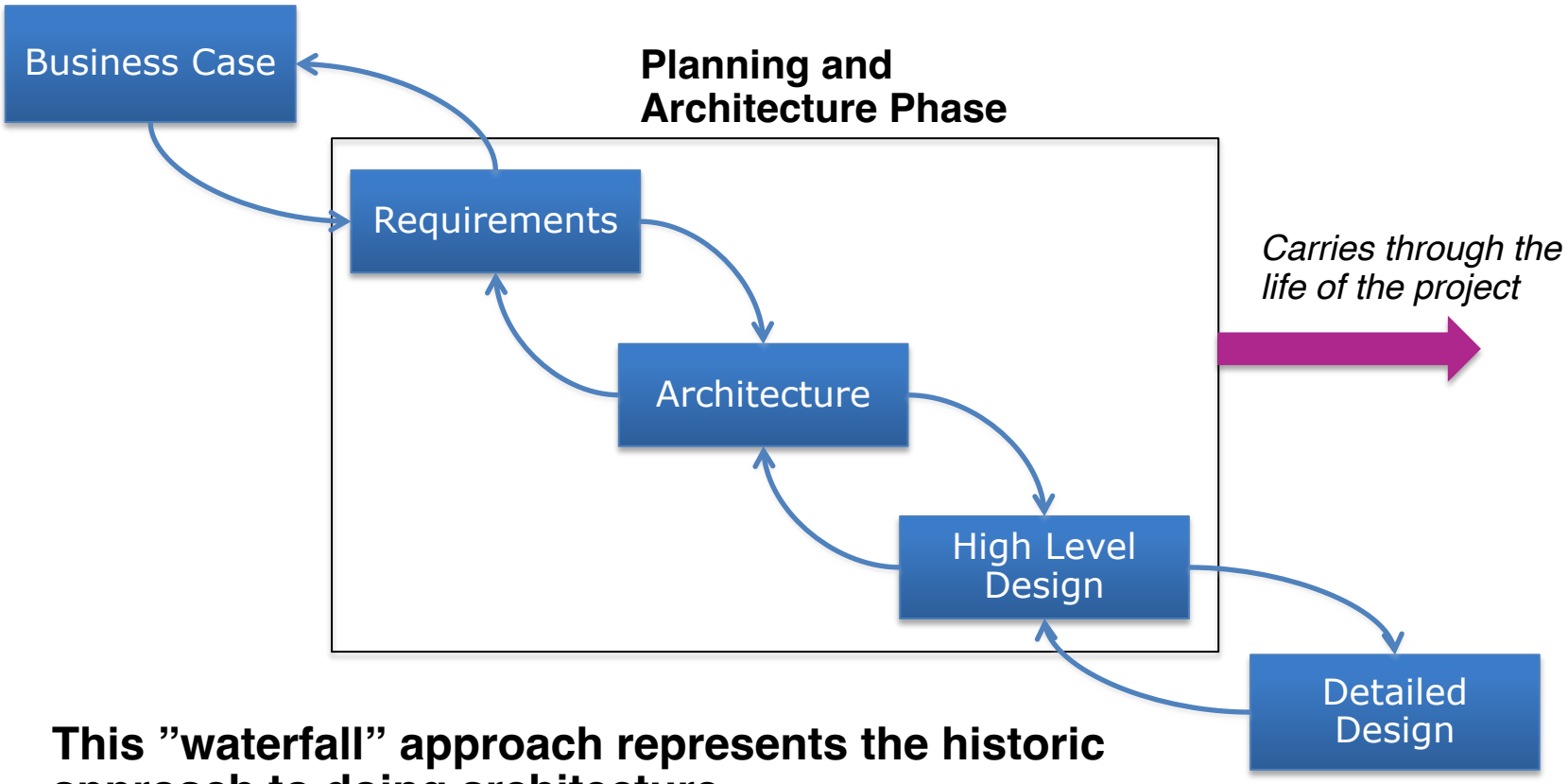
SE 577
Software Architecture

Basic SW Architectural Concepts

Acknowledgement

- Material from several sources including:
  - Ian Gorton. *Essential Software Architecture (2nd Edition)*, Springer-Verlag.
  - R.N. Taylor, N. Medvidovic, and E.M. Dashofy. *Software Architecture: Foundations, Theory and Practice*, Wiley.
  - Neil Ford & Mark Richards
    Software Architecture Fundementals

Architecture in a Traditional Project Life Cycle



This ”waterfall” approach represents the historic approach to doing architecture.

Architecture in an Agile Project Life Cycle

Create Reference Architecture –
"how the proposed solution fits into the big picture"

Iterate

Create Solution Architecture for a Program Increment (approx. 5 Sprints)

Update Existing Architecture Decisions

Define new architecture decisions that need to be made

Create Technical Architecture Guidance to Guide Engineering, Delivery and Testing

Monitor Engineering, ensuring architecture guidance is followed or creating a backlog for adjusting architecture

Perform Architecture Retrospective

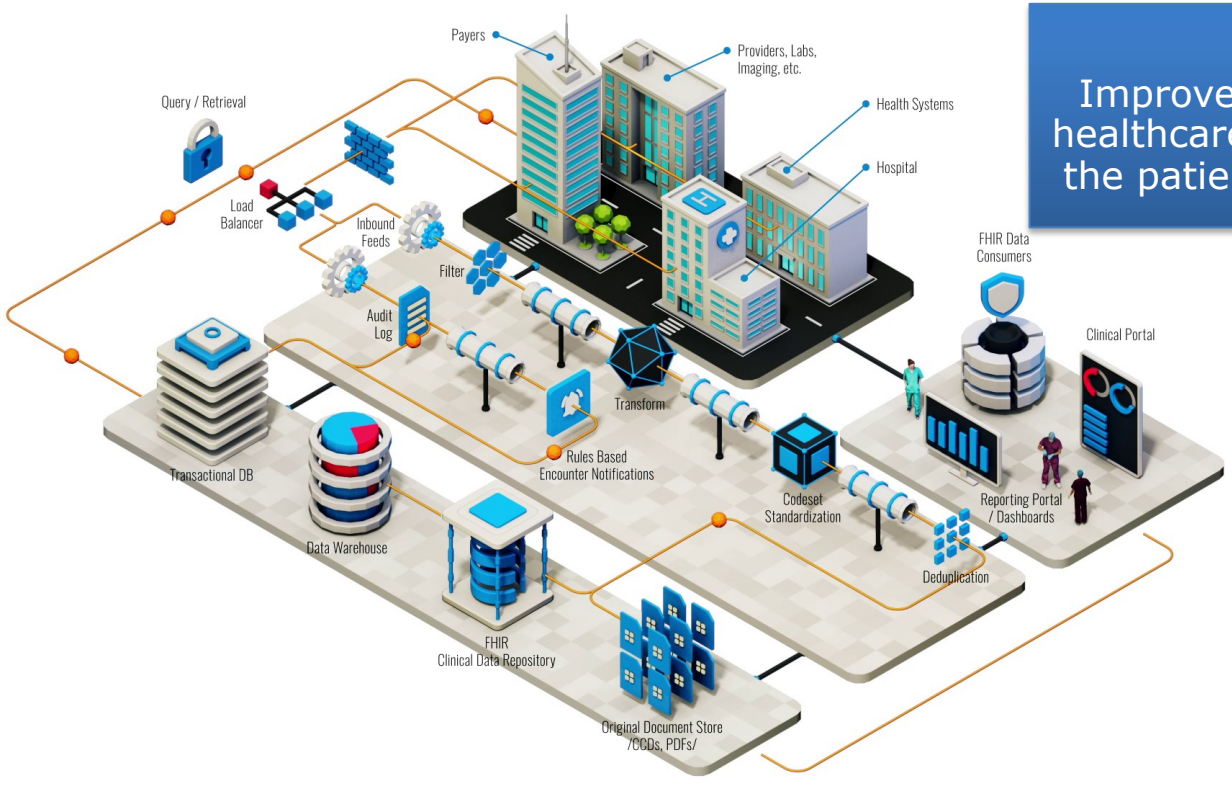
Monitor overall architecture against agreed objectives

Consistently support activities needed in support of quality attributes – e.g., security, compliance, etc

Negotiate and Influence on behalf of architecture and engineering ongoing progress and required changes

Example of Reference Architecture

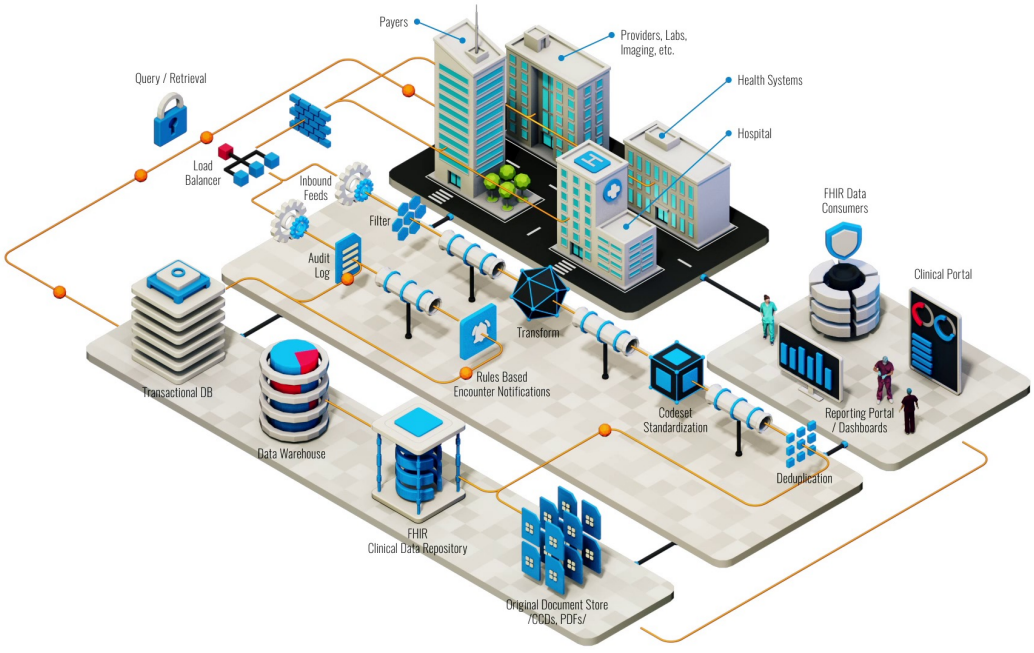
Create Reference Architecture –
"how the proposed solution fits into the big picture"



OBJECTIVE
Improve timeliness of insights from healthcare delivery system to improve the patient experience and outcomes

Example: Architecture Kickoff

Create Reference Architecture –
"how the proposed solution fits into the big picture"



OBJECTIVE

Improve timeliness of insights from healthcare delivery system to improve the patient experience and outcomes

KEY ARCHITECTURE NEEDS

Pick a healthcare interoperability partner – partner must support real-time FHIR events

Identity key systems that must be adjusted to react to real time patient data

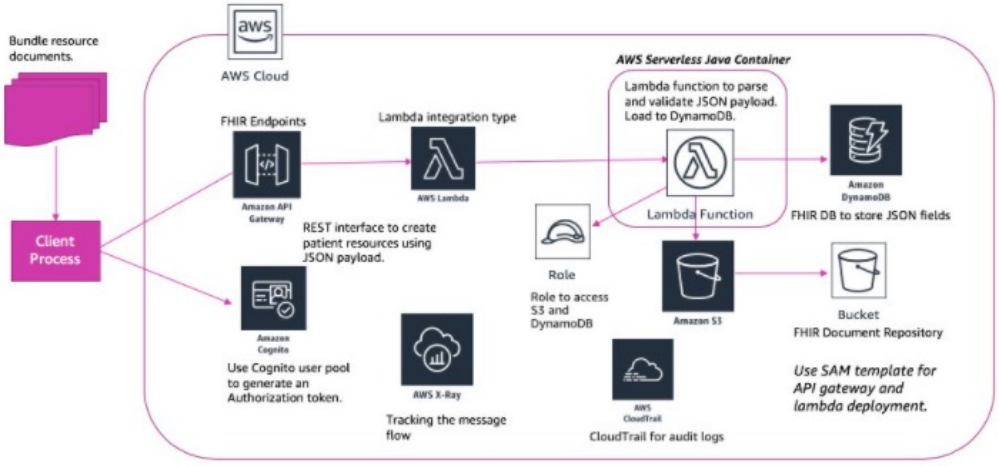
Build new data insights engine to process events real time and to build actionable insights

Modernize legacy systems to react in real time to key events of interest

Build analytics solution to measure effectiveness and patient outcomes

Example: Establish baseline solution architecture next

Create Solution Architecture –
"how the proposed solution will be structured"



OBJECTIVE

Employ cloud native architecture patterns to ensure resiliency, scale, and flexibility to ingest real-time clinical events

KEY ARCHITECTURE DECISIONS and GUIDENCE

Ensure that HIPAA privacy and security requirements are enforced in every component of the solution

Favor managed cloud services to support dynamic scalability and resiliency

Ensure audit and control policies are supported by the solution via CloudFront

Ensure data is flexibly stored to support FHIR resource standards

Major Areas of Concern for Architectural Design

- Requirements
  - Both *functional* and *non-functional*
- OA&M (Operation Administration & Maintenance)
  - The set of procedures for initializing and administering the system.
- Error Recovery
  - How does the system handle *rainy day* scenarios?

Requirements

- Traditional SE suggests requirements analysis should remain unsullied by any consideration for a design.
- However, without reference to existing architectures it becomes difficult to assess practicality, schedules, or costs.
- Requirements analysis and consideration of design must be pursued at the same time.

Non-Functional Requirements

- Non-Functional Requirements (NFRs) define “how” a system achieves its functional requirements.

- NFRs include:
  - Quality attributes
    - Application performance must provide sub-four second response times for 90% of requests.
  - Technical constraints
    - The system must be written in Java so that we can use existing development staff.
  - Business constraints
    - “We want to work closely with and get more development funding from *MegaHugeTech Corp*, so we need to use their technology in our application.”

Non-Functional Requirements (cont'd)

- NFRs rarely captured in functional requirements
  - They are also known as "architecture requirements"
  - Must be elicited by architect
- Specification of NFR might require an architectural framework to even enable their statement.
- An architectural framework will be required for assessment of whether the properties are achievable.

What is OA&M?

- OA&M includes all the steps necessary to get and keep the system operational.
  - Manufacture the software for delivery and installation
  - Install the software
  - Convert required data
  - Initialize the database
  - Prime the tables
  - Start up the system
  - Connect to the outside world
  - Take it down for backups
  - Restart it after a failure
  - Update the software in the field
  - Etc., Etc., Etc.

What is OA&M? (cont'd)

- OA&M is one of the most overlooked parts of the architecture.
- Many systems leave this last and assume that the operators in the field will do what they have to do to keep the system operational.
- OA&M has to be included in the original architecture because it is difficult (if not impossible) to add later.
- Some view it as just the set of instructions that the system administrator is given, but this is just the tip of the iceberg.

Error Recovery

- The system has to be designed from the standpoint of errors.
- The *sunny day* paths are easy; the error handling is what complicates the system.
- Must have a consistent approach for handling errors.
- Error recovery (like OA&M) is often overlooked in the system architecture.

Error Recovery: Kinds Of Errors

- Communication
  - Intermittent / interrupted communication
  - Unexpected / incorrect network changes
- Process
  - Abnormal termination
  - Asleep waiting (*patiently*) for an event (which may never occur, or that has been missed).
  - Resource leaks
- Coordination
  - Someone forgot to tell the up/downstream systems about the new data format.
  - Other systems in the network do not react kindly to a system going down due to a failure or for a scheduled upgrade

Software Architecture's Elements

- A software system's architecture should be a composition and interplay of different elements
  - Processing
  - Data, also referred as information or state
  - Interaction

Components

- Elements that encapsulate processing and data in a system's architecture are referred to as software components
- A software component is an architectural entity that:
  - Encapsulates a subset of the system's functionality and/or data.
  - Restricts access to that subset via an explicitly defined interface.
  - Has explicitly defined dependencies on its required execution context.
- Components typically provide application-specific services.

Connectors

- A software connector is an architectural building block tasked with effecting and regulating interactions among components.
- In complex systems interaction may become more important and challenging than the functionality of the individual components.
- In many software systems connectors are usually simple procedure calls or shared data accesses.
  - Much more sophisticated and complex connectors are possible!
- Connectors typically provide application-independent interaction facilities.

Examples of Connectors

- Procedure call connectors
- Shared memory connectors
- Message passing connectors
- Streaming connectors

Architecture Defines Structure

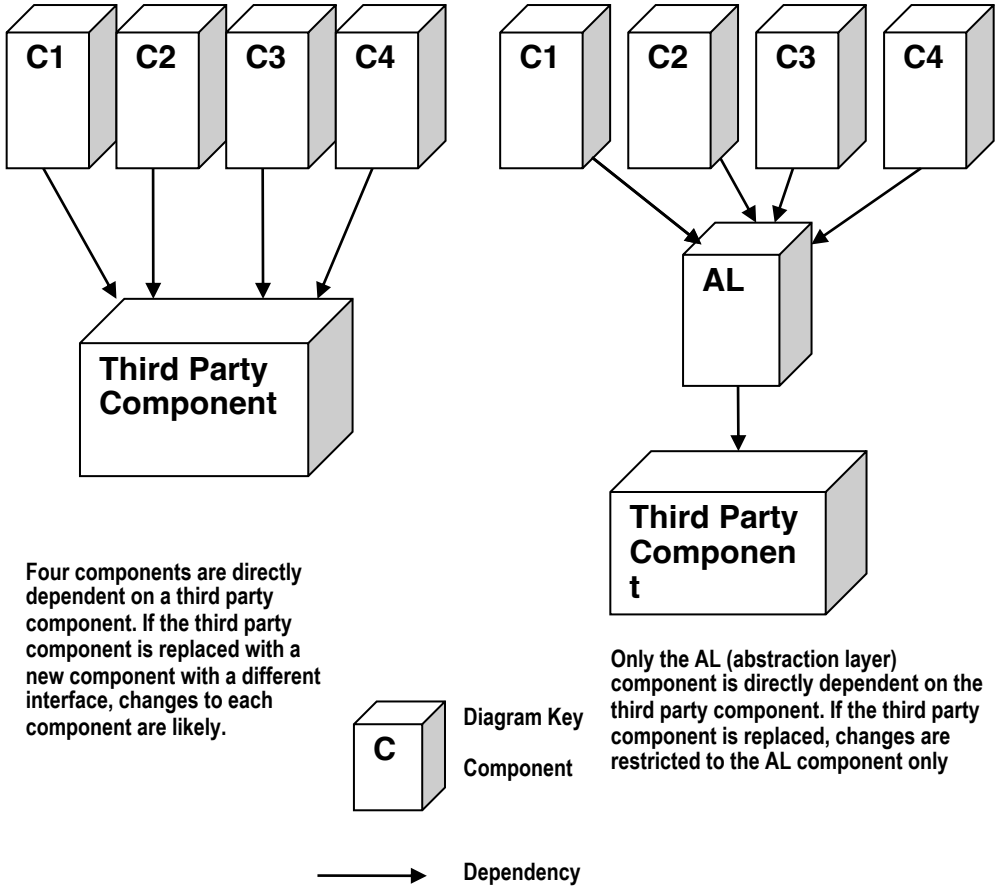
- Decomposition of system into components/modules/subsystems
- Architecture defines:
  - Component responsibilities
    - Precisely what a component will do when you ask it
  - Component interfaces
    - What a component can do
  - Component communications and dependencies
    - How components communicate

Architecture Specifies Component Communication

- Communication involves:
  - Data passing mechanisms
    - Remote method invocation
    - Asynchronous message
  - Control flow
    - Sequential
    - Concurrent/parallel

An Important Concern: Structure and Dependencies

- Excessive component dependencies are bad!
- Key architecture issue
  - Identifying components that may change
  - Reduce direct dependencies on these components
- Creates more modifiable systems



Inter-Process Communication

- Inter-process communication (IPC) refers to the activity of sharing data across multiple processes.
- Processes, or applications, using IPC are categorized as *clients* and *servers*.
  - (In what follows we'll use process to refer to either process or application.)
  - A *client* is a process that requests a service from some other process, or application.
  - A *server* is a process that responds to a client request.

IPC (cont'd)

- Many processes act as both a client and a server, depending on the situation.
  - For example, a word processing application might act as a client in requesting a summary table of manufacturing costs from a spreadsheet application acting as a server.
  - The spreadsheet application, in turn, might act as a client in requesting the latest inventory levels from an automated inventory control application.

IPC (cont'd)

- Reasons for implementing IPC systems:
  - Sharing information
    - Web servers use IPC to share web documents and media with users through a web browser.
  - Distributing labor across systems
    - An application uses multiple servers that communicate with one another using IPC to process user requests.
  - Privilege separation
    - Some GUI software is separated into layers based on privileges to minimize the risk of attacks. These layers communicate with one another using encrypted IPC.

IPC (cont'd)

- Methods for achieving IPC are divided into categories which vary based on requirements such as performance and modifiability, network bandwidth, latency, etc.

Configurations

- Components and connectors are composed in a specific way in a given system's architecture to accomplish that system's objective
- An *architectural configuration*, or topology, is a set of specific associations between the components and connectors of a software system's architecture.

Architecture Styles/Patterns

- Patterns catalogue successfully used structures that facilitate certain kinds of component communication
  - Client-server
  - Message broker
  - Pipe-and-filter
- Patterns have well-known characteristics appropriate for particular types of requirements
- Patterns are very useful…
  - Reusable architectural blueprints
  - Help efficiently communicate a design
  - Large systems comprise a number of individual patterns
- "Patterns and Styles are the same thing – the patterns people won" [anonymous SEI member]

Three-Tiered Pattern

- Front Tier
  - Contains the user interface functionality to access the system's services
- Middle Tier
  - Contains the application's major functionality
- Back Tier
  - Contains the application's data access and storage functionality

Architectural Models, Views, and Visualizations

- Architecture Model
  - An artifact documenting some or all of the architectural design decisions about a system
- Architecture View
  - A subset of related architectural design decisions
- Architecture Visualization
  - A way of depicting some or all of the architectural design decisions about a system to a stakeholder

Architecture Views

- A software architecture represents a complex design artifact
- Many possible ‘views’ of the architecture
  - Cf. with buildings – floor plan, external, electrical, plumbing, air-conditioning

Architecture Views (cont'd)

- Logical view: describes architecturally significant elements of the architecture and the relationships between them.
- Process view: describes the concurrency and communications elements of an architecture.
- Physical view: depicts how the major processes and components are mapped on to the applications hardware.
- Development view: captures the internal organization of the software components as held in e.g. a configuration management tool.

Architecture Views (cont'd)

- *Marketecture*
  - Informal depiction of system's structure and interactions.
  - Portray the design philosophies embodied in the architecture.
- Every system should have a marketecture:
  - Easy to understand.
  - Helps discussion during design, build, review, sales (!) activities.

WWW in a (Big) Nutshell

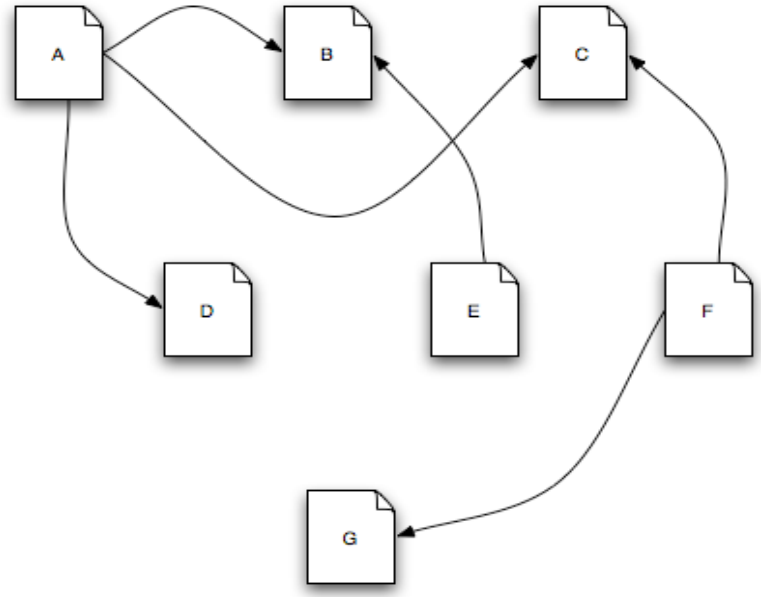
- The Web is a collection of resources, each of which has a unique name known as a uniform resource locator, or “URL”.
- Each resource denotes, informally, some information.
- URLs can be used to determine the identity of a machine on the Internet, known as an origin server, where the value of the resource may be ascertained.
- Communication is initiated by clients, known as user agents, who make requests of servers.
  - Web browsers are common instances of user agents.

WWW in a (Big) Nutshell (cont'd)

- Resources can be manipulated through their representations.
  - HTML is a very common representation language used on the Web.
- All communication between user agents and origin servers must be performed by a simple, generic protocol (HTTP), which offers the command methods GET, POST, etc.
- All communication between user agents and origin servers must be fully self-contained. (So-called "stateless interactions")

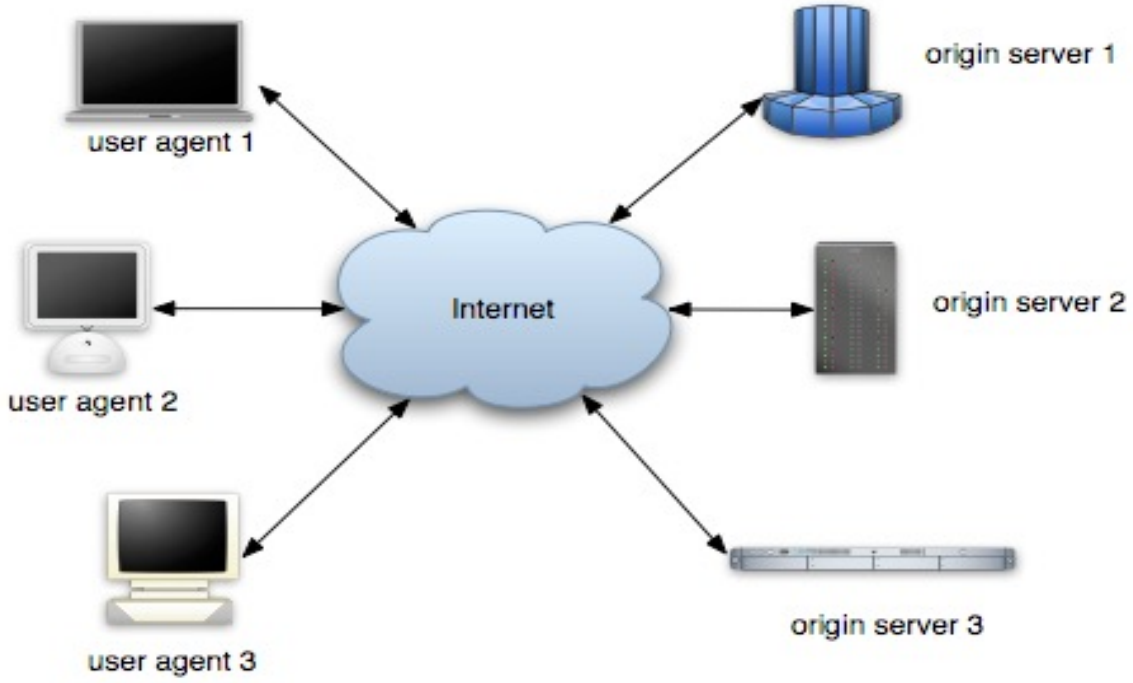
The Architecture of the WWW

- This is the Web



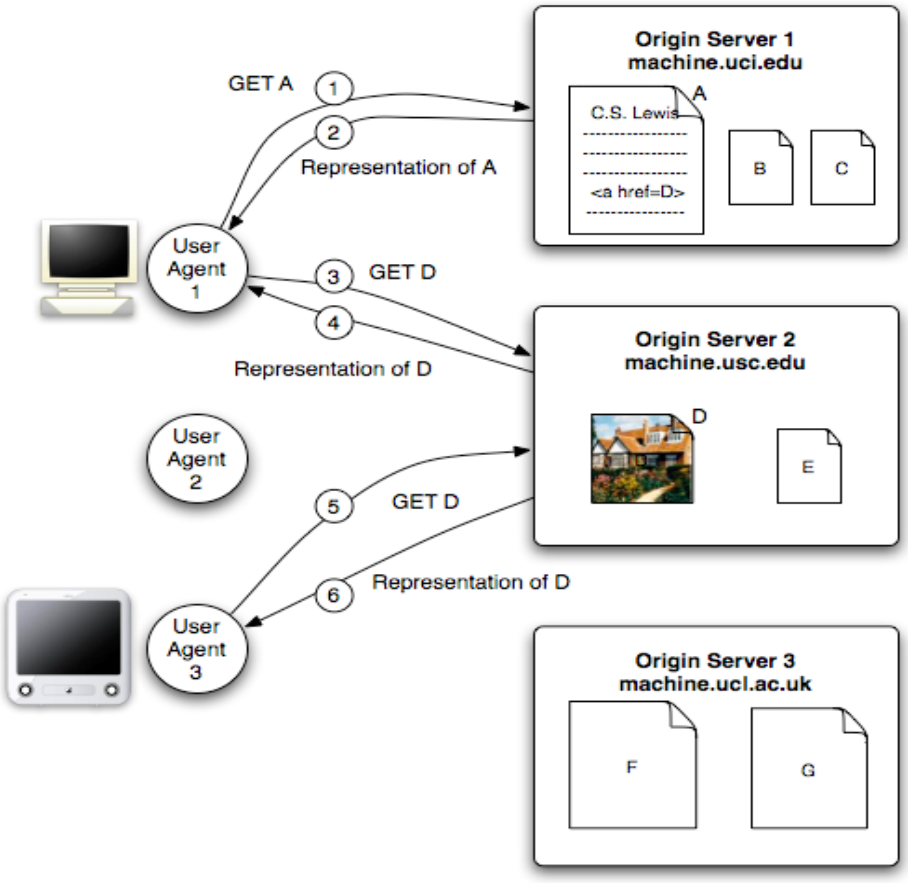
The Architecture of the WWW (cont'd)

- So is this



The Architecture of the WWW (cont'd)

- And this



WWW's Architecture

- Architecture of the Web is wholly separate from the code.
- There is no single piece of code that implements the architecture.
- There are multiple pieces of code that implement the various components of the architecture.
  - E.g., different Web browsers

WWW's Architecture (cont'd)

- Stylistic constraints of the Web's architectural style are not apparent in the code
  - The effects of the constraints are evident in the Web

- One of the world's most successful applications is only understood adequately from an architectural vantage point.

Prescriptive vs. Descriptive Architecture

- A system's *prescriptive architecture* captures the design decisions made prior to the system's construction
  - It is the *as-conceived* or *as-intended* architecture
- A system's *descriptive architecture* describes how the system has been built
  - It is the *as-implemented* or *as-realized* architecture

Temporal Aspect

- Architecture has a temporal aspect
  - Design decisions are made over a system's lifetime
  - A system's architecture will change over time
- At any given point in time the system has only one architecture

As-Designed vs. As-Implemented Architecture

- Which architecture is “correct”?
  - Intent vs. added experience of implementation
- Are the two architectures consistent with one another?
  - Architectural inconsistencies can be much more complex and/or insidious.
- What criteria are used to establish the consistency between the two architectures?
  - Sophisticated techniques might be needed
- On what information is the answer to the preceding questions based?
  - Diagrams may not contain adequate information

Architectural Evolution

- When a system evolves, ideally its prescriptive (*as-designed*) architecture is modified first.
- In practice, the system – and thus its descriptive (*as-implemented*) architecture – is often directly modified.
- This happens because of:
  - Developer sloppiness
  - Perception of short deadlines which prevent thinking through and documenting
  - Lack of documented prescriptive architecture
  - Need or desire for code optimizations
  - Inadequate techniques or tool support

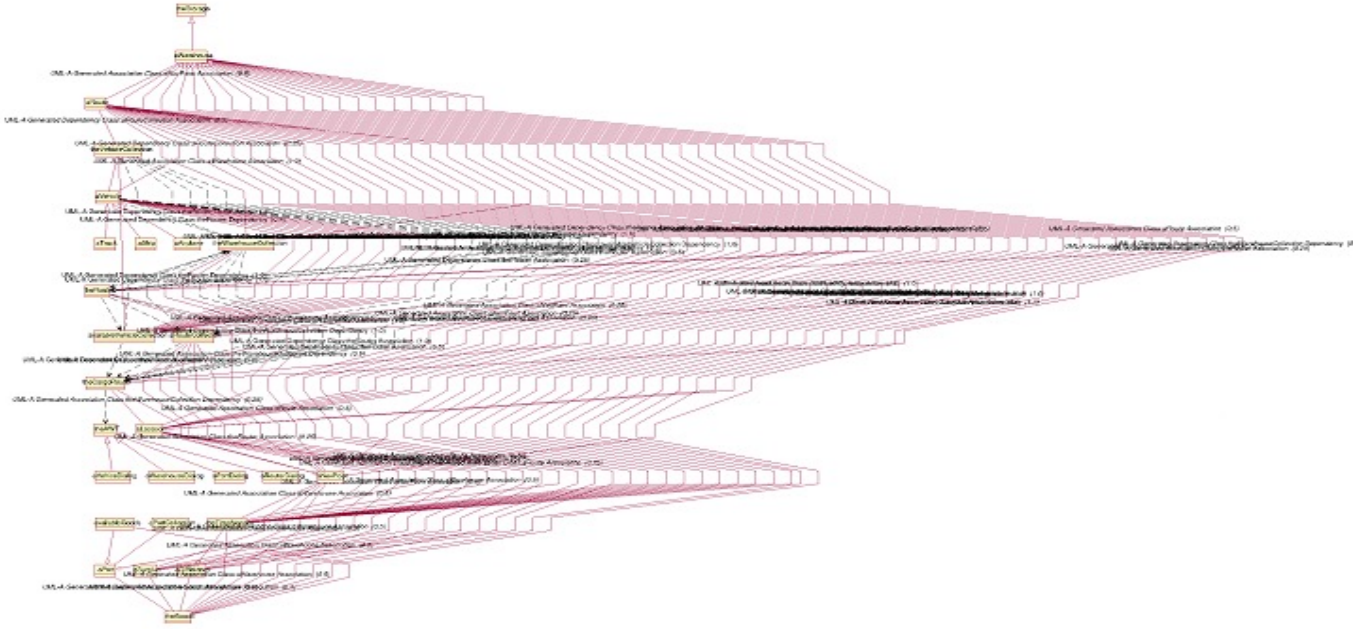
Architectural Degradation

- Two related concepts
  - Architectural drift
  - Architectural erosion
- *Architectural drift* is introduction of principal design decisions into a system's descriptive architecture that
  - Are not included in, encompassed by, or implied by the prescriptive architecture
  - But which do not violate any of the prescriptive architecture' s design decisions
- *Architectural erosion* is the introduction of architectural design decisions into a system's descriptive architecture that violate its prescriptive architecture

Architectural Recovery

- If architectural degradation is allowed to occur, one will be forced to *recover* the system's architecture sooner or later
- *Architectural recovery* is the process of determining a software system's architecture from its implementation-level artifacts
- Implementation-level artifacts can be
  - Source code
  - Executable files
  - Java .class files

Implementation-Level View of an Application



Complex and virtually incomprehensible!