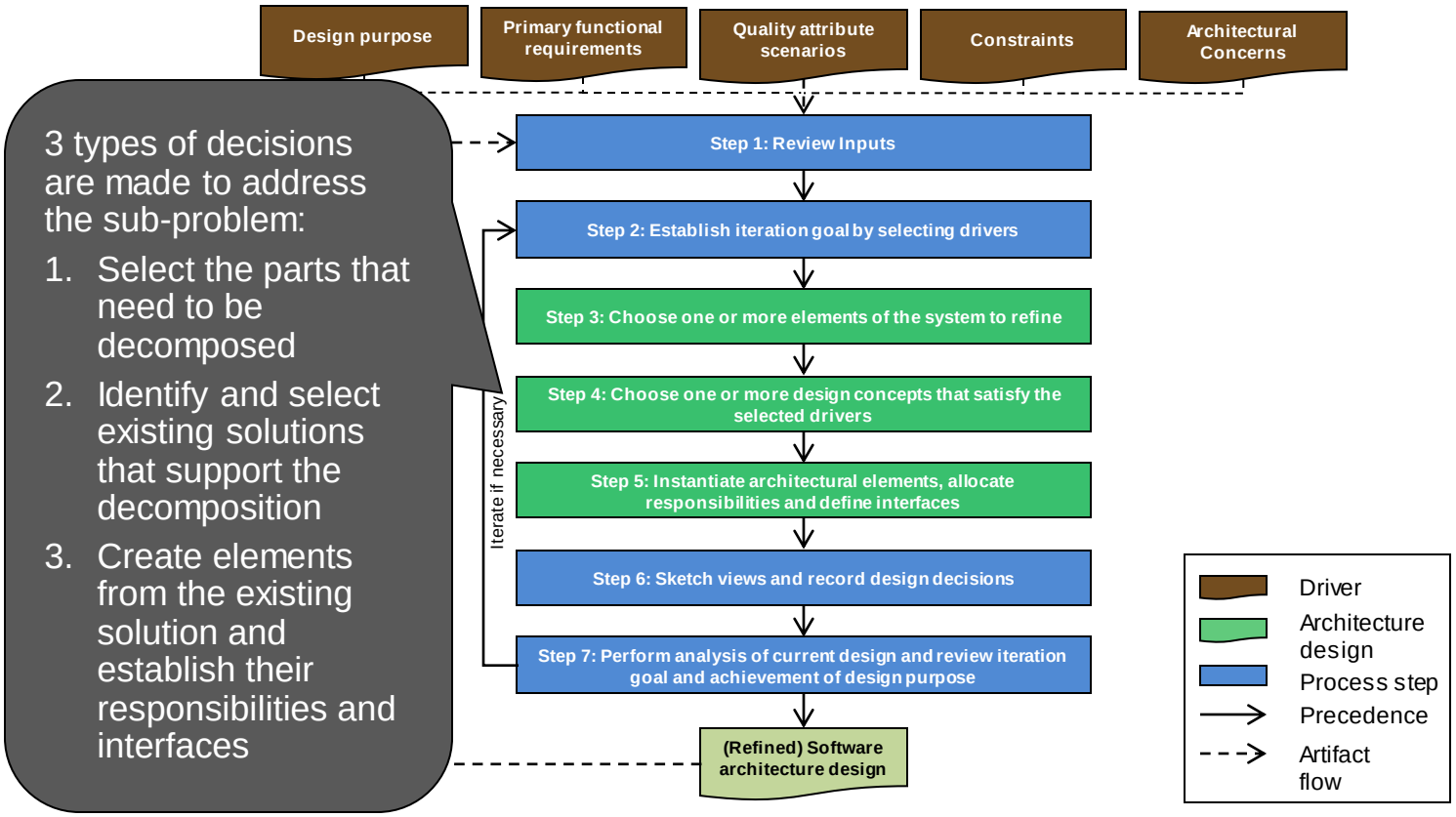
If the quality attribute scenarios haven't been prioritized, a technique can be borrowed from the ATAM.

Scenarios are prioritized across two dimensions:

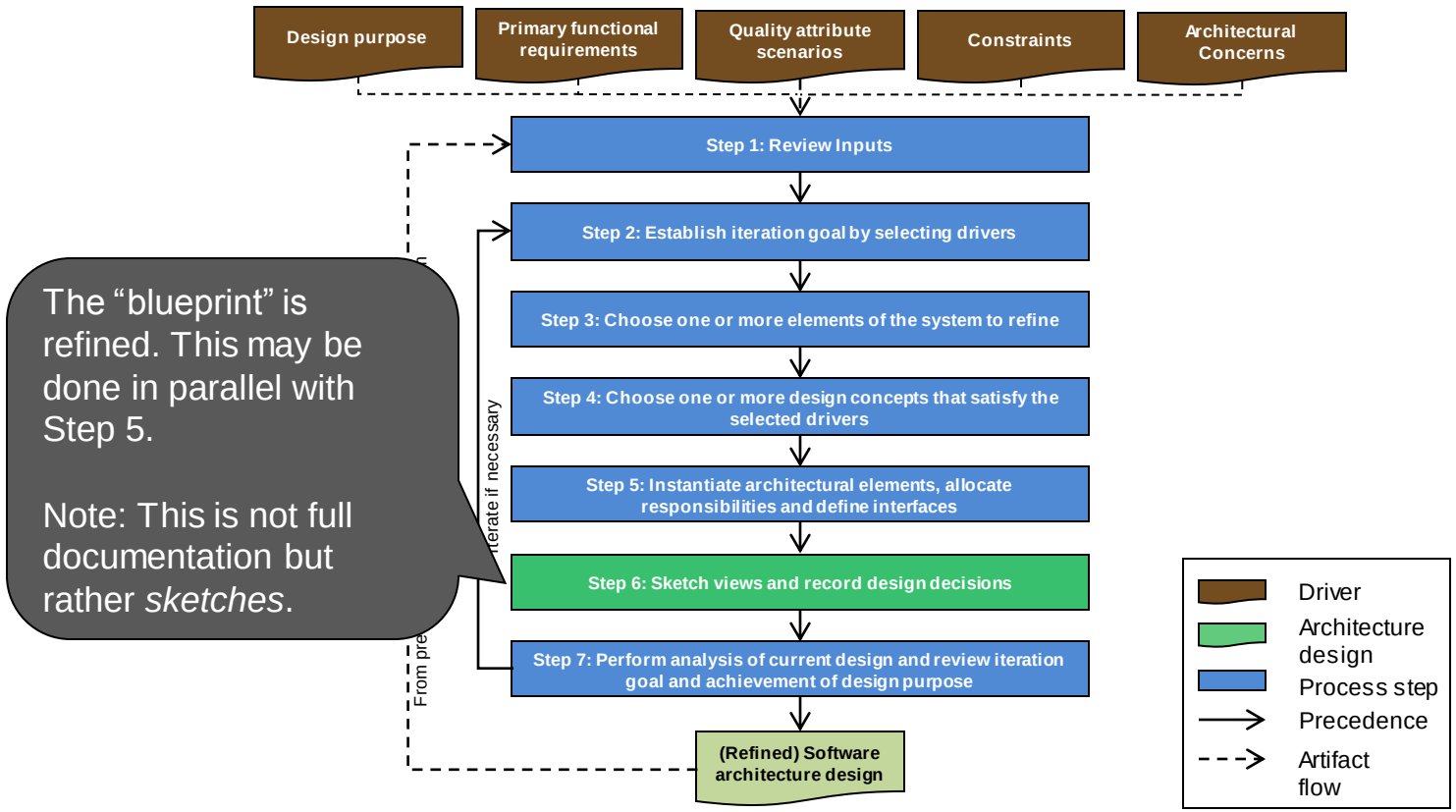
- importance to the success of the system
- degree of technical risk

Scenario ID	Importance	Technical Risk	Prioritized?
QA-001	H	H	Yes
QA-002	H	M	Yes
QA-003	H	L	Maybe
QA-004	M	H	Yes
QA-005	M	M	Yes
QA-006	M	L	No
QA-007	L	H	No
QA-008	L	M	No
QA-009	L	L	No

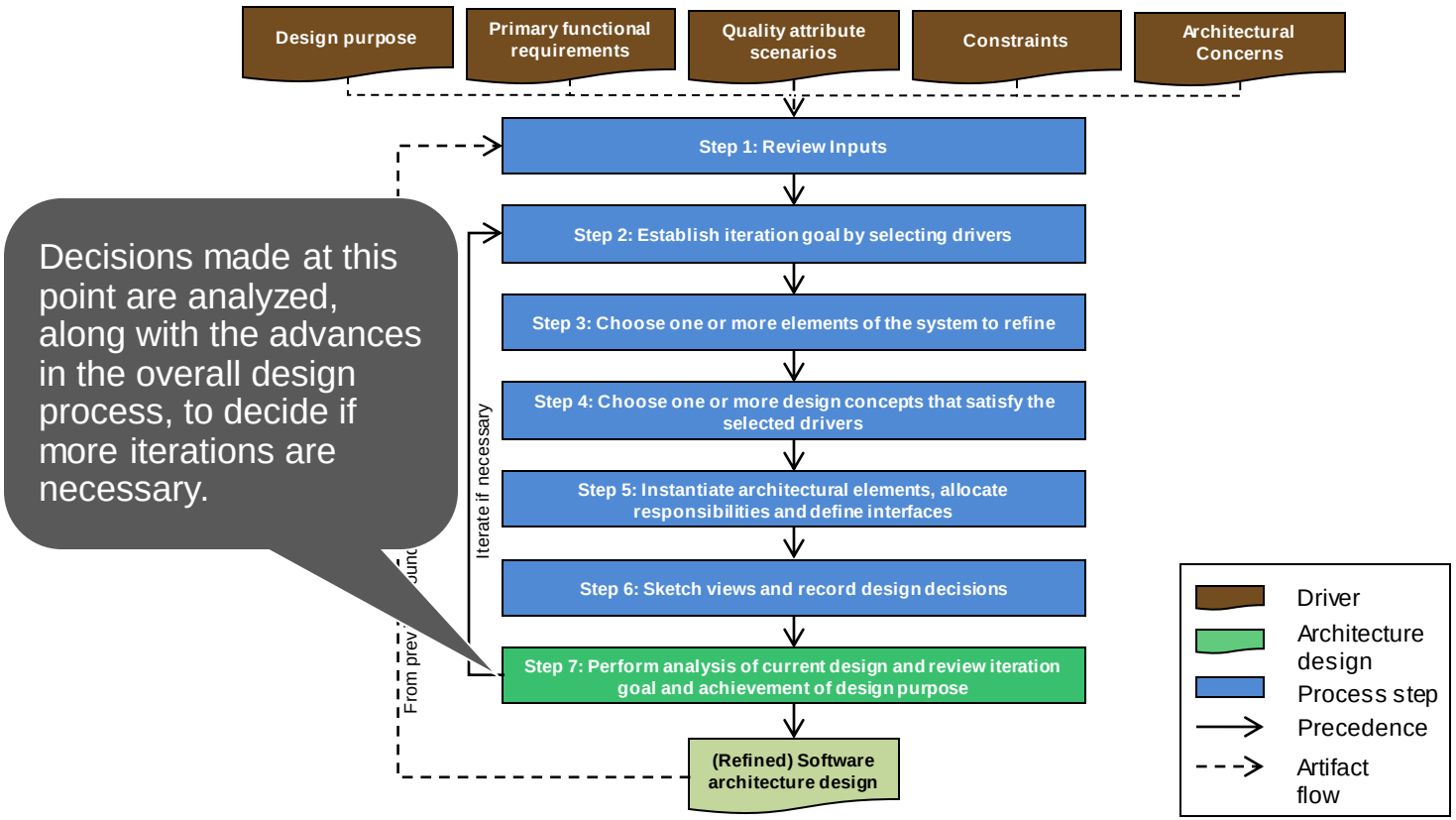
ADD Steps 3-5



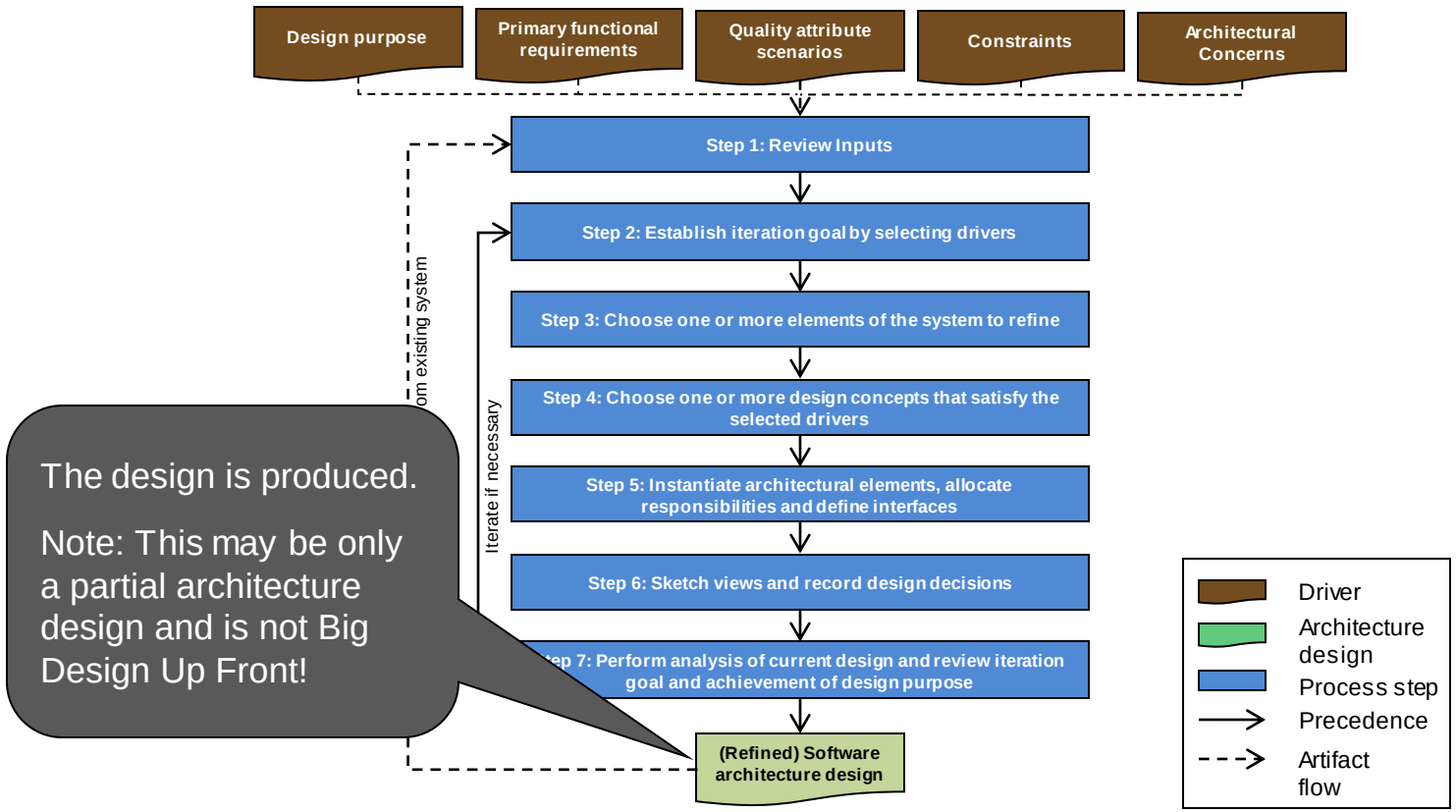
ADD Step 6



ADD Step 7



ADD Output/Iteration



Design Decisions

The design process involves making different design decisions

- Step 3: Selecting elements to refine
- Step 4: Choosing one or more design concepts that satisfy the selected drivers
- Step 5: Instantiating architectural elements, allocating responsibilities and defining interfaces

Step 4 (selecting design concepts) can be particularly challenging…

Design Concepts

Most sub-problems that are addressed during an iteration can be solved using existing solutions (i.e., design concepts)

- We want to avoid re-inventing the wheel.
- It is better (and faster) to use a proven solution to a problem for which we may not be experts.
- Creativity in design involves identifying, adapting, and combining solutions.

There are several categories of design concepts, some abstract and some more concrete.
Here we consider

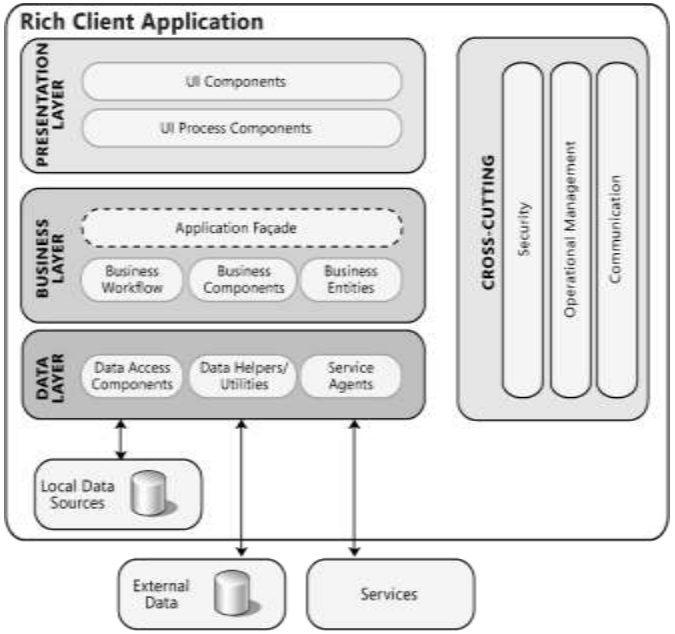
- reference architectures
- deployment patterns
- architectural/design patterns
- tactics
- externally developed components (e.g., frameworks)

vs.

Reference Architectures

Reference architectures provide a blueprint for structuring an application. Examples for the enterprise application domain include

- mobile applications
- rich client applications
- rich internet applications
- service applications
- web applications



Source: Microsoft Patterns and Practices Team. *Microsoft Application Architecture Guide*, 2nd ed. Microsoft Press. 2009. https://msdn.microsoft.com/en-us/library/ff650706.aspx

Used with permission from Microsoft.

Deployment Patterns

Deployment patterns provide guidance on how to structure the system from a physical standpoint. Good decisions with respect to the deployment of the software system are essential to achieve quality attributes such as availability.

Examples

- 2, 3, 4 and n-tier deployment
- Load-balanced cluster
- failover cluster
- private/public cloud

Source: Microsoft Patterns and Practices Team. *Microsoft Application Architecture Guide*, 2nd ed. Microsoft Press. 2009. https://msdn.microsoft.com/en-us/library/ff650706.aspx

Used with permission from Microsoft.

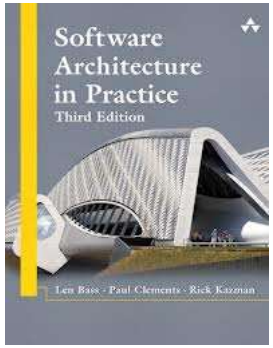
Tactics - 1

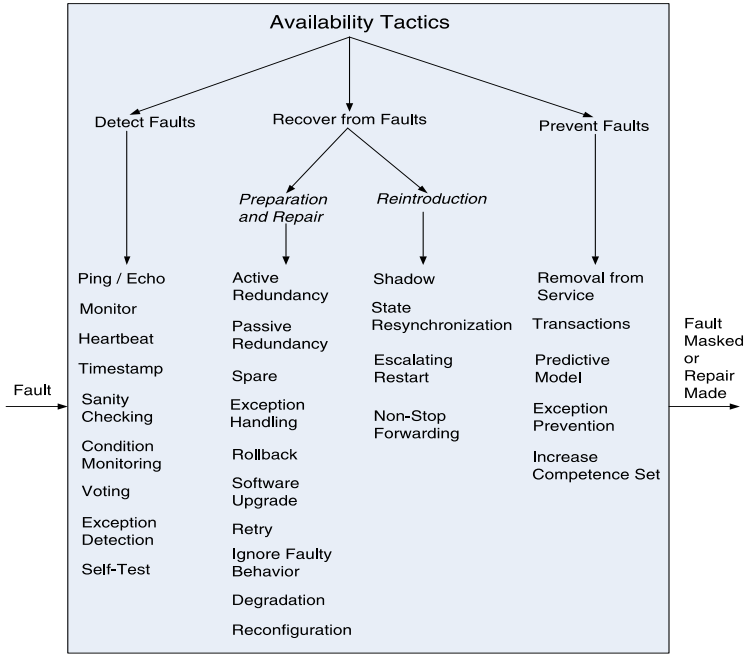
What are tactics?

- Design decisions that influence the control of a quality attribute response.

There are tactics categorizations for the quality attributes of

- availability
- interoperability
- modifiability
- performance
- security
- testability
- usability

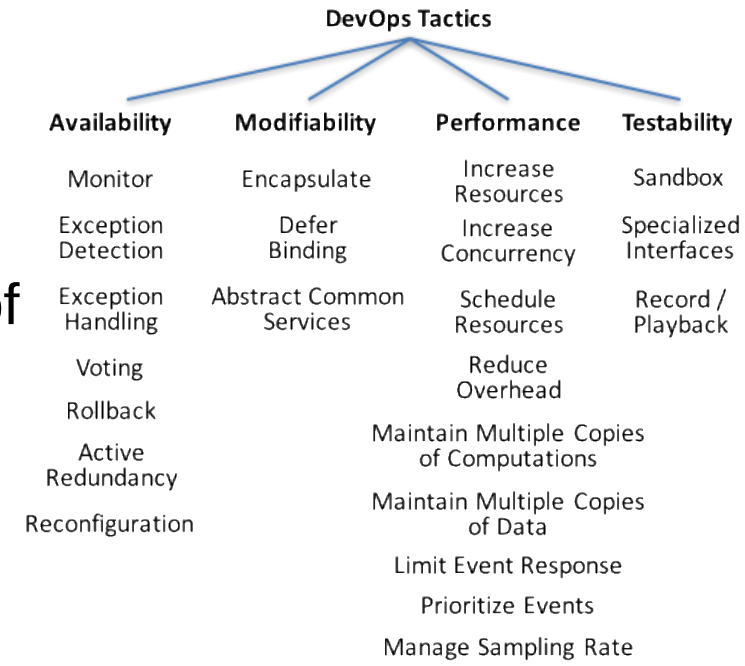




Tactics - 2

What if there are no tactics for my QA?

- You can create your own, based on your experience.
- This is a useful organizational knowledge base.
- Consider this categorization of DevOps tactics[1].



Chen, H-M, Kazman, R., Haziyev, S., Kropov, V., Chtchourov, D. "Architectural Support for DevOps in a Neo-Metropolis BDaaS Platform", *IEEE 34th Symposium on Reliable Distributed Systems Workshop (SRDSW),* (Montreal, Canada), Sept. 2015

Architectural / Design Patterns

Patterns are proven (conceptual) solutions to recurring design problems.

Patterns originated in building architecture.

Many patterns exist (thousands), and they are documented across several pattern catalogs.

It is difficult to draw a clear boundary between “design” and “architectural” patterns.

Externally Developed Components

These are reusable code solutions

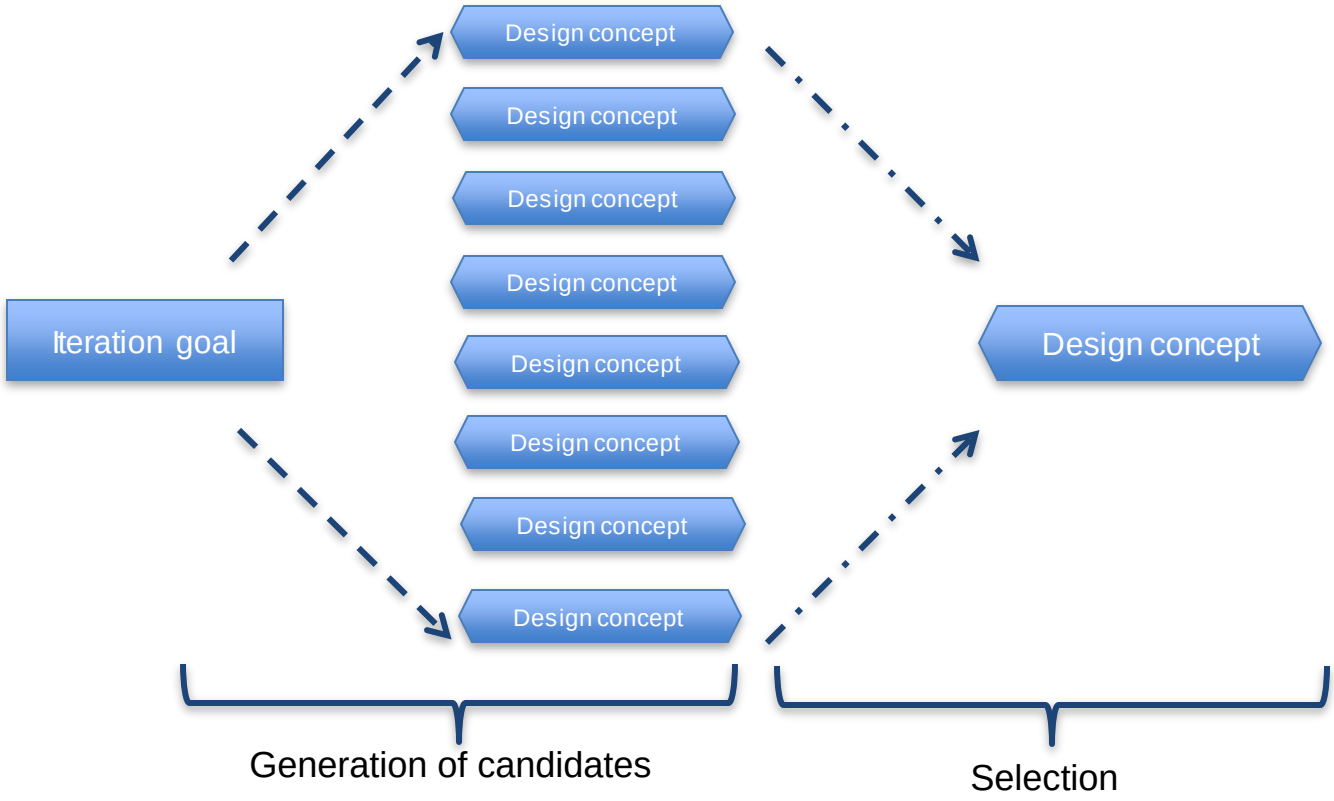
- Examples include middleware and frameworks.

A framework is a reusable software element that provides generic functionality, addressing recurring concerns across a range of applications.

- Examples for Java:

Concern	Framework	Usage
Local user interface	Swing	Inheritance
Web UI	Java Server Faces (JSF)	XML, Annotations
Component connection	Spring	XML, Annotations
Security (authentication, auth)	Spring-Security	XML, Annotations
OO–Relational Mapping	Hibernate	XML, annotations

Selecting Design Concepts (Step 4)



Continuous Architecture Evaluation

ATAMs are substantial undertakings

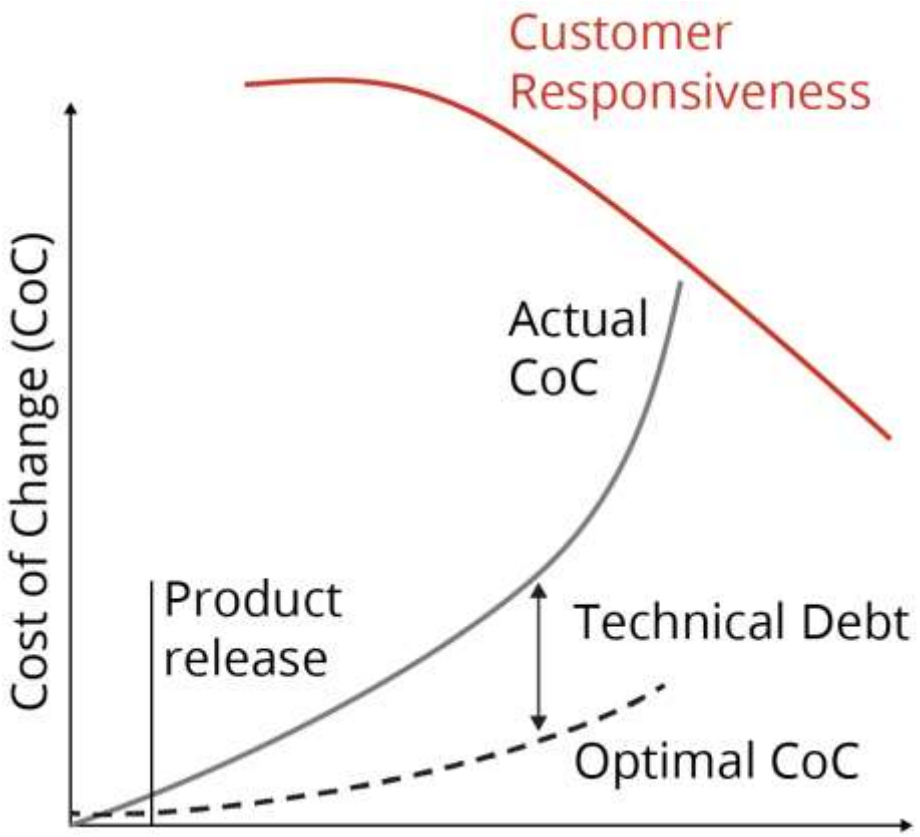
- 20 to 30 person-days of effort from evaluation team
- Only makes sense for large/expensive projects where architecture mistakes are unacceptable

For small, less risky projects, use Lightweight Architecture Evaluation (1/2 to 1 day meeting or less)

- Participants are fewer and internal to organization, so process & technologies are familiar to all

Design Decisions and Technical Debt

- We refer to design decisions with long-term negative consequences, such as increased cost of change or reduced customer responsiveness, as technical debt
- Technical debt, if not paid down, can increase over time



Summary

Architecture design transforms drivers into structures.

Architectural drivers include functional requirements, quality attributes, and constraints but also design purpose and architectural concerns.

ADD is a method that structures architecture design so it may be performed systematically.

Design concepts are building blocks from which the design is created. There are several important types: reference architectures, deployment patterns, architectural patterns, tactics, and externally developed components such as frameworks.

ADD can be performed in an agile way by using initial documentation (sketches) and a design Kanban board to track design advancement.

Final Discussion and Questions