



Software Architecture Principles and Practices

V13.0

Software Engineering Institute
Carnegie Mellon University
Pittsburgh, PA 15213

Software Architecture Principles and Practices

- Copyright 2018 Carnegie Mellon University. All Rights Reserved.
- 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.
- The view, opinions, and/or findings contained in this material are those of the author(s) and should not be construed as an official Government position, policy, or decision, unless designated by other documentation.
- NO WARRANTY. THIS CARNEGIE MELLON UNIVERSITY AND SOFTWARE ENGINEERING INSTITUTE MATERIAL IS FURNISHED ON AN "AS-IS" BASIS. CARNEGIE MELLON UNIVERSITY MAKES NO WARRANTIES OF ANY KIND, EITHER EXPRESSED OR IMPLIED, AS TO ANY MATTER INCLUDING, BUT NOT LIMITED TO, WARRANTY OF FITNESS FOR PURPOSE OR MERCHANTABILITY, EXCLUSIVITY, OR RESULTS OBTAINED FROM USE OF THE MATERIAL. CARNEGIE MELLON UNIVERSITY DOES NOT MAKE ANY WARRANTY OF ANY KIND WITH RESPECT TO FREEDOM FROM PATENT, TRADEMARK, OR COPYRIGHT INFRINGEMENT.
- [DISTRIBUTION STATEMENT A] This material has been approved for public release and unlimited distribution. Please see Copyright notice for non-US Government use and distribution.
- This material is distributed by the Software Engineering Institute (SEI) only to course attendees for their own individual study.
- Except for any U.S. government purposes described herein, this material SHALL NOT be reproduced or used in any other manner without requesting formal permission from the Software Engineering Institute at permission@sei.cmu.edu.
- Although the rights granted by contract do not require course attendance to use this material for U.S. Government purposes, the SEI recommends attendance to ensure proper understanding.
- Architecture Tradeoff Analysis Method[®], ATAM[®] and Carnegie Mellon[®] are registered in the U.S. Patent and Trademark Office by Carnegie Mellon University.
- DM18-0138



Software Architecture: Principles and Practices

Course Introduction

Introductions



Instructor introductions

Participant introductions

- name
- company/position
- background
- course expectations

Course Objectives



The Software Architecture: Principles and Practices course is a two-day course designed to

- familiarize participants with software architecture concepts and principles
- introduce participants to the relevance and role of software architectures and their impact on an organization
- provide participants with examples of software architectures in practice through case studies

Course-Entry Criteria

This course is designed for

- professionals who design and develop software or software-intensive systems
- acquisition professionals procuring software-intensive systems
- professionals who manage the development of software or software-intensive systems

Participants should have an understanding of

- the software development lifecycle
- modern software engineering concepts

Course Outcomes

Participants will have a better understanding of

- the relationships between system quality attributes and software architectures
- software architecture patterns and tactics, and their relationship to system quality attributes
- software architecture artifacts and documentation
- software architecture design
- software architecture evaluation
- architectural reuse
- how architecture practices relate to Agile practices

Course Strategy

Lectures will be used to introduce concepts.

Case studies will be used to illustrate architectural principles in practice.

Discussion sessions and group exercises will be used to engage students.

Course Agenda – 1



Day 1 sessions

- Introduction
- What Is Software Architecture?
- The Architecture Influence Cycle
- Case Study: The World Wide Web
- Understanding Quality Attributes
- Achieving Quality Attributes (Part 1)

Course Agenda – 2



Day 2 sessions

- Achieving Quality Attributes (Part 2)
- Documenting Software Architecture
- Architecture Evaluation
- Architectures in Agile Projects
- Final Thoughts

Rules of Engagement

We will be very busy over the next two 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 might 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 try to start on time.
- Participants must be present for all sessions in order to earn a course completion certificate.

Any Questions So Far?





Software Architecture: Principles and Practices

WHAT IS SOFTWARE ARCHITECTURE?

Module Objectives



After this module, students will understand

- what the term *software architecture* means
- the role and importance of software architecture in an organization
- how a software architecture comprises many different software structures

What Are Your Thoughts?

Apply your experience and background to define the following:

- enterprise architecture
- system architecture
- software architecture

Enterprise Architectures

Enterprise architecture is a term commonly used in business today, but what does it really mean?

Enterprise architecture is a means for describing business structures and the processes that connect them.¹

- It describes the flow of information and activities between various groups within the enterprise that accomplish some overall business activity.
- Enterprise architectures may or may not be supported by computer systems.
- Software and its design are not typically addressed explicitly in an enterprise architecture.

¹ Zachman, John A. "A Framework for Information Systems Architecture." IBM Systems Journal 26, 3 (1987): 276–292.

System Architecture

A system architecture is a means for describing the elements and interactions of a complete system including its hardware and software elements.

System architecture is concerned with the elements of the system and their contribution toward the system's goal but not with their substructure.

Where Does Software Architecture Fit?

Enterprise architecture and system architecture provide an environment in which software lives.

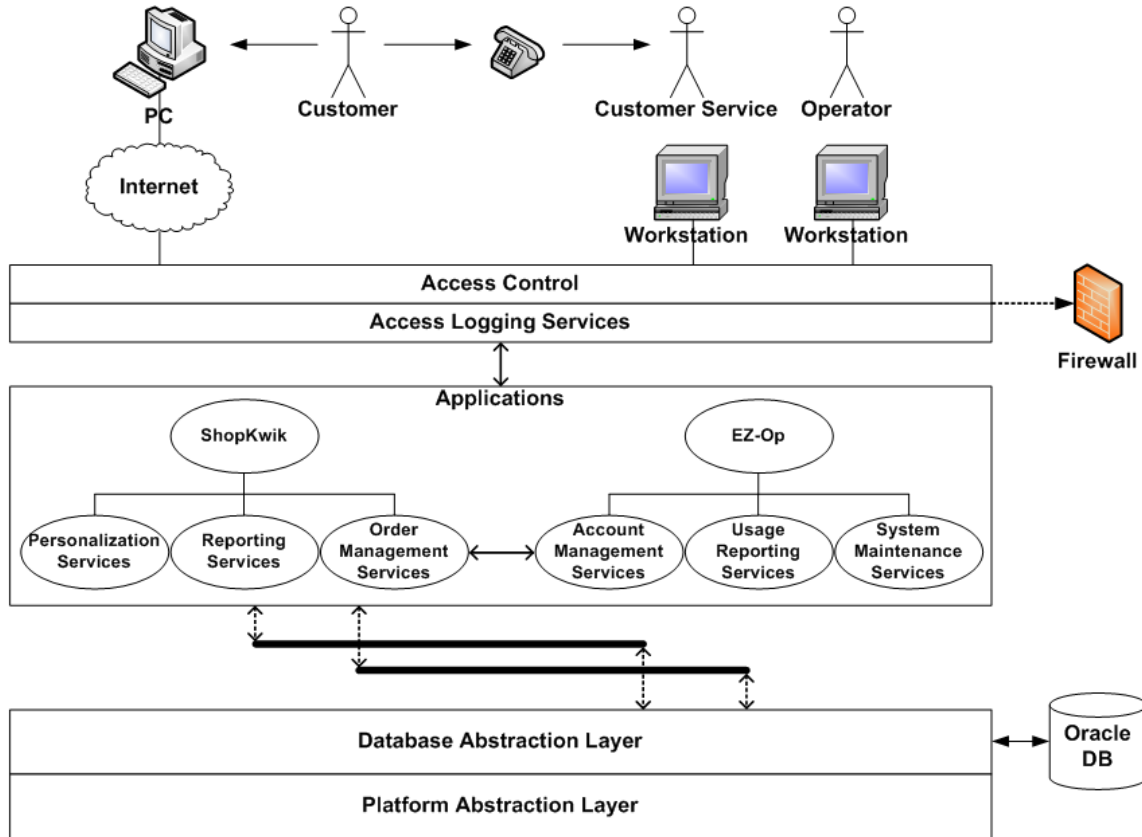
- Both provide requirements and constraints to which software architecture must adhere.
- Elements of both are likely to contain software architecture.

Typical Software Architecture – 1

A software architecture is often depicted using an ad hoc box-and-line drawing of the system that is intended to solve the problems articulated by the specification.

- Boxes show elements or “parts” of the system.
- Lines show relationships among the parts.

Typical Software Architecture – 2



What Can We Tell from This Picture?

The system consists of many elements?

The elements interact with each other over various networks?

Some elements represent layers and their relationships to one another?

The applications involved and the elements they comprise?

The system has multiple tiers?

The data, control, and communication mechanisms that are used?

What Does This Picture Omit? – 1

What is the nature of the elements?

- What is the significance of their separation?
- Do they exist at runtime?
- Do they run at separate times?
- Are they processes, programs, or hardware?
- Are they objects, tasks, functions, or processes?
- Are they distributed programs or systems?

What are the responsibilities of the elements?

- What do they do?
- What functions do they provide in the context of the system?

What Does This Picture Omit? – 2

What is the significance of the connections between the elements?

Do the elements

- communicate with each other?
- control each other?
- send data to each other?
- use each other?
- invoke each other?
- synchronize with each other?

What is the significance of how the elements are positioned on the diagram?

- For example, does ShopKwik use, call, control, and/or contain Personalization Services, Reporting Services, and Order Management Services?

What Is a Software Architecture?

“The software architecture of a system is the set of structures needed to reason about the system, which comprise software elements, relations among them, and properties of both.”¹

¹ Bass, L.; Clements, P.; & Kazman, R. *Software Architecture in Practice, Third Edition*. Boston, MA: Addison-Wesley, 2012.

Implications of Our Definition – 1

A software architecture is an abstraction of a system. The architecture

- defines elements and how they relate to one another
- suppresses details of what the elements do internally and purely local information about them; internal details are not architectural
 - The internal details of the elements do not affect how the elements are used or how they relate to or interact with other elements.

Implications of Our Definition – 2

The term “properties” refers to those assumptions that one element can make about another element such as

- which services it provides
- how it performs
- how it handles faults
- how it uses shared resources

Elements interact with each other via interfaces that partition details into public and private parts.

Architecture is concerned with the public side of this division.

Implications of Our Definition – 3

Systems can and do have many structures.

- No single structure can be the architecture.
- The set of candidate structures is not fixed or prescribed.
- Relationships and elements might be runtime related such as
 - “sends data to,” “invokes,” or “signals”
 - processes or tasks
- Relationships and elements might be nonruntime related such as
 - “is a submodule of,” “inherits from,” or “is allocated to team X for implementation”
 - a class or library

Implications of Our Definition – 4

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!

Implications of Our Definition – 5

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.

Patterns, Reference Models, and Reference Architectures – 1



Architects are likely to encounter the following terms:

- architectural patterns
- reference models
- reference architectures

But what exactly are they, and how do they relate to software architecture?

Patterns, Reference Models, and Reference Architectures – 2

An architectural pattern is a description of element and relationship types and a set of constraints on how they are used.

- Patterns define families of architectures such as client-server, pipe-and-filter, and so forth.
- Patterns exhibit known quality attribute properties.
- The selection of an architectural pattern is often the architect's first major design choice.
- The term *architectural style* has also been widely used to describe architectural patterns.

Patterns, Reference Models, and Reference Architectures – 3

A reference model is a division of functionality into elements and the data flow between them.

- Reference models include databases and compilers, among other things.
- For example, the elements of a compiler are well known:
 - parser
 - lexical analyzer
 - code generator
 - optimizer
 - ...and so forth
- Data flow and connectivity between these pieces are well established.

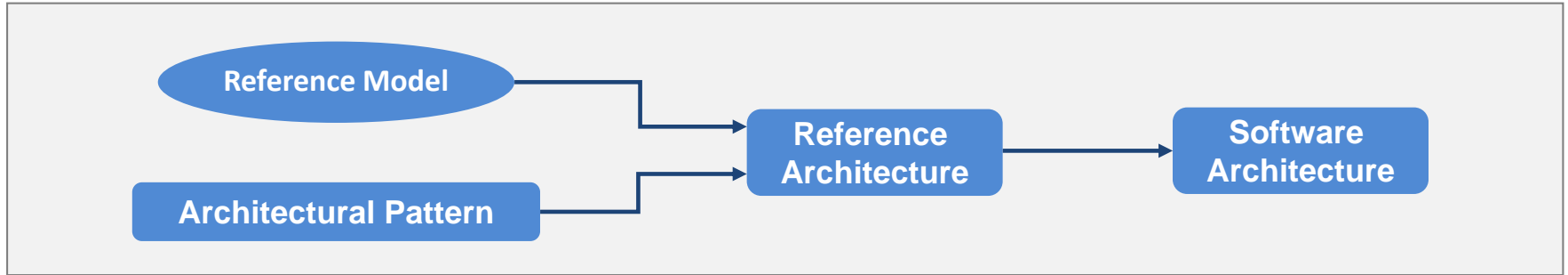
Patterns, Reference Models, and Reference Architectures – 4

A reference architecture is a reference model that is mapped onto the software elements that implement the functionality defined in the model.

- Mapping does not have to be one to one.
- Software elements might implement one function, parts of a single function, or many functions.
- Templates, standards, and tools speed up development and help implementers adhere to the constraints prescribed by the reference architecture.

Patterns, Reference Models, and Reference Architectures – 5

Architectural patterns, reference models, and reference architectures are not architectures; instead, they provide well-known solutions to various design problems that we can tailor and incorporate into our architectures.



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.

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.

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.

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.**

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.

Transferable, Reusable Abstraction – 3

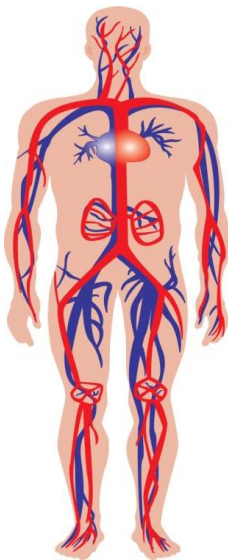
Architecture enables *template-based development*.

- An architecture embodies design decisions about how elements interact. Such decisions can be localized and written once.
- Templates can be used to code element interaction frameworks.
 - The developer fills in the unique part and reuses the common part.
- Templates speed up development and increase reliability.
 - The source of many errors is eliminated.
 - Fixing one error causes improvements in many places.
- Aspect-oriented design is a modern-day approach to address many of these concerns.

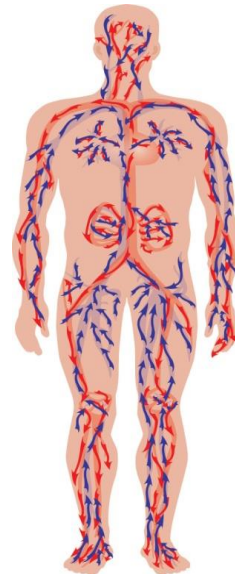
Architectural Structures – 1



A human body comprises multiple structures.



a static view of one of those structures



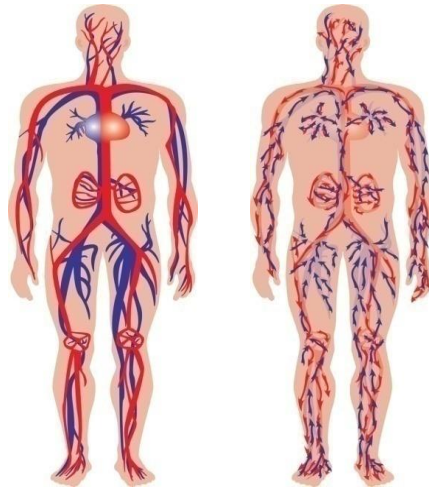
a dynamic view of that same structure

One body has many structures. So it is with software...

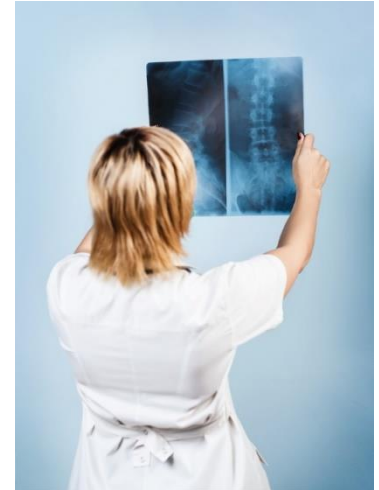
Architectural Structures – 2



These views are needed by a cardiologist...



**Different stakeholders are interested in different structures.
So it is with software...**



...but they won't do for an orthopedist.

Architectural Structures – 3



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.

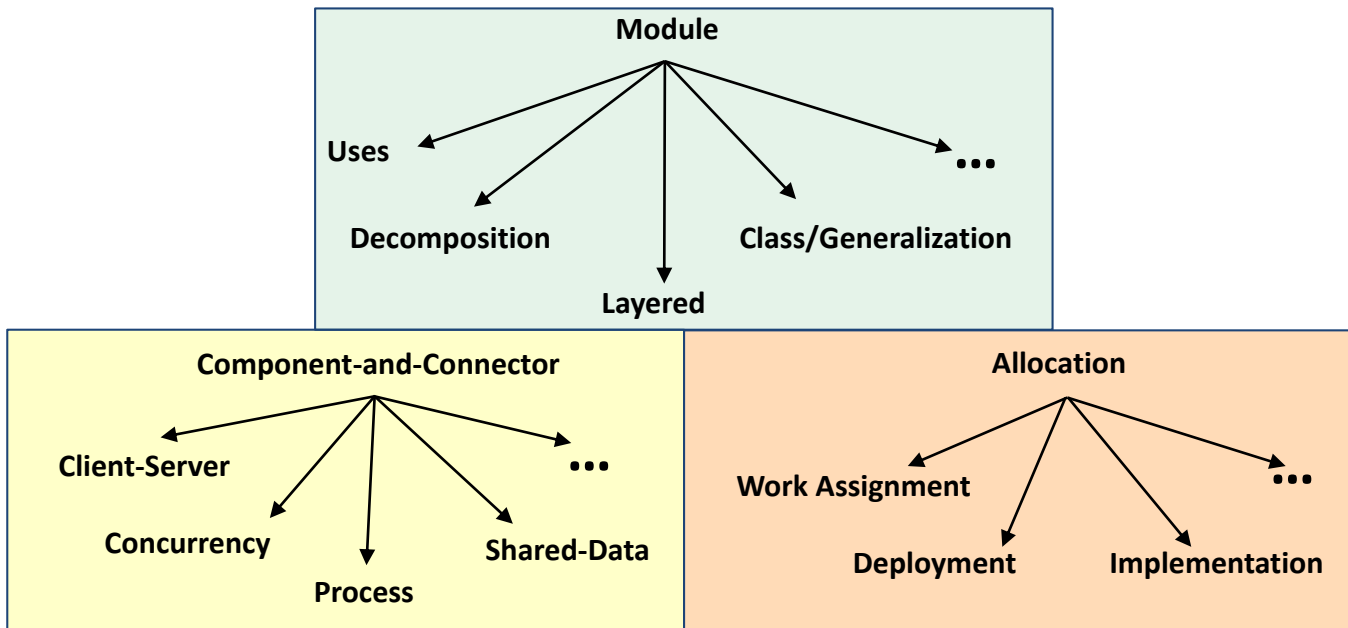
To communicate meaningfully about an architecture, we must make clear which structure or structures we are discussing.

Architectural Structures – 4

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

Architectural Structures Summary



Architects must focus on whatever structures will provide them with the most leverage in achieving the desired quality attributes of a system.

Module Summary

Architecture is important because it

- provides a communication vehicle among stakeholders
- is the embodiment of the most important design decisions
- is a transferable, reusable abstraction of a system

An architecture is composed of many structures, each of which comprises software elements and their relationships.

- Each structure provides engineering leverage and insight on different quality attributes.
- Architects select those structures that help to achieve the desired quality attributes in the implementation.



Software Architecture: Principles and Practices

THE ARCHITECTURE INFLUENCE CYCLE

Module Objectives



This session will familiarize participants with

- factors influencing software architectures
- factors influenced by software architectures
- the cycle of architecture influences

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

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

Influence of System Stakeholders

Stakeholders have an interest in the construction of a software system and might include

- customers
- users
- developers
- project managers
- marketers
- maintainers

Stakeholders have different concerns that they want to guarantee and/or optimize.

Customers

Customers are the people who pay for the system's development.

Customers are not always users.

Customers' concerns include the system's

- cost
- functionality
- lifetime
- development time/time to market
- quality
- flexibility to do many things on delivery day and over its lifetime

Users

Users are the people who will use the system including

- end users
- system administrators

End users are concerned primarily with a system's ease of use in terms of functionality to help them do their jobs.

System administrators are concerned with a system's ease of use in terms of their ability to

- configure the system
- manage users
- establish security and detect security breaches
- back up information
- recover and rebuild the system

Management

Management stakeholders include those managers from the

- development organization
- customer organization

Management's concerns include

- amortizing development costs
- maintaining the workforce's core competency and organizational training
- investing to achieve strategic goals
- keeping development costs as low as possible
- adhering to the development schedule
- maintaining product quality

Other Stakeholders

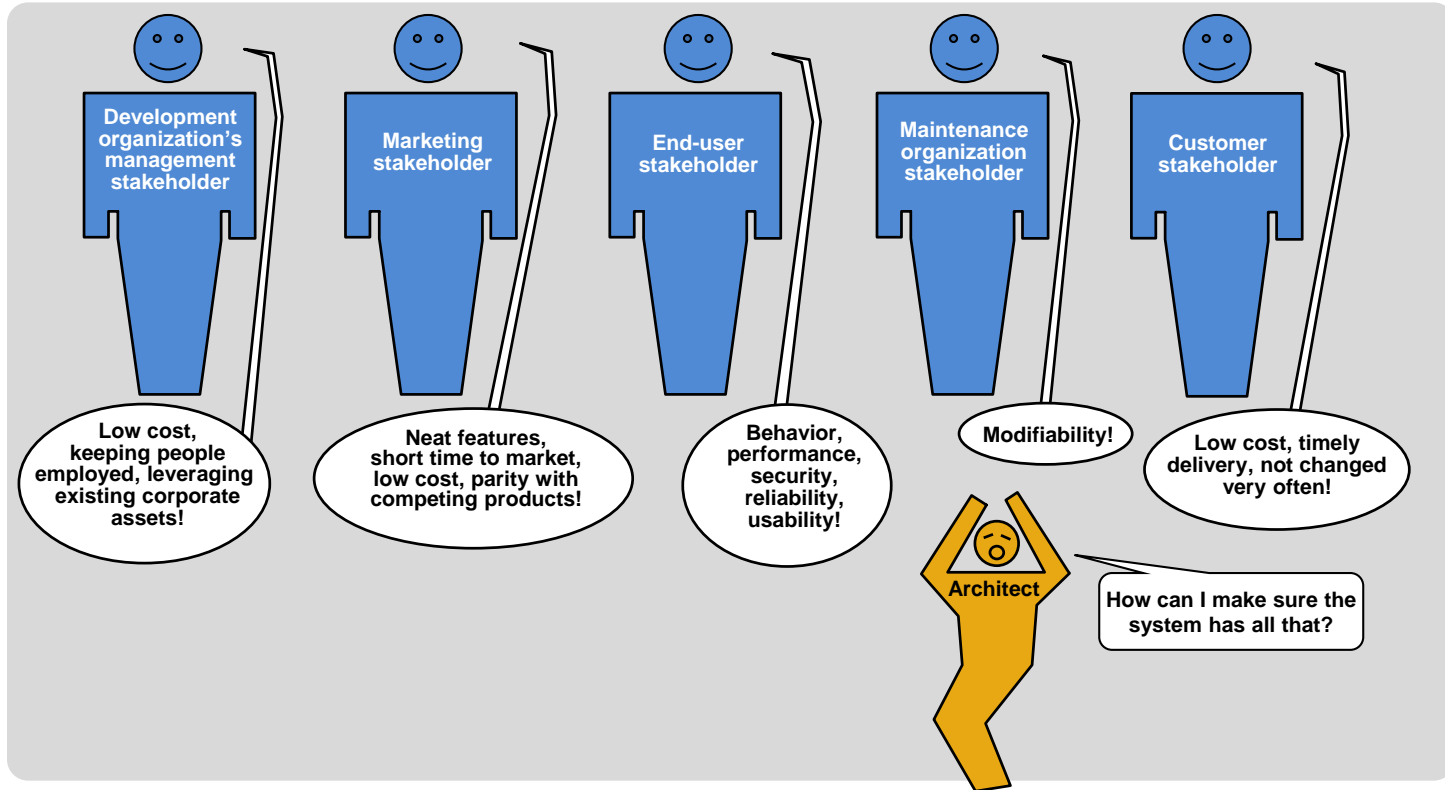
Developers are concerned about languages, technology, and the best mix to solve the problem.

Maintainers want a system they can fix, improve, tune, configure, deploy, extend, and so forth.

Marketers want features that meet or exceed those of the competition at a competitive price.

Who are the stakeholders for your systems?

Concerns of System Stakeholders



Stakeholder Involvement

The organizational goals and system properties required by the business/mission are rarely understood, let alone fully articulated.

Customers' quality attribute requirements are seldom documented, which results in

- goals not being achieved
- inevitable conflict between stakeholders

Architects must identify and actively engage stakeholders early in the lifecycle to

- understand the real constraints of the system
- manage the stakeholders' expectations
- negotiate the system's priorities
- make tradeoffs

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

Development Organization's Business Environment

Software architecture can be shaped by business/mission concerns, including

- time to market
- rollout schedule
- use of legacy systems
- available expertise
- support for existing products
- targeted markets
- political interests
- existing architectures
- plans for long-term infrastructure
- organizational structure
- projected lifetime of the system
- workforce utilization
- cost
- investment in existing assets

What factors influence your organization?

Factors Influencing Architectures

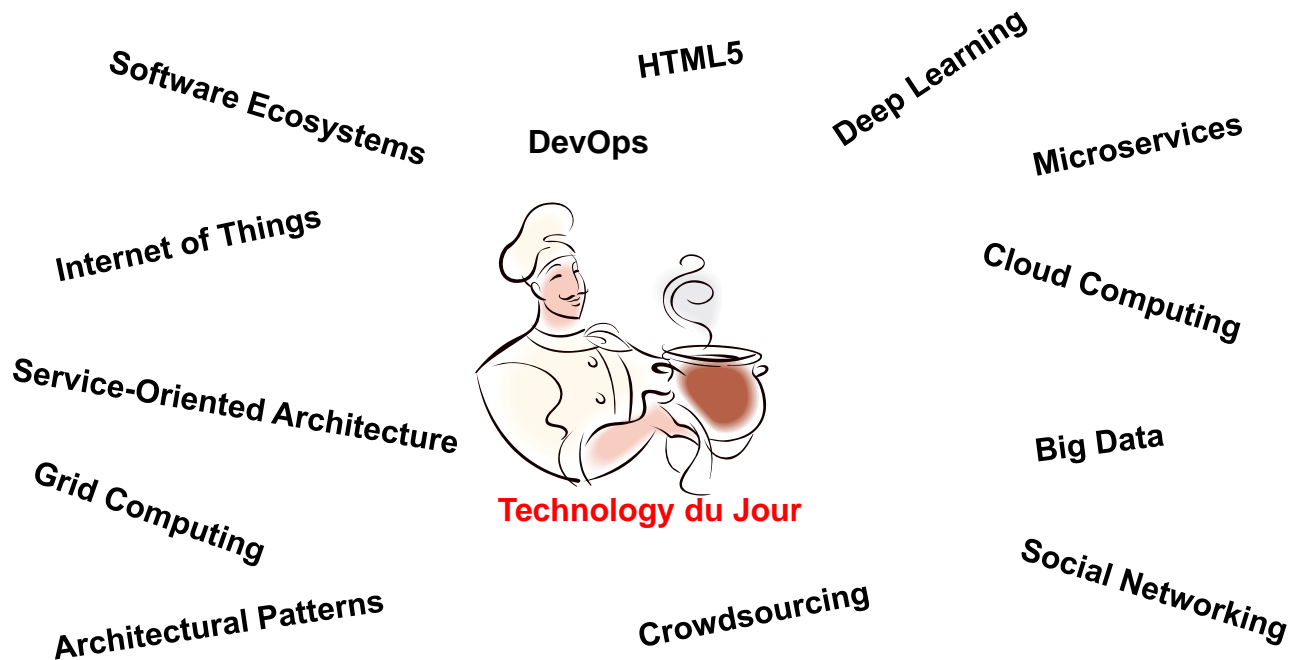


Architectures are influenced by

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

Influence of Technical Environment on Architectures

The technical environment that is current when an architecture is designed will influence that architecture.



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**

Influence of Architect's Professional Background on Architectures

Architects make choices based on their past experiences:

- Good experiences will lead to the replication of those prior designs that worked well.
- The methods, techniques, and/or technology that led to bad experiences will be avoided in new designs, even if those methods or techniques might work better in subsequent designs.
- An architect's choices might be influenced by education and training.

Influences on the Architecture

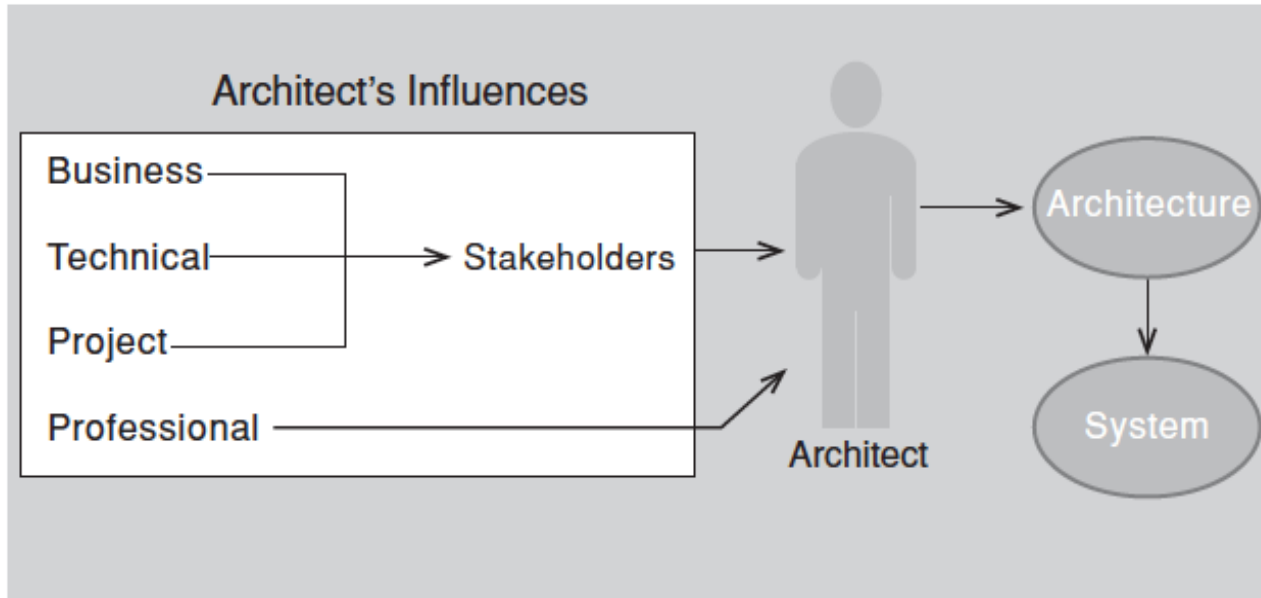


FIGURE 3.4 Influences on the architect

All citations are from the course textbook, *Software Architecture in Practice*, Third Edition, unless otherwise noted.

▶ Architectures Affect the Factors That Influence Them

Once the architecture is created and a system is built, both will affect

- the structure and goals of the development organization
- stakeholder requirements
- the architect's experience
- the technical environment

Architectures Affect the Factors That Influence Them

Once the architecture is created and a system is built, both will affect

- ▶ **• the structure and goals of the development organization**
- stakeholder requirements
- the architect's experience
- the technical environment

How Architectures Affect the Development Organization – 1

Architectures can influence the structure of the development organization.

By prescribing the structure for a system, architectures also prescribe the units of software that must be implemented and integrated.

In turn, software units are the basis for

- team formation
- development, test, and integration activities
- resource allocation in schedules and budgets

How Architectures Affect the Development Organization – 2

Architectures can influence the goals of an organization.

A successful system built from an architecture can enable a company to establish a foothold in a particular market area:

- The architecture can provide opportunities for the efficient production and deployment of similar systems.
- The organization might adjust its goals to take advantage of new market opportunities.

Architectures Affect the Factors That Influence Them

Once the architecture is created and a system is built, both will affect

- the structure and goals of the development organization
- ▶ • **stakeholder requirements**
- the architect's experience
- the technical environment

How Architectures Affect Stakeholder Requirements

Architectures can influence customers' requirements:

- Knowledge of similarly fielded systems leads customers to ask for particular kinds of features.
 - Stakeholders learn the language of the architecture, perceive the benefits of the architecture, and want similar kinds of architectures—such as service-oriented, client-server, Java Enterprise Edition (JEE), .NET, peer-to-peer, and so forth.
 - Stakeholders will demand quality architectures in future systems.
- Stakeholders will alter their system requirements based on the availability of existing systems and components.

Architectures Affect the Factors That Influence Them



Once the architecture is created and a system is built, both will affect

- the structure and goals of the development organization
- stakeholder requirements
- ▶ **• the architect's experience**
- the technical environment

How Architectures Affect the Architect's Experience



The process of building systems influences the architect's experience base. This, in turn, influences how subsequent systems in the organization are constructed:

- Successful systems built around a technology, tool, or method will engender future systems that are built in the same way.
- The architecture for a failed system is less likely to be chosen for future projects.

Architectures Affect the Factors That Influence Them



Once the architecture is created and a system is built, both will affect

- the structure and goals of the development organization
- stakeholder requirements
- the architect's experience
- ▶ **• the technical environment**

How Architectures Affect the Technical Environment

Occasionally, a system, an architecture, or a collection of technologies, such as those listed below, will change the foundations of the technical environment in which architects operate and learn:

- relational databases
- compiler generators
- the Internet
- n-tier client-server
- spreadsheets
- GUIs/windowing systems
- the World Wide Web
- cloud infrastructure
- dependency injection
- lambda architecture
- NoSQL databases
- service-oriented architecture
- microservices
- container infrastructure

A Cycle of Influences

The relationships among business/mission goals, product requirements, architect experience, architectures, and fielded systems form a cycle with feedback loops that an organization can use and manage to

- handle growth
- expand its enterprise area
- take advantage of previous investments in architectures and system building

Architecture Influence Cycle (AIC)

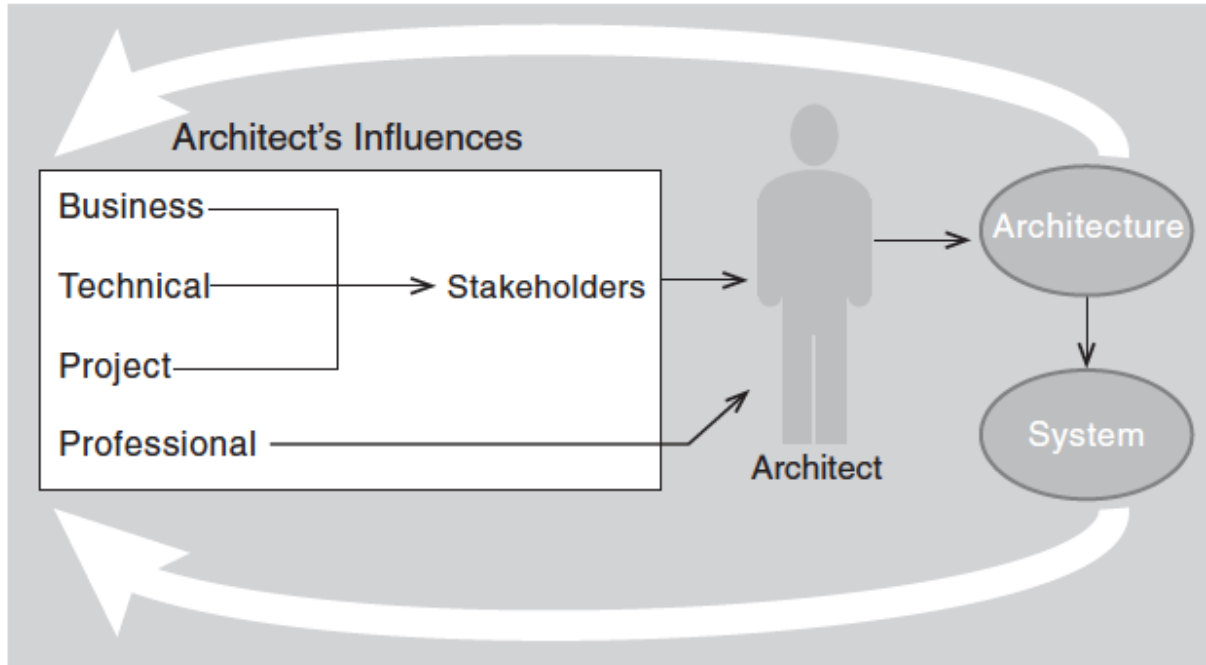


FIGURE 3.5 Architecture Influence Cycle

Module Summary

Architecture involves more than just the technical requirements for a given system. It also involves nontechnical factors such as the

- architect's background
- organization (structure, core competency, experience)
- business/mission goals

Architecture influences the factors that affect it, such as

- architects learning from experience
- the organization being changed
- new markets being entered
- the state of the practice being advanced

Software Architecture: Principles and Practices

THE WORLD WIDE WEB— A CASE STUDY IN ACHIEVING QUALITY ATTRIBUTES

Module Objectives

This case study will use the World Wide Web (WWW) to illustrate how architectural decisions lead to the achievement of quality attribute requirements. Participants will

- understand the relationship between the architecture of the WWW and the architectural environments that spawned it
- appreciate the architectural decisions made by the WWW's software developers to support scalability
- appreciate how shifting requirements can cause an architecture to change over time
- see how a system traverses the Architecture Influence Cycle (AIC)

Historical Notes



Image credit: CC BY-SA 3.0
https://commons.wikimedia.org/wiki/Tim_Berners-Lee#/media/File:First_Web_Server.jpg

March, 1989 – While at CERN,¹ Tim Berners-Lee wrote a proposal titled “Information Management: A Proposal.”

- He used the concept of hypertext, the foundation of the WWW.
- The proposal was rejected initially; he recirculated it in 1990.
- He developed the initial browser on his NeXT system by November 1990.
- A WYSIWYG (what you see is what you get) browser and editor illustrated the concept of using hypertext to display information.
- A line-mode browser was released by anonymous File Transfer Protocol (FTP) to the general public in January 1992.

¹ CERN is the Organisation Européenne pour la Recherche Nucléaire, which in English translates to the European Organization for Nuclear Research.

Original Requirements – 1

Provide remote access across networks – Any information had to be accessible from any machine on the CERN network.

Support heterogeneity – The system could not be limited to specific software, a piece of hardware, or an operating system.

Avoid centralized control of the system.

Support access to existing data and databases.

Support private links to data and databases.

Display on 24 X 80 character ASCII terminal – At the inception of the Web, using graphics was optional.

Original Requirements – 2

Support data analysis – Let users search across various data sources to look for anomalies, regularities, irregularities, and so forth.

Support live links – Given that information changes all the time, there must be some way of updating a user's view of it.

Original Quality Attributes

Platform portability

- had to allow a variety of users with different desktop hardware and operating systems to share the same data
 - Hardware included Intel, Apple, NeXT, Unix, Commodore, and others.
 - Different versions of operating systems were used on these machines as well.

Scalability/extensibility

- had to allow adding and removing users
- had to support emerging hardware and operating systems
- had to allow the capacity for servers and data to be extended

Original Non-Requirements

Graphics and multimedia

Different link types

Visual history

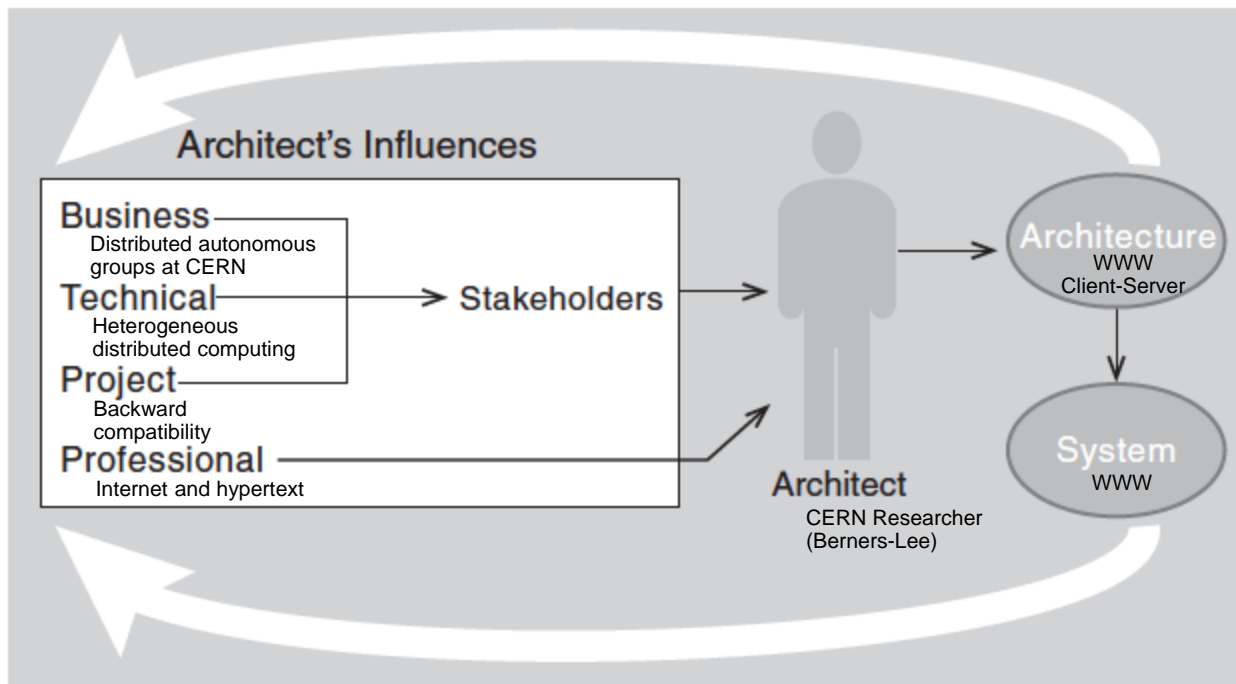
Copyright enforcement

Security

Privacy

Markup format

Initial Architecture Influence Cycle for the Web

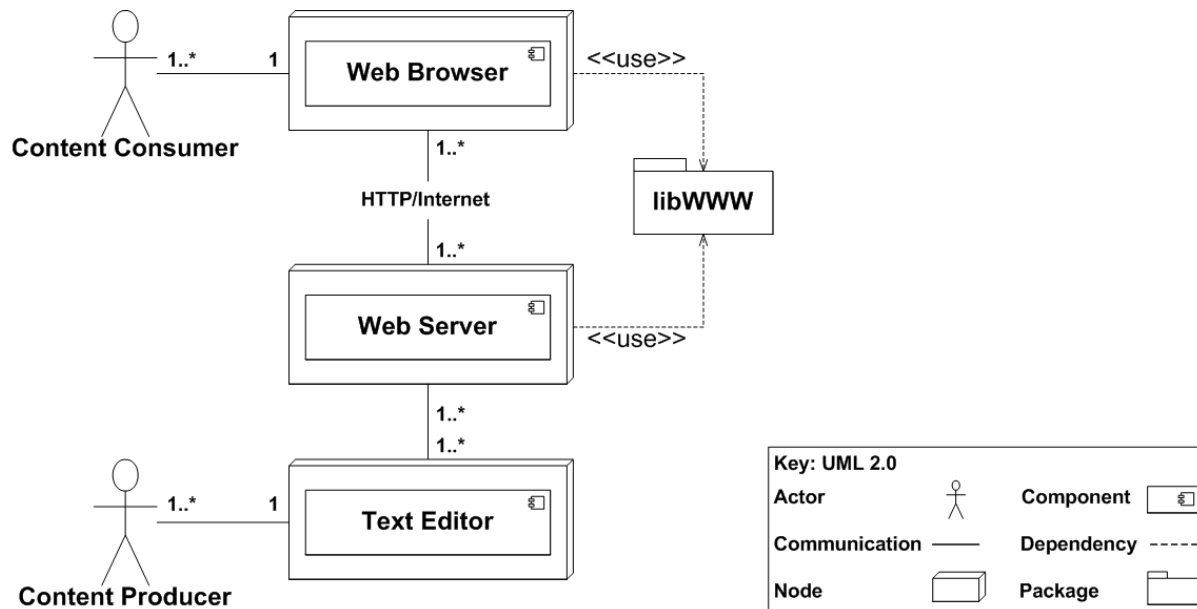


Architectural Choices

Berners-Lee made these key architectural choices:

- used an existing internet protocol (TCP/IP), but added another layer to encapsulate other network protocols (e.g., Hypertext Transfer Protocol [HTTP])
- provided a layered architecture
- used a client-server architecture
 - Very thin, portable clients called browsers interpret Hypertext Markup Language (HTML) documents.
 - Browsers talk to servers that host data in many formats.

WWW Architectural Approach

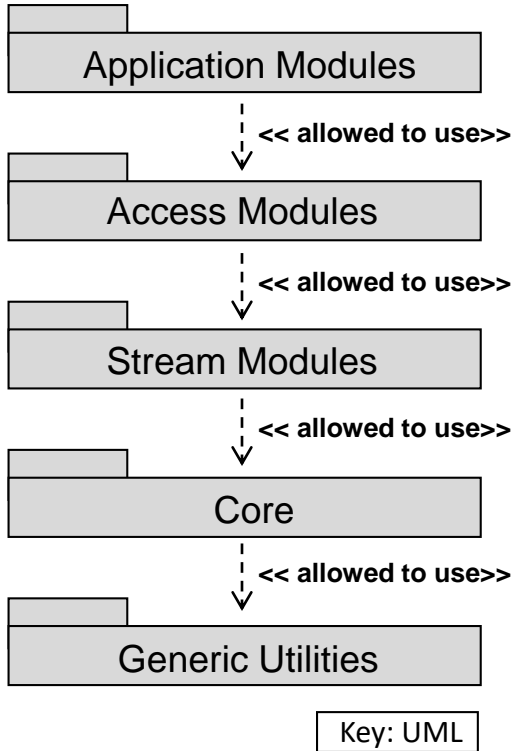


This is a Client-Server view of the system.

libWWW provides protocol support and masks platform details.

Remote access is achieved by building WWW on top of the Internet.

Meeting the Requirements: libWWW



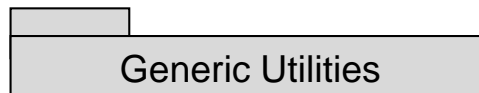
The developer's view was a layered architecture that

- illustrated major layers of libWWW
- assigned responsibilities to each layer

Generic Utilities

Basic building blocks for the system

- network management
- data types
- string manipulation utilities



Key: UML

Core

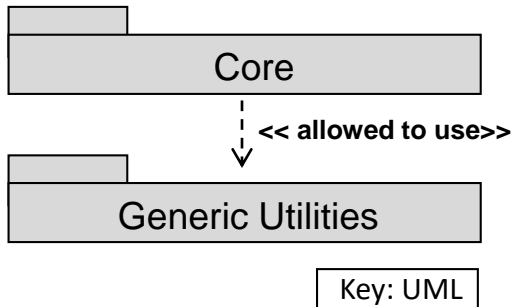
Is the skeletal functionality of a Web application such as

- network access
- data management
- parsing
- Logging

Provides a standard interface for Web applications

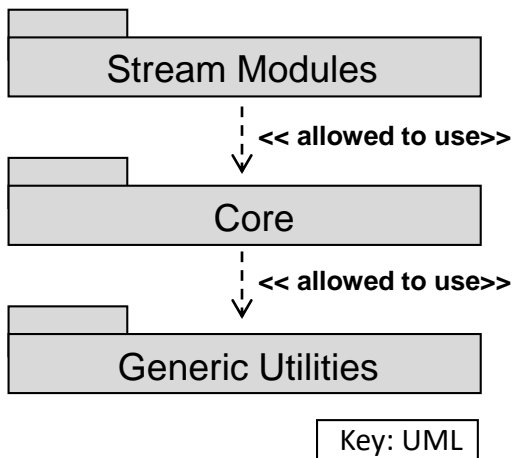
The actual functionality is provided by plug-ins:

- Plug-ins are registered at runtime and support various protocols, data formats, and translation.

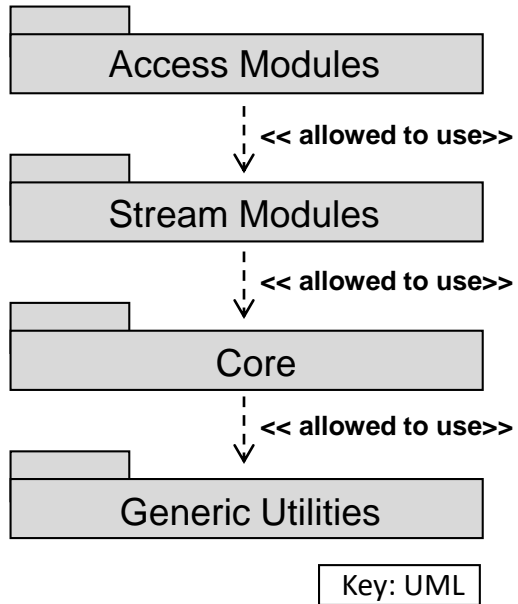


Stream Modules

Provide an abstraction of stream data used by all data transported between the application and the network.



Access Modules

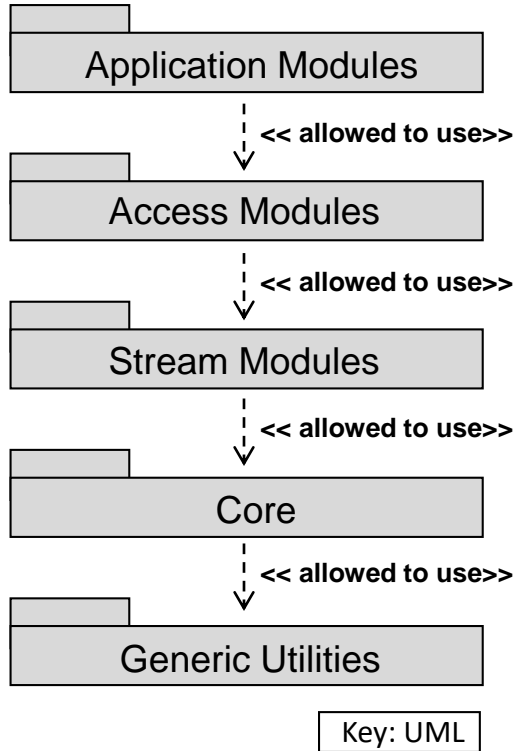


Provide a set of network-protocol-aware modules

- HTTP/HTTP Secure (HTTPS)
- Wide Area Information Server (WAIS)
- FTP
- Telnet
- rlogin
- Gopher
- local file system

Make it easy to add new protocols

Application Modules

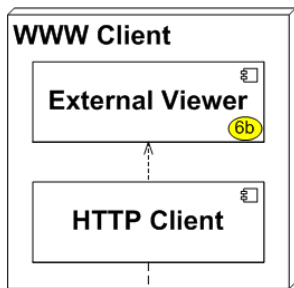


Are not actual applications; rather, they are sets of application programming interfaces (APIs) that provide functionality useful for writing end-user applications.

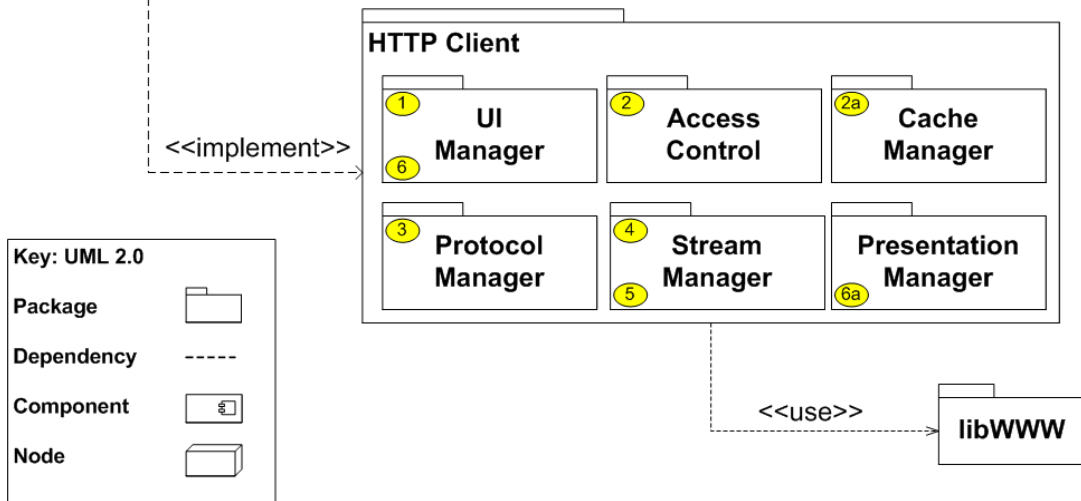
Provide services such as

- caching
- logging
- registering proxy servers and gateways
- history maintenance

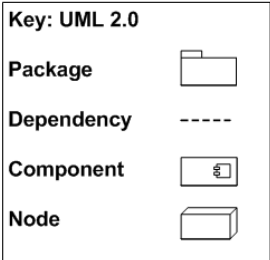
Typical WWW Client



- 1 Receive user request
- 2 Check cache for requested URL
- 3 Determine request type and protocol
- 4 Send request to server
- 5 Receive requested information
- 6 Display requested information



<<implement>>



Elements of the HTTP Client

User Interface (UI) Manager is responsible for the look and feel of the UI (e.g., Web browser).

Presentation Manager delegates document types to viewers:

- external: QuickTime movies, MP3 audio
- internal: HTML, Graphics Interchange Formats (GIFs) to UI Manager

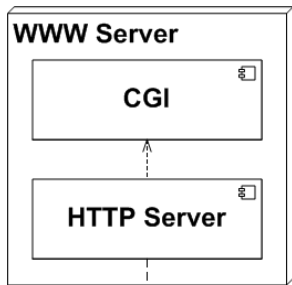
UI Manager captures the user's URL request and passes it to Access Manager.

Access Manager determines whether the URL has been cached; if it hasn't, the component initiates retrieval through Protocol Manager and Stream Manager.

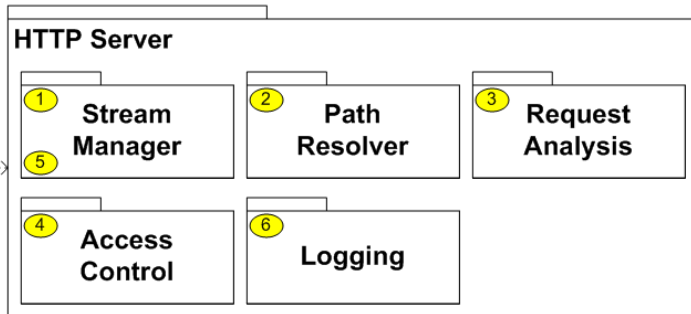
Typical WWW Server

Serve HTML back to Client

- ① Receive URL request
- ② Determine file location
- ③ Analyze request
- ④ Determine if access is permitted
- ⑤ Serve HTML back to Client
- ⑥ Log activity



<<implement>>



<<use>>



Key: UML 2.0

Package

Dependency

Component

Node

Elements of the HTTP Server

The HTTP server receives a URL request and passes it to the Path Resolver, which determines the file location for the document (assuming local).

The HTTP server checks the access list to see if access is permitted and then gets the document from the file system.

Common Gateway Interface (CGI) is a special document type that allows customized access to other data or programs.

HTML is served by the HTTP server back to the Client.

Extending the Server

CGI allows

- dynamic documents
- the addition of data to existing databases
- customized queries
- clickable images

CGI scripts

- can be written in a variety of languages
- run as separate processes

Problems with CGI

Extensibility of the server is principally supported by CGI scripts in libWWW applications.

However, CGI

- has security holes
- is not portable

This part of the architecture limited the future growth of applications based on libWWW.

Achieving Quality Attributes

Goal	How Achieved
Remote access	Build the Web on top of the Internet, and adhere to defined protocols.
Interoperability	Use the Layers pattern in libWWW.
Extensibility of software	Isolate protocols and data types in libWWW. Abstract common services. Hide information. Use configuration files.
Extensibility of data	Keep data items independent (HW, SW, format, etc.) except for URL references.
Scalability	Use the Client-Server pattern. Allow concurrency.

libWWW Lessons Learned – 1

libWWW has had many releases. Lessons learned from the many iterations include the following:

- Well-defined APIs are required.
- Functionality must be layered.
- APIs must support a dynamic, open-ended set of features that can be added or replaced at runtime.
- Processes must be thread safe.

libWWW Lessons Learned – 2

Not all these goals were met!

- The Core makes assumptions about the essential services implemented in other layers.
 - Not all features can be replaced dynamically.
 - Feature replacement still takes a restart so that new services can be registered.
- Since libWWW is meant to run on many different platforms, it cannot depend on any one thread model.
 - It uses pseudo-threads that provide some, but not all, of the required functionality.

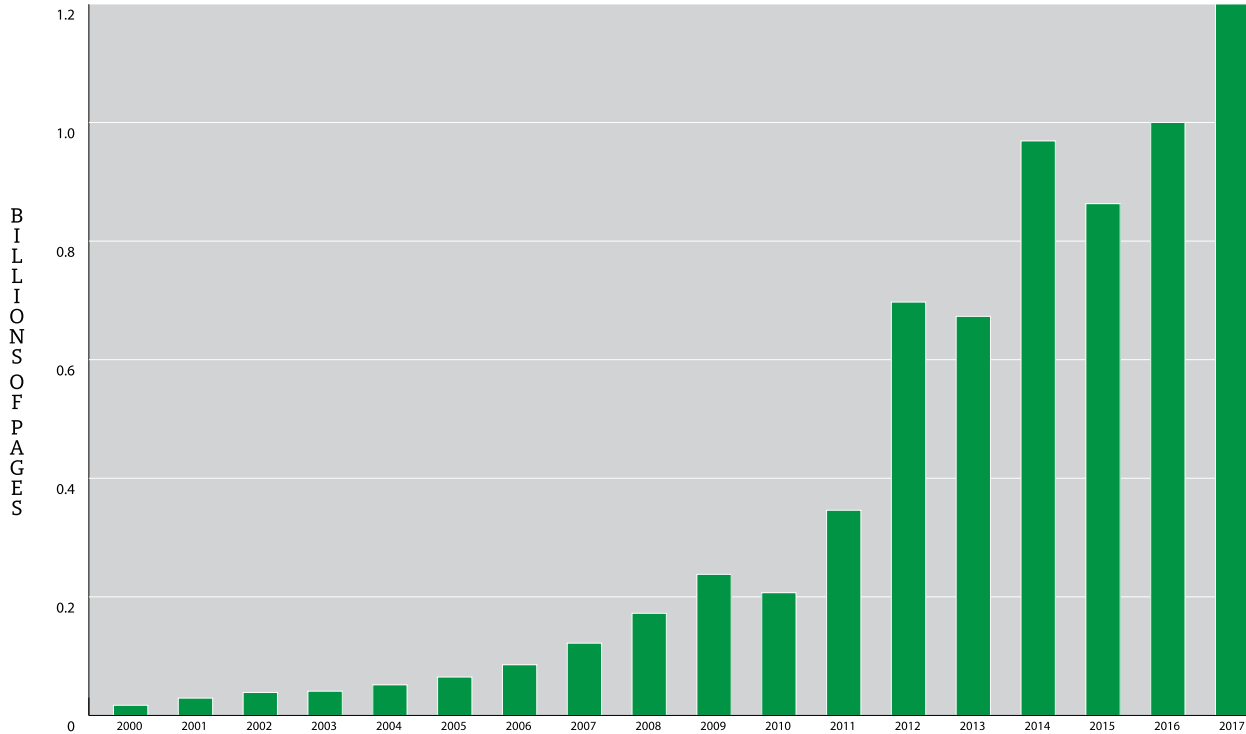
Another Cycle Through the AIC

The incredible success of the Web has resulted in unprecedented interest from business and hence unprecedented pressure on the architecture, via the AIC.

Business requirements have dominated the Web architecture.

These requirements have now strayed considerably from the original vision at CERN.

WWW Growth¹



¹ Graphic adapted from <http://www.internetlivestats.com/total-number-of-websites/#trend>

Requirements Come and Go

With the incredible success and growth of the Web came new requirements.

For example,

- with e-commerce came requirements for security and privacy
- with cyberporn came requirements for content labeling (Platform for Internet Content Selection)

If these requirements had appeared in the original proposal, what would their fate have been?

New Requirements – 1

High performance – Popular websites will have tens of millions of hits per day. Users expect low latency and will not tolerate their requests being refused.

High availability – E-commerce sites are expected to be available 24/7.

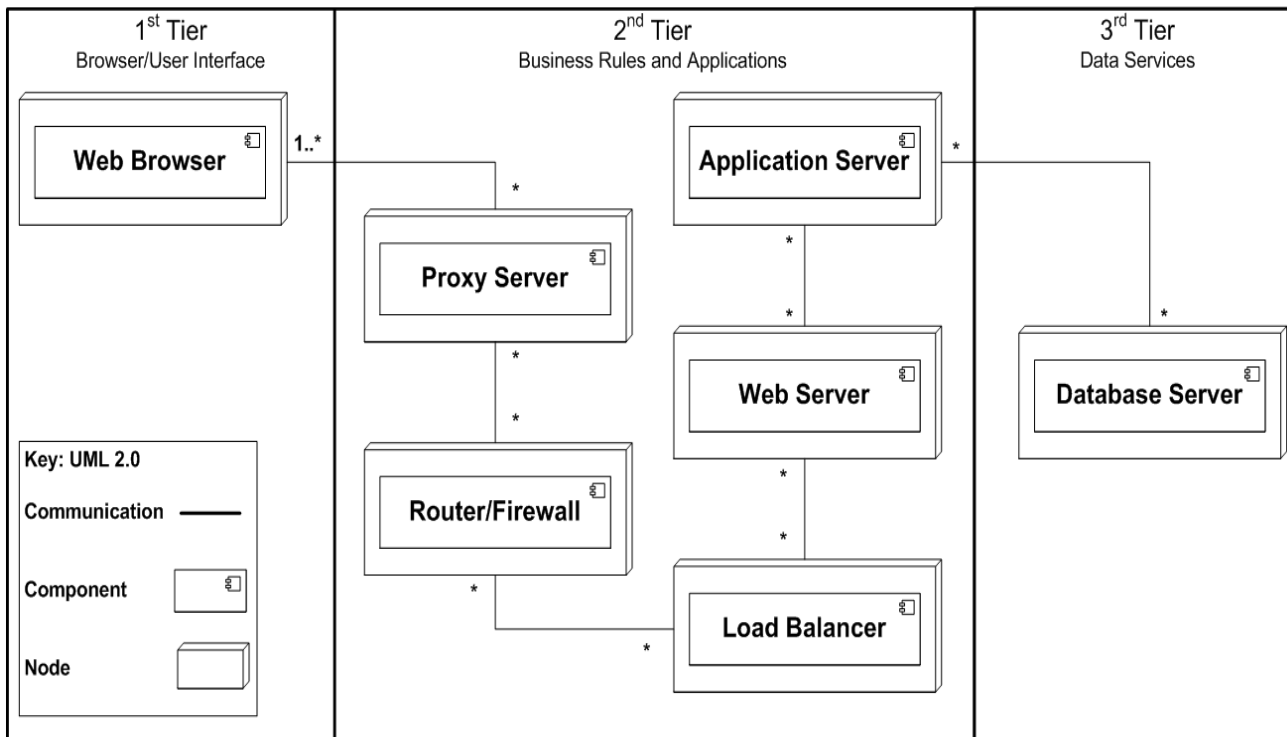
Scalability – As businesses and their websites grow, the amount of data that can be stored and accessed on the Web must also grow without impacting performance.

New Requirements – 2

Security – Users must be assured that any sensitive information will not be compromised. Operators of the Web must be able to prevent/detect attacks that would render it unusable.

Modifiability – E-commerce sites change frequently, in many cases daily, so their content must be very simple to change.

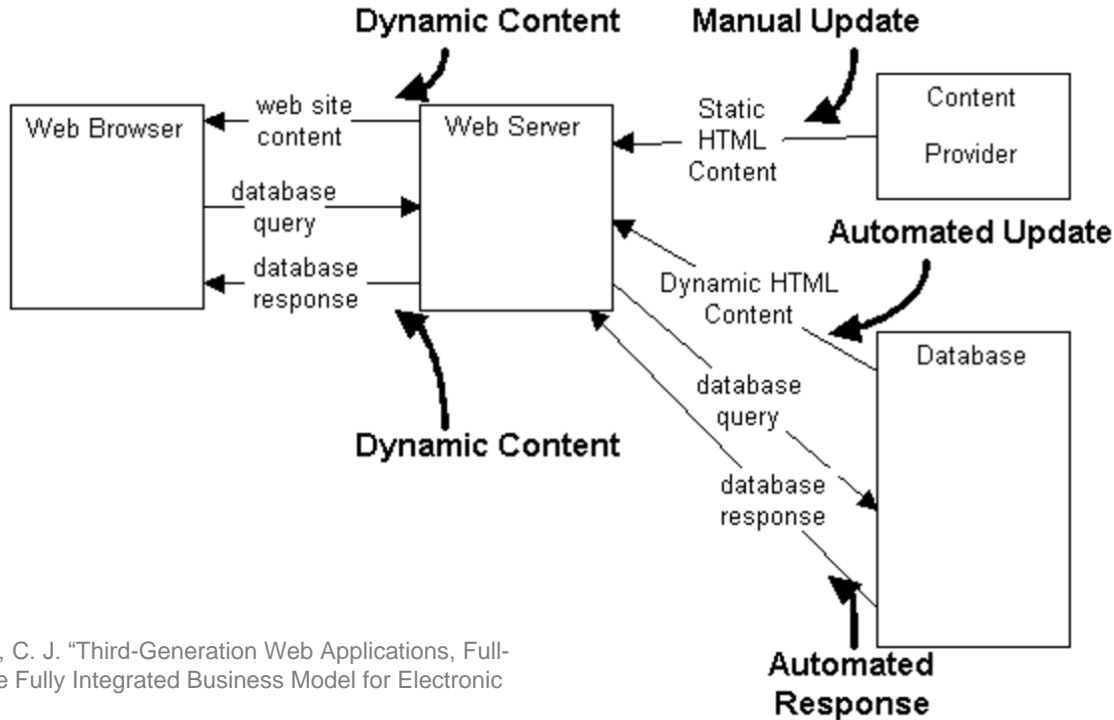
Second-Generation System Architecture



Group Exercise

How do decisions in the second-generation WWW system architecture address the new quality attribute requirements?

Third-Generation System Architecture¹



¹ Copeland, W. K. & Hwang, C. J. "Third-Generation Web Applications, Full-Service Intranets, EDI: The Fully Integrated Business Model for Electronic Commerce." INET'97. https://www.isoc.org/inet97/proceedings/C5/C5_2.HTM#s3

Changes in the AIC for the Web – 1

Several types of organizations (service providers and content providers) provide the technical environment.

Service providers produce the software—browsers, servers, databases, application servers, security technologies (such as firewalls), transaction servers, networks, and routers.

Content providers produce the data.

Changes in the AIC for the Web – 2

There is heavy competition in all these areas.

Open source projects, aside from the World Wide Web Consortium (W3C), continue to have considerable influence (such as the Apache project).

CERN has *no* role.

Languages such as PHP, JavaScript, HTML5, AJAX, and Ruby are changing the way functionality is developed and delivered on the Web.

Module Summary

The Web has been so successful because of how its key quality attributes were satisfied, for example, through

- a flexible client-server architecture
- well-defined protocols
- no centralized control
- libWWW layers

These structures have withstood the test of time and been reinvented in the face of dramatic new requirements.

The success of the Web has meant that the AIC has been traversed multiple times in just a few years, creating new business environments, opportunities, and technologies.



Software Architecture: Principles and Practices

UNDERSTANDING QUALITY ATTRIBUTES

Module Objectives

This module will familiarize participants with

- quality attributes
- the effect of architectural decisions on quality attributes
- how to express quality attribute requirements via scenarios
- a method for eliciting quality attributes

Quality Attributes

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

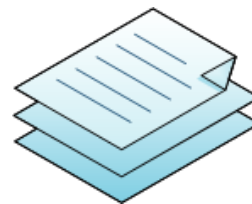
Some examples of quality attributes by which stakeholders judge the quality of software systems are

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

Stakeholders and Quality Attributes



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 Attributes and Architecture

Quality attribute requirements for software systems have a significant influence on the architecture of those systems.

The degree to which a software system meets its quality attribute requirements depends on its architecture.

Architectural decisions are made to promote various quality attributes.

A change in an architecture to promote one quality attribute often affects other quality attributes.

Architecture provides the foundation for achieving quality attributes but is useless if not adhered to in the implementation.

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.

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.

Quality Attribute Scenarios – 1

A solution to the problem of describing quality attributes is to use quality attribute scenarios to clearly characterize them.

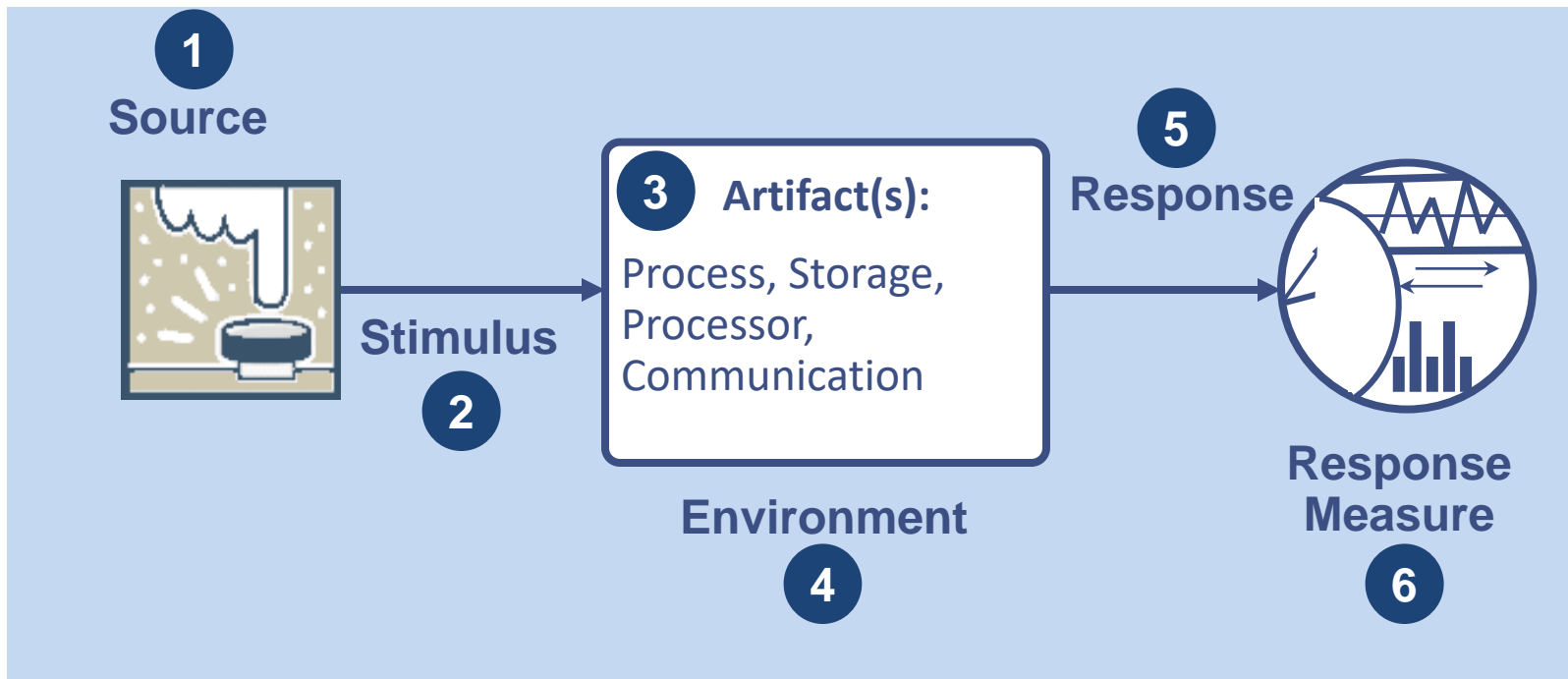
A quality attribute scenario is a short description of how a system is required to respond to some stimulus.

Quality Attribute Scenarios – 2

A quality attribute scenario consists of six parts:

1. **source** – an entity that generates a stimulus
2. **stimulus** – a condition that affects the system
3. **artifact(s)** – the part of the system that was stimulated by the stimulus
4. **environment** – the condition under which the stimulus occurred
5. **response** – the activity that results because of the stimulus
6. **response measure** – the measure by which the system's response will be evaluated

Parts of a Quality Attribute Scenario



Concrete Scenarios

Quality attribute scenarios that are created for a specific system are called **concrete scenarios. For example:**

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

General Scenarios

General scenarios are system-independent scenarios that allow stakeholders to communicate more effectively about quality attribute requirements and can assist stakeholders in developing concrete scenarios.

General scenarios can be developed for any quality attribute.

General scenarios are given in your textbook for the following quality attributes:

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

Using General Scenarios – 1

General scenario for availability (see p. 86, Table 5.3)

Source	Internal/external: people, hardware, software, physical infrastructure or environment
Stimulus	Fault: omission, crash, incorrect timing, incorrect response
Artifact(s)	Processors, communications channels, persistent storage, processes
Environment	Normal operation, startup, shutdown, repair mode, degraded operation, overloaded operation
Response	<p>Prevent the fault from becoming a failure.</p> <p>Detect the fault:</p> <ul style="list-style-type: none"> • Log the fault. • Notify the appropriate entities (people or systems). <p>Recover from the fault:</p> <ul style="list-style-type: none"> • Disable source of events causing the fault. • Be temporarily unavailable while repair is being effected. • Fix or mask the fault/failure, or contain the damage it causes. • Operate in a degraded mode while repair is being effected.
Response Measure	<p>Time or time interval when the system must be available</p> <p>Availability percentage (e.g., 99.999%)</p> <p>Time to detect the fault / Time to repair the fault</p> <p>Time or time interval in which the system can be in degraded mode</p> <p>Proportion (e.g., 99%) or rate (e.g., up to 100 per second) of a class of faults that the system prevents or handles without failing</p>

Using General Scenarios – 2

Select items from a general scenario to form a concrete scenario.

Source	Internal/external: people , hardware , software , physical infrastructure or environment
Stimulus	Fault: omission , crash , incorrect timing , incorrect response
Artifact(s)	Processors , communications channels , persistent storage , processes
Environment	Normal operation , startup , shutdown , repair mode , degraded operation , overloaded operation
Response	<p>Prevent the fault from becoming a failure.</p> <p><u>Detect the fault:</u></p> <ul style="list-style-type: none"> • Log the fault. • Notify the appropriate entities (people or systems). <p><u>Recover from the fault:</u></p> <ul style="list-style-type: none"> • Disable source of events causing the fault. • Be temporarily unavailable while repair is being effected. • Fix or mask the fault/failure, or contain the damage it causes. • Operate in a degraded mode while repair is being effected.
Response Measure	<p>Time or time interval when the system must be available</p> <p>Availability percentage (e.g., 99.999%)</p> <p>Time to detect the fault / Time to repair the fault</p> <p>Time or time interval in which the system can be in degraded mode.</p> <p>Proportion (e.g., 99%) or rate (e.g., up to 100 per second) of a class of faults that the system prevents or handles without failing.</p>

Using General Scenarios – 3

Resulting concrete scenario:

When a processor crash occurs during normal operation, the system should detect the failure, record it, notify the operator, and continue to operate in a degraded mode. The system must be back to normal operation in less than 15 minutes.

Source	Internal hardware
Stimulus	Crash
Artifact(s)	Processors
Environment	Normal operation
Response	Detect the fault: <ul style="list-style-type: none"> • Log the fault. • Notify the appropriate entities (people or systems). Recover from the fault: <ul style="list-style-type: none"> • Operate in a degraded mode while repair is being effected.
Response Measure	System can be in degraded mode no more than 15 minutes.

Group Exercise: Developing Concrete Scenarios – 1

Develop a six-part concrete scenario for the system and quality attribute of your choice.

Using the worksheets on the following pages,

Choose a system.
Develop a concrete scenario for the chosen system.
Present your results.

Group Exercise: Developing Concrete Scenarios – 2

Worksheet

Choose a system for which you will develop a concrete quality attribute scenario. For example, you could choose a work-related system, an automated teller machine (ATM), or a patient-monitoring system.

Group Exercise: Developing Concrete Scenarios – 3

Worksheet

Develop a concrete scenario that will characterize one of the quality attribute requirements for the chosen system (e.g., performance, availability).

Use one of the following general scenarios from your textbook to assist you:

Availability – p. 86, Table 5.3	Security – p. 150, Table 9.1
Interoperability – p. 108, Table 6.2	Testability – p. 163, Table 10.1
Modifiability – p. 120, Table 7.1	Usability – p. 176, Table 11.1
Performance – p. 134, Table 8.1	

Group Exercise: Developing Concrete Scenarios – 4

Worksheet

Develop a concrete scenario (continued).

Stimulus Source	
Stimulus	
Artifact(s)	
Environment	
Response	
Response Measure	

Group Exercise: Developing Concrete Scenarios – 5

Worksheet

Present your concrete scenario.

Quality Attributes in Practice

The quality attributes we have discussed in this module are fairly common in nearly all systems.

In practice, many other quality attributes exist that are domain and product specific (e.g., “calibratability”).

Specific quality attributes may be important to system stakeholders but have no general relevance.

It’s not important for all domain-specific quality attributes to be mapped to one of those described in this module.

Software architects need to be aware of the *driving* quality attributes—whatever they are!

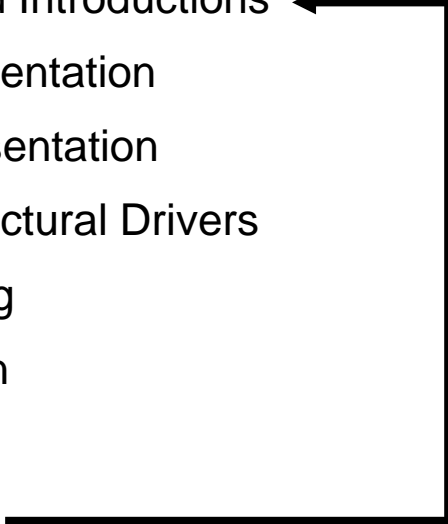
SEI Quality Attribute Workshop

The Quality Attribute Workshop (QAW) is a facilitated method that engages system stakeholders early in the lifecycle to discover the driving quality attributes of a software-intensive system.

Key points about the QAW are that it is

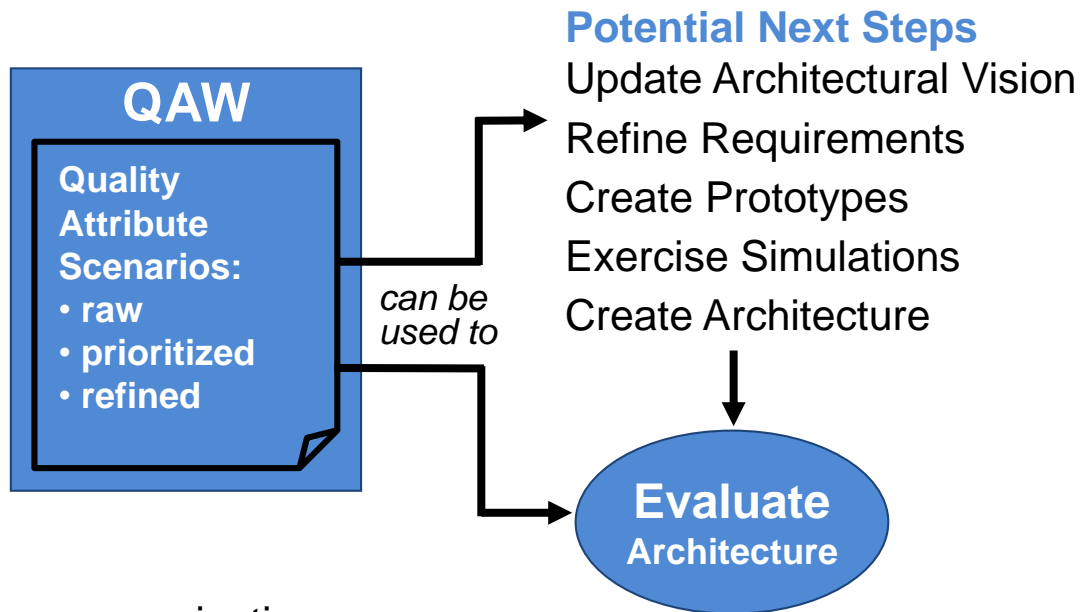
- system-centric
- stakeholder focused
- used before the software architecture has been created
- scenario based

QAW Steps

1. QAW Presentation and Introductions
 2. Business/Mission Presentation
 3. Architectural Plan Presentation
 4. Identification of Architectural Drivers
 5. Scenario Brainstorming
 6. Scenario Consolidation
 7. Scenario Prioritization
 8. Scenario Refinement
- 

Iterate as necessary with broader stakeholder community.

QAW Benefits and Next Steps



Potential Benefits

- increased stakeholder communication
- clarified quality attribute requirements
- informed basis for architectural decisions

More on the QAW

A detailed discussion of the QAW is provided in

- your textbook, Section 16.2, pages 294–296
- the technical report titled *Quality Attribute Workshops, Third Edition*
- the course notes from the SEI Software Architecture Design and Analysis course

Module Summary – 1

If the only thing that matters is getting the right answer, an unstructured, monolithic system will suffice.

However, other things do matter such as performance, modifiability, availability, and so forth. We call these things quality attributes.

The degree to which a system meets quality attribute requirements depends on architectural decisions.

Quality attribute requirements are often vaguely understood and/or weakly articulated.

Module Summary – 2

Quality attribute scenarios can help us better describe quality attribute requirements.

General scenarios are available to assist us in developing concrete scenarios for specific systems.

The Quality Attribute Workshop is a system-centric, stakeholder-focused process that helps us elicit quality attribute requirements early in the lifecycle.



Software Architecture: Principles and Practices

ACHIEVING QUALITY ATTRIBUTES

Module Objectives

This module will familiarize participants with

- the categories of design concepts:
 - design principles
 - reference architectures
 - externally developed components
 - deployment patterns
 - architectural design patterns
 - tactics
- the Attribute-Driven Design (ADD) method

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

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

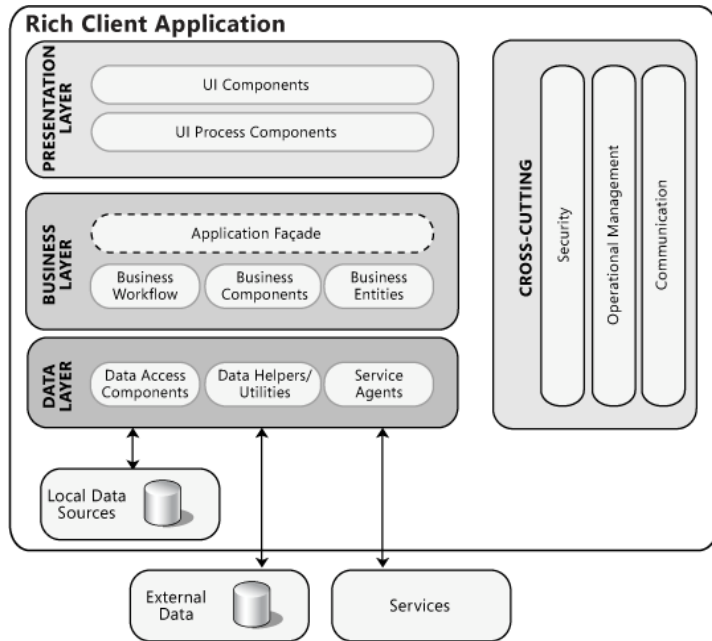
There are more specific design principles. For example, the SOLID principles aid in designing modifiable, extensible OO-based architectures.

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



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

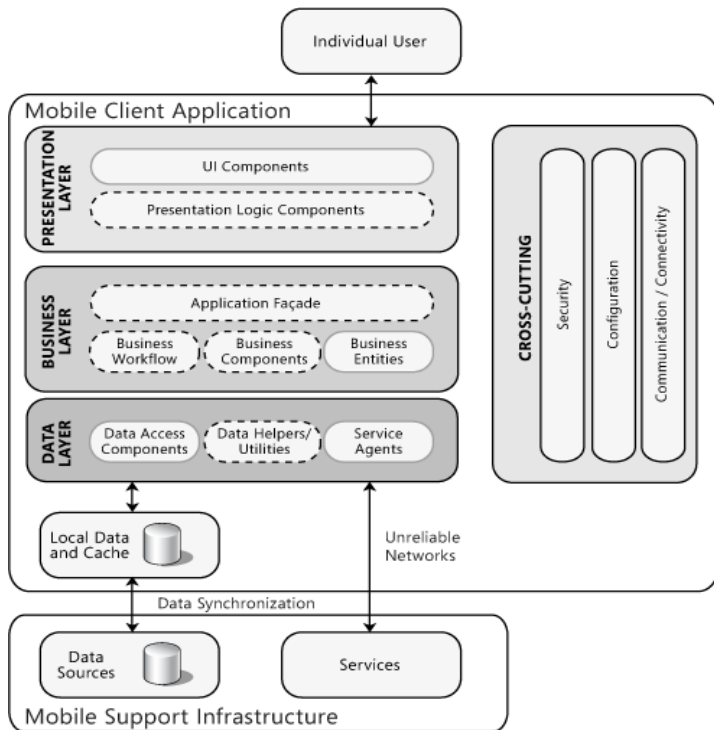
- Web application
- mobile application
- lambda architecture

They typically employ and combine patterns.

They aid in planning and reasoning.

From: Microsoft Application Architecture Guide

Reference Architectures



Different reference architectures focus on optimizing different quality attributes.

They are NOT complete architecture design solutions.

They are often accompanied by tools, frameworks, and platforms.

From: Microsoft Application Architecture Guide

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

Example: Frameworks

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

Example frameworks (for Java):

Concern	Framework	Use
OO – relational mapping	Hibernate	XML, annotations
Local user interface	Swing	Inheritance
Component connection	Spring	XML, annotations
Unit testing	JUnit	Inheritance, annotations
Web UI	JSF	XML

Selection of Externally Developed Components

Selecting components can be challenging because of their complexity and wide variety.

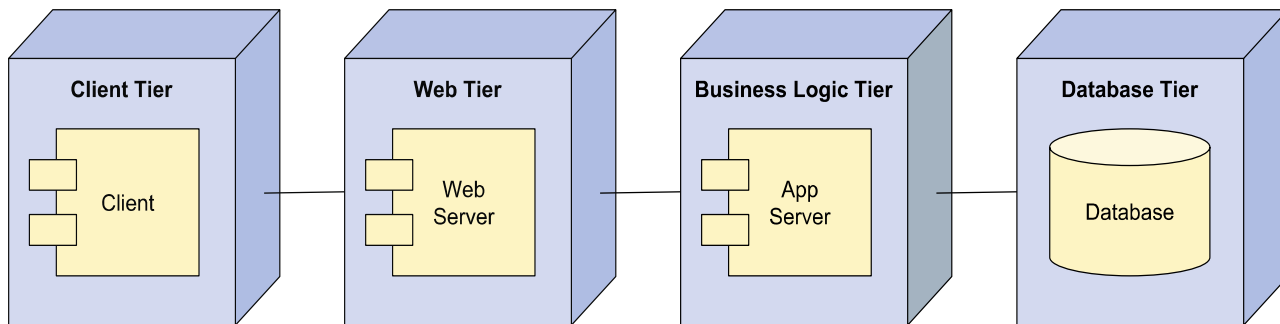
Some useful criteria:

- *Problem that it addresses*
- *Cost*
- *Type of license*
- *Support*
- *Learning curve*
- *Maturity*
- *Popularity*
- *Evolution tempo*
- *Compatibility and ease of integration*
- *Support for critical quality attributes*
- *Size*

Deployment Patterns

Deployment patterns provide models of how to *physically* structure the system so that it can be deployed.

Example: Four-tier deployment pattern



Key: UML

Architectural Design Patterns

- An architect for a software system designs its architectural structures to solve a variety of design problems.
- The architectural structures designed by an architect are often based on one or more patterns.
- But what exactly is a pattern?

Patterns Defined – 1

A pattern describes a particular recurring design problem that arises in specific design contexts and presents a well-proven solution for the problem. The solution is specified by describing the roles of its constituent participants, their responsibilities and relationships, and the ways in which they collaborate.

Patterns Defined – 2

A pattern establishes a relationship between

- a context – a situation that gives rise to a problem
- a problem – a recurring problem that arises in the given context
 - provides a general statement of the problem
 - describes any complementary/opposing forces
- a solution – a proven resolution to the problem
 - describes how to solve the problem (i.e., balance the forces)
 - describes the responsibilities of and static relationships between elements
 - describes the runtime behavior of and interaction between elements

It's up to the architect to decide how patterns are instantiated.

Layers Pattern

Context: All complex systems experience the need to develop and evolve portions of the system independently. For this reason, the developers of the system need a clear and well-documented separation of concerns, so that modules of the system may be independently developed and maintained.

Problem: The software must to be segmented in such a way that the modules can be developed and evolved separately with little interaction among the parts, supporting portability, modifiability, and reuse.

Solution: To achieve this separation of concerns, the Layers pattern divides the software into units called layers. Each layer is a grouping of modules that offers a cohesive set of services. The usage must be unidirectional. Layers completely partition a set of software, and each partition is exposed through a public interface.

Layers Pattern Solution

Overview: The Layers pattern defines layers (groupings of modules that offer a cohesive set of services) and a unidirectional *allowed-to-use* relation among the layers.

Elements: *Layer*, a kind of module. The description of a layer should define what modules the layer contains.

Relations: *Allowed to use*. The design should define the layer usage rules and any allowable exceptions.

Constraints:

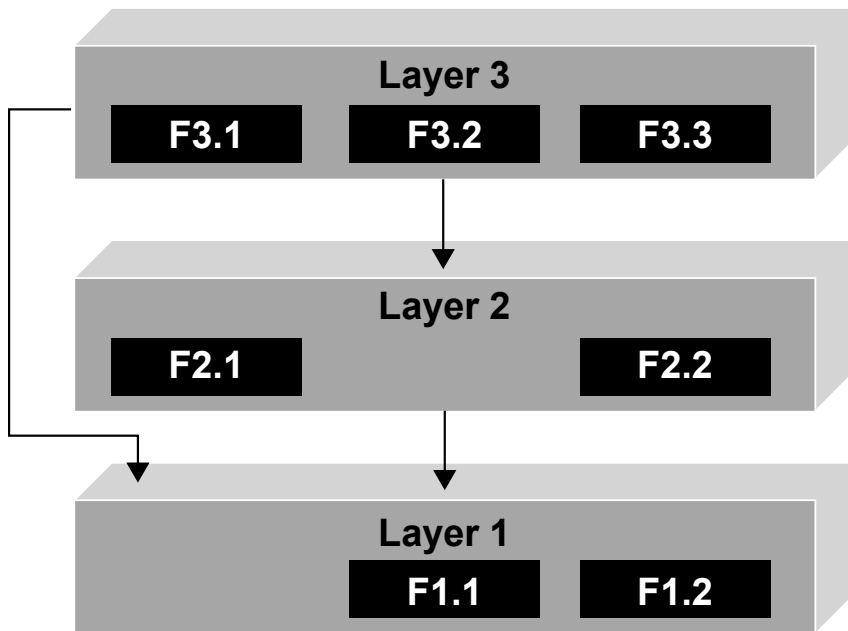
- Every piece of software is allocated to exactly one layer.
- There are at least two layers (but usually there are three or more).
- The allowed-to-use relations should not be circular (i.e., a lower layer cannot use a layer above).

Weaknesses:

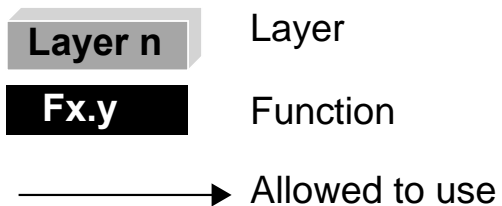
- The addition of layers adds up-front cost and complexity to a system.
- Layers contribute a performance penalty.

Layers Pattern Example

Solution (continued)



Key



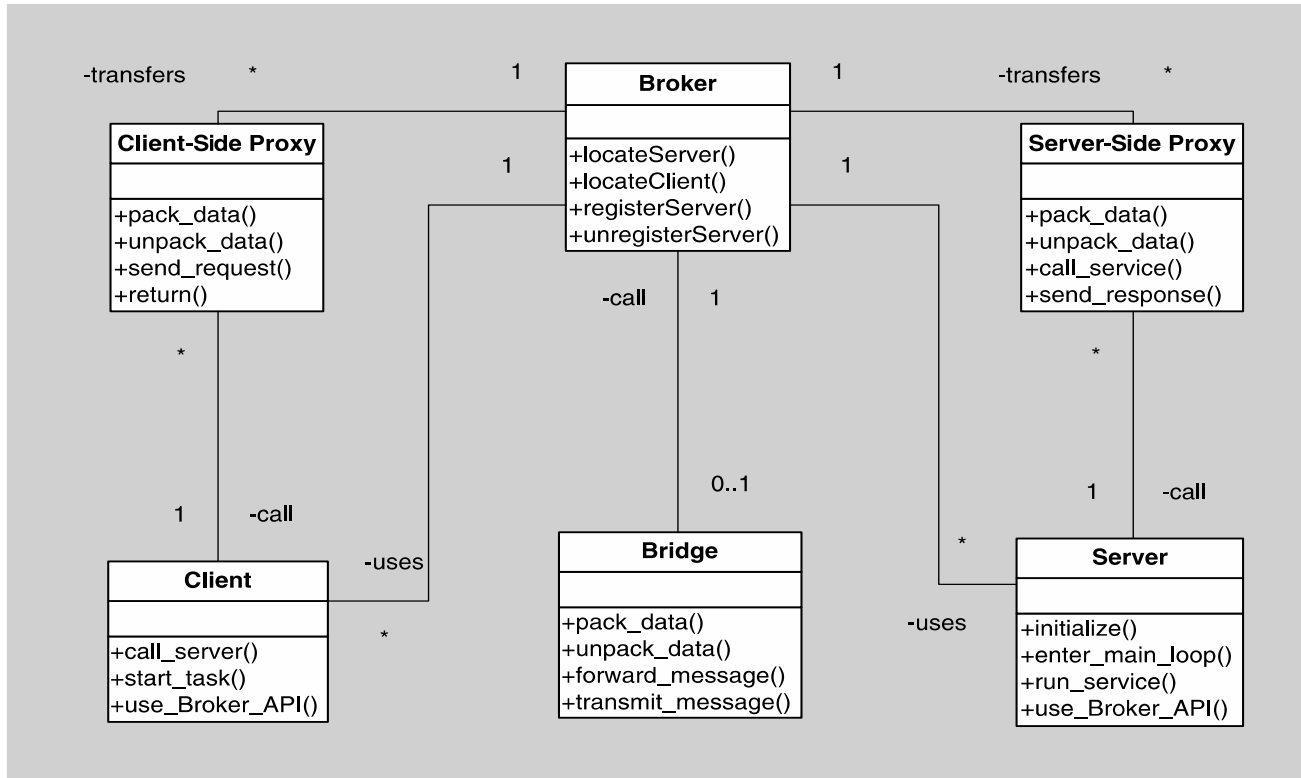
Broker Pattern

Context: Many systems are constructed from a collection of services distributed across multiple servers. Implementing these systems is complex because you need to worry about how the systems will interoperate—how they will connect to each other and how they will exchange information—as well as the availability of the component services.

Problem: How do we structure distributed software so that service users do not need to know the nature and location of service providers, making it easy to dynamically change the bindings between users and providers?

Solution: The Broker pattern separates users of services (clients) from providers of services (servers) by inserting an intermediary, called a broker. When a client needs a service, it queries a broker via a service interface. The broker then forwards the client's service request to a server, which processes the request.

Broker Pattern Example



Broker Pattern Solution – 1

Overview: The Broker pattern defines a runtime component, called a broker, that mediates the communication between a number of clients and servers.

Elements:

- *Client*, a requester of services
- *Server*, a provider of services
- *Broker*, an intermediary that locates an appropriate server to fulfill a client's request, forwards the request to the server, and returns the results to the client
- *Client-side proxy*, an intermediary that manages the actual communication with the broker, including marshaling, sending, and unmarshaling of messages
- *Server-side proxy*, an intermediary that manages the actual communication with the broker, including marshaling, sending, and unmarshaling of messages

Broker Pattern Solution – 2

Relations: The *attachment* relation associates clients (and, optionally, client-side proxies) and servers (and, optionally, server-side proxies) with brokers.

Constraints: The client can only attach to a broker (potentially via a client-side proxy). The server can only attach to a broker (potentially via a server-side proxy).

Weaknesses:

- Brokers add a layer of indirection, and hence latency, between clients and servers, and that layer may be a communication bottleneck.
- The broker can be a single point of failure.
- A broker adds up-front complexity.
- A broker may be a target for security attacks.
- A broker may be difficult to test.

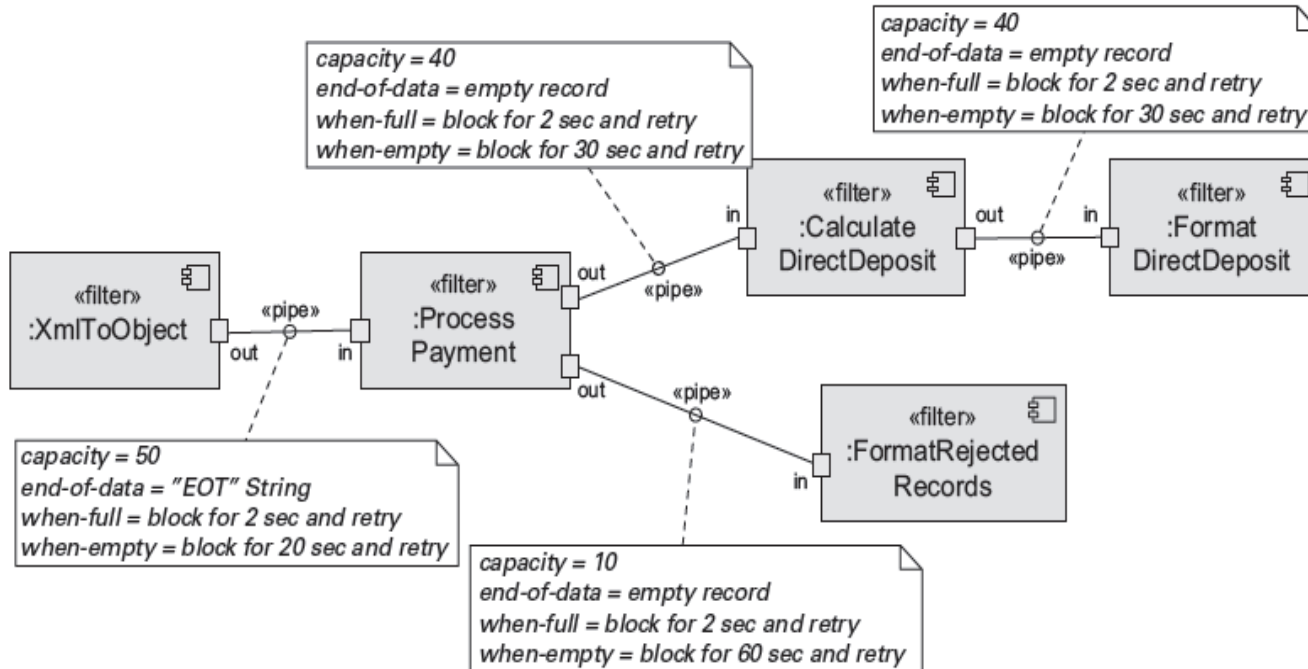
Pipe-and-Filter Pattern

Context: Many systems are required to transform streams of discrete data items, from input to output. Many types of transformations occur repeatedly in practice, so it is desirable to create these as independent, reusable parts.

Problem: Such systems need to be divided into reusable, loosely coupled components with simple, generic interaction mechanisms. In this way, they can be flexibly combined with each other. The components, being generic and loosely coupled, are easily reused. The components, being independent, can execute in parallel.

Solution: The pattern of interaction in the Pipe-and-Filter pattern is characterized by successive transformations of streams of data. Data arrives at a filter's input port(s), is transformed, and then is passed via its output port(s) through a pipe to the next filter. A single filter can consume data from, or produce data to, one or more ports.

Pipe-and-Filter Pattern Example



Pipe-and-Filter Pattern Solution

Overview: Data is transformed from a system's external inputs to its external outputs through a series of transformations performed by its filters connected by pipes.

Elements:

- *Filter*, which is a component that transforms data read on its input port(s) to data written on its output port(s).
- *Pipe*, which is a connector that conveys data from a filter's output port(s) to another filter's input port(s). A pipe has a single source for its input and a single target for its output. A pipe preserves the sequence of data items, and it does not alter the data passing through.

Relations: The *attachment* relation associates the output of filters with the input of pipes and vice versa.

Constraints:

- Pipes connect filter output ports to filter input ports.
- Connected filters must agree on the type of data being passed along the connecting pipe.

Group Exercise: Patterns and Quality Attributes

Worksheet

Discuss with your group which quality attributes are likely to be positively or negatively affected by the use of the Layers pattern in designing an architecture.

Present your results.

Buildability	Security	Interoperability
Modifiability	Testability	Availability
Reusability	Usability	Safety
Portability	Subsetability	Performance
Reliability	Dependability	
Others?		

Patterns Vary in Scale and Abstraction

Because patterns cover various ranges of scale and are applied at various levels of abstraction, it is sometimes useful to broadly classify them as

- *architectural patterns* – express a fundamental structural organization schema for software systems. An architectural pattern provides a set of predefined major architectural elements, specifies their responsibilities, and includes rules and guidelines for organizing the relationships between them.
- *design patterns* – provide a scheme for refining the major architectural elements of a software system or the relationships between them. A design pattern describes a commonly recurring structure of communicating elements that solves a general design problem within a particular context.
- *idioms* – are patterns specific to a programming language. An idiom describes how to implement particular aspects of components or the relationships between them using the features of the given language.

Note that there is overlap and ambiguity in these definitions.

Pattern Benefits

Patterns

- provide solutions to recurring design problems
- document existing, well-proven design expertise
- provide a common vocabulary and understanding for design principles
- provide a means to augment software architecture documentation
- support construction of software with predictable properties
- help build and manage complex and heterogeneous software architectures

Pattern Resources

There is no complete list or definitive source of patterns.

Some of the many excellent pattern resources include

- your textbook, Chapter 13
- other textbooks
 - *Pattern-Oriented Software Architecture*, Volumes 1–5, New York: Wiley, 1996–2007, Buschmann et al.
 - *Design Patterns: Elements of Reusable Object-Oriented Software*, Boston: Addison-Wesley, 1994, Gamma et al.
- Websites
 - <http://www.hillside.net>
 - <http://www.soapatterns.org>

Group Exercise: Designing with Patterns – 1

Your team has been assigned the task of designing a messaging infrastructure for a system that supports appraisal brokering for commercial and residential properties.

Using the worksheets on the following pages,

Read the problem statement.

Identify and determine the relative importance of quality attribute requirements.

Read about and discuss the Messaging, Publisher-Subscriber, and SOA patterns.

Discuss the benefits and liabilities of each pattern.

Select the pattern that you feel will best meet system requirements.

Discuss how you would instantiate the selected pattern.

Note any tradeoffs that result from the selected pattern and how you would instantiate it.

Prepare a sketch of your proposed design.

Record any assumptions you have made.

Present your results.

Group Exercise: Designing with Patterns – 2

Worksheet

Read the problem statement.

Problem Statement

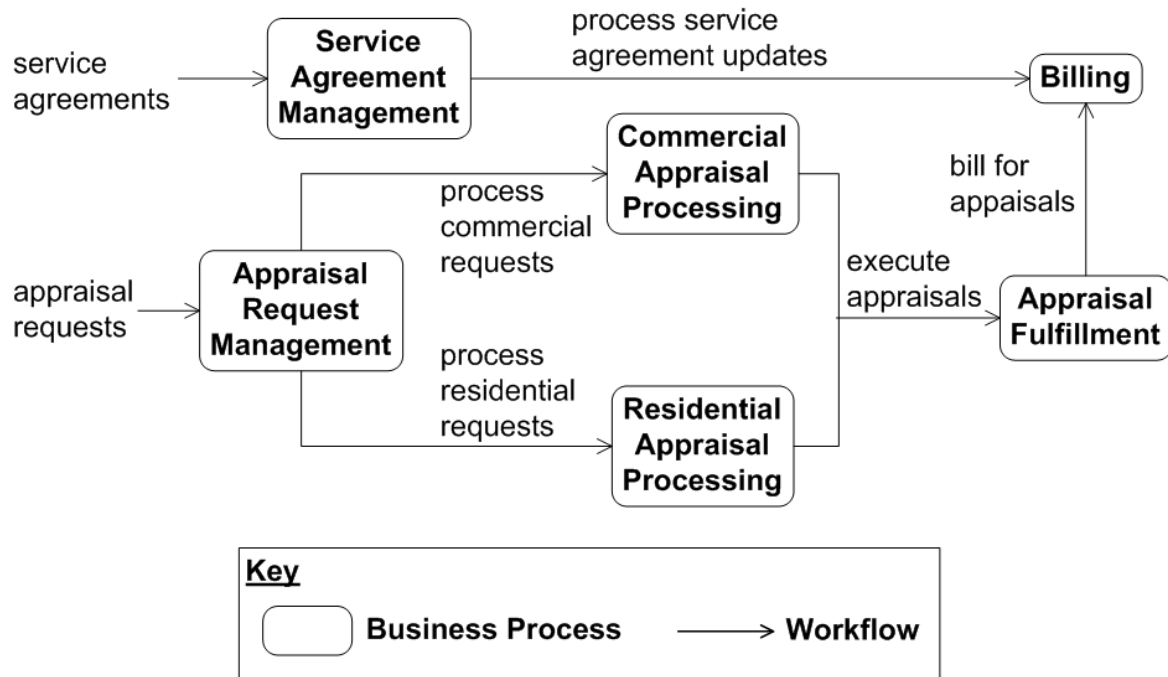
Our company, BizCo, would like to establish itself as an industry leader in brokering appraisals for commercial and residential properties. One step toward accomplishing this goal is to ensure that our main office has more detailed and timely information about the data entered and the activities that occur at our branch offices.

Group Exercise: Designing with Patterns – 3

Worksheet

Problem Statement (continued)

The workflow at branch offices typically proceeds as follows:



Group Exercise: Designing with Patterns – 4

Worksheet

Problem Statement (continued)

Business Process	Description
Service Agreement Management	Establishes and records service agreements between BizCo and lenders/appraisers.
Appraisal Request Management	Accepts and records incoming appraisal requests. Routes requests for commercial or residential processing as appropriate.
Commercial Appraisal Processing	Pairs commercial appraisal requests with appraisers. Routes requests to appraisers.
Residential Appraisal Processing	Pairs residential appraisal requests with appraisers. Routes requests to appraisers.
Appraisal Fulfillment	Performs and records the results of property appraisals.
Billing	Bills lenders and pays appraisers for appraisal services rendered according to established service agreements.

Group Exercise: Designing with Patterns – 5

Worksheet

Problem Statement (continued)

BizCo’s main office is located in Pittsburgh, PA, with branch offices throughout the United States.

The workflow at each branch office is supported by three systems—the Commercial Property System (CPS), the Residential Property System (RPS), and the Brokerage Billing System (BBS)—as follows:

	CPS	RPS	BBS
Appraisal Request Management	x*	x	
Commercial Appraisal Processing	x		
Residential Appraisal Processing		x	
Appraisal Fulfillment	x	x	
Billing			x
Service Agreement Management			x

* “X” indicates supported by/supports

Group Exercise: Designing with Patterns – 6

Worksheet

Problem Statement (continued)

Branch offices also use CPS, RPS, and BBS biweekly to create hard-copy reports that summarize brokerage activities. The reports are sent via courier to the main office for review. Although we would like to have direct, more extensive, and daily access to data and information related to brokerage activities, these systems are currently standalone and unable to interact with other systems within a branch or external to a branch.

To rectify this situation, we intend to develop a software system called the Brokerage Information System (BIS) that will provide detailed data and activity reports based on information acquired directly from CPS, RPS, and BBS.

BIS will allow users at our main office to display reports on

- branch office, lender, and appraiser business addresses and contacts
- contract terms and conditions for lenders and appraisers
- appraisal requests submitted, assigned, and fulfilled
- brokerage activities across and for individual branch offices
- brokerage activities across and for individual lenders, appraisers, and brokers
- accounts receivable

Group Exercise: Designing with Patterns – 7

Worksheet

Problem Statement (continued)

Our business goals include

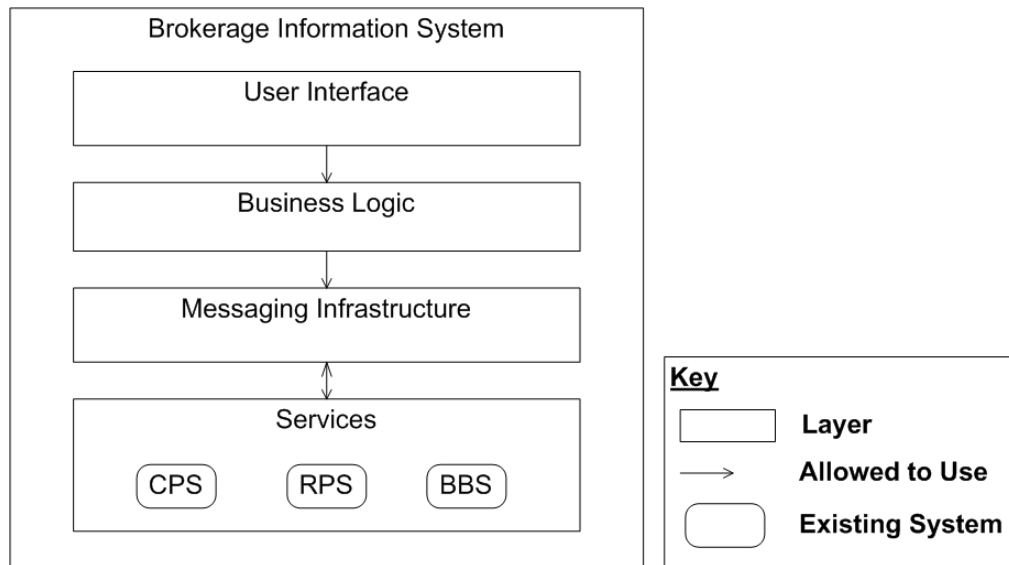
- establishing BizCo as an industry leader in brokering appraisals for commercial and residential properties
- increasing our market share of brokerage services
- dramatically decreasing our response time to market conditions
- providing direct and secure access to brokerage information 24/7
- making it easy to collate and understand brokerage information

Group Exercise: Designing with Patterns – 8

Worksheet

Problem Statement (continued)

Our chief architect has already decided on a layered architecture as a module structure for the system.



Group Exercise: Designing with Patterns – 9

Worksheet

Problem Statement (continued)

Layer	Description
User Interface	Allows users to select, customize, and display brokerage data and activity reports.
Business Logic	Interacts with remote services to collect data and collates the data into brokerage reports.
Message Infrastructure	Provides access to remote services.
Services	A collection of services that provide brokerage data. CPS, RPS, and BBS will provide these services.

Group Exercise: Designing with Patterns – 10

Worksheet

Problem Statement (continued)

Service Providers	Description
CPS	Services requests for data about appraisal requests submitted, assigned, and fulfilled for commercial properties including request details, lender/appraiser assignments, and status.
RPS	Services requests for data about appraisal requests submitted, assigned, and fulfilled for residential properties including request details, lender/appraiser assignments, and status.
BBS	Services requests for data about branch office, lender, and appraiser business addresses and contacts, contract terms and conditions, and accounts receivable.

Group Exercise: Designing with Patterns – 11

Worksheet

Problem Statement (continued)

Your team has been assigned to propose a design for the Messaging Infrastructure layer, which allows the Business Logic layer to communicate with remote services.

Select from the following patterns:

- Messaging
- Publisher-Subscriber
- SOA

Discuss the patterns; consider their pros, cons, and tradeoffs from a quality attributes perspective; sketch a design that contains enough information to support analysis; and present your results.

These patterns are presented in detail in the following slides.

Group Exercise: Designing with Patterns – 12

Worksheet

Identify and determine the relative importance (1 to 10, where 1 is most important) of quality attribute requirements for the messaging infrastructure. (Hint: Revisit the business goals. Also, use your knowledge of and experience with similar systems.)

Relative Importance	Quality Attribute

Group Exercise: Designing with Patterns – 13

Worksheet

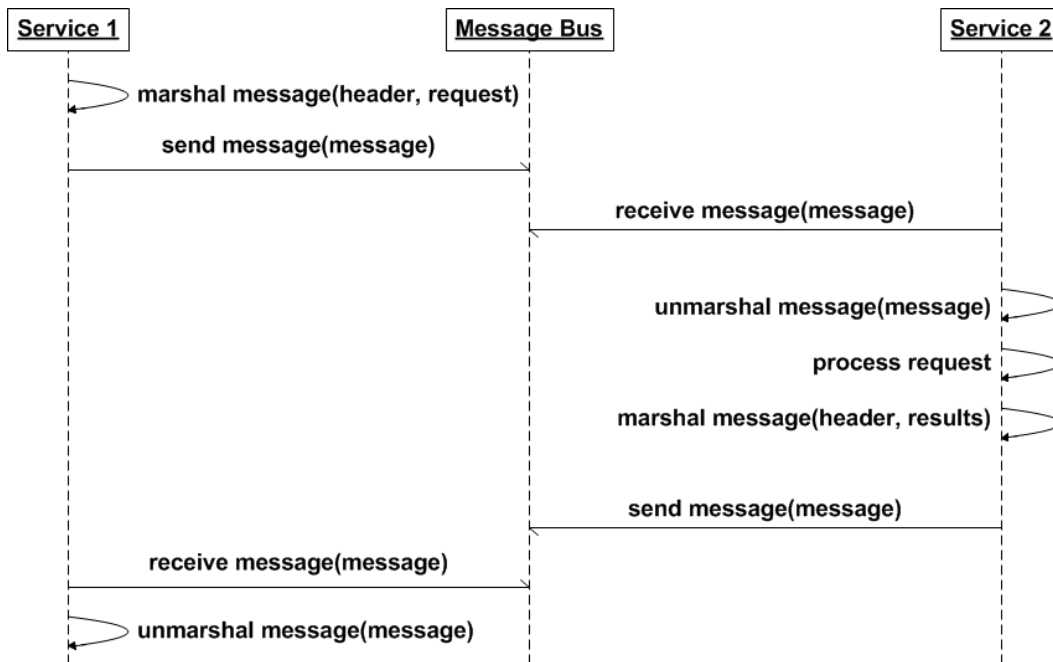
Read and discuss the Messaging pattern.

Context	Some distributed systems are composed of services that were developed independently. To form a coherent system, however, these services must interact reliably, but without incurring overly tight dependencies on one another.
Problem	Integrating independently developed services, each having its own business logic and value, into a coherent application requires reliable collaboration between services. However, since services are developed independently, they are generally unaware of each other's specific functional interfaces. Furthermore, each service may participate in multiple integration contexts, so using them in a specific context should not preclude their use in other contexts.
Solution	Connect the services via a message bus that allows them to transfer data messages asynchronously. Encode the messages (request data and data types) so that senders and receivers can communicate reliably without having to know all the data type information statically.

Group Exercise: Designing with Patterns – 14

Worksheet

Read and discuss the Messaging pattern. (continued)



Behavior Trace

Key: UML Diagram

Group Exercise: Designing with Patterns – 15

Worksheet

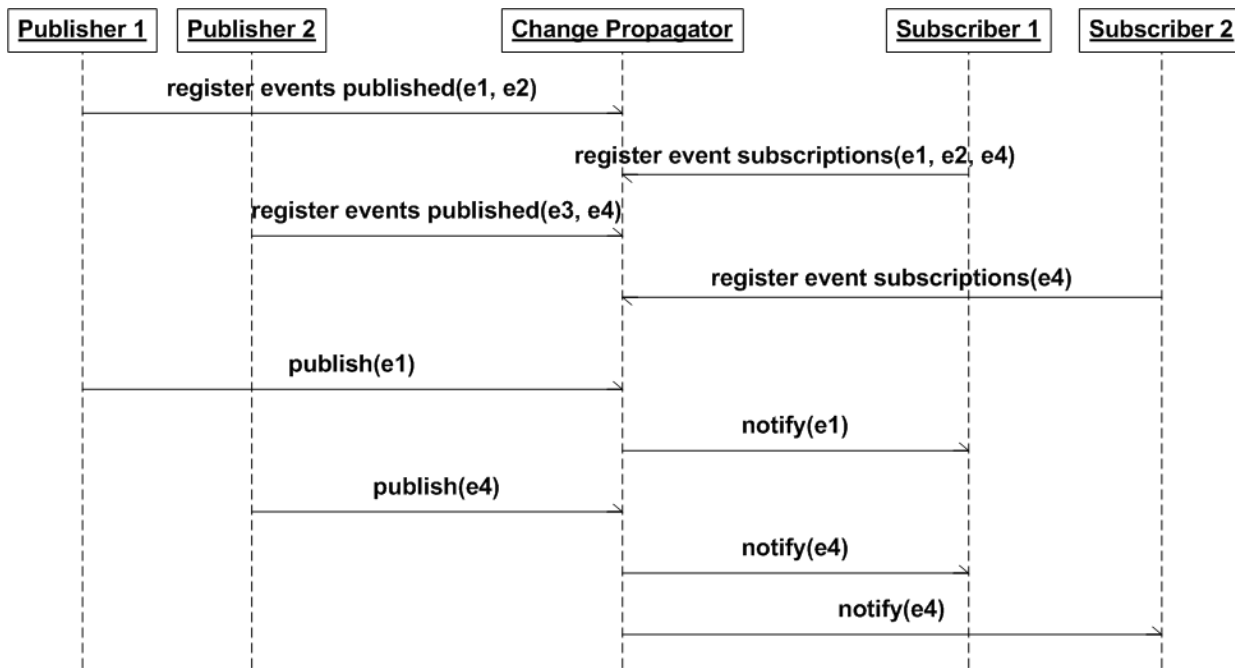
Read and discuss the Publisher-Subscriber pattern.

Context	Components in some distributed applications are loosely coupled and operate largely independently. If such applications need to propagate information to some or all of their components, a notification mechanism is needed to inform the components about state changes or events that affect or coordinate their own computation.
Problem	The notification mechanism should not couple application components too tightly, or they will lose their independence. Components want to know only that another component is in a specific state, not which specific component is involved. Components that disseminate events often do not care which other components want to receive the information. Components should not depend on how other components can be reached or on their specific location in the system.
Solution	Define a change propagation infrastructure that allows publishers in a distributed application to disseminate events that may be of interest to others. Notify subscribers interested in those events whenever such information is published.

Group Exercise: Designing with Patterns – 16

Worksheet

Read and discuss the Publisher-Subscriber pattern. (continued)



Behavior Trace

Key: UML Diagram

Group Exercise: Designing with Patterns – 17

Worksheet

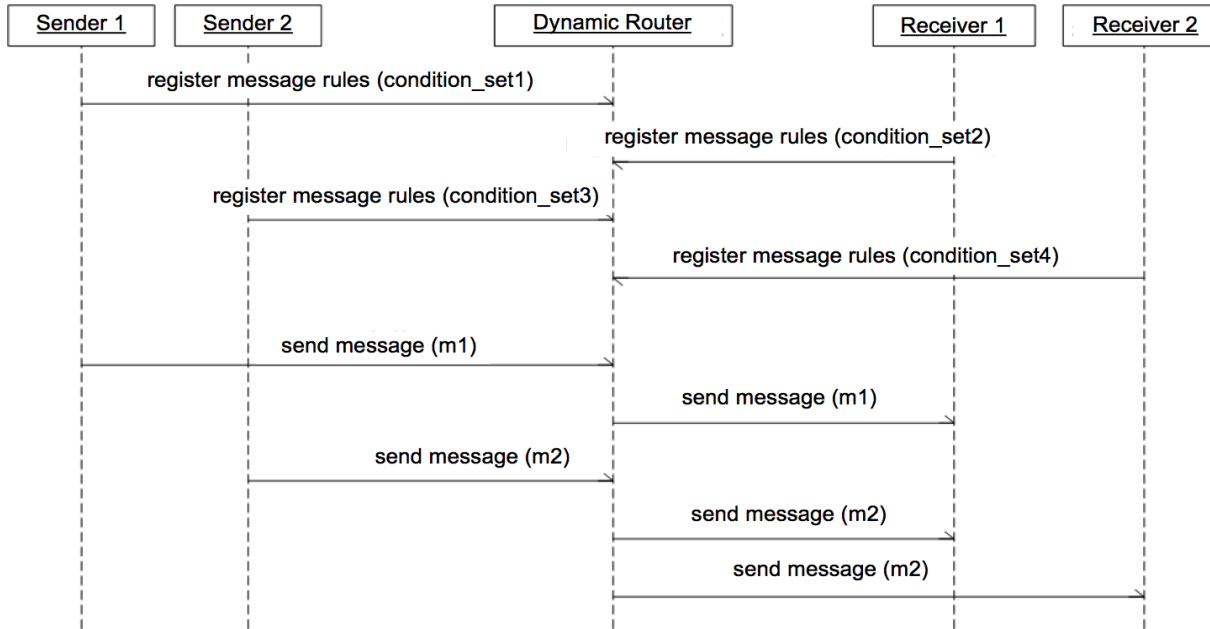
Read and discuss the Dynamic Routing pattern.

Context	It is often necessary to build complex business processes by wiring together a set of relatively simple services in a dynamic way.
Problem	Routing messages through a distributed system based on filtering rules is inefficient because messages are sent to every destination's filter and router for inspection and rules resolution, whether or not the message could be processed.
Solution	Define a message router that includes both filtering rules and knowledge about the processing destination paths so that messages are delivered only to the processing endpoints that can act on them. Unlike filters, message routers do not modify the message content and are concerned only with message destination.

Group Exercise: Designing with Patterns – 18

Worksheet

Read and discuss the Dynamic Routing pattern. (continued)



Behavior Trace

Key: UML Diagram

Group Exercise: Designing with Patterns – 19

Worksheet

Discuss the benefits and liabilities of each pattern.

	Benefits	Liabilities
Messaging	Services can interact without having to deal with networking and service location concerns.	Lack of statically typed interfaces makes it hard to validate system behavior prior to runtime.
	Asynchronous messaging allows services to handle multiple requests simultaneously without blocking.	Service requests are encapsulated within self-describing messages that require extra time and space for message processing.
	Allows services to participate in multiple application integration and usage contexts.	

Group Exercise: Designing with Patterns – 20

Worksheet

Discuss the benefits and liabilities of each pattern (continued).

	Benefits	Liabilities
Publisher-Subscriber	Publishers can asynchronously transmit events to Subscribers without blocking.	Publishing can cause unnecessary overhead if subscribers are interested in only a specific type of event.
	Asynchronous communication decouples Publishers from Subscribers, allowing them to be active and available at different times.	Filtering events to decrease event publishing and notification overhead can result in other costs (e.g., decrease in throughput, unnecessary notifications, breakdown of anonymous communication model).
	Publishers and Subscribers are unaware of each other's location and identity.	

Group Exercise: Designing with Patterns – 21

Worksheet

Discuss the benefits and liabilities of each pattern (continued).

	Benefits	Liabilities
Dynamic Routing	Services do not need to deal with networking concerns or know each other's locations since all requests are handled through the message router.	Performance overhead.
	Communications can be optimized as services dynamically become available or unavailable.	Single point of failure.
	Efficient, predictive routing.	Potentially complex, unintuitive behavior when rules conflict.

Group Exercise: Designing with Patterns – 22

Worksheet

Select one of the three patterns that you feel will best meet the messaging infrastructure requirements.

Discuss how you would instantiate the selected pattern.

Note any tradeoffs that result from the selected pattern and how you would instantiate the pattern.

Prepare a sketch of your proposed design.

Record any assumptions you have made.

Group Exercise: Designing with Patterns – 23

Worksheet

Present the following:

- quality attributes in relative order of importance
- which pattern your group selected
- what tradeoffs result from the selected pattern and how you would instantiate the pattern
- sketch of your proposed design
- assumptions you have made

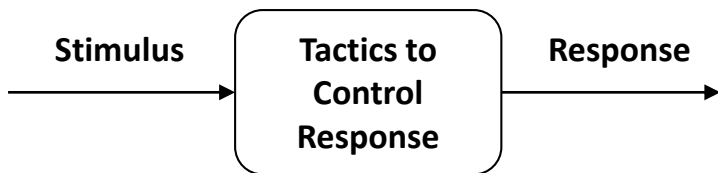
Tactics Defined – 1

Patterns are composed from fundamental design decisions we call tactics.

In other words, tactics are the “building blocks” of design from which architectural patterns are created.

But what exactly is a tactic?

Tactics Defined – 2



The software architecture for a system is a collection of design decisions.

- Some decisions are made to ensure we achieve the system's functional requirements.
- Other decisions are made to ensure we achieve the system's quality attribute requirements.
 - These decisions are called tactics.
 - Tactics are influential in controlling quality attribute responses.

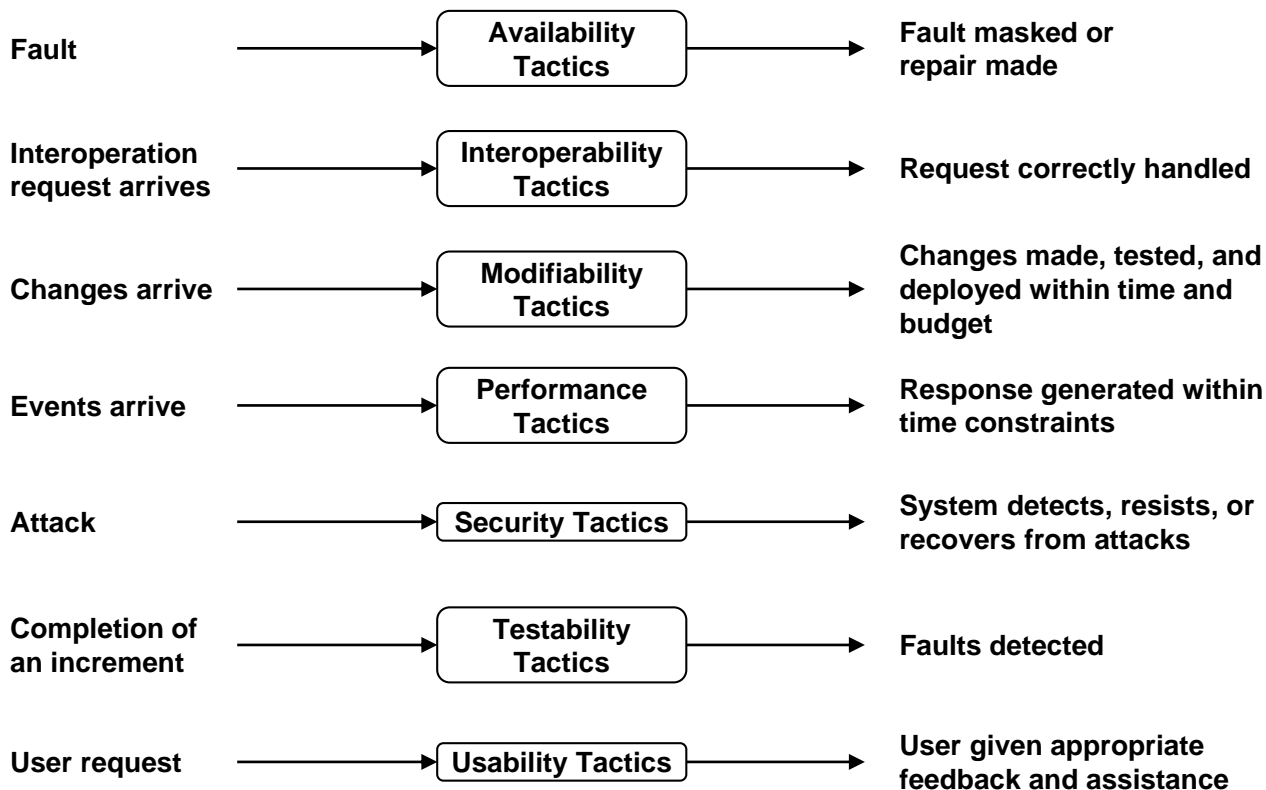
Tactic Defined – 3

Restrict dependencies
Passive redundancy
Maintain audit trail
Condition monitoring
Maintain system model

Each tactic is a design option for the architect.

For example, which of the following tactics is a design option an architect might choose to promote the availability of a system?

Tactics Categorized by Quality Attributes



Example: Availability Tactics – 1

Detect faults

- **Ping/echo:** asynchronous request/response message pair exchanged between nodes, used to determine reachability and the round-trip delay through the associated network path
- **Monitor:** a component used 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.
- **Heartbeat:** a periodic message exchange between a system monitor and a process being monitored
- **Timestamp:** used to detect incorrect sequences of events, primarily in distributed message-passing systems
- **Condition monitoring:** checking conditions in a process or device, or validating assumptions made during the design

Example: Availability Tactics – 2

Detect faults

- **Sanity checking:** checks the validity or reasonableness of a component's operations or outputs; typically based on knowledge of the internal design, the state of the system, or the nature of the information under scrutiny
- **Voting:** to check that replicated components are producing the same results. Comes in various flavors: replication, functional redundancy, analytic redundancy
- **Exception detection:** detection of a system condition that alters the normal flow of execution, e.g., system exception, parameter fence, parameter typing, timeout
- **Self-test:** procedure for a component to test itself for correct operation

Example: Availability Tactics – 3

Recover from faults (preparation and repair)

- **Active redundancy** (hot spare): All nodes in a protection group receive and process identical inputs in parallel, allowing redundant spare(s) to maintain synchronous state with the active node(s).
 - A protection group is a group of nodes where one or more nodes are “active,” with the remainder serving as redundant spares.
- **Passive redundancy** (warm spare): Only the active members of the protection group process input traffic; one of their duties is to provide the redundant spare(s) with periodic state updates.
- **Spare** (cold spare): Redundant spares of a protection group remain out of service until a failover occurs, at which point a power-on-reset procedure is initiated on the redundant spare prior to its being placed in service.
- **Exception handling**: dealing with the exception by reporting it or handling it, potentially masking the fault by correcting the cause of the exception and retrying
- **Rollback**: reverting to a previous known good state, referred to as the “rollback line”

Example: Availability Tactics – 4

Recover from faults (preparation and repair)

- **Software upgrade:** in-service upgrades to executable code images in a manner that does not affect service
- **Retry:** where a failure is transient, retrying the operation may lead to success
- **Ignore faulty behavior:** ignoring messages sent from a source when it is determined that those messages are spurious
- **Degradation:** maintains the most critical system functions in the presence of component failures, dropping fewer critical functions
- **Reconfiguration:** reassigning responsibilities to the resources left functioning, while maintaining as much functionality as possible

Example: Availability Tactics – 5

Recover from faults (reintroduction)

- **Shadow:** operating a previously failed or in-service upgraded component in a “shadow mode” for a predefined time prior to reverting the component back to an active role
- **State resynchronization:** partner to active redundancy and passive redundancy in which state information is sent from active to standby components
- **Escalating restart:** recover from faults by varying the granularity of the component(s) restarted and minimizing the level of service affected
- **Non-stop forwarding:** splits functionality into supervisory and data. If a supervisor fails, a router continues forwarding packets along known routes while protocol information is recovered and validated.

Example: Availability Tactics – 6

Prevent faults

- **Removal from service:** temporarily placing a system component in an out-of-service state for the purpose of mitigating potential system failures
- **Transactions:** bundling state updates so that asynchronous messages exchanged between distributed components are *atomic, consistent, isolated, and durable*
- **Predictive model:** monitoring the state of health of a process to ensure that the system is operating within nominal parameters; taking corrective action when conditions are detected that are predictive of likely future faults
- **Exception prevention:** preventing system exceptions from occurring by masking a fault, or preventing them via smart pointers, abstract data types, or wrappers
- **Increase competence set:** designing a component to handle more cases—faults—as part of its normal operation

Group Exercise: Applying Tactics

Worksheet

Tactic descriptions can be found in your textbook as follows:

Choose one of the following quality attributes of the Brokerage Information System, and read the descriptions for the associated tactics in your textbook:

Availability:	p. 87–95	Security:	p. 150–154
Interoperability:	p. 110–112	Testability:	p. 164–168
Modifiability:	p. 121–125	Usability:	p. 177–181
Performance:	p. 135–141		

Discuss the tactics with your group.

Which tactics would you choose to improve your design of the messaging infrastructure for the system?

What tradeoffs and issues arise from your selection of tactics?

Present your results.

The Relationship Between Tactics and Patterns

As we said earlier, tactics are the “building blocks” of design from which patterns are created.

Patterns typically employ several different tactics to promote various quality attributes.

- For example, a pattern that supports availability will likely use one or more redundancy tactics to achieve availability.

Group Exercise: Tactics and Patterns

Worksheet

Consider the Layers pattern, which promotes quality attributes such as portability, testability, modifiability, and reusability.

Review the tactics for modifiability.

Which modifiability tactics does the Layers pattern employ to achieve modifiability?

Tactics Resources

Get more information about tactics in

- your textbook: *Software Architecture in Practice, Third Edition*.
- SEI technical reports, including
 - *Modifiability Tactics* (CMU/SEI-2007-TR-002)
 - *Realizing and Refining Architectural Tactics: Availability* (CMU/SEI-2009-TR-006)
- articles, including
 - W. Wu, T. Kelly, “Safety Tactics for Software Architecture Design,” *Proc. 28th Annual International Computer Software and Applications Conference*, 2004.
 - S. Kim, D. Kim, L. Lu, S. Park, “Quality-Driven Architecture Development Using Architectural Tactics,” *Journal of Systems and Software*, 82, 2009.
 - N. Harrison, P. Avgeriou, “How Do Architecture Patterns and Tactics Interact? A Model and Annotation,” *Journal of Systems and Software*, 83, 2010.
 - J. Ryoo, P. Laplante, R. Kazman, “Revising a Security Tactics Hierarchy Through Decomposition, Reclassification, and Derivation,” *Proc. International Conference on Software Security and Reliability*, 2012.

Attribute-Driven Design

The Attribute-Driven Design (ADD) method, developed by the SEI, is an approach to defining a software architecture that bases the decomposition process on the quality attributes that the software must fulfill.

ADD follows a recursive decomposition process in which, at each stage in the decomposition, design concepts (tactics, patterns, etc.) are chosen to satisfy a set of quality attribute scenarios.

ADD is now in its third major revision, ADD 3.0.

ADD Method's Inputs and Outputs

Inputs include these architectural drivers:

- design purpose
- quality attribute requirements
- design constraints
- functional requirements
- architectural concerns

Outputs include

- first several levels of module decomposition
- various other views of the system as appropriate
- set of elements with assigned functionalities and the interactions among the elements
- the rationale associated with the decisions made

The Attribute-Driven Design Method

ADD is an iterative design method.

In each iteration, you

- choose a part of the system to design
- marshal all the architectural drivers for that part
- create and test (analyze) a design for that part

ADD does not result in a complete design. Its outputs are

- a set of design decisions documented as structures and interfaces
- a description of interactions and information flows among the containers
- the rationale for the decisions made

It does not produce an API or signature for containers.

ADD Inputs and Outputs

Drivers

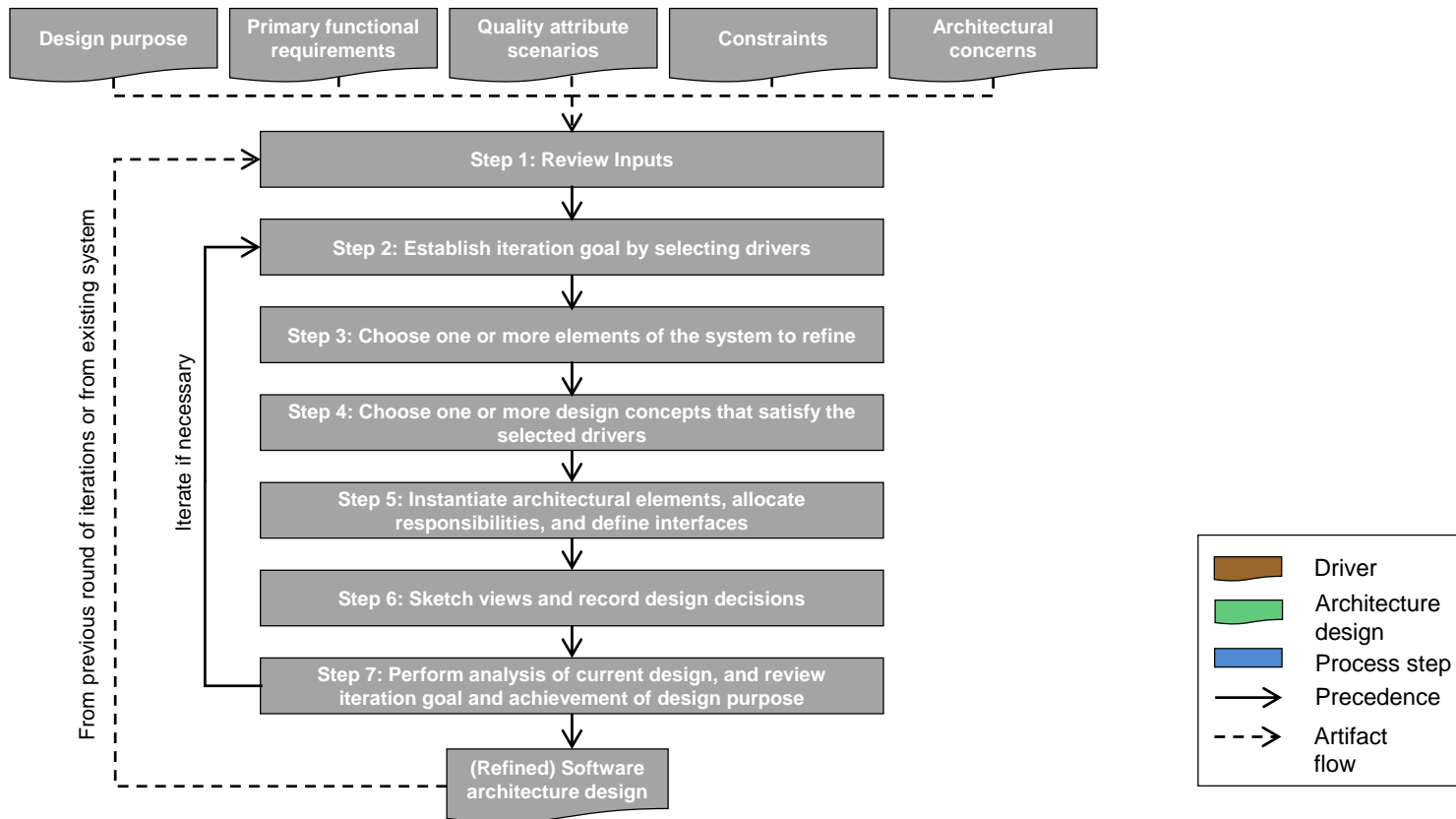
- Design purpose
- Quality attributes
- Primary functionality
- Architectural concerns
- Constraints



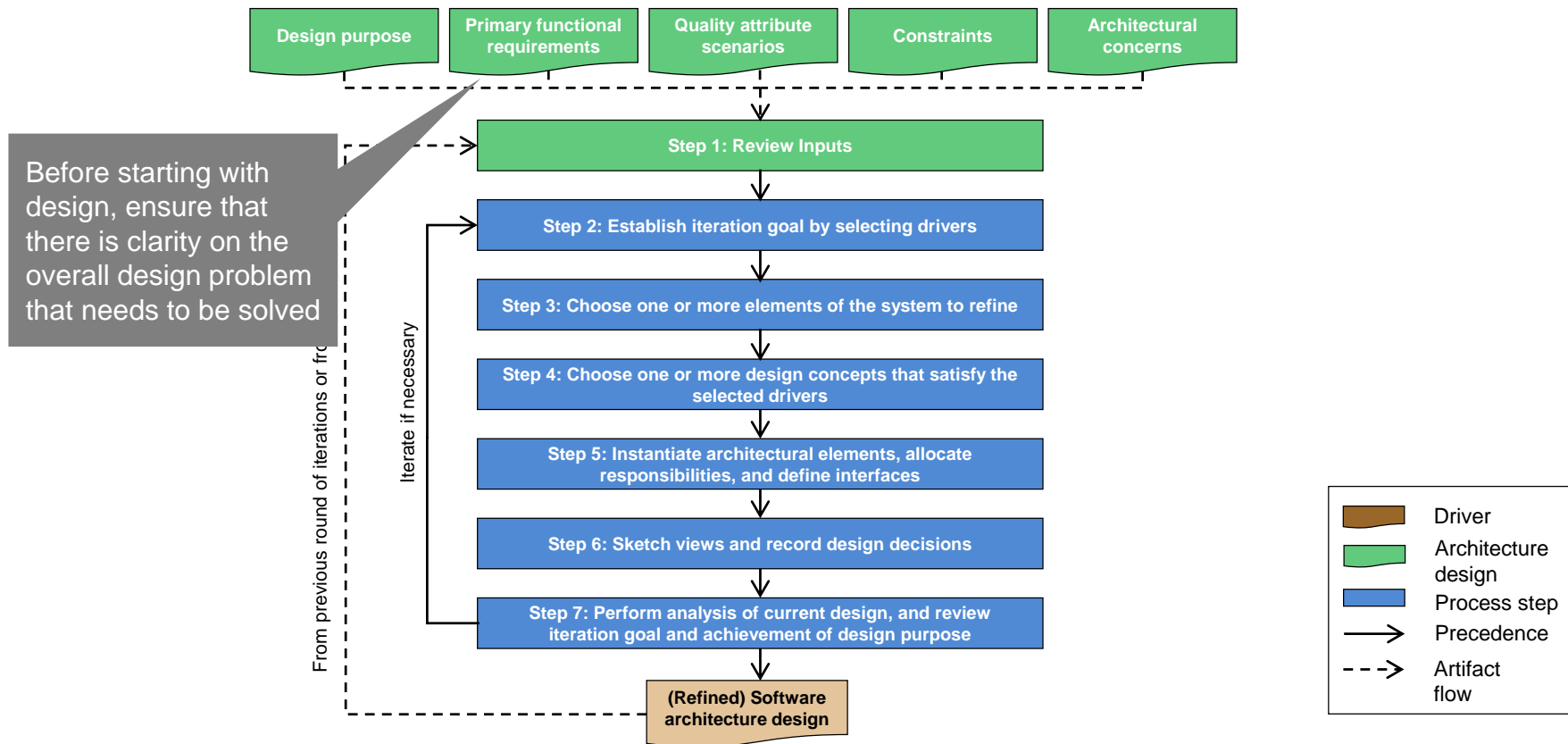
Software architecture design decisions

- Structures
- Interfaces
- Rationale

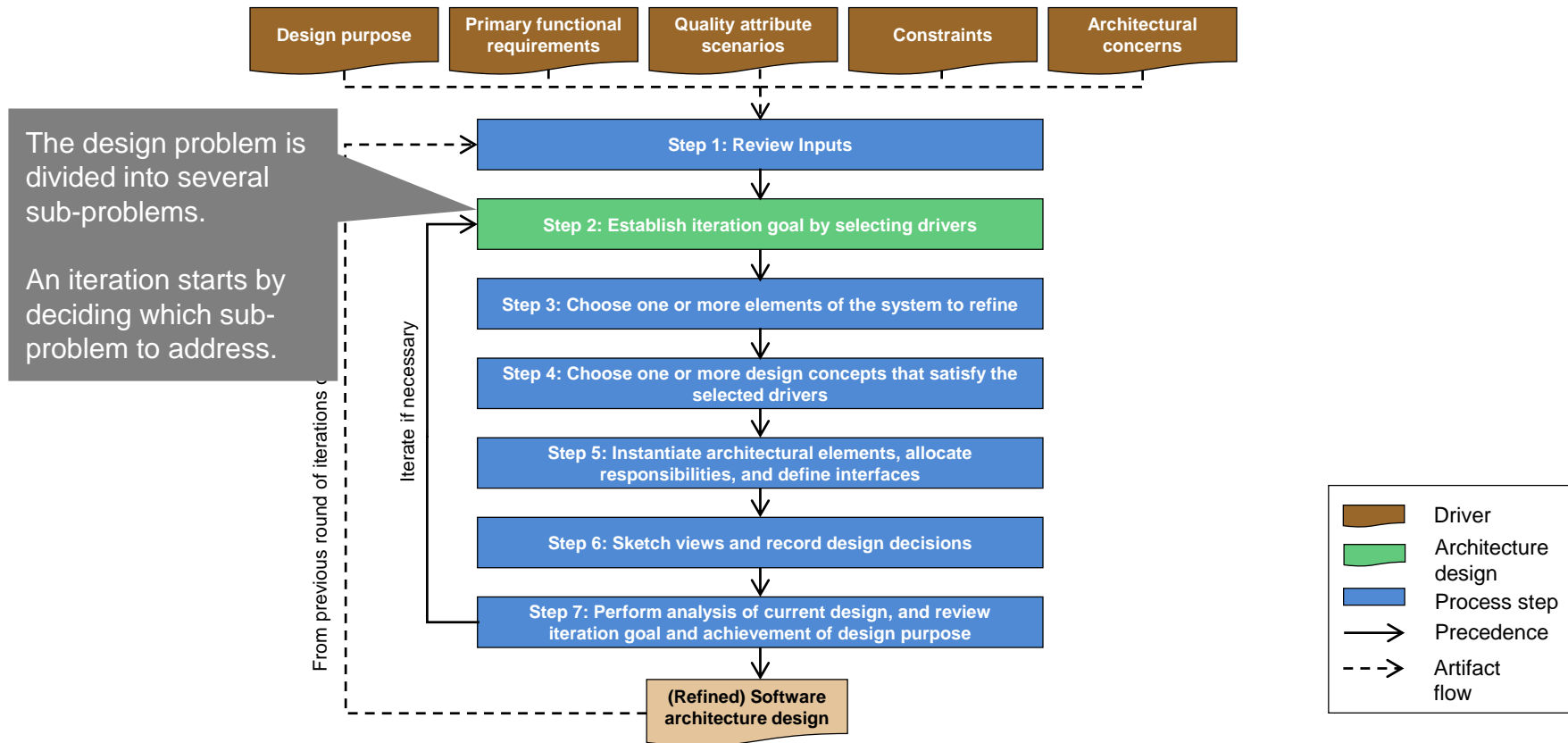
The Steps of ADD



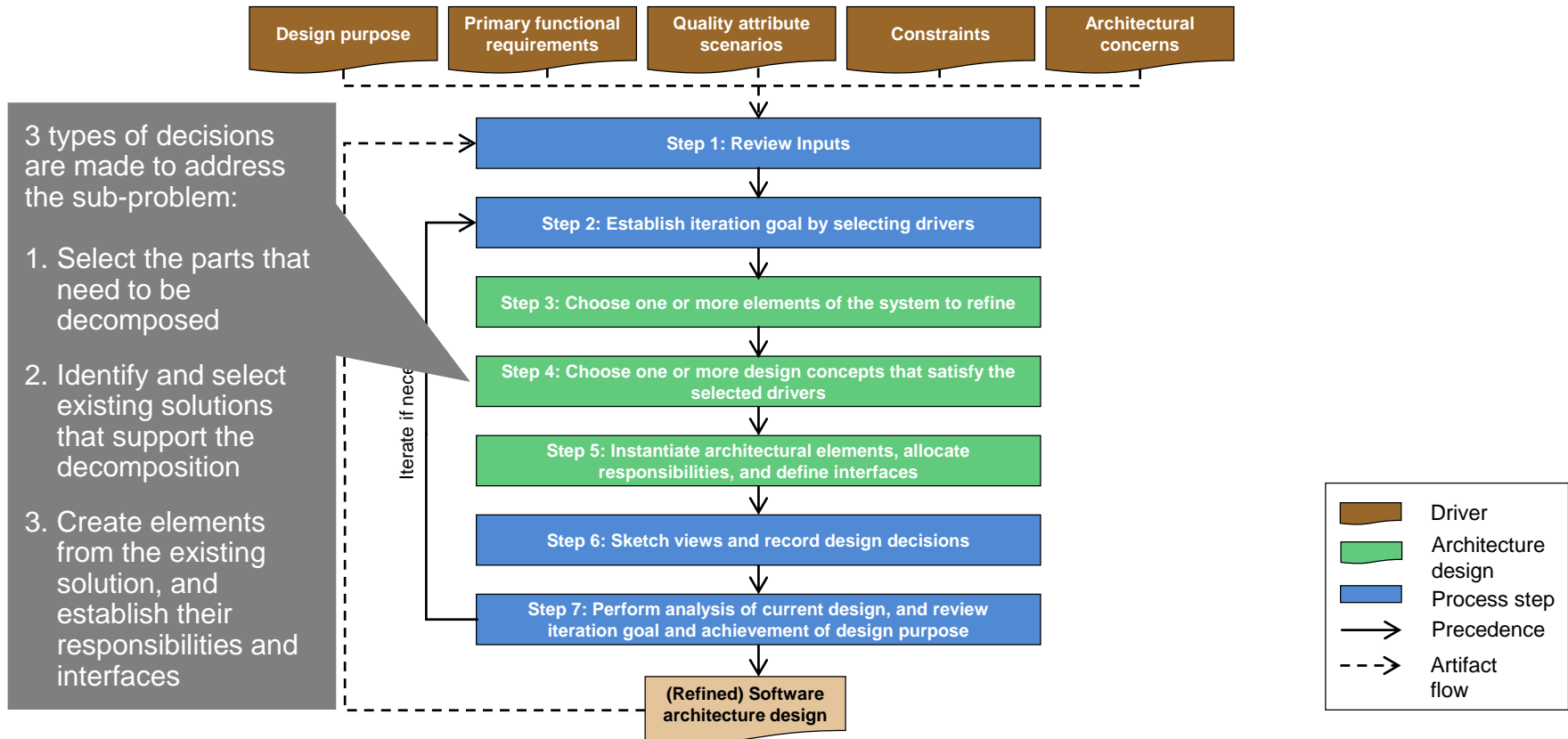
Step 1: Review Inputs



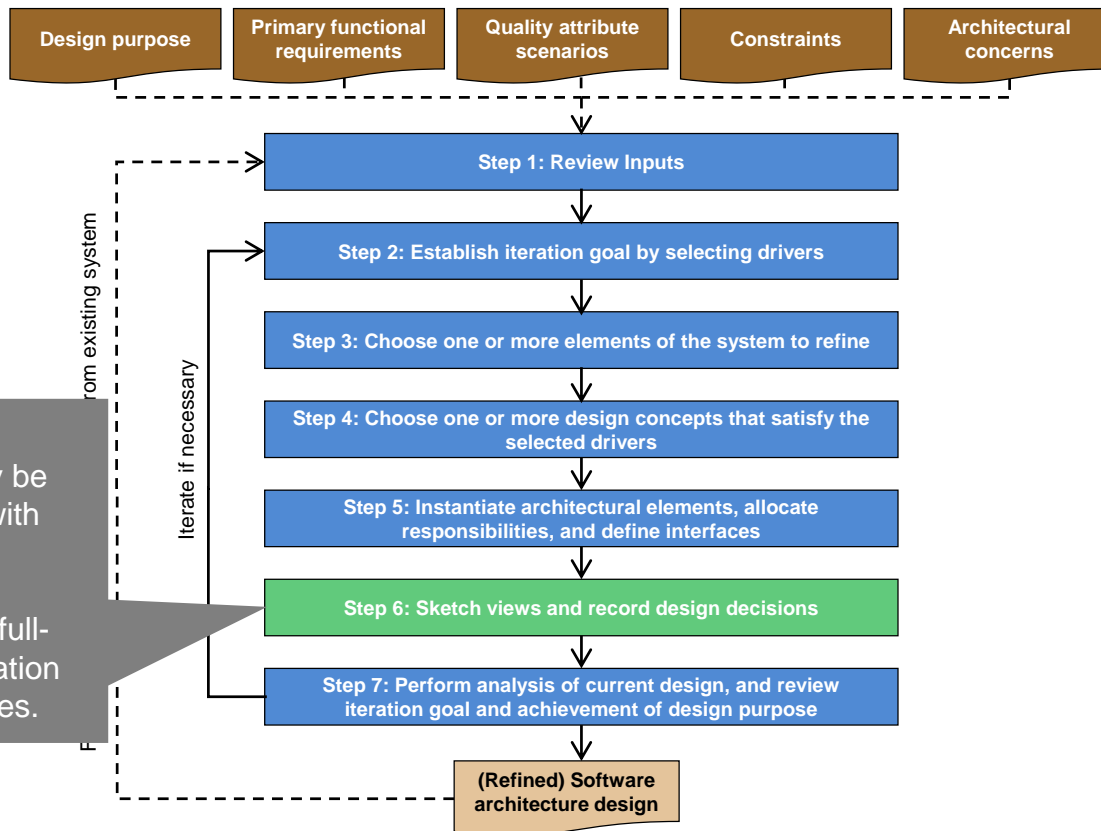
Step 2: Establish Iteration Goal



Steps 3–5: Choose and Instantiate Elements

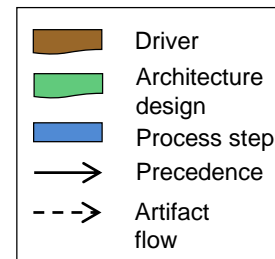


Step 6: Sketch Views

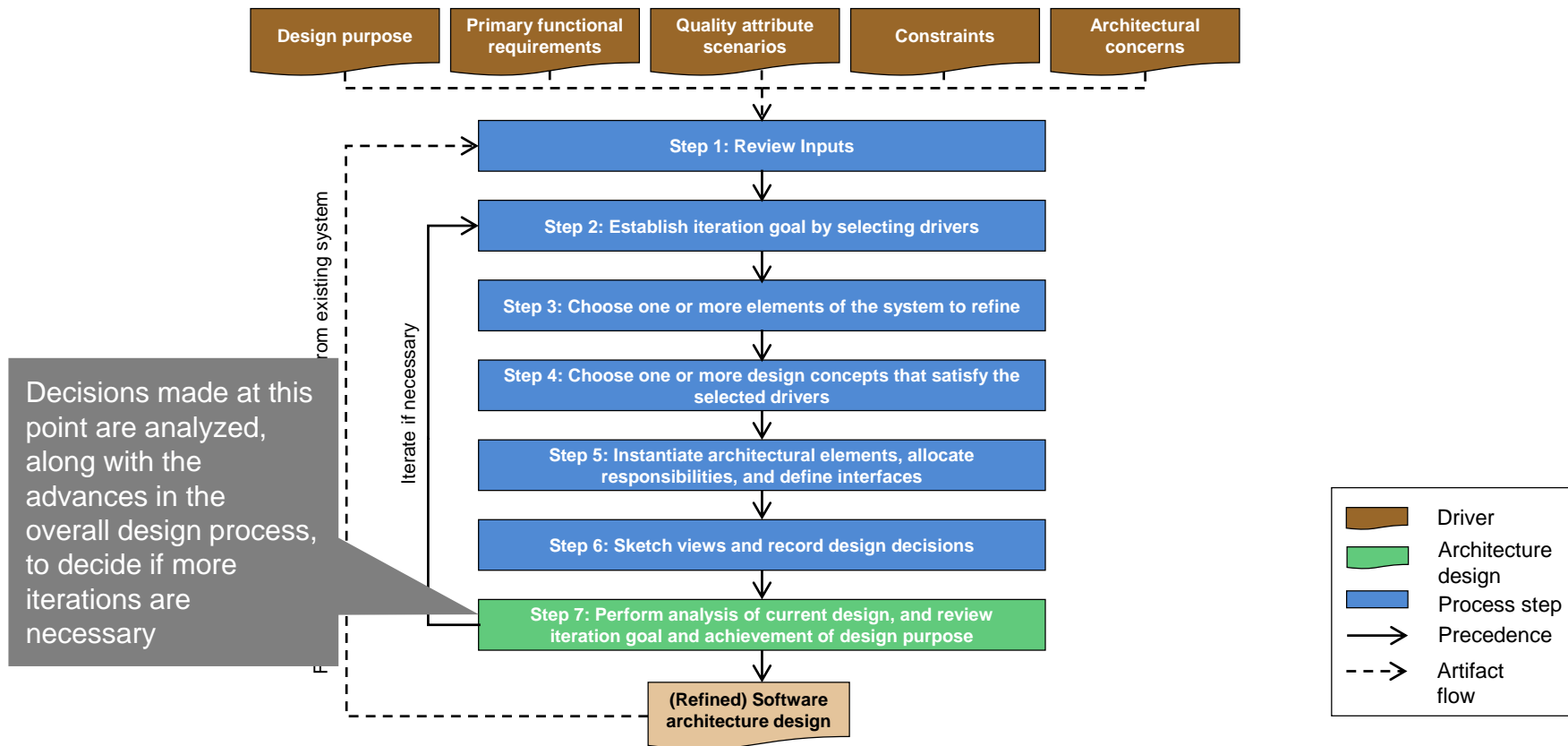


The “blueprint” is refined. This may be done in parallel with Step 5.

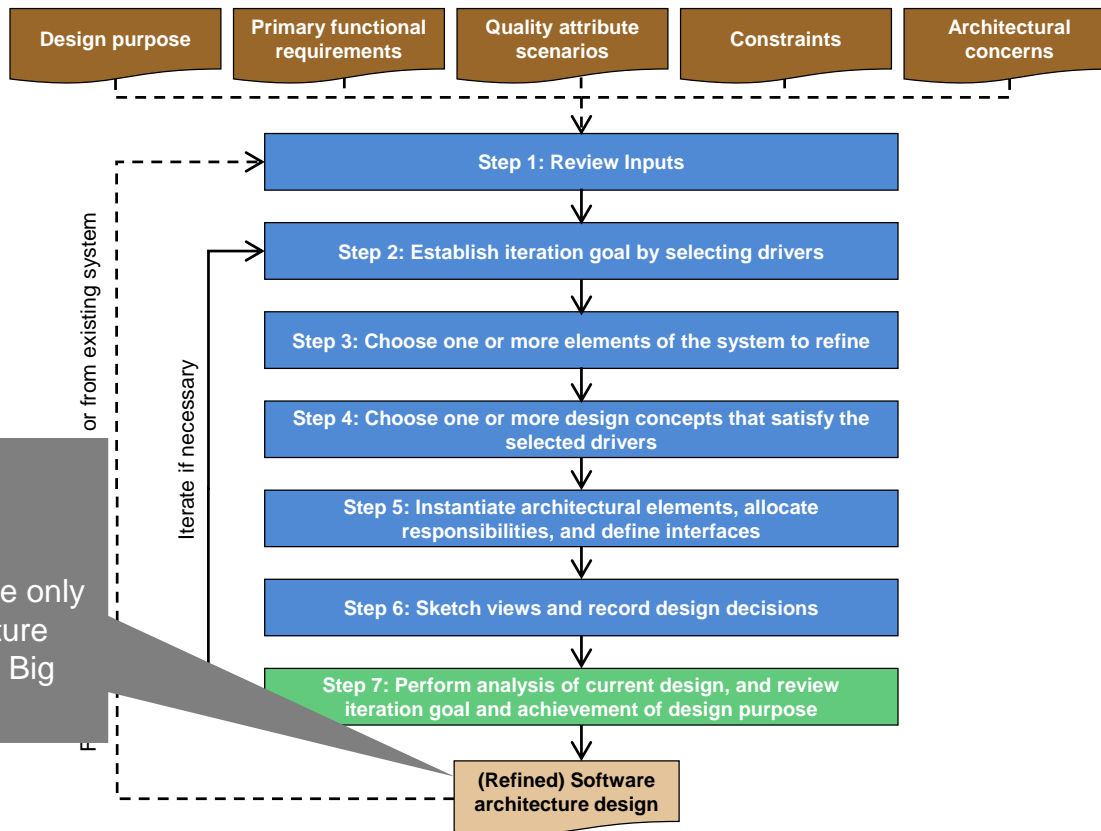
Note: This is not full-blown documentation but rather sketches.



Step 7: Perform Analysis

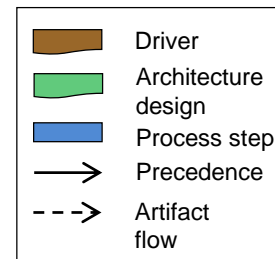


ADD Output/Iteration



The design is produced.

Note: This may be only a partial architecture design and is not Big Design Up Front!



ADD Resources

Resources for more information about ADD include

- books
 - *Software Architecture in Practice* describes ADD 2.0.
 - H. Cervantes, R. Kazman, *Designing Software Architectures: A Practical Approach*, Addison-Wesley, 2016, describes ADD 3.0 in detail.
- course
 - The SEI Software Architecture Design and Analysis course provides an in-depth look at ADD 3.0 and gives students hands-on experience with the method.

Module Summary

There are six major categories of design concepts:

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

Architectural patterns represent the collective experience of skilled software engineers in solving recurring design problems.

Tactics are fundamental design decisions that influence the control of quality attribute responses.

Attribute-Driven Design is a method for designing an architecture that bases the decomposition process on the architectural drivers that the software must meet.



Software Architecture: Principles and Practices

DOCUMENTING SOFTWARE ARCHITECTURES

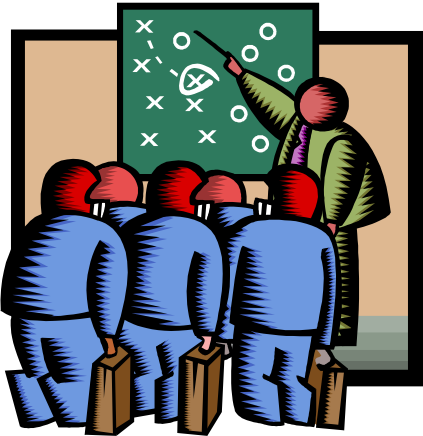
Module Objectives



This module will familiarize participants with

- architectural views and how they are used to document software architectures
- various types of views, why they are useful, and when to use them
- notations often used to describe architectures
- what a software architecture document should contain

Why Document an Architecture?



Architecture structures the system and the project that develops it.

- It defines the work assignments.
- It is the primary carrier of quality attributes.
- It is the best artifact for early analysis.
- It is the key to post-deployment maintenance and mining.

This blueprint must be understood if it is to be used. It must be communicated if it is to be understood.

Documentation speaks for the architect, today, tomorrow, and 20 years from now.

Architecture Documentation Also Contributes to Architecture Design

Documentation establishes the set of design decisions that must be made along the way to establishing and maintaining the architecture.

Documentation also clarifies the line between architectural and non-architectural design decisions.

- Non-architectural design is the term preferred over detailed design. Architectural decisions can be quite detailed!
- Architectural design decisions are those that affect the system's ability to deliver on its behavioral and quality attribute goals.

What Is Architecture Documentation Used For?

Construction: Documentation tells the developers what to build, how the pieces should behave, and how they should fit together.

Analysis: Analysts will use the documentation to assess the architecture for its ability to its job, particularly to provide the required behavior and quality attributes. Analyzing the documented architecture can uncover risks and problems long before the design is committed to code.

Education: Someone new to the project or unfamiliar with the solution approach can come up to speed using the architecture documentation.

Views: An Important Concept for Documentation

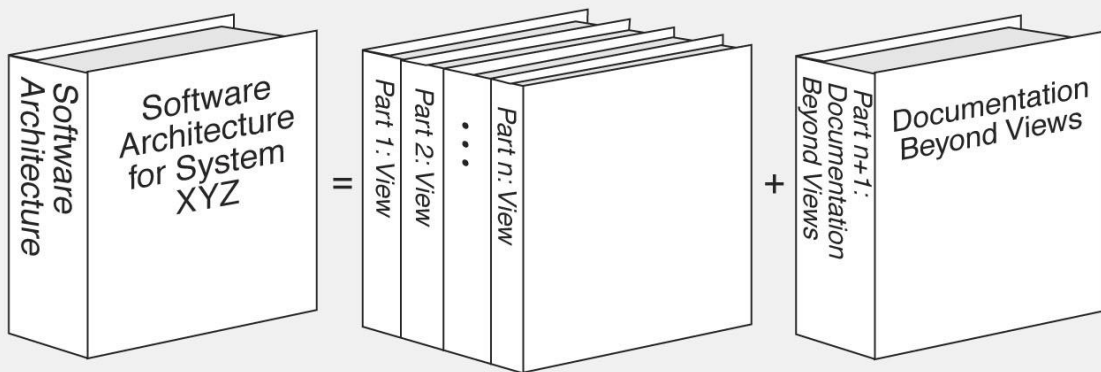
An architecture is a multidimensional construct, too involved to be seen all at once.

Systems are composed of many structures that show

- modules, their composition/decomposition, and mapping to code units
- processes and how they synchronize
- programs and how they call or send data to each other
- how software is deployed on hardware
- how teams cooperate to build the system
- how components and connectors work at runtime
- ...

A view is a representation of a structure. We use views to manage complexity by separating concerns.

View-Based Documentation



Documenting an architecture is a matter of documenting the relevant views, and then adding documentation that applies to more than one view.

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.

Three Types of Views

Recall that a view is a representation of a structure in the software.

Choosing to use a style or pattern usually leads to the creation and “filling out” of a software structure.

Recall that we showed three kinds of software structures.

Therefore, we have three kinds of views. Different kinds of views show different kinds of information:

1. **Module views** show how the system is structured as a set of code units.
2. **Component-and-connector views** show how the system is structured as a set of elements with runtime behaviors and interactions.
3. **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.

Information in a View

A view's primary function is to show the structure that it represents.

Hence, a view will show the elements that are in the structure and the relationships among them.

A view will also explain what the elements are, what their responsibilities are, and what their important properties are.

Properties are values used to describe the elements to help convey understanding and aid analysis. Properties are often quality-of-service values, such as an element's performance or reliability characteristics.

Architects can choose the properties they wish to document based on what the document's readers will wish to know.

Module Views

Elements: modules. A module is a code unit that implements a set of responsibilities.

Relations: Relations among modules include

- A is part of B. This defines a part-whole relation among modules.
- A depends on B. This defines a dependency relation among modules.
- A is a B. This defines specialization and generalization relations among modules.

Properties: Properties of a module usually include a name, responsibilities, and the visibility of the module and its interface.

What Are Module Views Used For?

Construction: These are the blueprints for the code.

Modules are assigned to teams for implementation and are often the basis for subsequent design (e.g., that of interfaces). Subsets and deployment packages are built using module styles.

Analysis: Traceability and impact analysis rely on implementation units. Project management, budgeting, planning, and tracking often use modules.

Education: A software developer can learn the development project's structure by understanding module views.

Component-and-Connector (C&C) Views

Elements: components and connectors

- Components are principal units of runtime interaction and data stores.
- Connectors are interaction mechanisms.

Relations: attachment of components' ports to connectors' roles (interfaces with protocols)

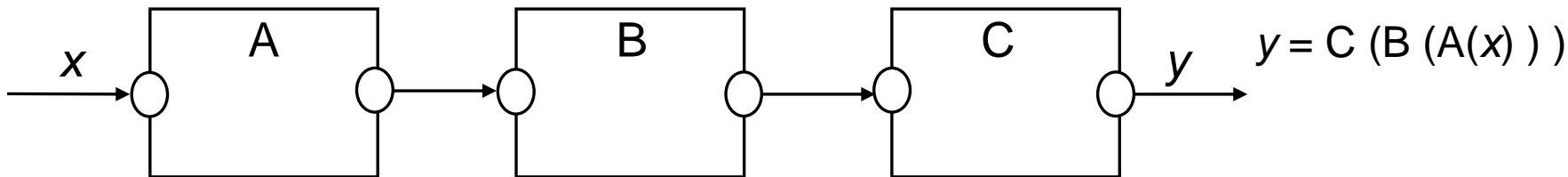
Properties: Properties of a component or connector often include a name and runtime quality-of-service information to facilitate analysis or prediction of runtime quality attributes.

What Are C&C Views Used For?

Construction: for specifying the behavior that elements must exhibit

Education: as a starting point for the architect to show how the system works

Analysis: for reasoning about runtime system quality attributes such as performance, security, and reliability



Allocation Views

Elements:

- software elements (as defined in module or C&C styles)
- environment elements

Relations: Software elements are “allocated to” environment elements.

Properties: Properties documented in an allocation view are “requires” properties of software elements and “provides” properties of the environment elements. If the properties are compatible, then the allocation is a sound one.

Applying Views to Heterogeneous Systems

Systems are usually built by employing several different styles or patterns. Architectures are almost guaranteed to use more than one architectural style.

As a result, systems usually contain large numbers of diverse types of elements. This heterogeneity might show up in the following ways:

1. Different “areas” of the system (e.g., different subsystems) might use different styles or patterns and be documented in different views.
2. An element of one style might be decomposed into elements arranged in another style. For example, a *service* in an SOA view might be designed internally to use the *shared-data* style and documented using a *shared-data* view.
3. The architect might have combined two styles or patterns to solve a design problem. The view will show element types and relation types from each.

Views help sort out these combinations and aid understanding.

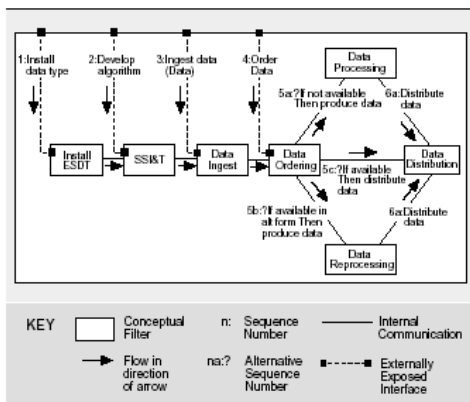
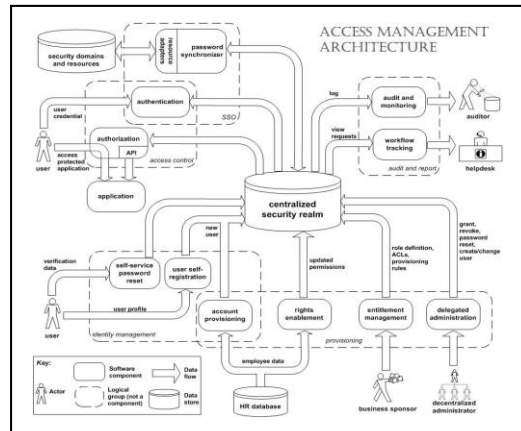
Other Information in a View

Architects must also be concerned with documenting

- **interfaces:** Elements cannot interact each other except through their interfaces. Interfaces must be documented as part of the architecture. Interfaces consist of syntax plus semantics—not just APIs!
- **rationale:** Documenting *why* the architect made the decisions he/she did can save critical time in the future, when the system is ready to evolve and the original architect is long gone.
- **variability** in architectures, such as components that are plug-replaceable or optional
- **dynamic architectures:** components and connectors that change at runtime
- **context:** the external environment and how it relates to our system
- **behavior:** How do the elements behave when the system runs? Structure just shows connections; behavior shows how and when those connections “fire.”

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 for Documenting Behavior

1. 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

2. Comprehensive languages

- show the complete behavior of a system
- are usually state based (e.g., statecharts)
- can infer all traces using a static model—even impossible traces
- support the documentation of alternatives

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

Which Is Which?

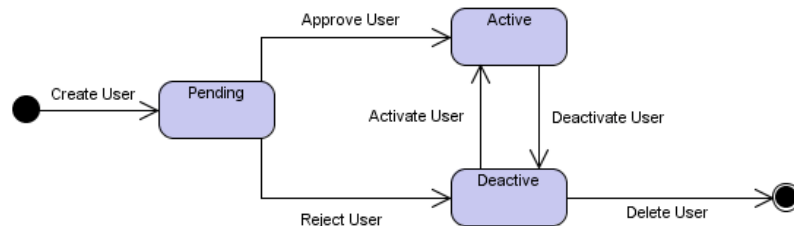
Trace-oriented languages

- use cases*
- communication diagrams*
- sequence diagrams*
- message sequence charts
- activity diagrams*
- timing diagrams*
- Business Process Execution Language (BPEL)
- ...

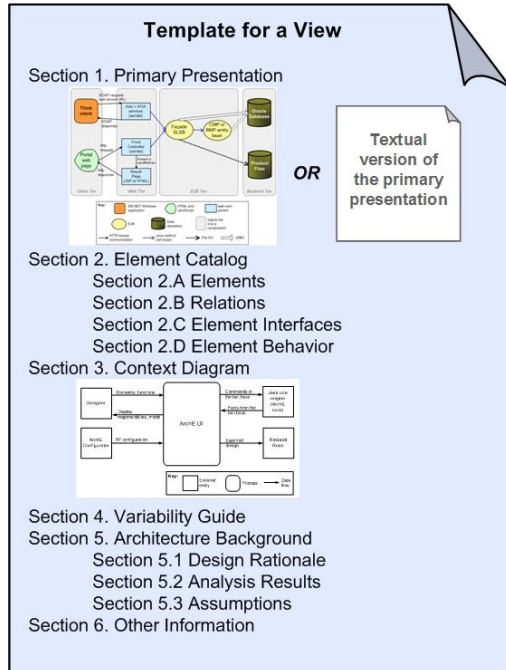
* available in UML

Comprehensive languages

- statecharts*
- SDL diagrams
- Z specifications
- some ADLs
- ...



Use a Standard Organization for Your Architecture Document



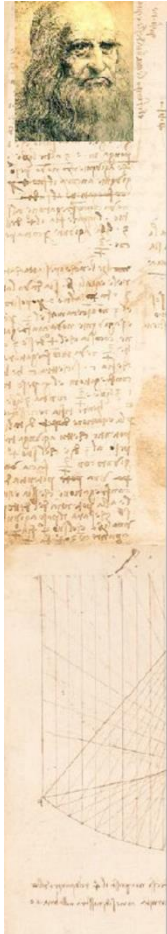
Establish it, make sure that your documents follow it, and make sure that readers know what it is.

A standard organization

- helps the reader navigate and find information
- helps the writer place information 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

Example Architecture Document

https://wiki.sei.cmu.edu/sad/index.php/The_Adventure_Builder_SAD



page discussion view source history

Adventure Builder - Software Architecture Document (SAD)

(Redirected from The Adventure Builder SAD)

Authors: Paulo Merson, Darpan Saini

[READ ME \(disclaimers\)](#)

Contents

[Documentation Roadmap](#)

[How a View is Documented](#)

[System Overview](#)

[Views](#)

1. Module Views
 1. Top Level Module Uses View
 2. OPC Module Decomposition View
 3. OPC Module Uses View
 - [OpcPurchaseOrderService Interface Documentation](#)
 - [OpcOrderTrackingService Interface Documentation](#)
 4. workflowmanager Module Uses View
 5. Data Model
2. C&C Views
 1. Top Level SOA View
 2. Consumer Website Multi-tier View
 3. OPC C&C View
3. Allocation Views
 1. Deployment View
 2. Install View
 3. **Implementation View**

[Mapping Between Views](#)

[Rationale](#)

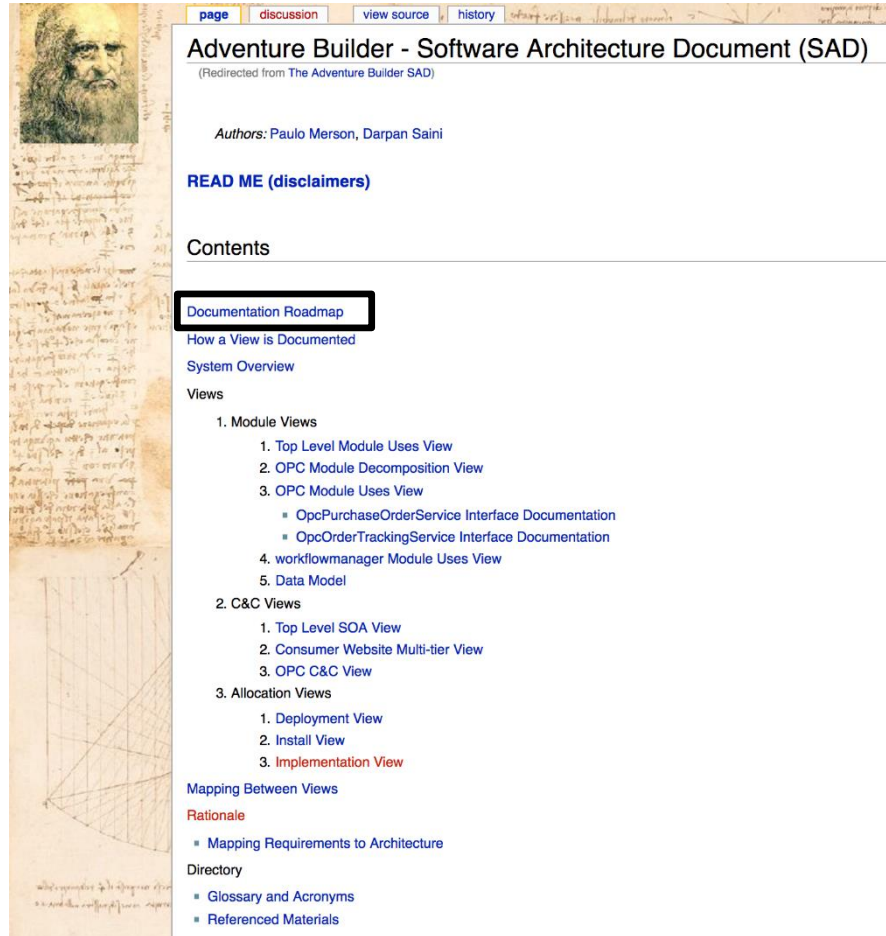
- [Mapping Requirements to Architecture](#)

[Directory](#)

- [Glossary and Acronyms](#)
- [Referenced Materials](#)

Example Architecture Document

https://wiki.sei.cmu.edu/sad/index.php/The_Adventure_Builder_SAD



page discussion view source history

Adventure Builder - Software Architecture Document (SAD)

(Redirected from The Adventure Builder SAD)

Authors: Paulo Merson, Darpan Saini

[READ ME \(disclaimers\)](#)

Contents

Documentation Roadmap

How a View is Documented

System Overview

Views

1. Module Views
 1. Top Level Module Uses View
 2. OPC Module Decomposition View
 3. OPC Module Uses View
 - OpcPurchaseOrderService Interface Documentation
 - OpcOrderTrackingService Interface Documentation
 4. workflowmanager Module Uses View
 5. Data Model
2. C&C Views
 1. Top Level SOA View
 2. Consumer Website Multi-tier View
 3. OPC C&C View
3. Allocation Views
 1. Deployment View
 2. Install View
 3. **Implementation View**

[Mapping Between Views](#)

Rationale

- [Mapping Requirements to Architecture](#)

Directory

- [Glossary and Acronyms](#)
- [Referenced Materials](#)

AdventureBuilder Documentation Roadmap

(Redirected from [Adventure DocumentRoadmap](#))

Contents [hide]

- 1 [Scope and summary](#)
- 2 [How the documentation is organized](#)
 - 2.1 [How to update this architecture document](#)
- 3 [View overview](#)
- 4 [How stakeholders can use the documentation](#)

Scope and summary

This wiki describes the architecture of the Java Adventure Builder Reference application, which is part of the BluePrints program by Sun Microsystems. The Adventure Builder is an e-commerce application that sells adventure packages for vacationers over the Internet. It uses web services to interact with external suppliers such as banks, airlines, hotels and adventure providers.

The architecture described here was reconstructed from the implementation artifacts of the Adventure Builder, but we made some changes to make the architecture documentation and the system itself look more interesting for our purpose (to provide an example of software architecture documentation).

How the documentation is organized

This architecture document is organized into the following sections:

- [Documentation Roadmap](#): provides information about this document and its intended audience. It provides the roadmap and document overview.
- [How a View is Documented](#): describes how to use the template for an architecture view.
- [System Overview](#): provides a brief description of the Adventure Builder application, its functional and quality attribute requirements.
- [Views](#): several architecture views are described, each in a separate wiki page. Each view describe a different structure of the system. There are module views, component & connector (C&C) views, and allocation views. The architecture views are the most important part of the document. More general information about the views can be found down below in the [View overview](#) section.
- [Mapping Between Views](#): specify how elements in one view map to elements in another view for a few selected pairs of views.
- [Rationale](#): provides information about the software architecture. It describes the background and rationale for the software architecture. It explains the constraints and influences that led to the current architecture, and it describes the major architectural approaches that have been utilized in the architecture.
- [Directory](#): contains a glossary of technical terms and acronyms used in the documentation that may not be familiar to all readers. It also contains a list of referenced materials.

Places where the documentation is not complete are marked with "TODO" or "TODO: <description of what's missing>". If a section is marked as "N/A", it means the contents of that particular section are deemed not applicable or not necessary.

[Edit this TOC](#) 🗑️ [View this alone](#)

[Documentation Roadmap](#)

[How a View is Documented](#)

[System Overview](#)

[Views](#)

1. Module Views

1. [Top Level Module Uses View](#)
2. [OPC Module Decomposition View](#)
3. [OPC Module Uses View](#)
 - [OpcPurchaseOrderService Interface Documentation](#)
 - [OpcOrderTrackingService Interface Documentation](#)
4. [workflowmanager Module Uses View](#)
5. [Data Model](#)

2. C&C Views

1. [Top Level SOA View](#)
2. [Consumer Website Multi-tier View](#)
3. [OPC C&C View](#)

3. Allocation Views

1. [Deployment View](#)
2. [Install View](#)
3. [Implementation View](#)

[Mapping Between Views](#)

[Rationale](#)

- [Mapping Requirements to Architecture](#)

[Directory](#)

- [Glossary and Acronyms](#)
- [Referenced Materials](#)

Example Architecture Document

https://wiki.sei.cmu.edu/sad/index.php/The_Adventure_Builder_SAD

page discussion view source history

Adventure Builder - Software Architecture Document (SAD)

(Redirected from The Adventure Builder SAD)

Authors: Paulo Merson, Darpan Saini

[READ ME \(disclaimers\)](#)

Contents

[Documentation Roadmap](#)

How a View is Documented

[System Overview](#)

Views

1. Module Views
 1. Top Level Module Uses View
 2. OPC Module Decomposition View
 3. OPC Module Uses View
 - [OpcPurchaseOrderService Interface Documentation](#)
 - [OpcOrderTrackingService Interface Documentation](#)
 4. workflowmanager Module Uses View
 5. Data Model
2. C&C Views
 1. Top Level SOA View
 2. Consumer Website Multi-tier View
 3. OPC C&C View
3. Allocation Views
 1. Deployment View
 2. Install View
 3. **Implementation View**

[Mapping Between Views](#)

Rationale

- [Mapping Requirements to Architecture](#)

Directory

- [Glossary and Acronyms](#)
- [Referenced Materials](#)

Template:ArchitectureViewTemplate

(Redirected from [Template:ViewTemplate](#))

Contents [hide]

- [1 Primary Presentation](#)
- [2 Element Catalog](#)
- [3 Context Diagram](#)
- [4 Variability Guide](#)
- [5 Rationale](#)
- [6 Related Views](#)

Primary Presentation

TODO: Add here the diagram (or non-graphical representation) that shows the elements and relations in this view. Indicate the language or notation being used. If it's not a standard notation such as UML, add a notation key.

Element Catalog

TODO: This section can be organized as a dictionary where each entry is an element of the Primary Presentation. For each element, provide additional information and properties that the readers would need that would not fit in the Primary Presentation. Optionally, you can add interface specifications and behavior diagrams (e.g., UML sequence diagrams, statecharts).

Context Diagram

TODO: Add here a context diagram that graphically shows the scope of the part of the system represented by this view. A context diagram typically shows the part of the system as a single, distinguished box in the middle surrounded by other boxes that are the external entities. Lines show the relations between the part of the system and the external entities.

Variability Guide

TODO: Describe here any variability mechanisms used in the portion of the system shown in this view, along with how and when (build time, deploy time, run time) those mechanisms may be exercised. Examples of variability include: optional components (e.g., plug-ins, add-ons); configurable replication of components and connectors; selection among different implementations of an element or different vendors; parameterized values set in build flags, .properties files, .ini files, or other config files.

Rationale

TODO: Describe here the rationale for any significant design decisions whose scope is limited to this view. Also describe any significant rejected alternatives. This section may also indicate assumptions, constraints, results of analysis and experiments, and architecturally significant requirements that affect the view.

[Edit this TOC](#)  [View this alone](#)

[Documentation Roadmap](#)

[How a View is Documented](#)

[System Overview](#)

Views

1. Module Views

1. Top Level Module Uses View
2. OPC Module Decomposition View
3. OPC Module Uses View
 - [OpcPurchaseOrderService Interface Documentation](#)
 - [OpcOrderTrackingService Interface Documentation](#)
4. workflowmanager Module Uses View
5. Data Model

2. C&C Views

1. Top Level SOA View
2. Consumer Website Multi-tier View
3. OPC C&C View

3. Allocation Views

1. Deployment View
2. Install View
3. [Implementation View](#)

[Mapping Between Views](#)

[Rationale](#)

■ [Mapping Requirements to Architecture](#)

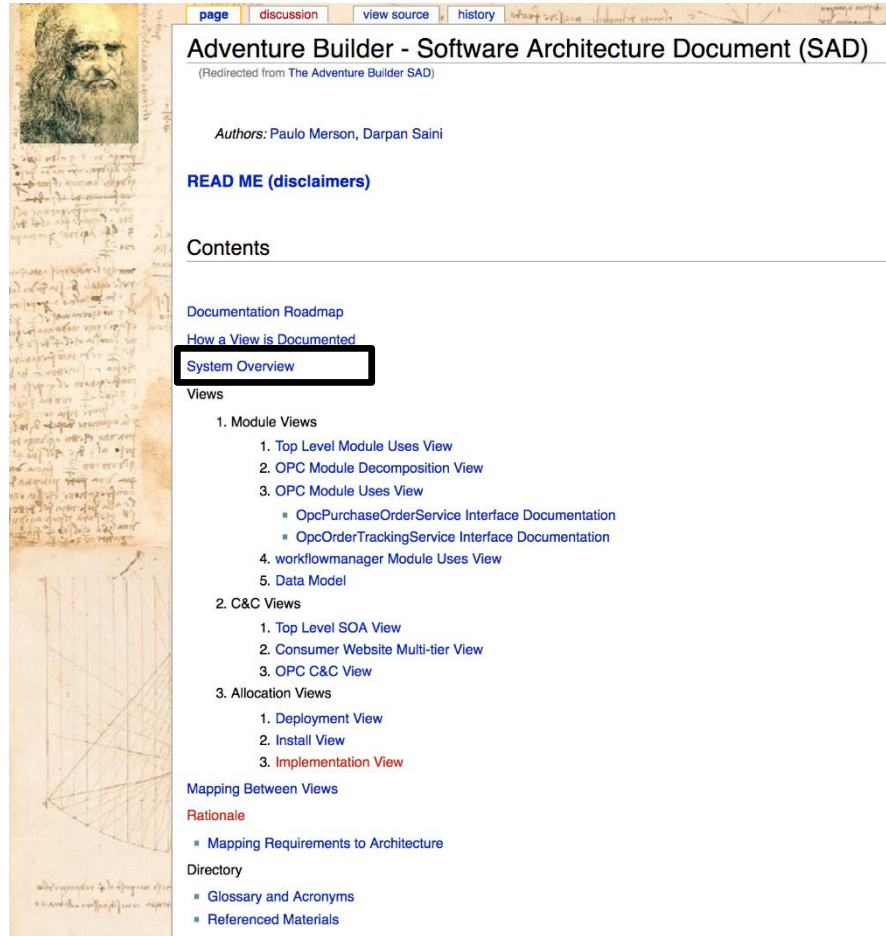
[Directory](#)

■ [Glossary and Acronyms](#)

■ [Referenced Materials](#)

Example Architecture Document

https://wiki.sei.cmu.edu/sad/index.php/The_Adventure_Builder_SAD



page discussion view source history

Adventure Builder - Software Architecture Document (SAD)

(Redirected from The Adventure Builder SAD)

Authors: Paulo Merson, Darpan Saini

[READ ME \(disclaimers\)](#)

Contents

Documentation Roadmap

How a View is Documented

System Overview

Views

1. Module Views
 1. Top Level Module Uses View
 2. OPC Module Decomposition View
 3. OPC Module Uses View
 - OpcPurchaseOrderService Interface Documentation
 - OpcOrderTrackingService Interface Documentation
 4. workflowmanager Module Uses View
 5. Data Model
2. C&C Views
 1. Top Level SOA View
 2. Consumer Website Multi-tier View
 3. OPC C&C View
3. Allocation Views
 1. Deployment View
 2. Install View
 3. Implementation View

Mapping Between Views

Rationale

- Mapping Requirements to Architecture

Directory

- Glossary and Acronyms
- Referenced Materials

System Overview

Adventure Builder Reference Application

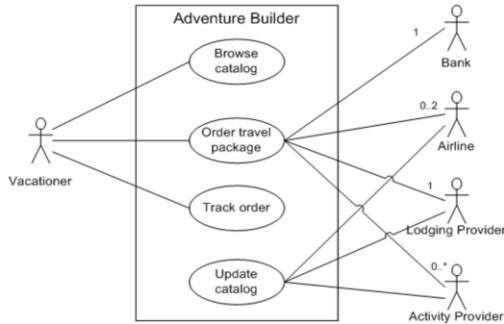
The system used as an example in this architecture document is an adapted version of the [Adventure Builder Reference application](#), which was developed in the context of the Java BluePrints program at Sun Microsystems. This application was chosen because the functionality is easy to understand, and the source code, documentation, and other artifacts are publicly available for download. Also available is a book on Web services that explains the design and implementation of the application [Singh 2004].

The architecture documented here does not reflect exactly the implementation provided by Sun. To make it a more interesting and realistic example of an SOA solution, we made several assumptions about the business context and requirements of the system, and documented design elements and relations that deviate from the original implementation. [Click for more information.](#)

Functionality

Adventure Builder is a fictitious company that sells adventure packages for vacationers over the Internet. The system provides four basic use cases (UC):

- UC1. The user can visit the Adventure Builder Web site and browse the catalog of travel packages, which include flights to specific destinations, lodging options, and activities that can be purchased in advance. Activities include mountain biking, fishing, surfing classes, hot air balloon tours, and scuba diving. The user can select transportation, accommodation, and various activities to build his/her own adventure trip.
- UC2. The user can place an order for a vacation package. To process this order, the system has to interact with several external entities. A bank will approve the customer payment, airline companies will provide the flights, lodging providers will book the hotel rooms, and businesses that provide vacation activities will schedule the activities selected by the customer.
- UC3. After an order is placed, the user can return to check the status of his/her order. This is necessary because some interactions with external entities are processed in the background and may take hours or days to complete.
- UC4. The internal system periodically interacts with its business partners (transportation, lodging, and activity providers) to update the catalog with the most recent offerings.



[Visio file](#)

Quality Attribute Requirements

The quality attribute scenarios (QAS) are listed below, separated by quality attribute.

Modifiability

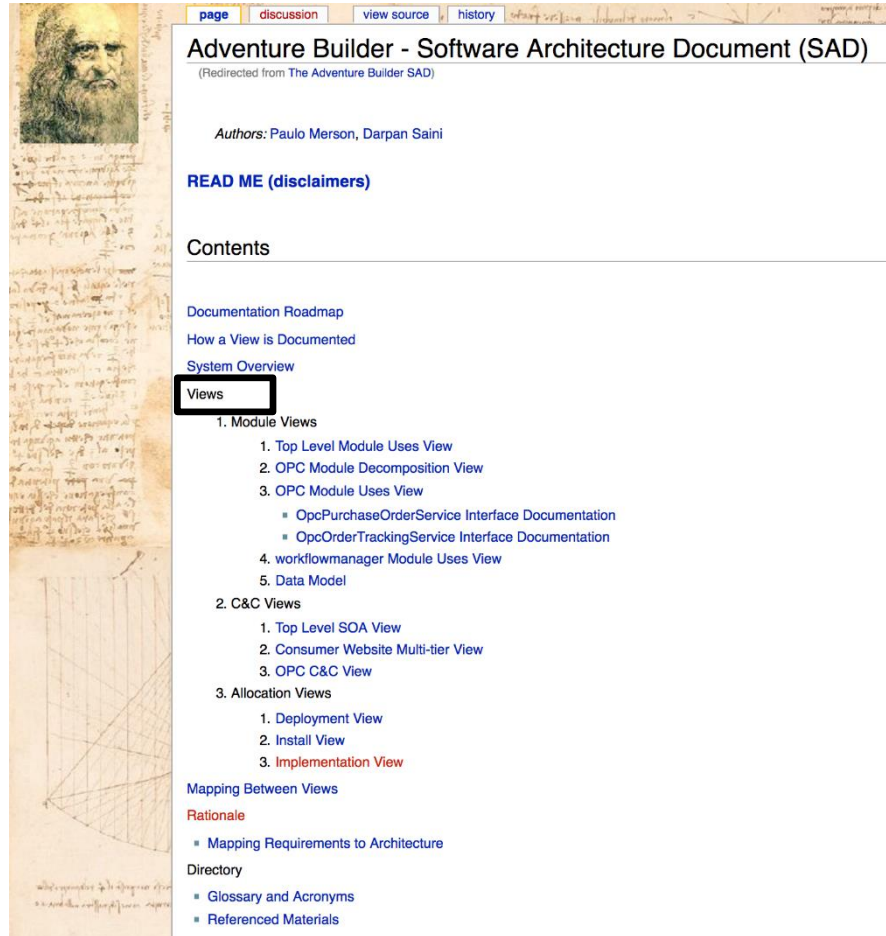
- QAS1. A new business partner (airline, lodging, or activity provider) that uses its own web services interface is added to the system in no more than 10 person-days of effort for the implementation. The business goal is e

Performance

- QAS2. A user places an order for an adventure travel package to the Consumer Web site. The user is notified on screen that the order has been successfully submitted and is being processed in less than five seconds.

Example Architecture Document

https://wiki.sei.cmu.edu/sad/index.php/The_Adventure_Builder_SAD



page discussion view source history

Adventure Builder - Software Architecture Document (SAD)

(Redirected from The Adventure Builder SAD)

Authors: Paulo Merson, Darpan Saini

[READ ME \(disclaimers\)](#)

Contents

Documentation Roadmap

How a View is Documented

System Overview

Views

1. Module Views
 1. Top Level Module Uses View
 2. OPC Module Decomposition View
 3. OPC Module Uses View
 - OpcPurchaseOrderService Interface Documentation
 - OpcOrderTrackingService Interface Documentation
 4. workflowmanager Module Uses View
 5. Data Model
2. C&C Views
 1. Top Level SOA View
 2. Consumer Website Multi-tier View
 3. OPC C&C View
3. Allocation Views
 1. Deployment View
 2. Install View
 3. **Implementation View**

Mapping Between Views

Rationale

- Mapping Requirements to Architecture

Directory

- Glossary and Acronyms
- Referenced Materials

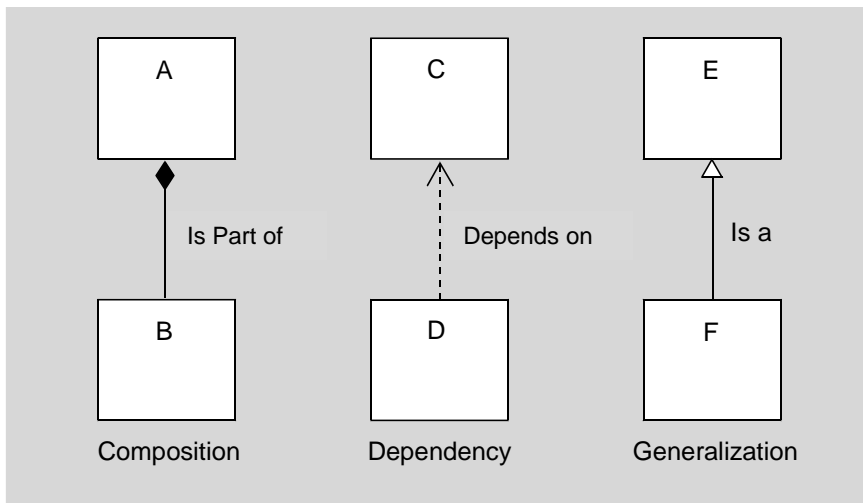
Notations for Architecture Views

Notations for documenting views differ in their degree of formality. There are three main categories of notation:

- **Informal notations:** Views are depicted graphically using general-purpose diagramming tools and visual conventions chosen for the system at hand. The semantics of the description are characterized in natural language and generally cannot be formally analyzed.
- **Semi-formal notations:** Views are expressed in a standardized notation that prescribes graphical elements and rules of construction but does not provide a complete semantic treatment of the meaning of those elements. Rudimentary analysis can be applied to determine whether a description satisfies syntactic properties.
- **Formal notations:** Views are described in notation that has a precise (usually mathematically based) semantics. Formal analysis of both syntax and semantics is possible.

More formal notations take more time and effort to create, but they repay this effort in reduced ambiguity and better opportunities for analysis.

Notations for Module Views' Primary Presentations



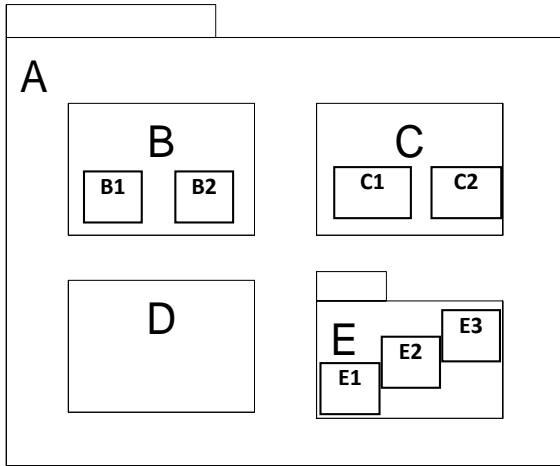
Key:
UML 2.0

Informal:

- box-and-line drawings (Nesting can represent “is part of” relations.)
- tables or lists (Indenting can represent “is part of” relations.)

Semi-formal: UML class diagrams and package diagrams

Decomposition View



Elements: modules

Relations: “is part of”

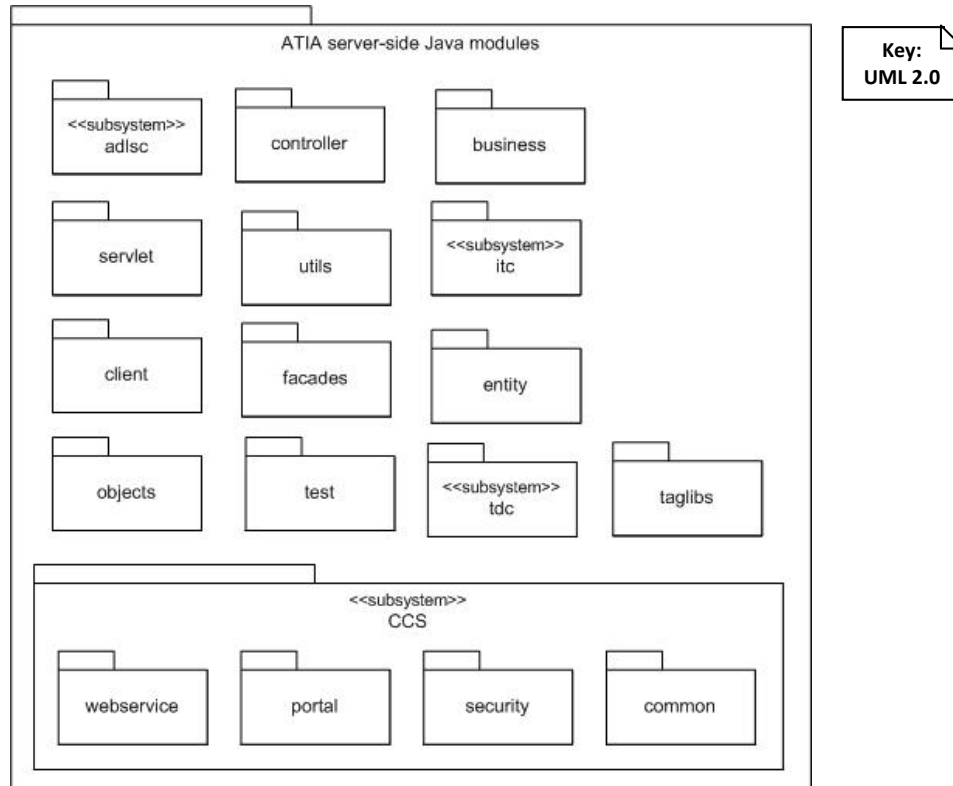
- The criteria for decomposition vary:
 - achievement of modifiability
 - build versus buy
 - software product lines: common versus unique parts
 - developers’ skills

Topology: A child can have only one parent.

What it’s for:

- assigning responsibilities to modules as a prelude to subsequent, downstream design
- conducting change/impact analysis
- developing work assignments
- developing unit testing

Decomposition View Primary Presentation: UML



Uses View

Elements: modules

Relations: “uses,” a specialization of “depends on”

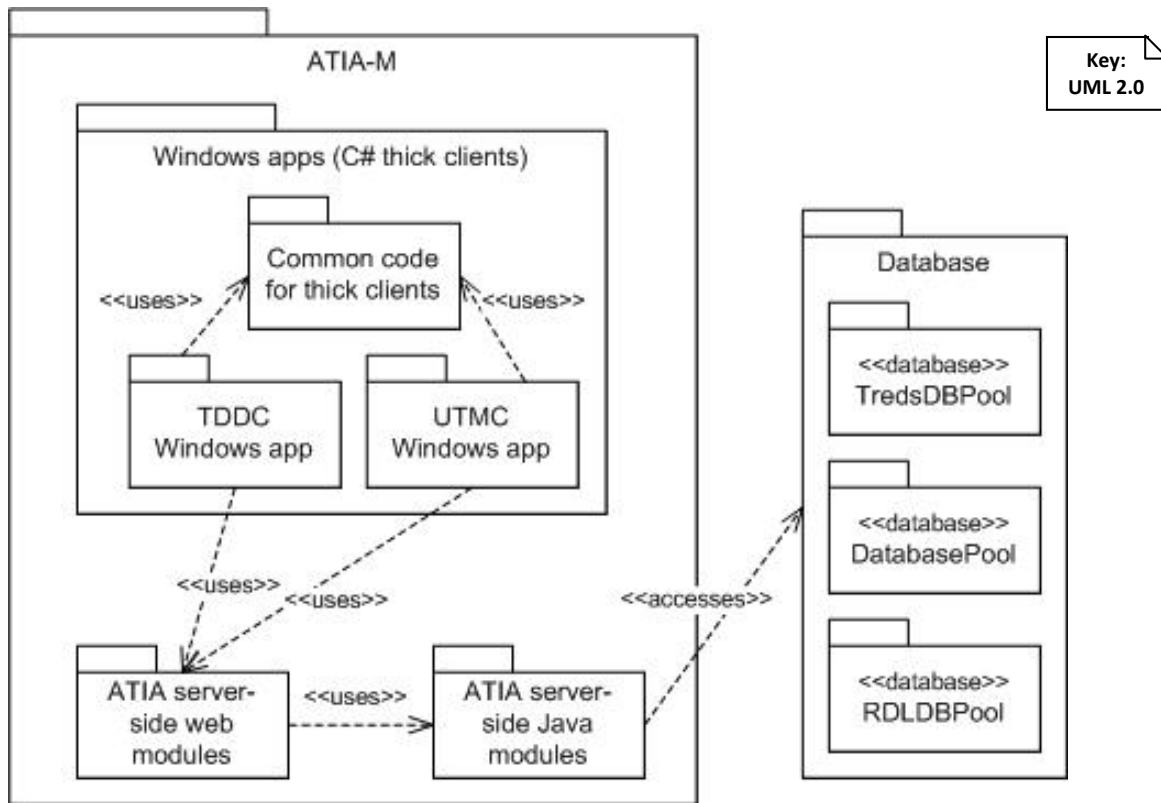
- A uses B if A depends on the presence of a correctly functioning B to satisfy its (A’s) own requirements.

Topology: no constraints (However, loops can cause problems with incremental system delivery.)

What it’s for:

- planning incremental development
- use in system extensions and subsets
- debugging and testing
- gauging the effects of specific changes

Uses View Primary Presentation: UML



Generalization View

Elements: modules

Relations: generalization, an “is a” relation

Properties: abstract? (i.e., is this an interface without a complete implementation?)

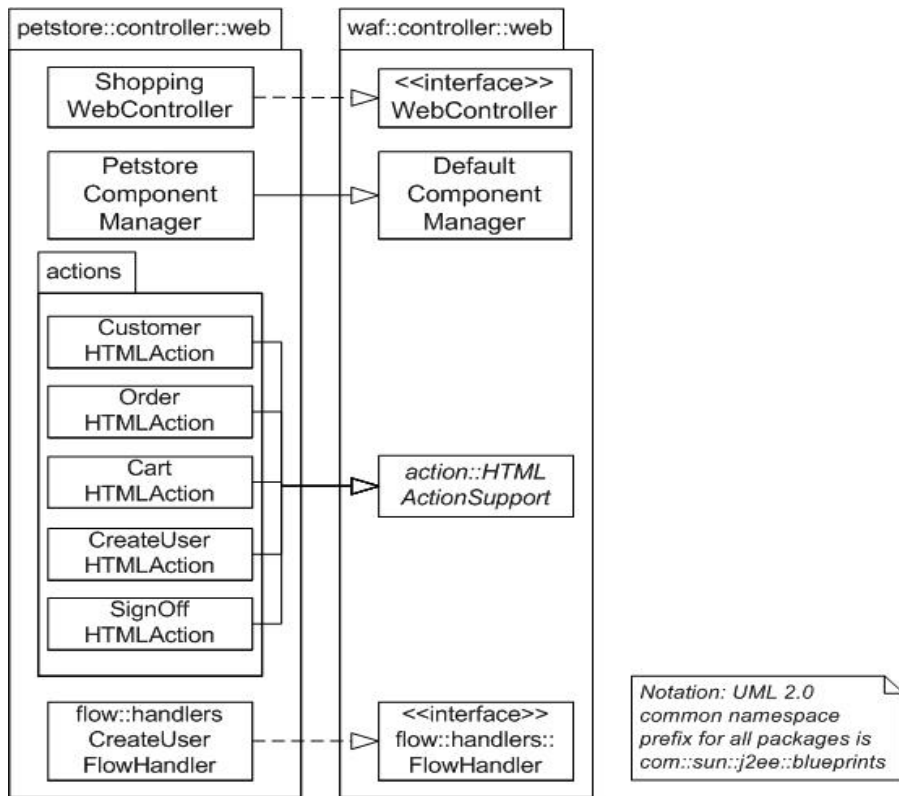
Topology:

- A module can have multiple parents.
- Cycles are prohibited.

What it's for:

- basis for object-oriented designs
- incrementally describing evolution and extension
- capturing commonalities, with variations as children
- supporting reuse

Generalization View Primary Presentation: UML



Layered View – 1

A		
B1	B2	B3
C		

Elements: layers, a virtual machine

Relations: “allowed to use,” a specialization of the “depends on” relation

- Recall that A uses B if A’s correctness depends on the presence of a correct B.
- The relation is not necessarily transitive.

Topology:

- Every piece of software is assigned to exactly one layer.
- Software in a layer is allowed to use software in {any lower layer, next lower layer}.
- Software in a layer {is, is not} allowed to use other software in the same layer.

Layered View – 2

A		
B1	B2	B3
C		

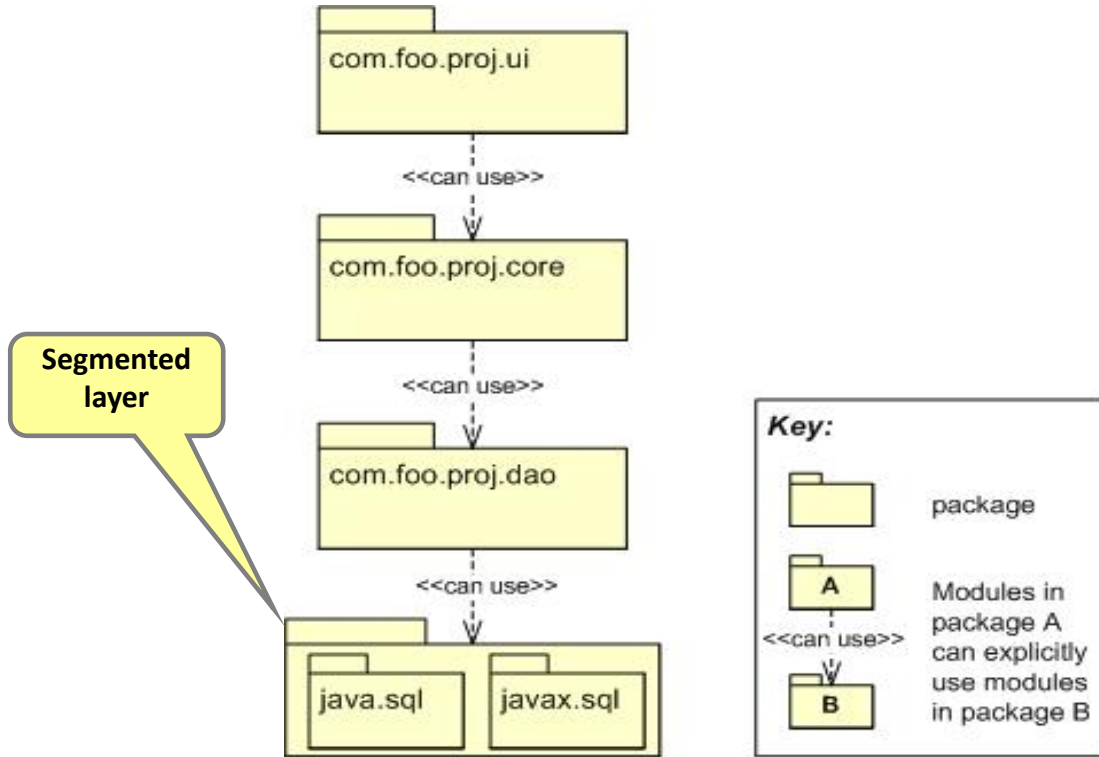
What it's for:

- promotes portability
- fielding subsets and incremental development
- separation of concerns
- promotes reusability

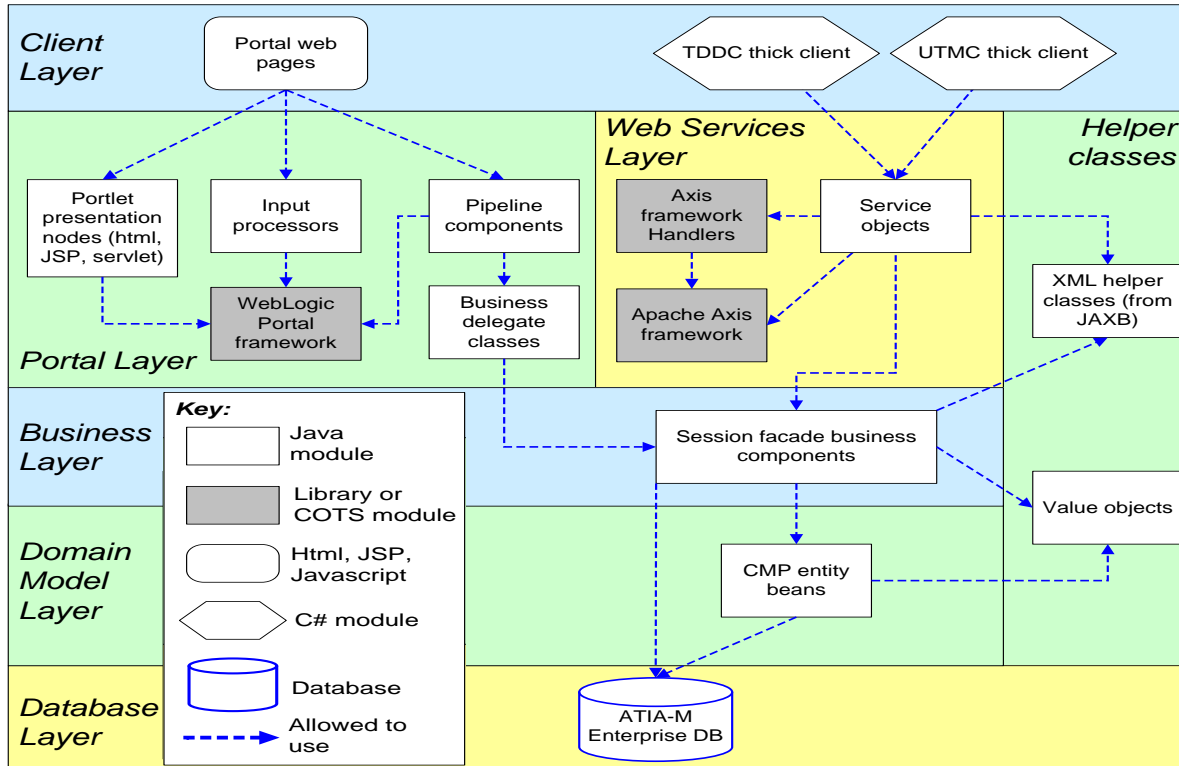
Variations:

- segmented layers: dividing a layer into segments (or submodules), with “allowed to use” relations between those segments and segments from other layers

Layered View Primary Presentation: UML Packages



Layered View Primary Presentation: Informal Notation



Notations for C&C Views' Primary Presentations

Informal:

- box-and-line diagrams
- Most box-and-line diagrams showing runtime behavior are in fact attempting to be C&C views.

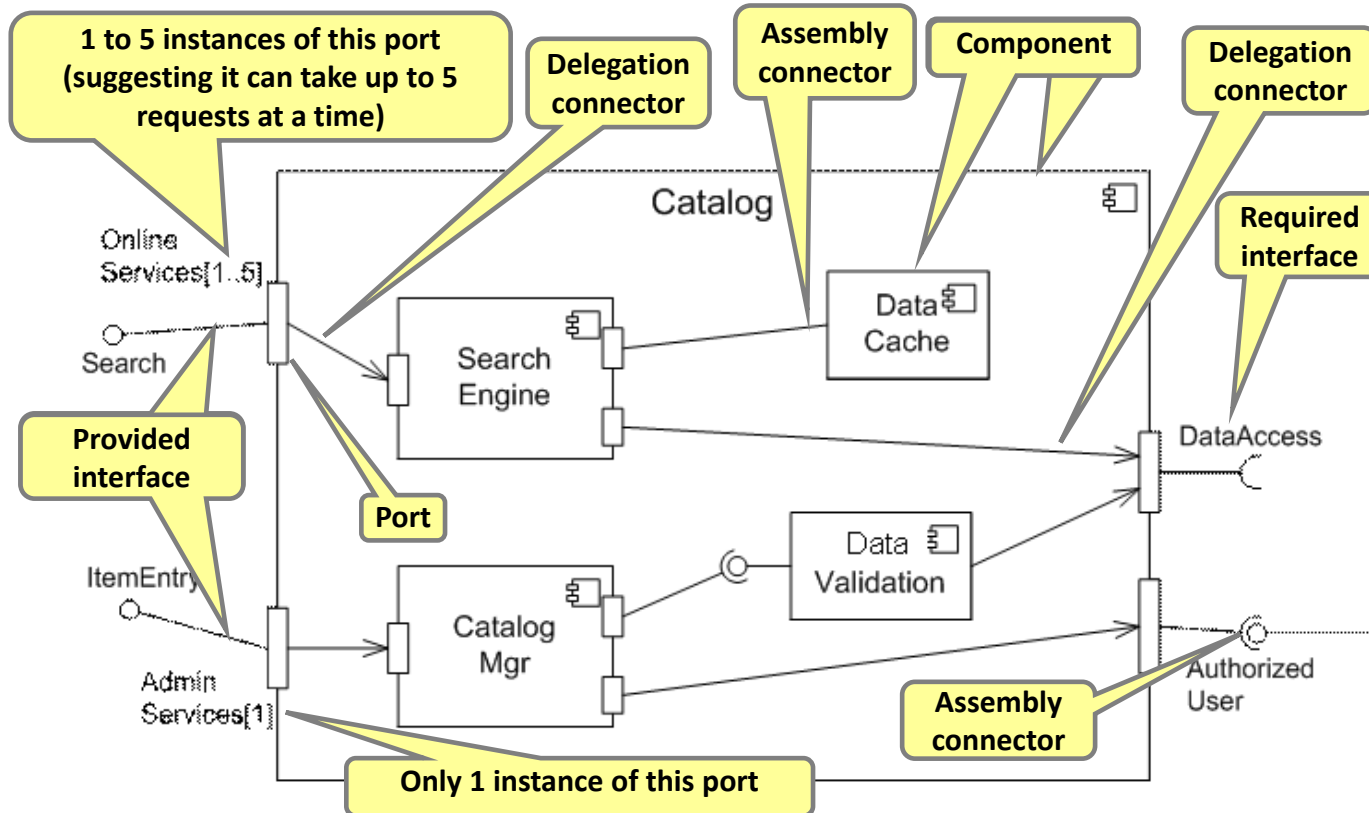
Formal:

- architecture description languages such as Acme, Wright, UniCon, the Architecture Analysis and Design Language (AADL), and Rapide

Semi-formal:

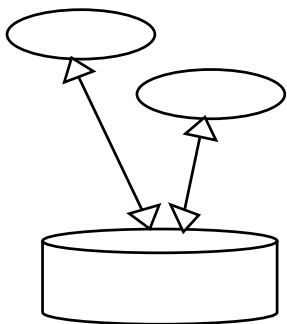
- UML
 - not always a straightforward mapping
 - major improvements in UML 2.0

Notations for C&C Views: UML 2.0



Key:
UML 2.0

Shared-Data View

**Elements:**

- component types: data stores and accessors
- connector types: data reading and writing

Relations: attachment

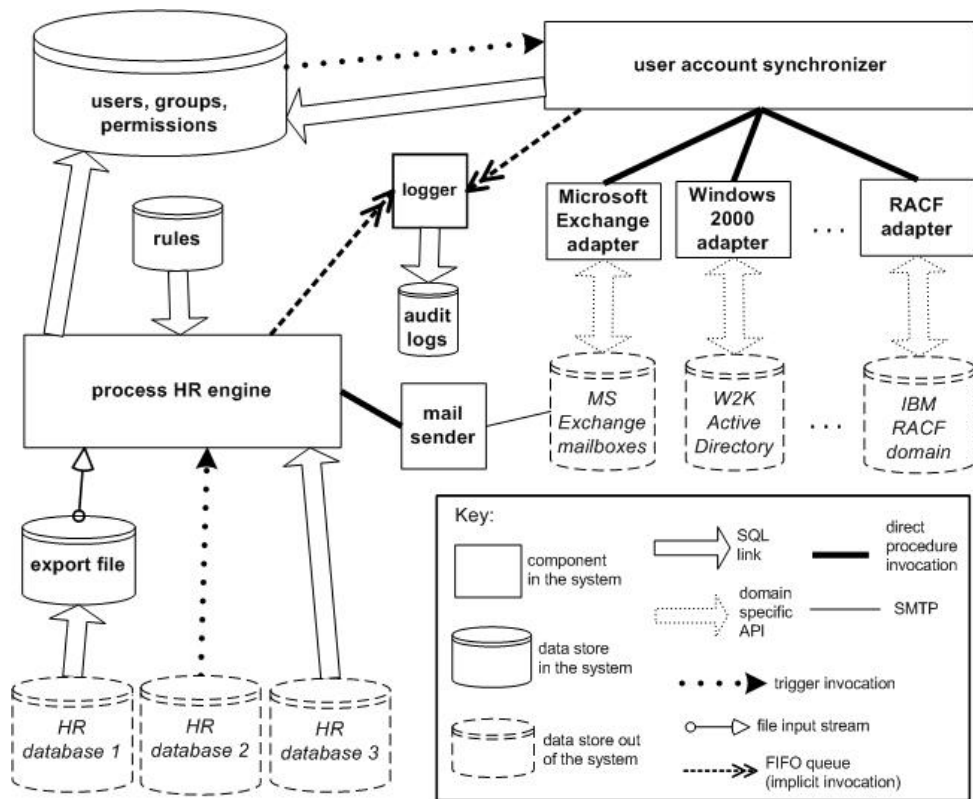
Properties: can include type of data, data-related performance properties, and data distribution

Topology: The data store is attached to the data accessors via connectors.

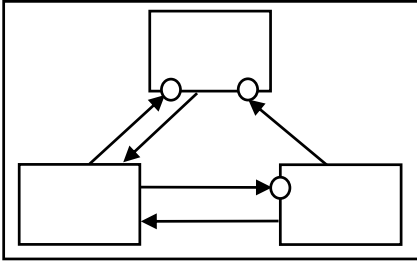
What it's for: when there are multiple accessors of persistent data

Variations: Data store is a blackboard that announces updates to interested subscribers.

Shared-Data View Primary Presentation: Informal Notation



Service-Oriented Architecture (SOA) View – 1



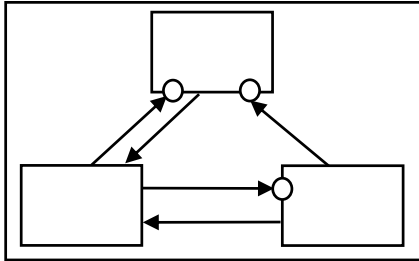
Elements:

- component types: service users, service providers, ESB, registry of services
- connector types: service calls (e.g., Simple Object Access Protocol [SOAP] messages)

Relations: attachment of a service call to a service endpoint

Properties: Security (e.g., authorization), cost, reliability, performance, and other qualities of service can be associated with services.

Service-Oriented Architecture (SOA) View – 2



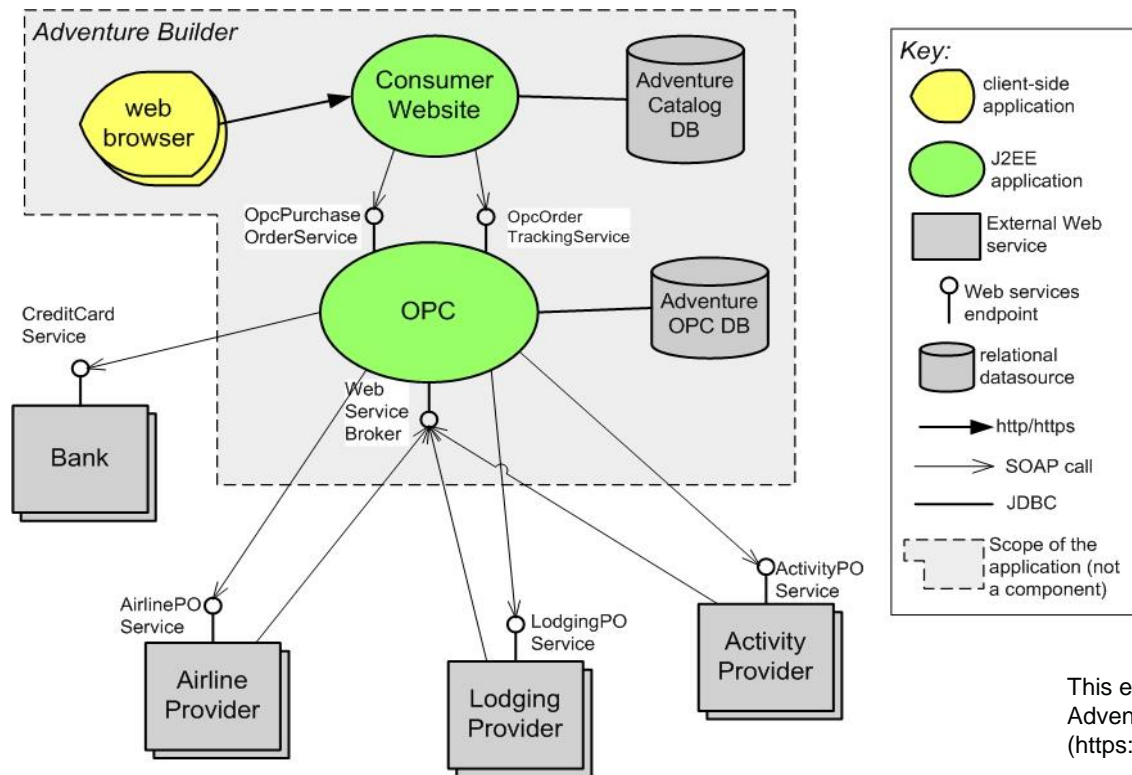
Topology:

- Service users are connected to service providers.
- Special components may intermediate the interaction between service users and service providers:
 - Registry of services: naming and location of services
 - ESB: routing, data and format transformation, technology adapters
- A service user may also be a service provider.

What it's for: Reasoning about

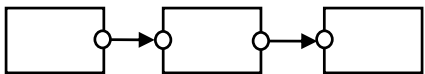
- how best to distribute applications across a system
- how well interoperability of components developed in different languages and platforms will be achieved
- how external components and legacy systems will be integrated

SOA View Primary Presentation: Informal Notation



This example was adapted from the Java Adventure Builder Reference application (<https://adventurebuilder.dev.java.net/>).

Pipe-and-Filter View



Elements:

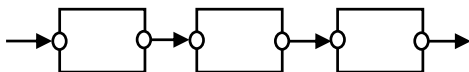
- component type: filter, which transforms data
- connector type: pipe, a unidirectional data conduit that preserves the order and value of data

Relations: attachment of pipes to filters

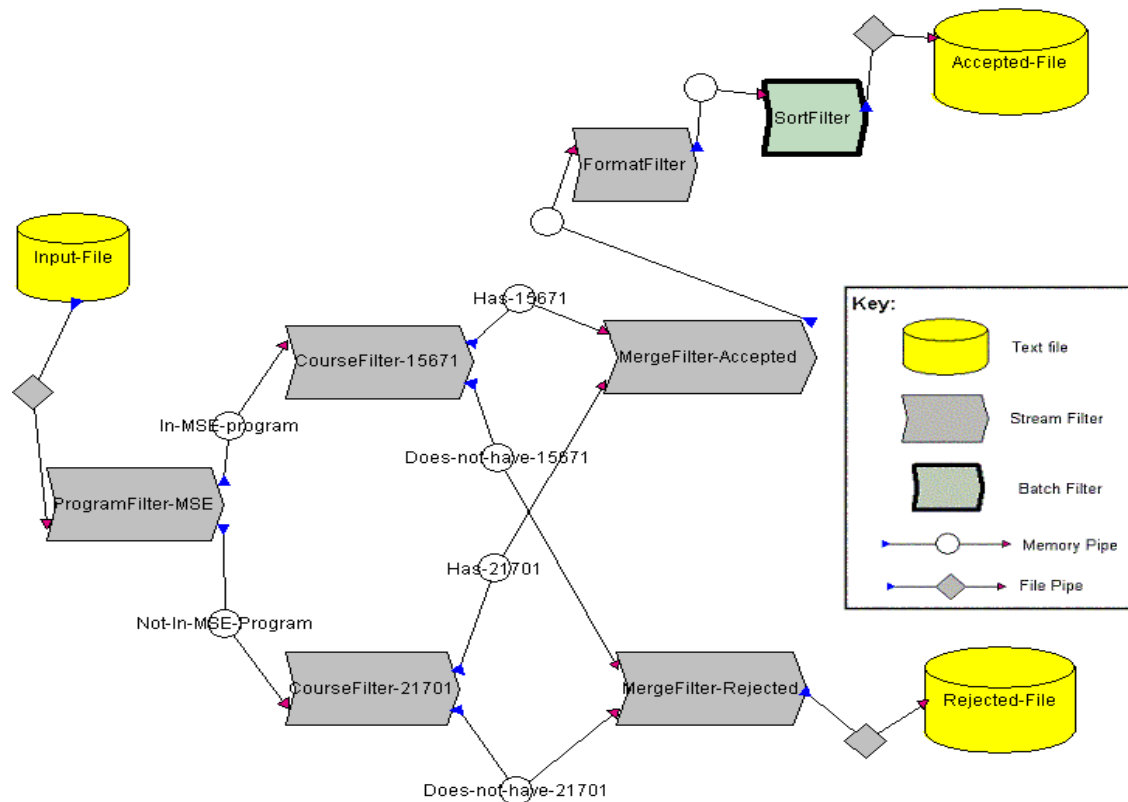
Topology: Pipes connect filters. Further specializations of the style may prohibit loops or branching.

What it's for:

- systems in which data is transformed serially
- supporting functional composition data analysis



Pipe-and-Filter View Primary Presentation: Informal Notation



Notations for Allocation Views' Primary Presentations

Informal:

- box-and-line diagrams
- tables
- snapshots of tool interfaces that manage the allocations

Semi-formal:

- UML can show various kinds of allocation, such as software to hardware or software to a file container.

Deployment View – 1

Elements:

- software element—usually processes from C&C views
- environmental element—computing hardware

Relations:

- “allocated to”—physical elements on which software resides
- “migrates to,” “copy migrates to,” and/or “execution migrates to” with dynamic allocation

Deployment View – 2

Properties:

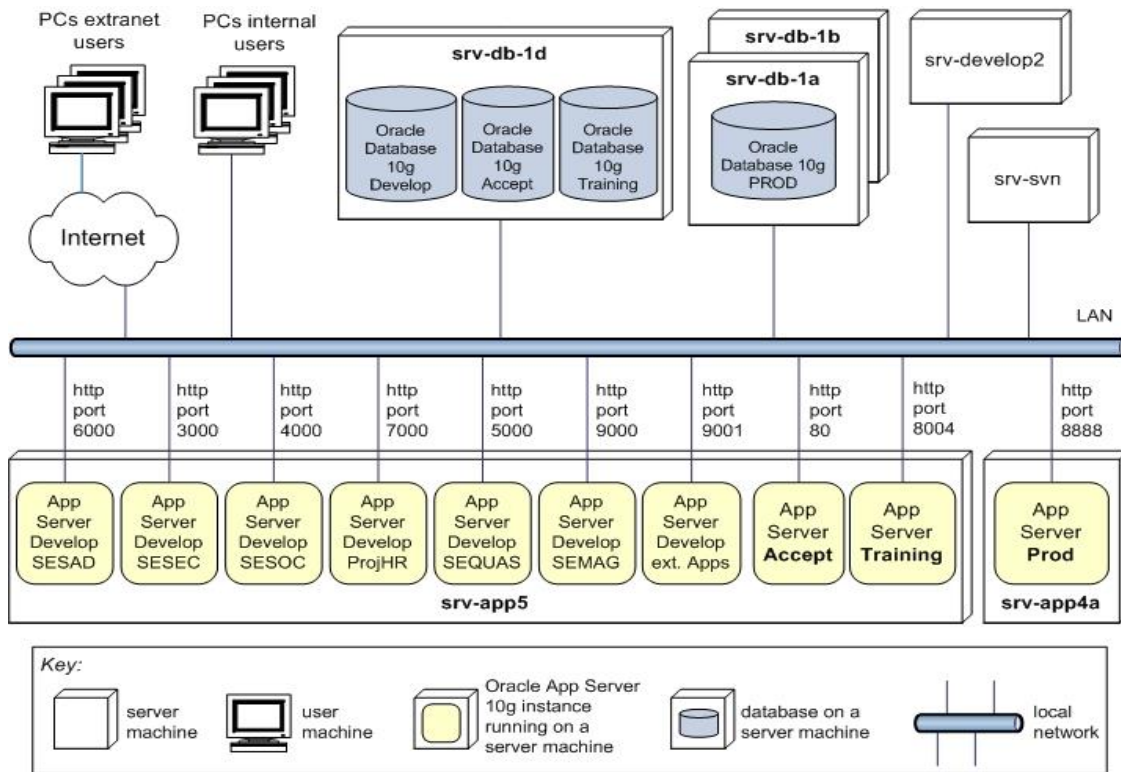
- requires property for software elements
 - significant features required from hardware
- provides property of environment elements
 - significant features provided by hardware

Topology: unrestricted

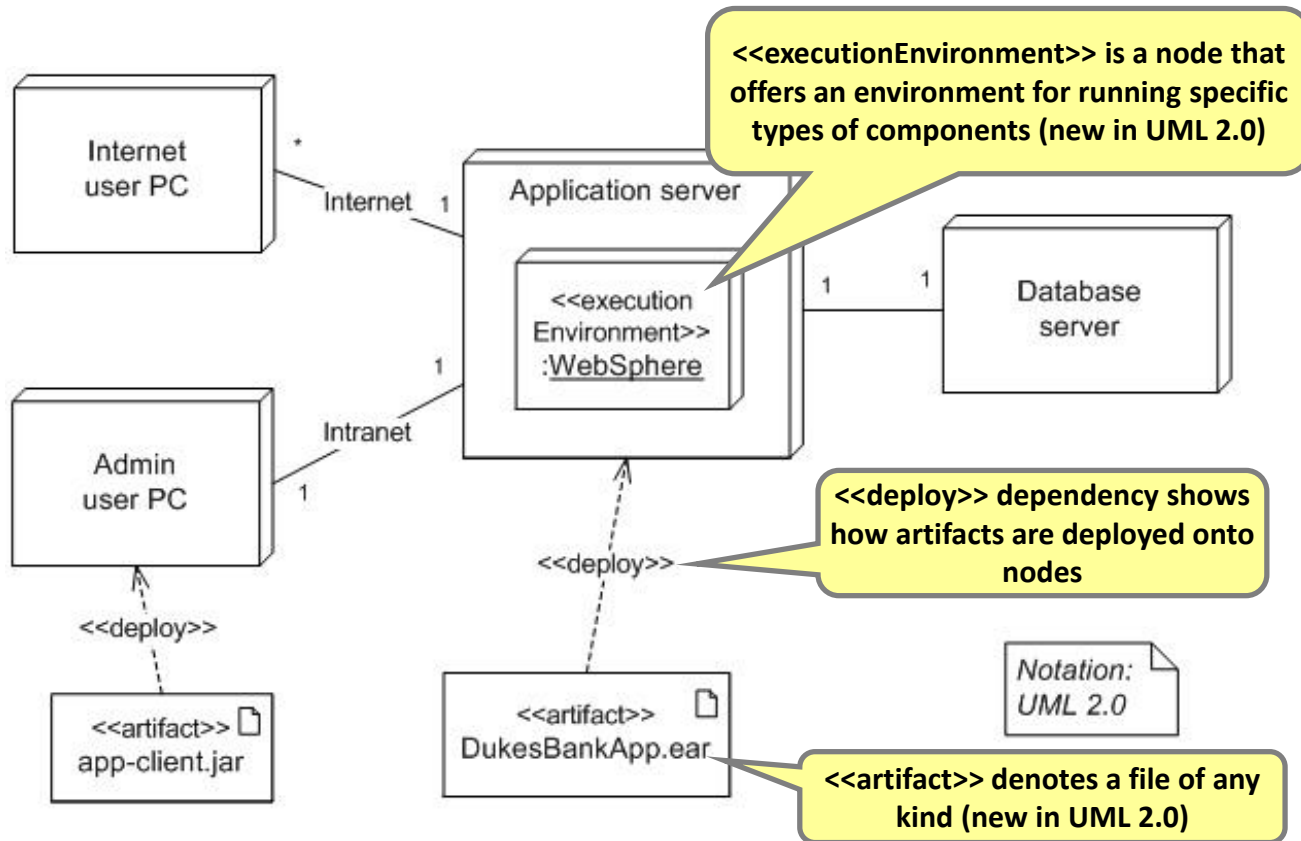
What it's for: analysis of

- performance
- bandwidth utilization
- availability
- security
- purchasing options for hardware

Deployment View Primary Presentation: Informal Notation



Deployment View Primary Presentation: UML



Implementation View – 1

Elements:

- software element—a module or component
- environmental element—configuration item; for example, a file or directory

Relations:

- containment—A configuration item is contained in another.
- “allocated to”—A module is allocated to a configuration item.

Implementation View – 2

Properties:

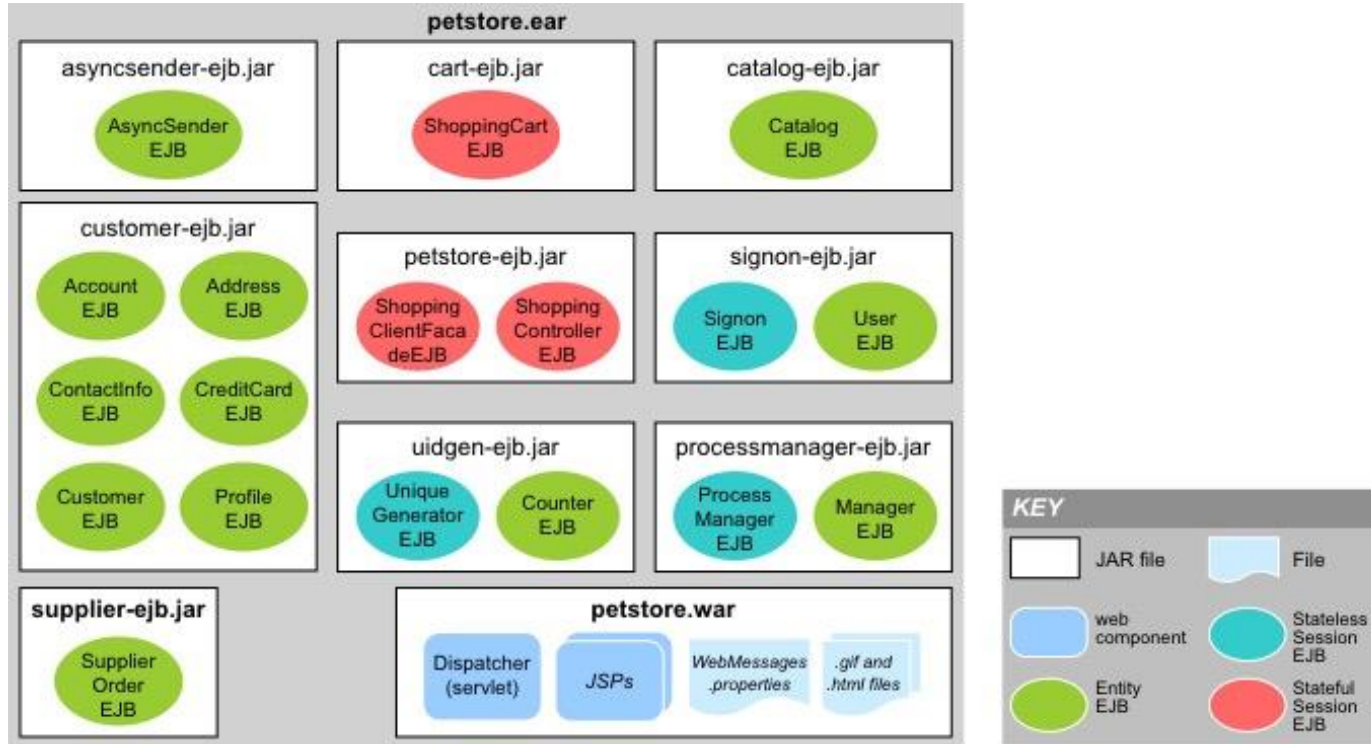
- *requires* property for software element—usually requirements on the development or production environments; for example, Java
- *provides* property of environment elements—characteristics provided by development or production environments

Topology: configuration items are hierarchical, following “is contained in”

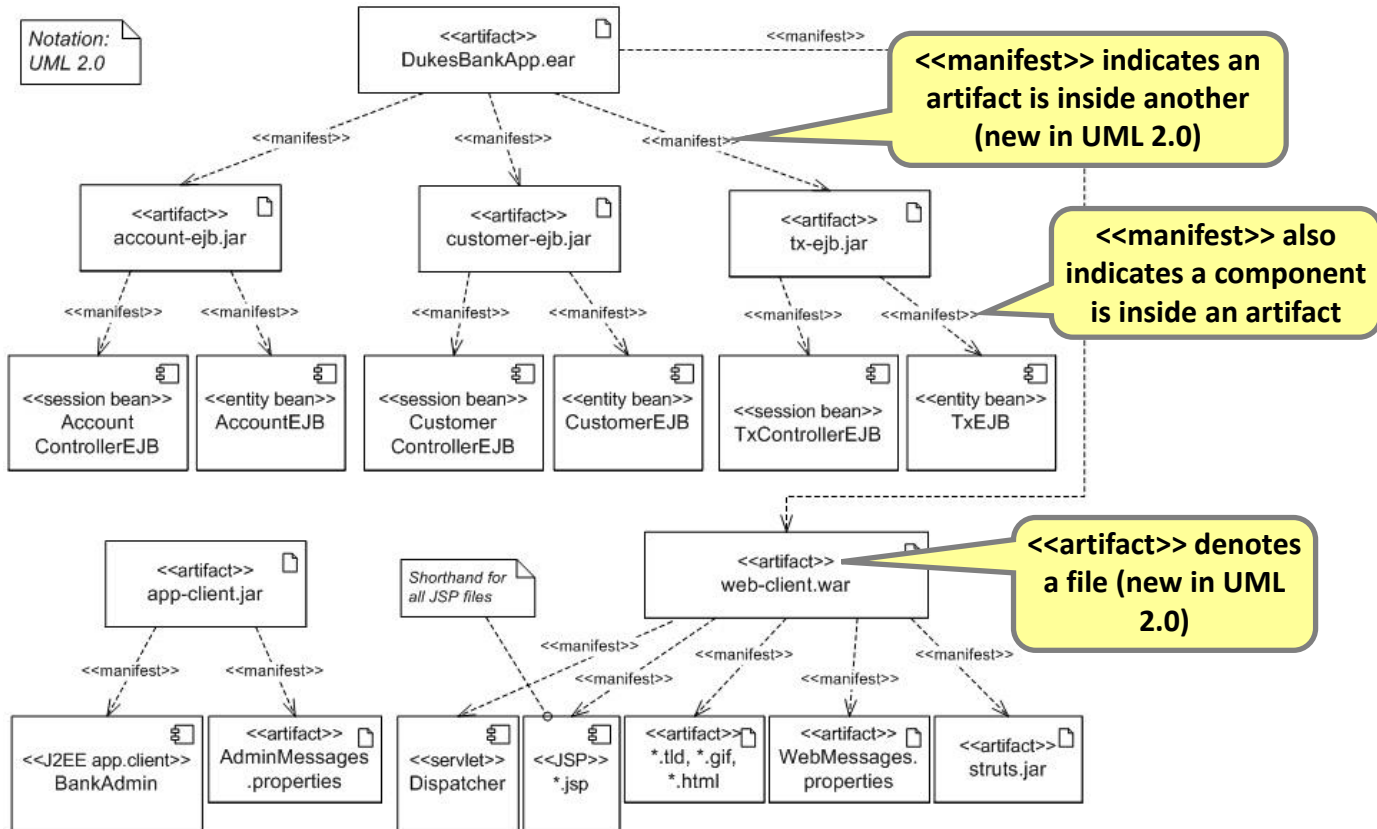
What it’s for:

- managing and maintaining files that correspond to software elements
- analyzing purchasing options for development or production environments
- defining deployment and production procedures

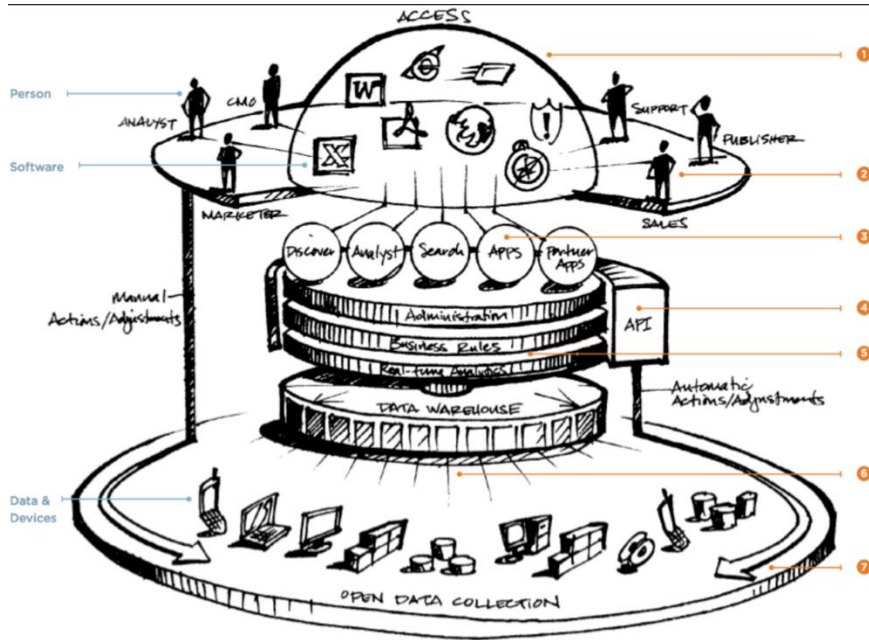
Implementation View Primary Presentation: Informal Notation



Implementation View Primary Presentation: UML



And, finally, *Marketecture*



Marketecture is a one-page, typically informal description of the architecture.

It shows major components and their relationships.

It facilitates discussion and provides a starting point for deeper analysis.

Choosing Views to Document



Remember that

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

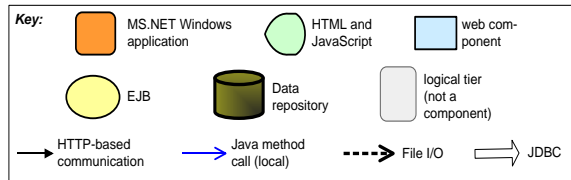
So we need to choose the views that

- are relevant to the stakeholders and their intended use for the views
- reflect structures inherent in the system

A Method for Choosing the Views

1. Build a stakeholder/view table.
 - a. ROWS (x) enumerate the stakeholders.
 - b. COLUMNS (y) enumerate the set of styles that could apply to the architecture being documented—the possible views.
 - c. Check the cell where x and y meet if stakeholder x needs view y in order to do his/her job.
2. Combine views appropriately to reduce their number.
3. Prioritize and stage the documentation, based on need.

Always Use a Notation Key



Notation:
UML 2.0

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

Precisely defined notations and languages help avoid ambiguity.

If your documentation uses a graphical language, always include a key that either

- points to the language's formal definition or
- gives the meaning of each symbol. (Don't forget the lines or arrows!)

If color or position is significant, indicate how.

Exercise

Consider the architecture sketch(es) that you created in the BizCo design exercise.

Improve your architecture documentation. Create one or more views, using the principles and examples in this module. Consider both structure and behavior.

Which views?

Choose your most important scenario, and document views that allow you to precisely reason about that scenario.

Module Summary – 1

Primary uses of architecture documentation include construction, education, and analysis.

Documenting an architecture is a matter of documenting its views and then documenting information that applies to more than one view.

A view is a representation of a structure. Choose the views to document that best represent the structures inherent in the architecture and that best serve your stakeholders' needs.

Use a standard organization.

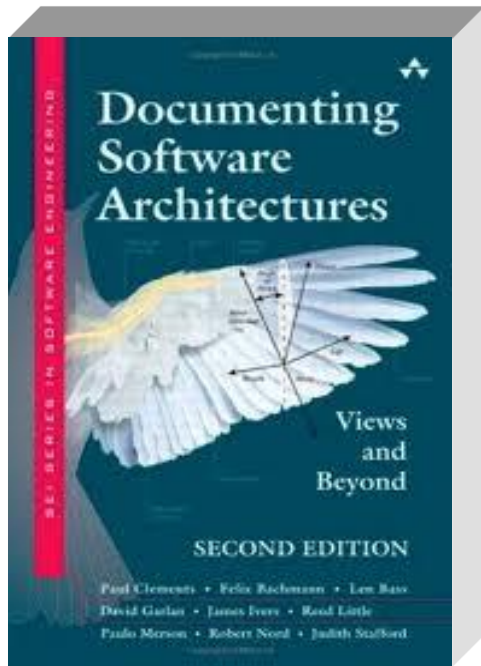
Make sure your diagrams always have a notation key.

Module Summary – 2

Diagrams are not enough! The elements in the diagrams must be explained.

Views by themselves are not enough! The views must be augmented with an explanation of the documentation organization and the system as a whole.

For More Information About Architecture Documentation



- Take the SEI Documenting Software Architectures two-day course.
- Read *Documenting Software Architectures: Views and Beyond, 2nd ed.*, by P. Clements, F. Bachmann, L. Bass, D. Garlan, J. Ivers, R. Little, R. Nord, & J. Stafford; published by Addison-Wesley.
- See a Microsoft Word template for a software architecture document based on the Views & Beyond / ISO 42010 approach, available at http://www.sei.cmu.edu/architecture/arch_doc.html



Software Architecture: Principles and Practices

ARCHITECTURE EVALUATION

Module Objectives



This module will familiarize participants with

- why we evaluate architectures
- when it's appropriate to evaluate architectures
- the benefits of evaluating architectures
- the cost of evaluating architectures
- techniques for evaluating architectures
- the SEI Architecture Tradeoff Analysis Method[®] (ATAM[®])

[®] Architecture Tradeoff Analysis Method and ATAM are registered in the U.S. Patent and Trademark Office by Carnegie Mellon University.

Outline



► Architecture Evaluation

- Why, When, Benefits, and Costs
- Evaluation Techniques

Architecture Tradeoff Analysis Method (ATAM)

Summary

Outline

Architecture Evaluation

- ▶ • **Why, When, Benefits, and Costs**
- Evaluation Techniques

Architecture Tradeoff Analysis Method (ATAM)

Summary

Why Evaluate an Architecture?

Because so much is riding on it!

- An unsuitable architecture will precipitate disaster.
- Architecture determines the structure of the project.

Because we can!

- Repeatable, structured methods offer a low-cost risk mitigation capability that can be employed early in the development lifecycle.
- Making sure an architecture is the right one simply makes good sense.

Architecture evaluation should be a standard part of every development methodology.

When to Evaluate an Architecture – 1

Analyzing for system quality attributes early in the lifecycle allows for a comparison of architectural options.

When building a system

- Architecture is the earliest artifact in which tradeoffs are visible and about which the most far-reaching design decisions are made.
- Evaluation should be done when deciding on an architecture.
- The reality is that evaluation is often done during damage control, later in the project.

When to Evaluate an Architecture – 2

When acquiring a system

- Architecture evaluation is particularly useful if the system will have a long lifetime within an organization.
- Evaluation provides a mechanism for understanding how the system will evolve.
- Evaluation provides early insight into system capabilities and quality attributes.

When putting a system through major changes

- Evaluation can provide insight into how well the architecture will support the changes.

Architecture Evaluation Benefits – 1



The representatives of four major companies—Millennium Services, Lucent Technologies, AT&T Labs, and Avaya Labs—had this to say:

Since 1988, we've conducted more than 700 project reviews. Starting with AT&T, architecture reviews have grown across all our companies. We estimate that projects of 100,000 non-commentary source lines of code have saved an average of U.S. \$1 million each by identifying and resolving problems early.¹

¹ Maranzano, J.F.; Rozsypal, S.A.; Zimmerman, G.H.; Warnken, G.W.; Wirth, P.E.; & Weiss, D.M. "Architecture Reviews: Practice and Experience." [IEEE Software](#) 22, 2 (2005): 34–43.

Architecture Evaluation Benefits – 2



Forced preparation for evaluations

- Documentation/specifications must be provided; hence they must exist.
- Some evaluations use standard questions, and the architect can make sure ahead of time that the architecture scores well.
- The criteria for evaluations is made explicit by prioritizing requirements and quality goals.

Early detection of problems

- Problems detected early are much less expensive to repair.

Architecture Evaluation Benefits – 3

Improved architectures

- Pre-positioning for evaluations means that requirements and quality goals are prioritized earlier in the lifecycle.
- Problems eliminated result in higher quality architectures.

Validation of requirements

- Architectures are evaluated against functional and quality attribute requirements.
- Often, the requirements are not clear.
- Evaluations uncover requirement ambiguities and gaps.
- Evaluations can be the basis for renegotiating requirements that turn out to be difficult to achieve.

Architecture Evaluation Benefits – 4



Improved stakeholder communication

- Evaluations usually put all the stakeholders in the same room for the first time.
- Evaluations often put stakeholders in the same room as the architect for the first time.
- Communication channels remain open even after the evaluation. Having stakeholders present
 - uncovers conflicts and tradeoffs
 - provides a forum for the negotiated resolution of problems
 - promotes buy-in of the architecture
 - builds a community of support that is a resource for the architect

Architecture Evaluation Costs – 1

Boeing¹

- An average of 91 person days per ATAM evaluation. On average, the breakdown is
 - 4 evaluators, 70 hours each
 - 1 facilitator, 120 hours
 - architecture team, 90 hours
 - 15 stakeholders, 16 hours each

Bosch²

- An average of 49 person days per ATAM evaluation. This does not include effort for the evaluators.

1 O'Connell, Don. "Architecture Analysis – Boeing's Experiences Using the SEI ATAM and QAW Processes." Second SEI Software Architecture Technology User Network Conference, Pittsburgh, PA, April 2006. www.sei.cmu.edu/library/abstracts/presentations/donoconnellarchanalysisoverview.cfm

2 Ferber, Stefan. "Architecture Reviews @ Bosch" (Keynote). First SEI Software Architecture Technology User Network Conference, Pittsburgh, PA, April 2005. www.sei.cmu.edu/library/assets/Bosch_Architecture_Reviews_ferber.pdf

Architecture Evaluation Costs – 2

Organizational overhead of establishing a corporate evaluation unit

- management overhead
- communication expenses
- staffing the unit
- relocating personnel to a central location
- training

Personnel costs include having senior designers conduct evaluations instead of designing

- loss of productivity (due to the reassignment of senior designers)
- time spent training staff in evaluation techniques

Outline



Architecture Evaluation

- Why, When, Benefits, and Costs



- **Evaluation Techniques**

Architecture Tradeoff Analysis Method (ATAM)

Summary

Evaluation Techniques

There are a variety of techniques for performing architecture evaluations, each having a different cost and providing different information.

1. questioning techniques

- are applied to evaluate an architecture for any given reason

2. measuring techniques

- are applied to answer questions about specific quality attributes

Questioning Techniques – 1

Scenario-based techniques

- Scenarios describe a specific interaction between stakeholders and a system.
- The architect explains how the architecture supports each scenario posed by the evaluators.
- The ATAM is the best-known scenario-based evaluation method.

Questionnaire-based techniques

- Evaluation consists of the architect answering a prepared list of questions.
- Some questions apply to all architectures (especially those in a single domain).
- Some questions ask about the details of an architecture.
- Some ask about process or development:
 - “Is there a single architect?”
 - “How do you ensure conformance?”

Questioning Techniques – 2

Checklist-based techniques

- Checklists contain detailed sets of yes/no questions.
- Checklists result from evaluating many architectures in a domain and “maturing” the scenarios or questions into a standard check-off procedure.
- Checklists often focus on particular quality attributes.
- Examples
 - “Have you verified that the peak load is within acceptable limits?”
 - “Have you performed a fault-tree analysis for software safety?”

Checklists and questionnaires reflect more maturity or experience with a class of similar systems and their architectures. Scenarios are specially developed for each system (but can be saved and used to “seed” future evaluations).

Measuring Techniques – 1

Metrics are quantitative interpretations of observable measures.

- Complexity metrics suggest areas where modifiability is poor or errors are likely to occur.
- Performance metrics help identify the presence of bottlenecks.

Metric-based evaluations tend to focus on

- choosing an appropriate set of metrics
- the results of applying the metrics
- the assumptions underlying the interpretation of the metrics (e.g., assumed event distribution)

Measuring Techniques – 2

Simulations, prototypes, experiments

- involve building domain-specific or system-specific models of an architecture
 - for example, a performance model or a queuing model
 - high-fidelity models fairly expensive to create
 - often exist as part of development anyway—in which case, they can be leveraged to evaluate the architecture
- may resolve issues raised by a questioning technique
 - for example, “What evidence do you have that performance is adequate?”

Analysis at Different Stages of the Lifecycle

What kinds of analysis can you do, at what time in the lifecycle, and with what level of confidence?

Lifecycle Stage	Form of Analysis	Cost	Confidence
Requirements	Experience-based analogy	Low	
Requirements	Back-of-the-envelope	Low	
Architecture	Thought experiment	Low	
Architecture	Checklist	Low	
Architecture	Analytic model	Low-Medium	
Architecture	Simulation	Medium	
Architecture	Prototype	Medium	
Implementation	Experiment	Medium-High	
Fielded system	Instrumentation	Medium-High	

Typical Outputs from Evaluations

Set of ranked issues, risks, or problem areas that

- have supporting data
- are contained in a formal report
- are used as feedback to the project

Enhanced system documentation

Set of scenarios, questions, or checklists for future use

Identification of potentially reusable components

Outline



Architecture Evaluation

- Why, When, Benefits, and Costs
- Evaluation Techniques

▶ Architecture Tradeoff Analysis Method (ATAM) Summary

The ATAM

The SEI developed the Architecture Tradeoff Analysis Method (ATAM) and has applied it to architectures for systems of wide-ranging sizes and domains.

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

The ATAM brings together three groups in an evaluation:

1. a trained evaluation team
2. an architecture's "decision makers" (architect, senior designers, project managers, customers)
3. representatives of the architecture's stakeholders

Purpose of the ATAM – 1

The ATAM is a method that helps stakeholders ask the right questions to discover potentially problematic architectural decisions.

Discovered risks can then be made the focus of mitigation activities such as further design, further analysis, and prototyping.

Surfaced tradeoffs can be explicitly identified and documented.

Purpose of the ATAM – 2

The purpose is NOT to provide precise analyses. An ATAM evaluation will not generate a queuing model of performance, or a Markov model of availability, or a coupling model of modifiability, ...

The purpose IS to discover any risks created by architectural decisions.

When to Use the ATAM

The ATAM can be used throughout the lifecycle when there is a software architecture to evaluate.

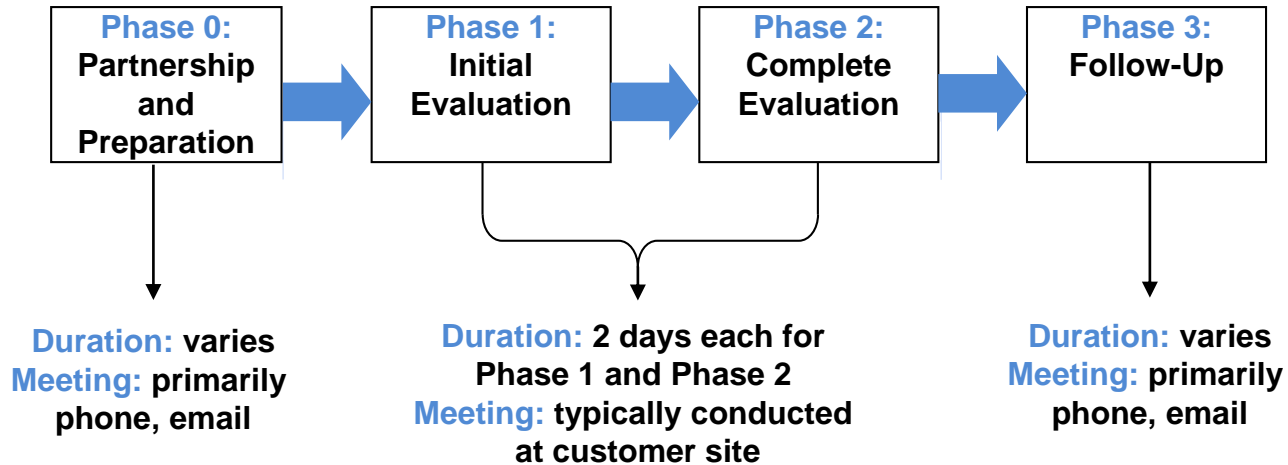
The ATAM can be used

- after an architecture has been specified but there is little or no code
- to evaluate architectural alternatives
- to evaluate the architecture of an existing system

An ATAM evaluation is inappropriate if the software architecture of the system has not been created yet.

ATAM Phases

ATAM evaluations are conducted in four phases.



ATAM Phase 0

Phase 0 precedes the technical part of the evaluation:

- The customer and a subset of the evaluation team discuss their understanding of the method and the system whose architecture is to be evaluated.
- An agreement to perform the evaluation is worked out.
- A core evaluation team is fielded.

ATAM Phase 1

Phase 1 involves a small group of predominantly technically oriented stakeholders.

Phase 1 is

- architecture-centric
- focused on eliciting detailed architectural information and analyzing it
- a top-down analysis

ATAM Phase 1 Steps

1. Present the ATAM.
2. Present business drivers.
3. Present architecture.
4. Identify architectural approaches.
5. Generate quality attribute utility tree.
6. Analyze architectural approaches.
7. Brainstorm and prioritize scenarios.
8. Analyze architectural approaches.
9. Present results.

Phase 1

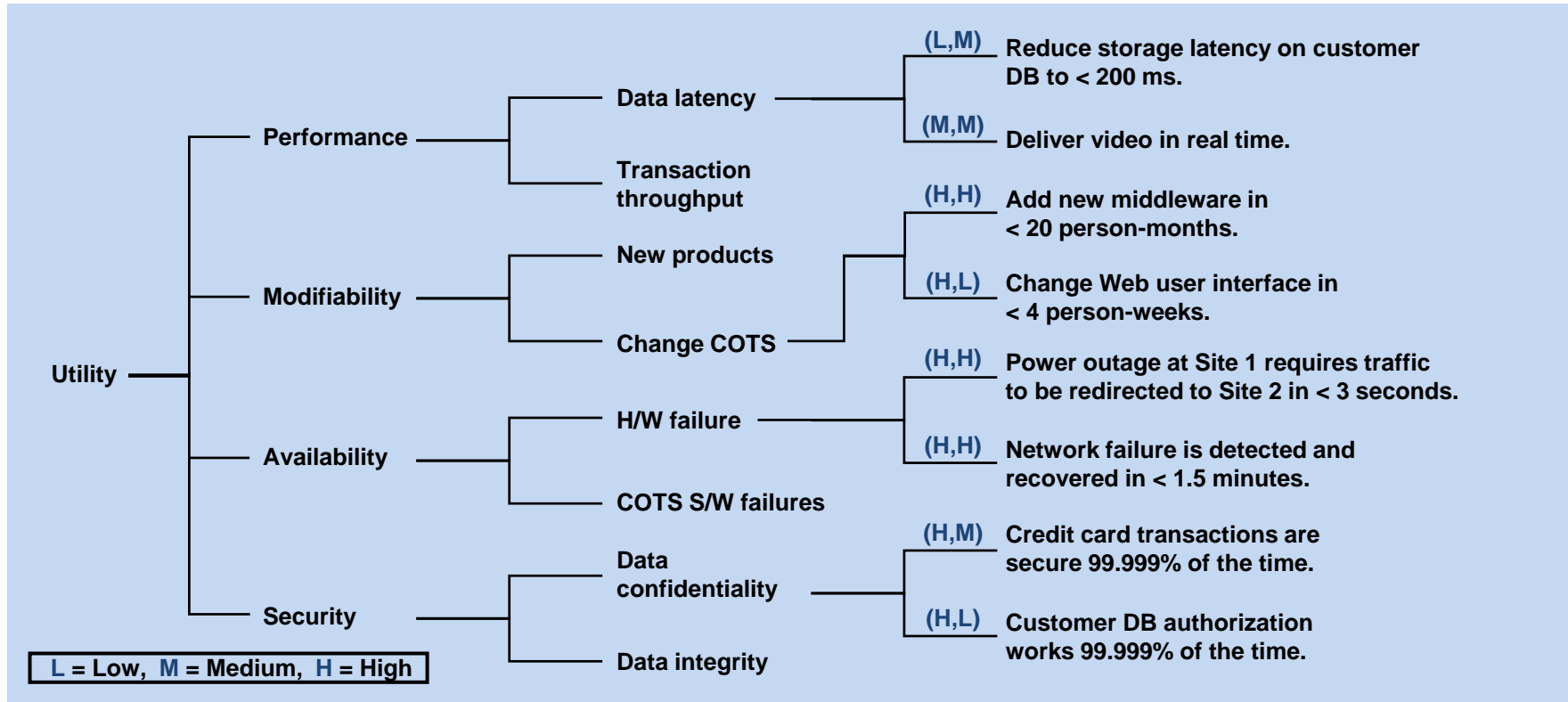
What Are Quality Attribute Utility Trees?

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 – 1

For design purposes, we use six-part scenarios as described earlier:

1. **source** – an entity that generates a stimulus
2. **stimulus** – a condition that affects the system
3. **artifact(s)** – the part of the system that was stimulated by the stimulus
4. **environment** – the condition under which the stimulus occurred
5. **response** – the activity that results because of the stimulus
6. **response measure** – the measure by which the system's response will be evaluated

How Scenarios Are Used – 2

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.

Stimuli, Environment, Responses

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.

Scenario Analysis Outputs

As each scenario is analyzed against the architecture, the evaluation team identifies risks, non-risks, sensitivity points, and tradeoffs.

- A risk is a potentially problematic architectural decision.
- Non-risks are good architectural decisions that are frequently implicit in the architecture.
- A sensitivity point is a place in the architecture that significantly affects whether a particular quality attribute response is achieved.
- A tradeoff is a property that affects more than one attribute and is a sensitivity point for more than one attribute.

Risks and Tradeoffs

Risk example

- “Rules for writing business logic modules in the second tier of your three-tier architecture are not articulated clearly. This could result in the replication of functionality, thereby compromising the modifiability of the third tier.”

Tradeoff example

- “Increasing the level of encryption will significantly increase security but decrease performance.”

Sensitivity Points and Non-Risks

Sensitivity point example

- “The response time to system events is sensitive to the number of processes running on the main processor.”

Non-risk example

- “Assuming message-arrival rates of no more than once per second and a processing time of less than 30 ms, the architecture should meet the 1-second soft deadline requirement.”

ATAM Phase 2

Phase 2 involves a larger group of stakeholders.

Phase 2 is

- stakeholder-centric
- focused on eliciting diverse stakeholders' points of view and verifying the results of Phase 1
- bottom-up analysis

ATAM Phase 2 Steps

1. Present the ATAM.
2. Present business drivers.
3. Present architecture.
4. Identify architectural approaches.
5. Generate quality attribute utility tree.
6. Analyze architectural approaches.
7. Brainstorm and prioritize scenarios.
8. Analyze architectural approaches.
9. Present results.

Phase 1

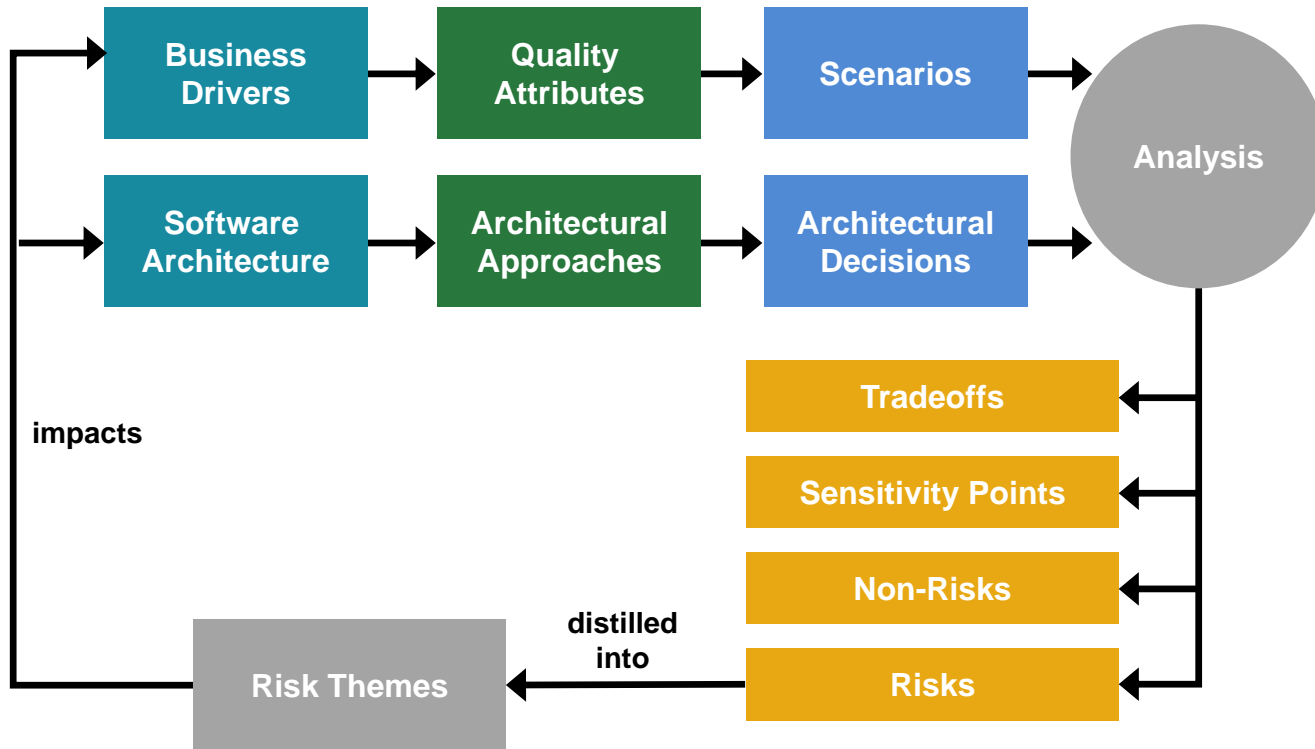
Phase 1

Do this

ATAM Phase 3

Phase 3 primarily involves producing a final report for the customer as well as assessing the quality of the evaluation and the ATAM materials.

Conceptual Flow of the ATAM



Group Exercise

Work together in *pairs* of groups, where one group is the evaluation team and one group is the architecture team.

Consider the following scenario:

BizCo server at a branch location stops responding. The failure is detected and the system is restored to normal operation within 90 seconds.

Complete the analysis template provided on the following slides.

Scenario Analysis Template: Part 1

ATAM: Scenario Analysis		
Scenario		
Business Goal(s)		
Attribute		
Attribute Concern		
Scenario Refinement	Stimulus	
	Stimulus Source	
	Environment	
	Artifact	
	Response	
	Response Measure	

Scenario Analysis Template: Part 2

Architectural Decisions and Reasoning	
Risks	1.
Sensitivities	1.
Tradeoffs	1.
Non-Risks	1.
Other Issues	1.

Outline



Architecture Evaluation

- Why, When, Benefits, and Costs
- Evaluation Techniques

Architecture Tradeoff Analysis Method (ATAM)

▶ Summary

Module Summary

Architecture evaluation

- why – to understand and analyze design tradeoffs
- when – throughout the lifecycle
- benefits – overall cost reduction, better documentation/specifications, better understanding and prioritization of requirements and quality goals, early detection of problems, improved architectures, validation of requirements

Evaluations are carried out via questioning and measuring techniques.

The ATAM is the most widely used method for evaluating architectures with respect to multiple quality attributes.

For More Information About Architecture Evaluation

Take the SEI Software Architecture Design and Analysis two-day course.

Read *Evaluating Software Architectures: Methods and Case Studies*, written by P. Clements, R. Kazman, & M. Klein and published by Addison-Wesley.



Software Architecture: Principles and Practices

APPLYING ARCHITECTURE PRACTICES ON AGILE PROJECTS

Module Introduction

We have spent a lot of time in this course discussing the importance of quality attributes and design.

We often get the question: How do these practices mesh with Agile development methodology?

The short answer is that architecture practices matter even more in an Agile context.

As a quick introduction to Agile, we will cover two areas:

- Agile philosophy
- incremental lifecycle

Philosophy: The Agile Manifesto

A common misconception is that Agile philosophy suggests writing code is the activity that matters, but that is not what the manifesto says...



Twelve Agile Principles – 1

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

<http://agilemanifesto.org>

Twelve Agile Principles – 2

7. Working software is the primary measure of progress.
8. Agile processes promote sustainable development. The sponsors, developers, and users should be able to maintain a constant pace indefinitely.
9. Continuous attention to technical excellence and good design enhances agility.
10. Simplicity—the art of maximizing the amount of work not done—is essential.
11. The best architectures, requirements, and designs emerge from self-organizing teams.
12. At regular intervals, the team reflects on how to become more effective, then tunes and adjusts its behavior accordingly.

<http://agilemanifesto.org>

Agile Lifecycle and Quality Attribute Focus

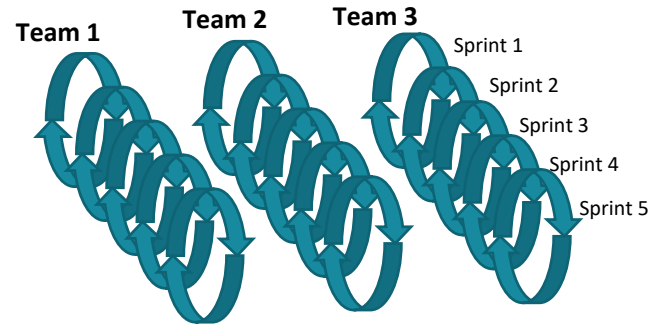
Another key aspect of Agile is the incremental life cycle:

- Sprints are increments of functionality
- Gold standard duration is two weeks

It is not uncommon to see multiple Agile teams working in parallel at this fast pace.

Software must be modular, flexible, and reliable.

For these reasons, a focus on quality attributes and strong evaluation practices are very important!



How Much Architecture?

Two classes of activities add time to the project schedule:

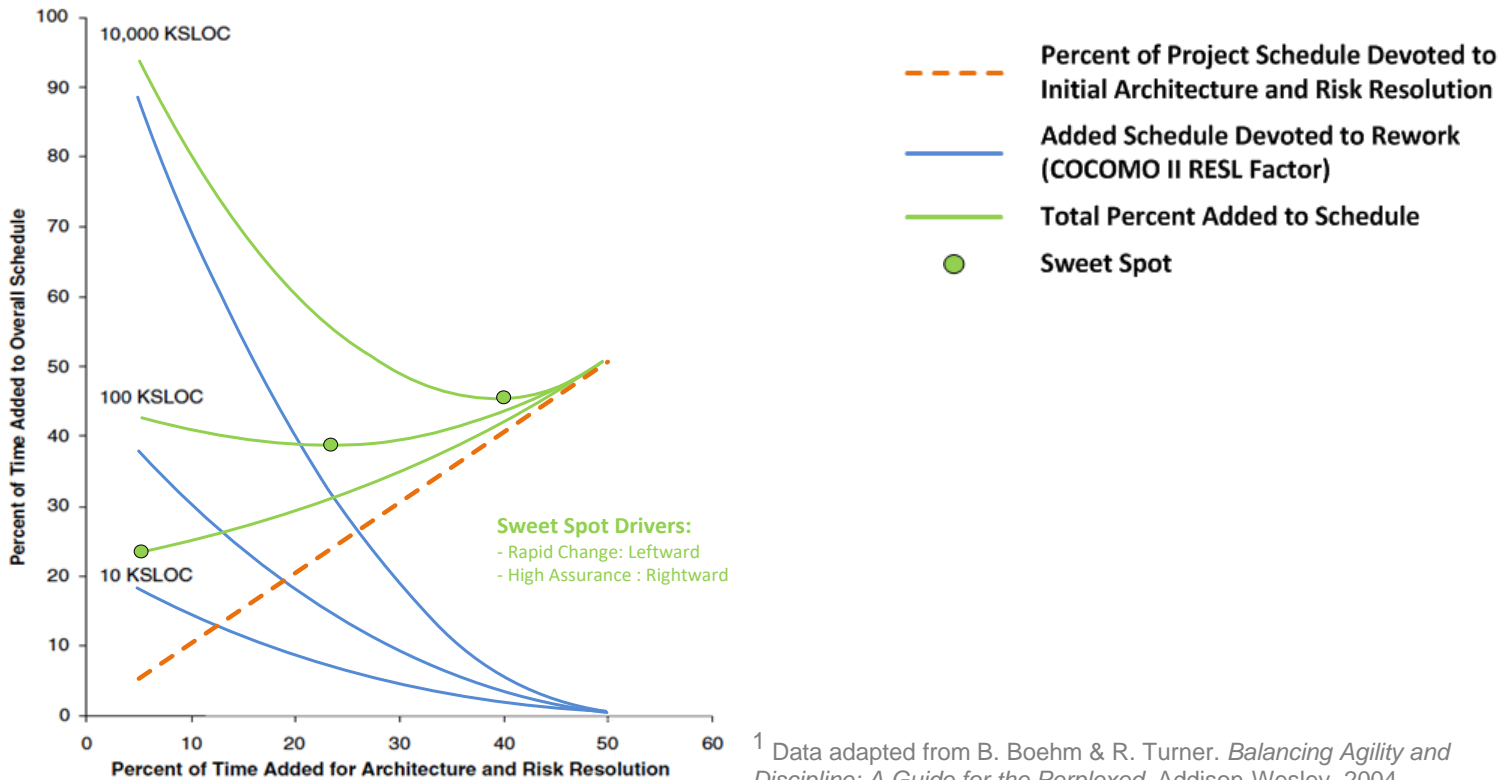
1. up-front design work on the architecture and up-front risk identification, planning, and resolution work
2. rework due to fixing defects and addressing modification requests

Intuitively, these two trade off against each other.

Boehm and Turner plotted these two values against each other for three hypothetical projects:

- one project of 10 KSLOC
- one project of 100 KSLOC
- one project of 10,000 KSLOC

How Much Architecture?



¹ Data adapted from B. Boehm & R. Turner. *Balancing Agility and Discipline: A Guide for the Perplexed*. Addison-Wesley, 2004.

How Much Architecture?

These lines show that there is a sweet spot for each project.

- For the 10-KSLOC project, the sweet spot is at the far left. Devoting much time to up-front work is a waste for a small project.
- For the 100-KSLOC project, the sweet spot is around 20 percent of the project schedule.
- For the 10,000-KSLOC project, the sweet spot is around 40 percent of the project schedule.

A project with millions of lines of code is enormously complex.

It is hard to imagine how Agile practices alone can cope with this complexity if there is no architecture to guide and organize the effort.

¹ B. Boehm & R. Turner. *Balancing Agility and Discipline: A Guide for the Perplexed*. Addison-Wesley, 2004.

Scaled Agile Framework-1

Agile processes were initially employed on small- to medium-sized projects. Today, we see Agile being used on large-scale projects.

This requires a blend of Agile and architecture.

A widely used framework that blends Agile and architecture is the Scaled Agile Framework (SAFe).

The purpose of this section is **NOT to teach SAFe.**

Rather, the purpose is to illustrate how practices in this course can be applied using SAFe as a backdrop.

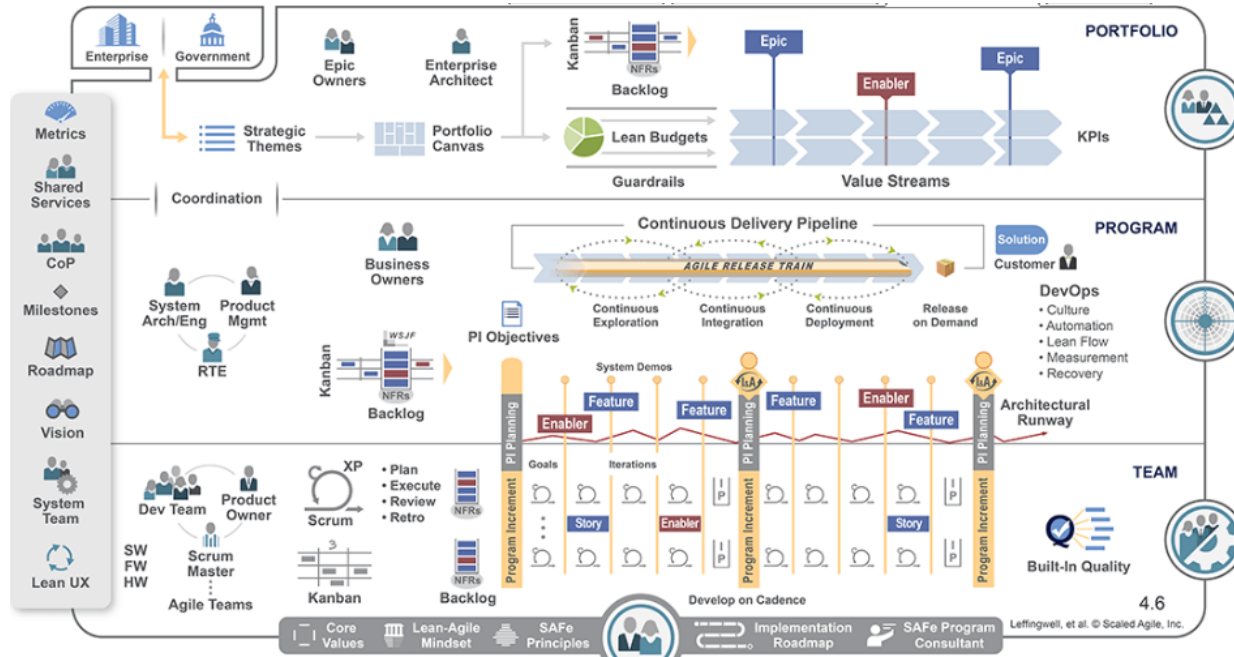
Scaled Agile Framework-2

Scaled Agile Framework (SAFe) advocates "Agile Architecture":¹ a set of principles aimed at finding the "sweet spot"

- supporting evolution of the architecture concurrent with the implementation
- avoiding the use of BDUF
- ensuring that the system “always runs,” supporting continuous delivery of value
- balancing emergent design and intentional architecture

¹ <http://www.scaledagileframework.com/Agile-Architecture/>

Scaled Agile Framework “Big Picture” Diagram



One of the biggest contributions from SAFe is this diagram.

It contains a lot of information!

Source: <https://www.scaledagileframework.com> (used with permission from Scaled Agile Academy)

Why Use SAFe as a Backdrop?

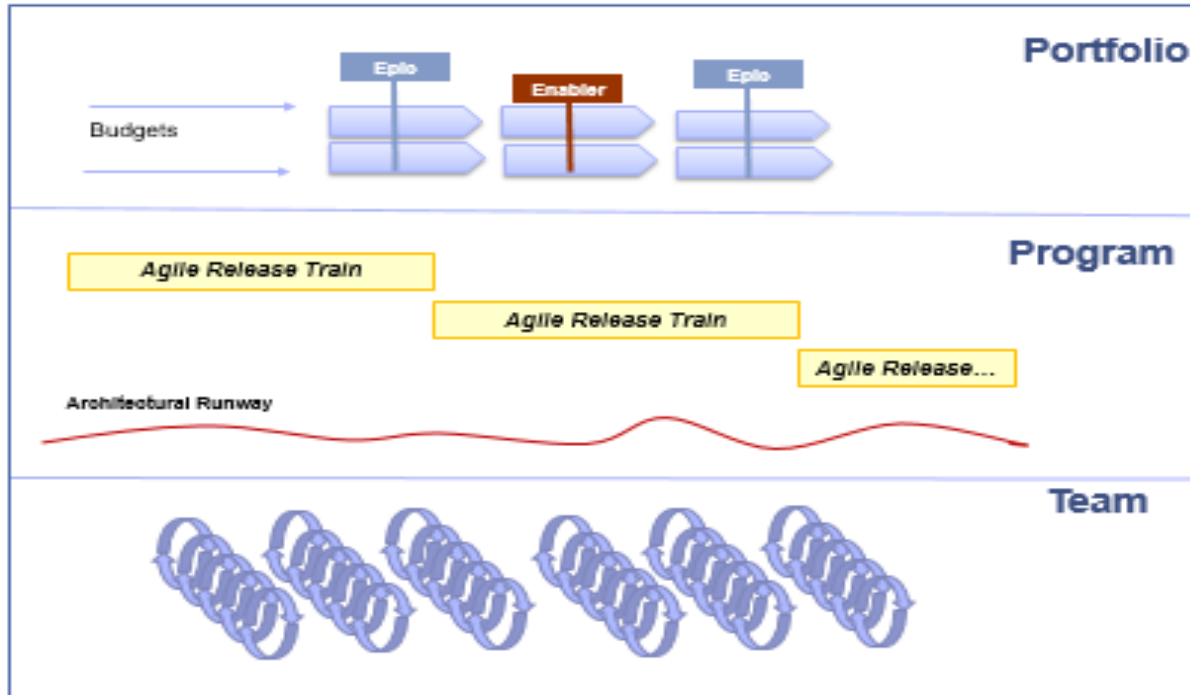
SAFe is a useful backdrop for this module because course participants have different roles in organizations of all sizes.

The SAFe “levels” allow for discussing Agile and architecture from various perspectives.

For example,

- some participants work on a single team
- some participants work on a program (multiple teams)
- some participants work at a portfolio level (multiple projects) and are concerned with enterprise-level architecture and investments

Simplified SAFe Diagram



Here is a simplified SAFe model that we use for illustrating concepts in this module.

Large scale environments require architecture practices to be applied at all 3 “levels” (Portfolio, Program, Team)

Applying Architecture Practices

The next few slides provide examples of practices discussed in this course against the SAFe backdrop.

We start with the Team level (bottom level) and work our way up.

Team Level-1

A common problem at the Project level (lowest level on the SAFe diagram) is that the heavy focus on features in a project leads to a lack of focus on quality attributes.

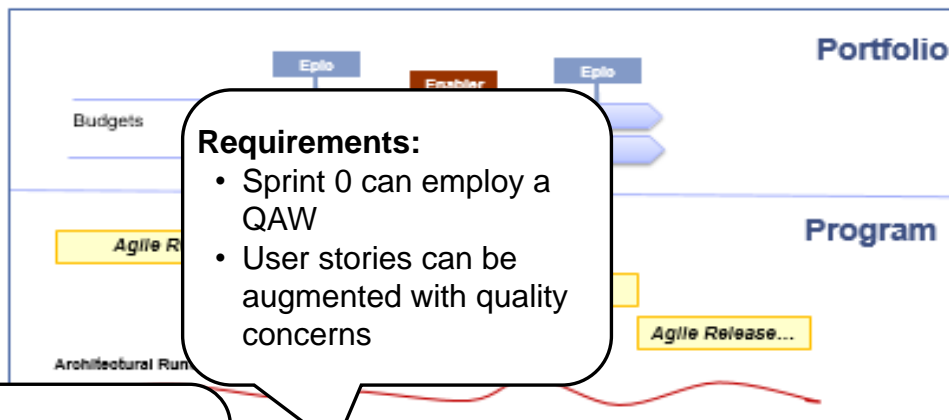
Systems become brittle, slow, and unreliable.

Technical debt accrues, and costly refactoring is necessary.

A quality attribute focus can help teams proactively minimize technical debt by focusing attention on architectural concerns (balancing the feature focus).

Team Level-2

The Team level provides several opportunities to apply techniques from this course, as shown below.



Requirements:

- Sprint 0 can employ a QAW
- User stories can be augmented with quality concerns

Design:

- “Architecture Envisioning” can employ techniques from ADD
- As the project matures, iterations can be fleshed out

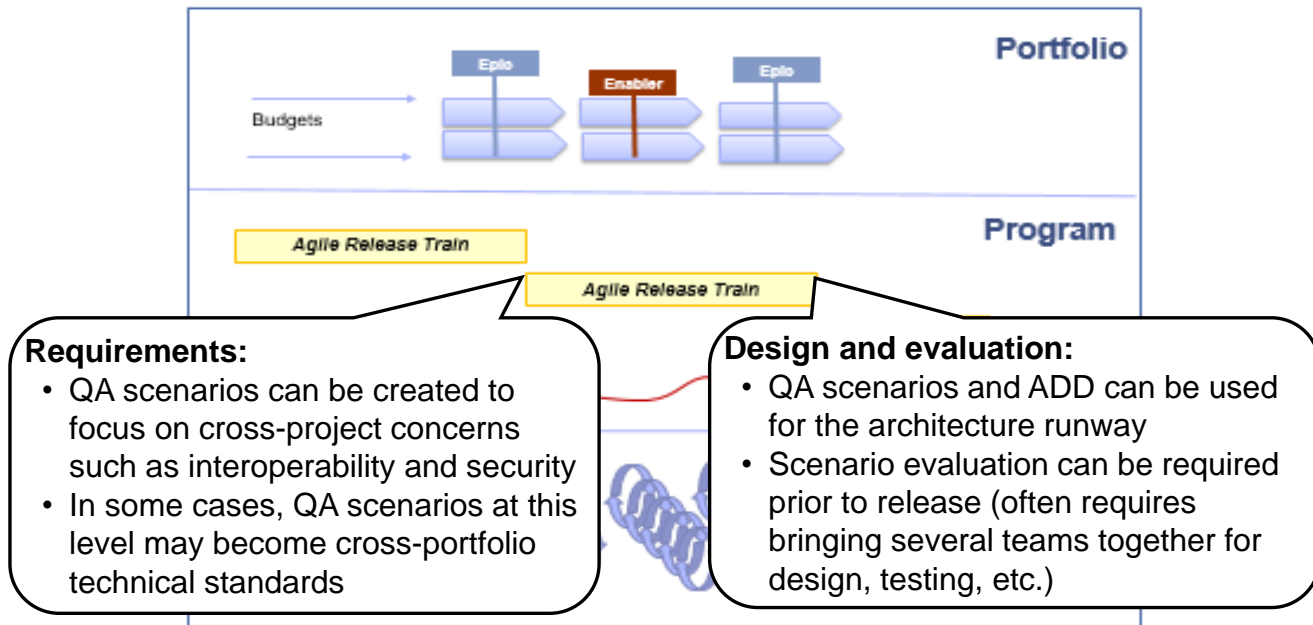
Evaluation:

In later sprints, QA scenarios are used to evaluate design decisions during sprint reviews

Program Level

At this level, multiple Agile projects need to integrate or coordinate.

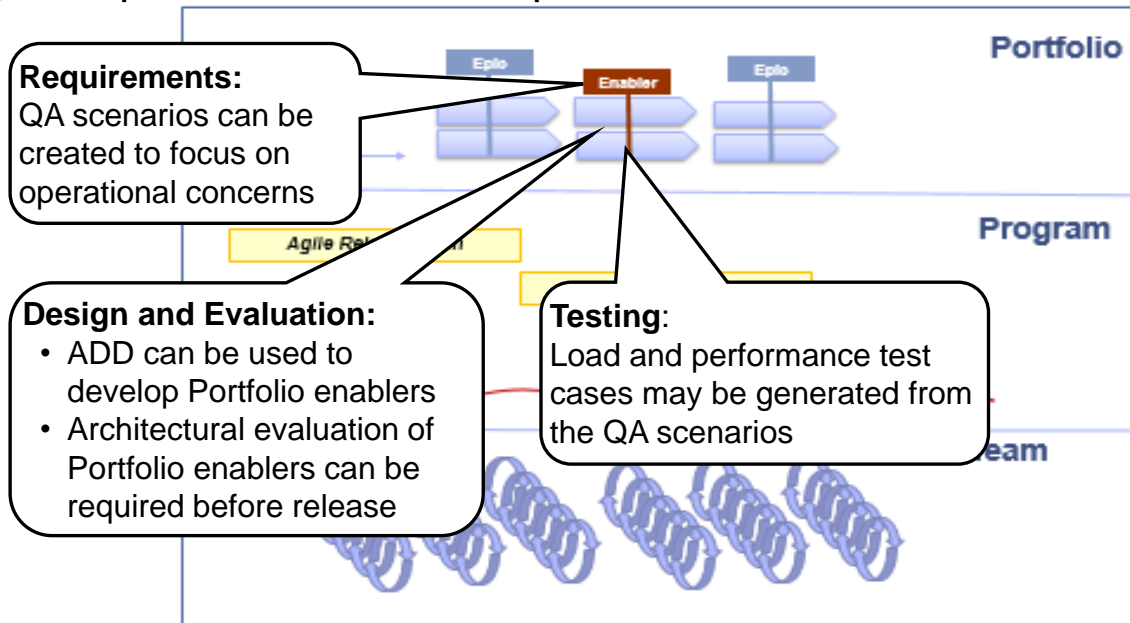
This is an important area; it is where risks often fall through cracks.



Portfolio Level

At this level, architectural investments may support multiple projects.

If capabilities are shared, operational concerns—such as reliability, scalability, availability, and performance—are important.



Agile-at-Scale Example: Wrap-up

That concludes an illustration of how the concepts in this course can be applied on a single Agile project *and* at scale.

What are your experiences?

Module Summary-1

We briefly described some key elements in the Agile philosophy and lifecycle:

- Agile philosophy centers on the Agile Manifesto.
- Lifecycle is incremental.

We talked about challenges that teams face as they try to determine how much to focus on architecture.

We also covered a little history.

- We explained that Agile processes were initially employed on small- to medium-sized projects with short time frames.
- Today, Agile is also applied to successful large-scale projects; this requires a blend of Agile and architecture.

Module Summary-2

We explained that Scaled Agile Framework is a popular approach, particularly among large organizations.

We showed how architecture practices from this course can be applied using the Scaled Agile Framework as a backdrop.

Further Reading

S. Bellomo, I. Gorton, R. Kazman, “Insights from 15 Years of ATAM Data: Towards Agile Architecture,” *IEEE Software*, September/October, 2015, 32:5, 38–45.

H.-M. Chen, R. Kazman, S. Haziyevev, V. Kropov, D. Chtchourov, “Architectural Support for DevOps in a Neo-Metropolis BDaaS Platform,” *Proceedings of DSSO*, 2015.

S. Bellomo, R. Nord, I. Ozkaya, “A Study of Enabling Factors for Rapid Fielding (Combined Practices to Balance Speed and Stability)”, *Proceedings of ICSE*, 2013.

J. Coplien, G. Bjornvig, *Lean Architecture for Agile Software Development*, Wiley, 2010.



Software Architecture: Principles and Practices

FINAL THOUGHTS

What Makes a “Good” Architecture?

There is no such thing as an inherently good or bad architecture.

Architectures are either more or less fit for some purpose.

Architectures can be evaluated, but only in the context of specific stated goals.

There are, however, good rules of thumb.

Process “Rules of Thumb”

The architecture should be the product of a single architect or a small group of architects with an identified technical leader.

- This approach leads to conceptual integrity and technical consistency.
- This recommendation holds for Agile and open source projects as well as “traditional” ones.
- There should be a strong connection between the architect(s) and the development team.

The architect (or architecture team) should base the architecture on a prioritized list of well-specified quality attribute requirements.

The architecture should be documented using views that address the concerns of the important stakeholders.

The architecture should be evaluated for its ability to deliver the system’s important quality attributes.

- This should occur early in the lifecycle and be repeated as appropriate.

The architecture should lend itself to incremental implementation.

- Create a “skeletal” system in which the communication paths are exercised but which at first has minimal functionality.

Structural “Rules of Thumb” – 1

The architecture should feature well-defined modules whose functional responsibilities are assigned on the principles of information hiding and separation of concerns.

- The information-hiding modules should encapsulate things likely to change.
- Each module should have a well-defined interface that encapsulates or “hides” the changeable aspects from other software.

Unless your requirements are unprecedented, your quality attributes should be achieved using well-known architectural patterns and tactics specific to each attribute.

The architecture should never depend on a particular version of a commercial product or tool. If it must, it should be structured so that changing to a different version is straightforward and inexpensive.

Modules that produce data should be separate from modules that consume data.

Structural “Rules of Thumb” – 2

Don’t expect a one-to-one correspondence between modules and components.

Every process should be written so that its assignment to a specific processor can be easily changed, perhaps even at runtime.

The architecture should feature a small number of ways for components to interact.

- The system should do the same things in the same way throughout.
- This will aid in understandability, reduce development time, increase reliability, and enhance modifiability.

The architecture should contain a specific (and small) set of resource contention areas, the resolution of which is clearly specified and maintained.

Discussion

What are good rules of thumb for architecture in your experience? What has worked for you—or failed you—in the past?

What Makes a Good Architect?

What does it mean for an architect to be "competent"?

How would you know?

And what should an organization do to foster such architects?

Improving Software Architecture Competence – 1



Most of the work in architecture to date has been technical:

- Design and creation
- Evaluation and analysis of architectures
- Styles and patterns
- Architectural reuse and software product lines
- Architectures for particular domains
- Architectural re-engineering and recovery

Improving Software Architecture Competence – 2



But architectures are created by architects...

- How can we help them do their best work?
- What does it mean for an architect to be competent?
- How can an architect improve his/her competence?

...working in organizations.

- How can we help an organization help its architects do their best work?
- What does it mean for an organization that produces architectures to be competent?
- How can an organization improve its competence in architecture?

What Is Competence?

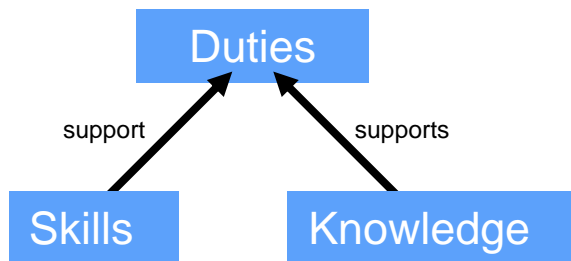
“The architecture competence of an organization is the ability of that organization to grow, use, and sustain the skills and knowledge necessary to effectively carry out architecture-centric practices at the individual, team, and organizational levels so as to produce high-quality architectures aligned with the organization’s business goals.”¹

This gives us a way to evaluate architects and organizations via

- past performance
- present performance

1. L. Bass, P. Clements, R. Kazman, M. Klein, “Models for Evaluating and Improving Architecture Competence,” CMU/SEI-2008-TR-006, 2008.

Duties/Skills/Knowledge Model



To measure how competent an architect is, we should be able to measure how well he or she

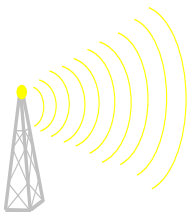
- performs architectural duties
- masters architectural skills
- possesses needed architectural knowledge

First step: Find out what those are!

- What are their duties?
- What skills and knowledge made them “capable of performing their allotted or required function?”

How can we find this out?

We Surveyed the “Community”



Three broad sources of information (with counts)

- “Broadcast” sources: information written by self-styled experts for mass anonymous consumption
 - Websites: e.g., Bredemeyer, SEI, HP, IBM (16*)
 - Blogs and essays (16*)
 - “Duties” list on SEI website
 - Books on software architecture (25 top sellers)
- Education and training sources:
 - University courses in software architecture (29*)
 - Industrial/non-university public courses (22*)
 - Certificate and certification programs in architecture; e.g., SEI, Open Group, Microsoft (7*)
- “Architecture for a living” sources
 - Position descriptions for software architects (60)
 - Résumés of software architects (12)
 - Questionnaires from practicing architects (30+)



* Exhaustive or near-exhaustive web search

Survey Results



We surveyed over 200 sources.

We cataloged

- 201 duties
- 85 skills
- 96 knowledge areas

We grouped the data into clusters using an affinity exercise.

Architectural Duties

Architecting

- Overall
- Creating the architecture
- Architecture evaluation and analysis
- Documentation
- Existing system and transformation

Life cycle phases other than architecture

- Requirements
- Testing
- Coding and development

Technology related

- Future technologies
- Tools and technology selection

Interacting with stakeholders

- Overall
- Clients
- Developers

Management

- Project management
- People management
- Support for project management

Organization and business related

- Organization
- Business

Leadership and team building

- Technical leadership
 - Team building
-

Architectural Skills

Communication skills

Out
Both (i.e., two-way)
In

Interpersonal skills

Within team
With other people
Leadership skills

Work skills

Effectively managing high workload
Skills to excel in a corporate environment
Skills for handling large amounts of information

Personal skills

Personal qualities
Skills for handling unknown
Skills for handling unexpected
Learning

Architectural Knowledge

Computer science knowledge

Knowledge of architecture concepts
Knowledge of software engineering
Design knowledge
Programming knowledge

Knowledge of technologies and platforms

Specific
Platforms
General
Domain

Knowledge about organizational context and management

Industry
Enterprise knowledge
Leadership and management

Discussion

How do you see the role of architecture evolving in your organization?

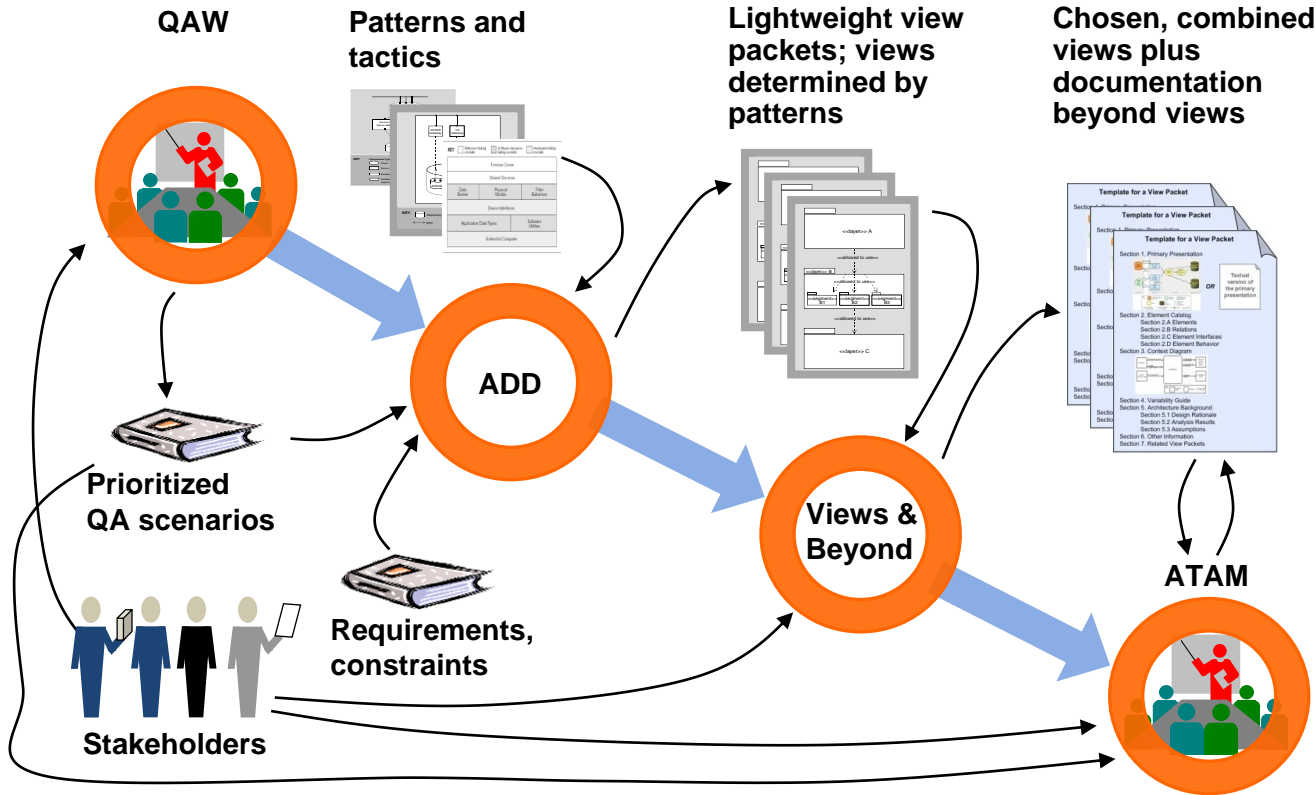
Where We Have Been

We have discussed

- the fundamentals of software architecture
- the Architecture Influence Cycle
- the precise characterization of quality attributes
- foundational design concepts (patterns, tactics, etc.)
- the documentation of architectures
- the evaluation of architectures

And we have explained how to package these concepts into *methods*.

QAW, ADD, V&B, and ATAM Together







Software Architecture: Principles and Practices

SEI Opportunities and Events

Software Architecture Professional Certificate

You can become a recognized expert in architecture-centric practices—on your team and in your organization. The Software Architecture Professional Certificate brings an obvious advantage to

- every project you serve
- your organization, which will enjoy an edge over other contractors during the bidding process. The SEI certificate reassures potential customers and enhances credibility.
- your value to your organization and potential to lead development efforts

You've already completed an important step. You can earn the SEI Software Architecture Professional Certificate by

- completing the Software Architecture: Principles and Practices course
- completing the Documenting Software Architecture course
- completing the Software Architecture Design and Analysis course
- completing the Software Architecture: Principles and Practices Examination

For more information, visit

<http://www.sei.cmu.edu/training/certificates/architecture/professional.cfm>

Software Architecture: Principles and Practices Examination

Individuals pursuing the Software Architecture Professional Certificate, ATAM Evaluator Certificate, or Service-Based Architecture Professional Certificate must demonstrate architecture proficiency through the Software Architecture: Principles and Practices Examination.

Individuals pursuing ATAM Leader Certification must also demonstrate architecture proficiency via the exam.

For more information about the exam, visit

<http://www.sei.cmu.edu/training/v19.cfm>

Other Software Architecture Certificate and Certification Opportunities

Successfully completing the Software Architecture: Principles and Practices Examination affords individuals the opportunity to pursue one or more of the following credentials:

- Software Architecture Professional Certificate
- ATAM Evaluator Certificate
- Service-Based Architecture Professional Certificate
- ATAM Leader Certification

For more information on the SEI's Software Architecture Certificate Programs visit:

<http://www.sei.cmu.edu/training/certificates/architecture/>

For more information on the SEI's Software Architecture Certification Programs visit:

<http://www.sei.cmu.edu/certification/opportunities/index.cfm>

ATAM Evaluator Certificate

You're on the path to becoming an ATAM Professional. The Software Architecture: Principles and Practices course is a prerequisite for expanding your architecture evaluation skills via the SEI's ATAM Evaluator Training.

Completing ATAM Evaluator Training results in

- an ATAM Evaluator certificate from the SEI
- recognition of your architecture evaluation knowledge and skills
- membership in a community of experts who are key contributors to their organizations' software success

ATAM Evaluator Training is a prerequisite for becoming an SEI-Certified ATAM Leader, which results in

- elevation to a higher level of expertise
- official recognition of skills
- career advancement opportunities
- increased influence: participation in a community of ATAM Leaders who collaborate to improve the method and practice

