

Applied Architectures, Part 2

**Software Architecture
Lecture 18**

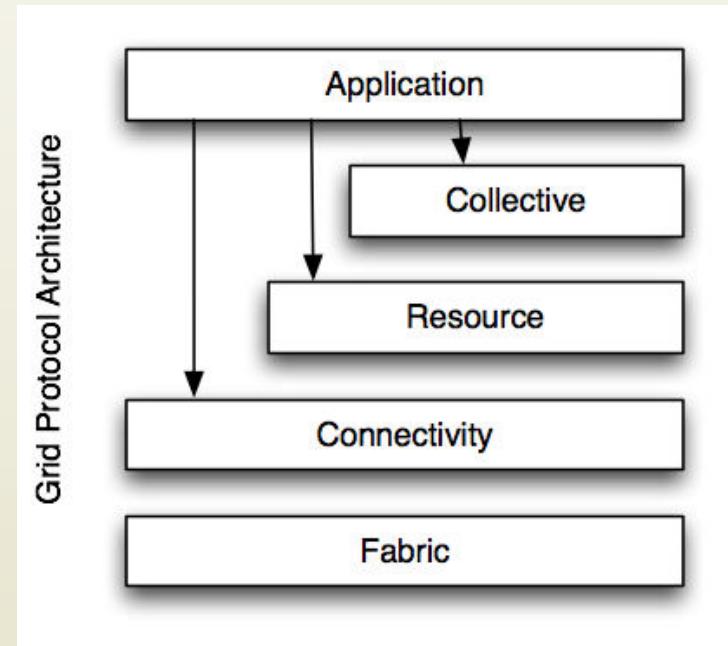
Decentralized Architectures

- Networked applications where there are multiple authorities
- In other words
 - ◆ Computation is distributed
 - ◆ Parts of the network may behave differently, and vary over time

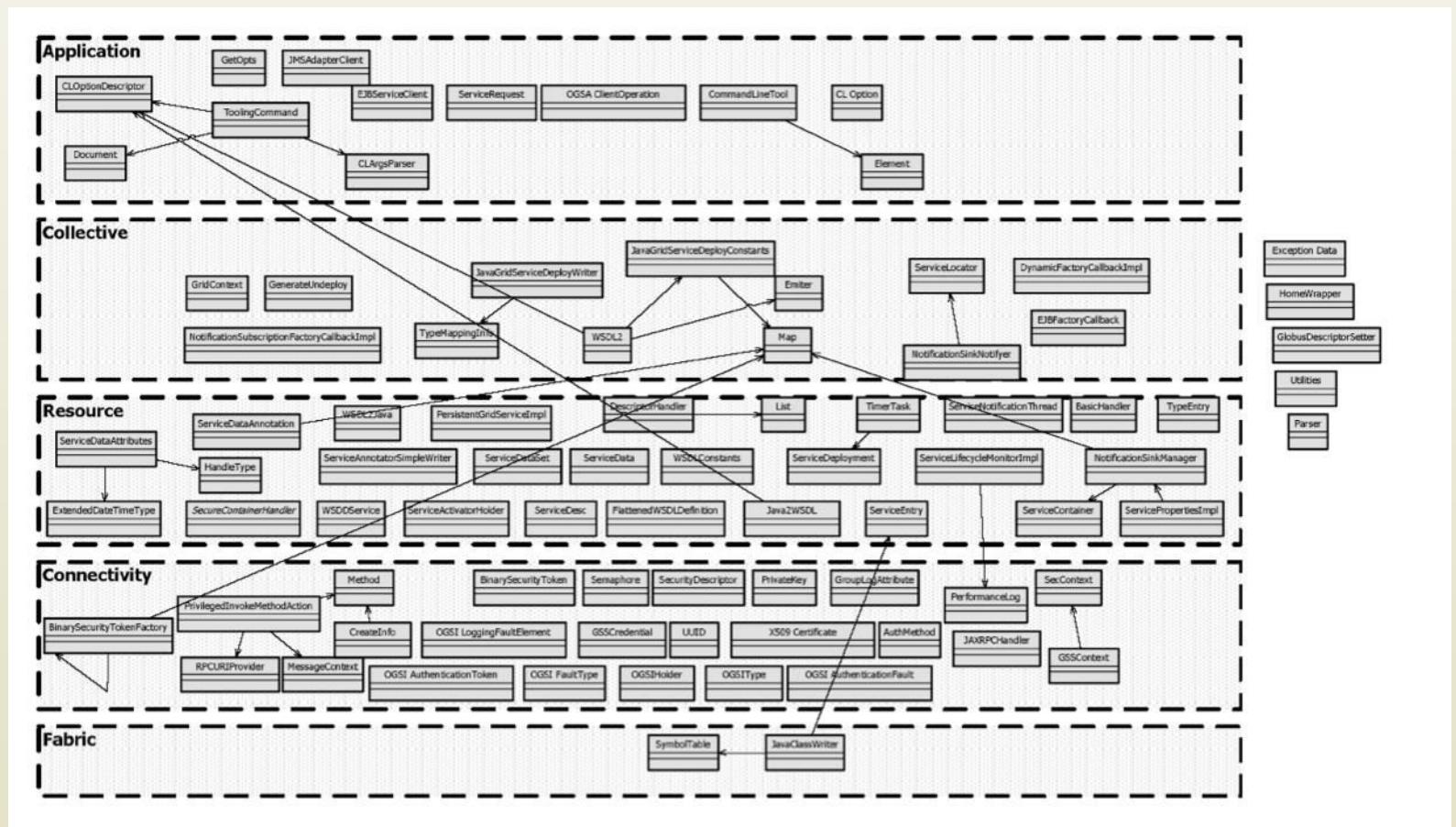
It's just like collaboration in the real world

Grid Protocol Architecture

- Coordinated resource sharing in a distributed environment
 - ◆ E.g., Folding-at-home
- GLOBUS
 - ◆ A commonly used infrastructure
- “Standard architecture”
 - ◆ Fabric manages low-level resources
 - ◆ Connectivity: communication and authentication
 - ◆ Resource: sharing of a single r.
 - ◆ Collective: coordinating r. usage



Grid GLOBUS (Recovered)



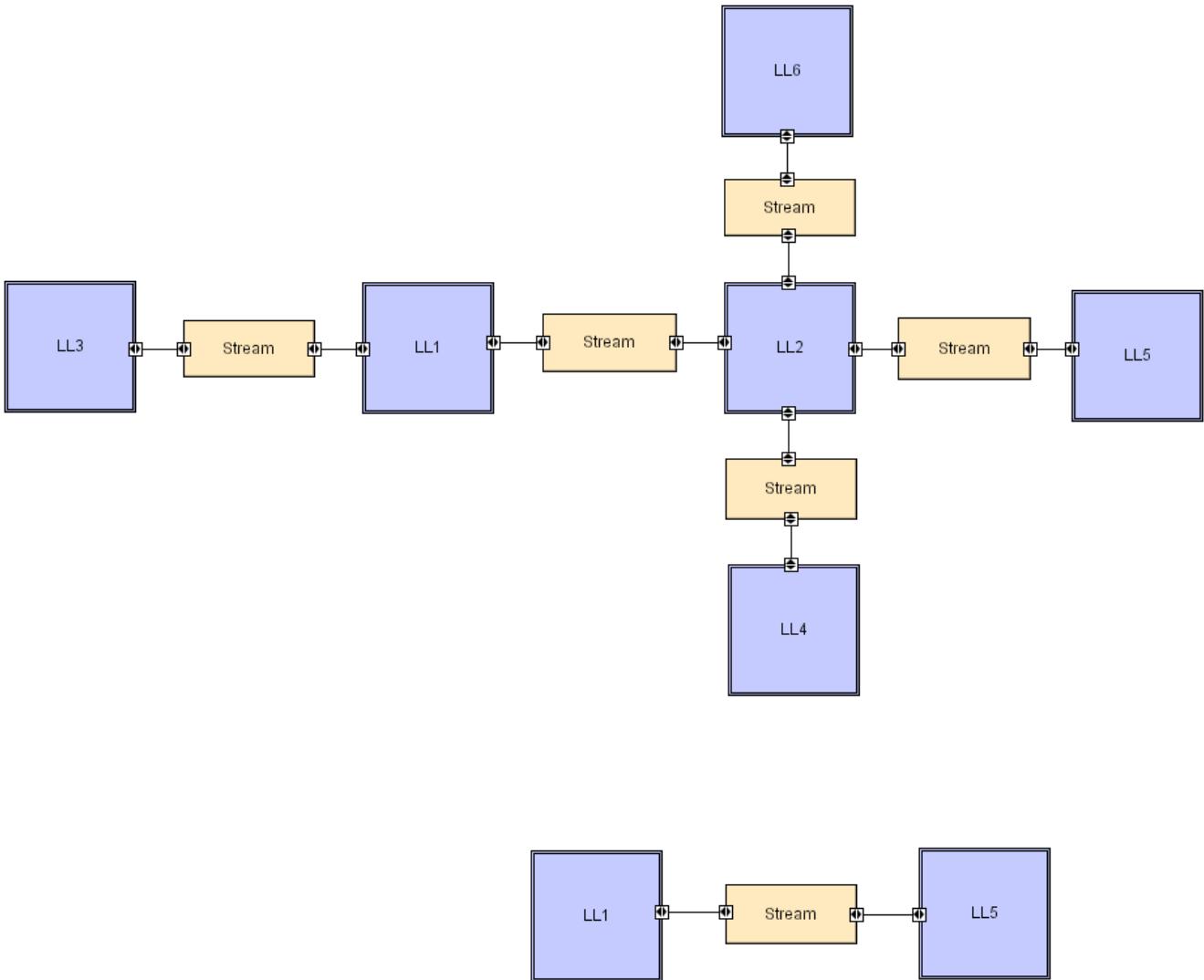
Peer-to-Peer Architectures

- Decentralized resource sharing and discovery
 - ◆ Napster
 - ◆ Gnutella
- P2P that works: Skype
 - ◆ And BitTorrent

Peer-to-Peer Style

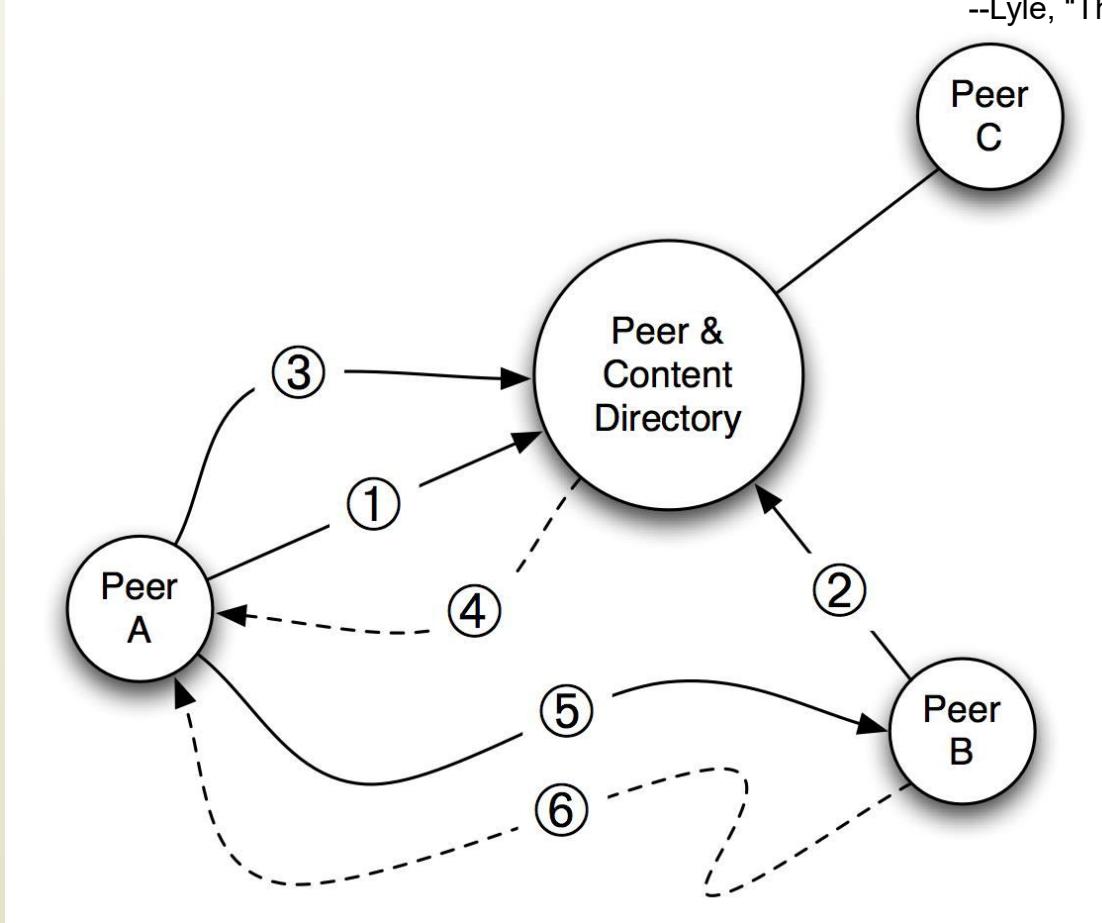
- State and behavior are distributed among peers which can act as either clients or servers.
- Peers: independent components, having their own state and control thread.
- Connectors: Network protocols, often custom.
- Data Elements: Network messages
- Topology: Network (may have redundant connections between peers); can vary arbitrarily and dynamically
- Supports decentralized computing with flow of control and resources distributed among peers.
Highly robust in the face of failure of any given node.
Scalable in terms of access to resources and computing power. But caution on the protocol!

Peer-to-Peer LL

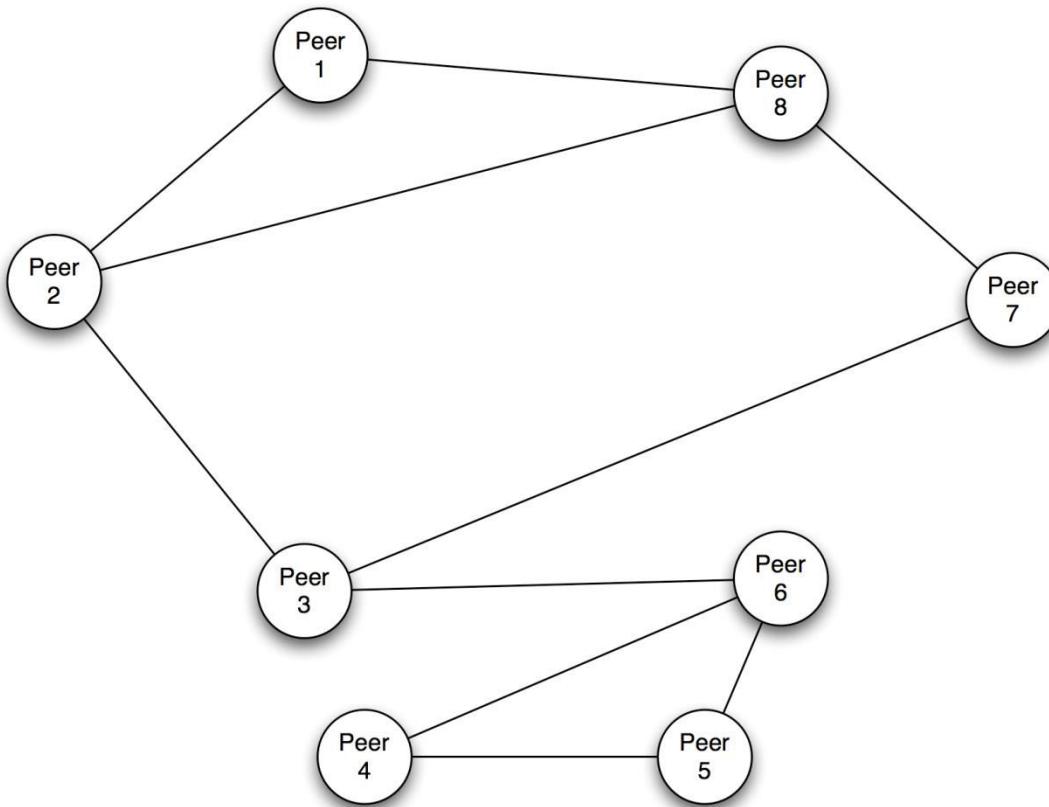


Napster ("I am the Napster!")

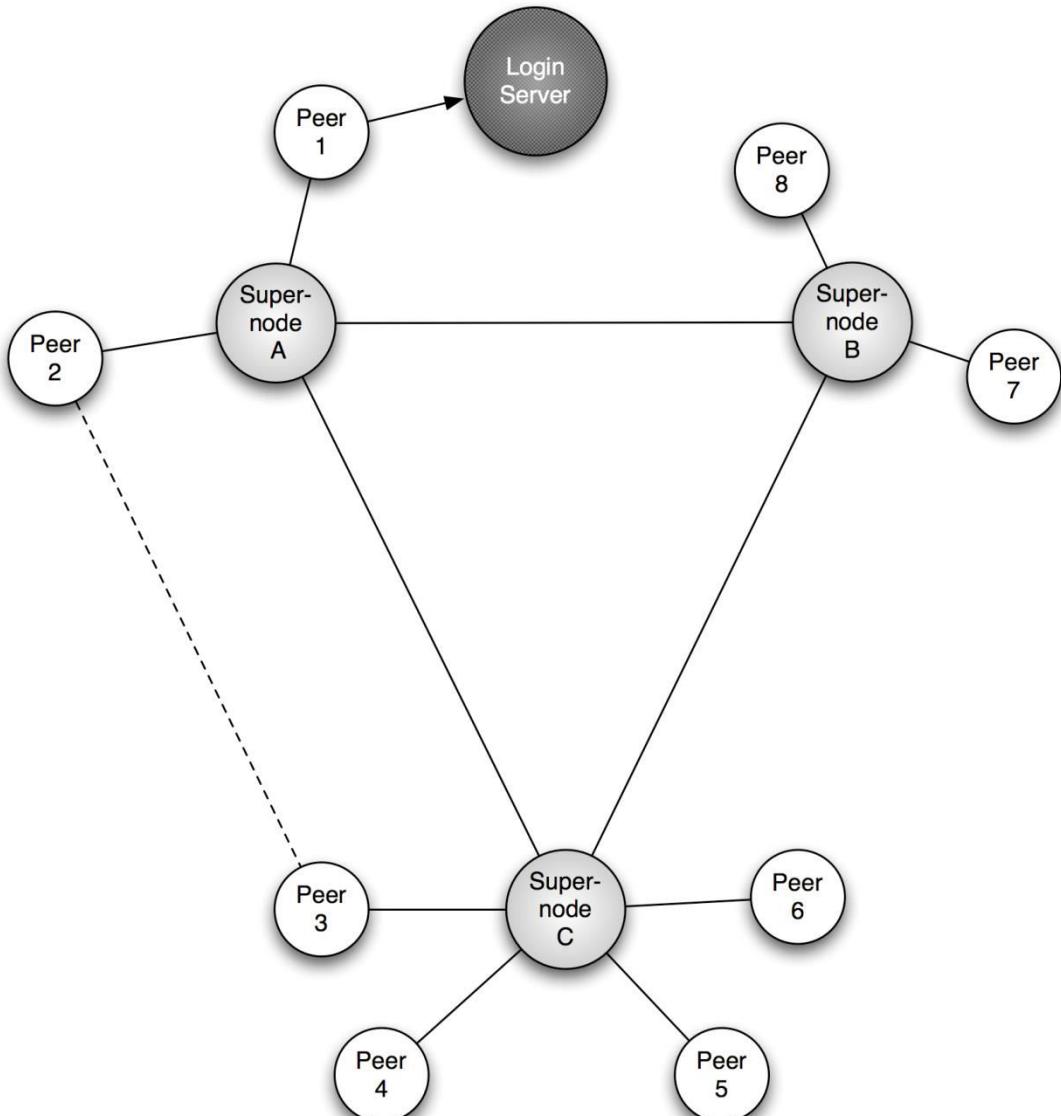
--Lyle, "The Italian Job" (2003)



Gnutella (original)



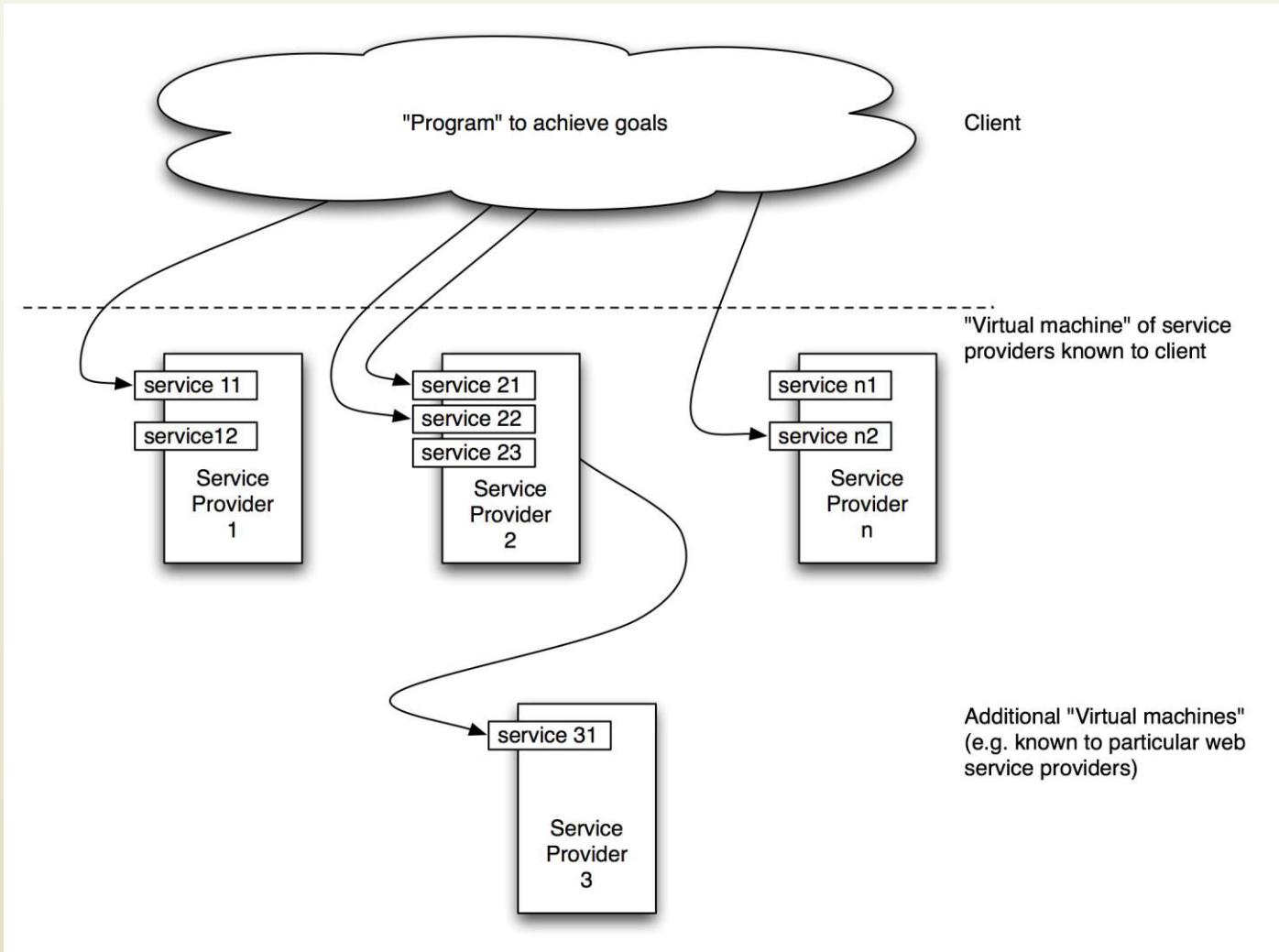
Skype



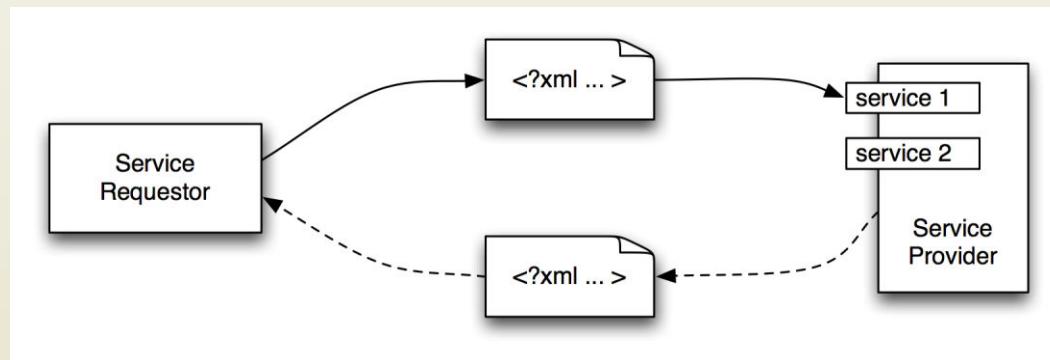
Insights from Skype

- A mixed client-server and peer-to-peer architecture addresses the discovery problem.
- Replication and distribution of the directories, in the form of supernodes, addresses the scalability problem and robustness problem encountered in Napster.
- Promotion of ordinary peers to supernodes based upon network and processing capabilities addresses another aspect of system performance: “not just any peer” is relied upon for important services.
- A proprietary protocol employing encryption provides privacy for calls that are relayed through supernode intermediaries.
- Restriction of participants to clients issued by Skype, and making those clients highly resistant to inspection or modification, prevents malicious clients from entering the network.

Web Services



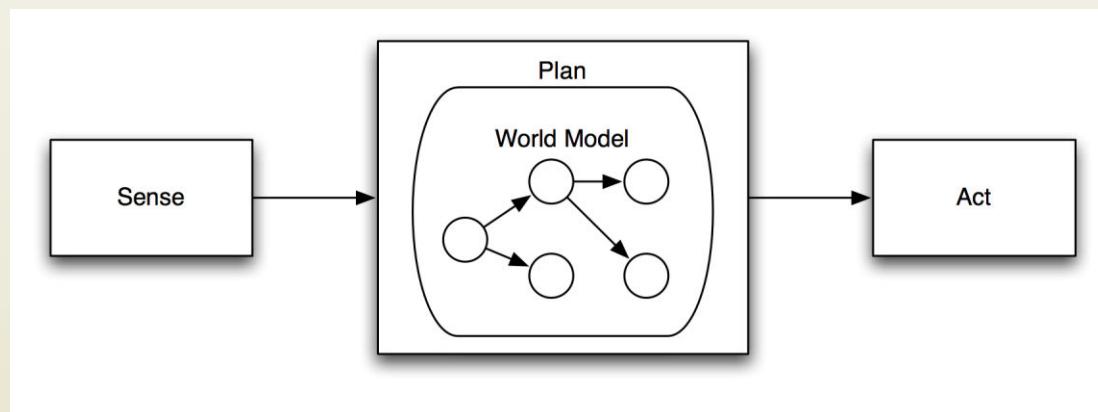
Web Services (cont'd)



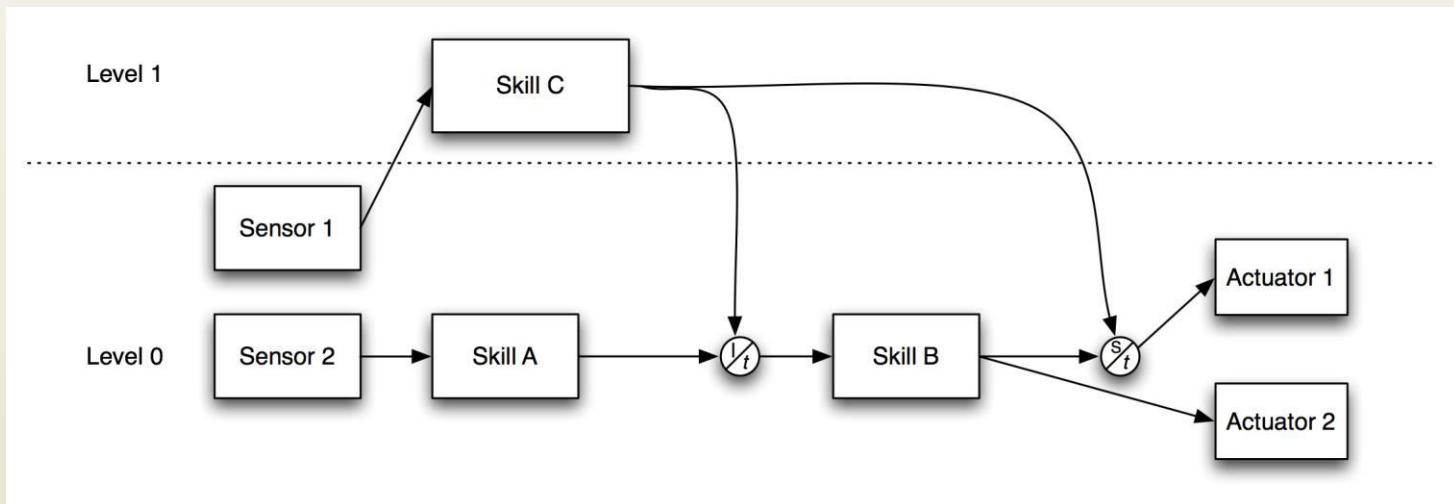
Mobile Robotics

- Manned or partially manned vehicles
- Uses
 - ◆ Space exploration
 - ◆ Hazardous waste disposal
 - ◆ Underwater exploration
- Issues
 - ◆ Interface with external sensors & actuators
 - ◆ Real-time response to stimuli
 - ◆ Response to obstacles
 - ◆ Sensor input fidelity
 - ◆ Power failures
 - ◆ Mechanical limitations
 - ◆ Unpredictable events

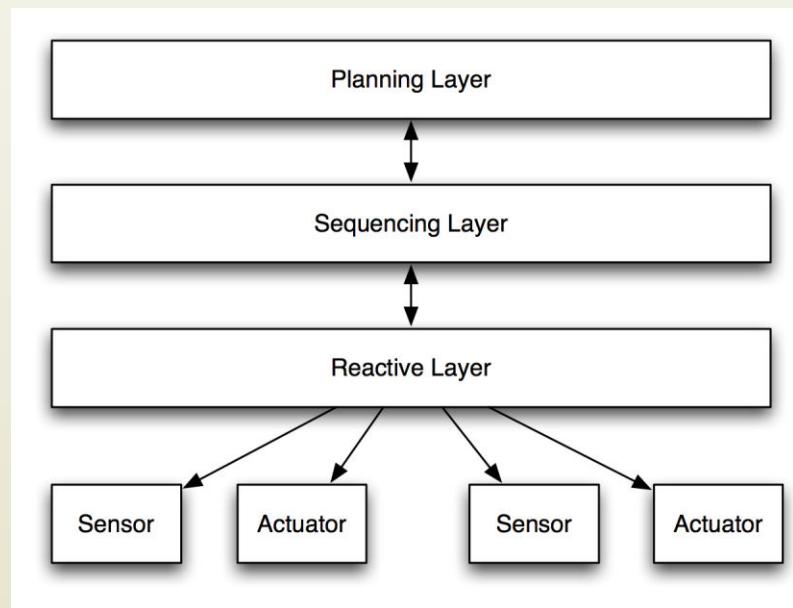
Robotics: Sense-Plan-Act



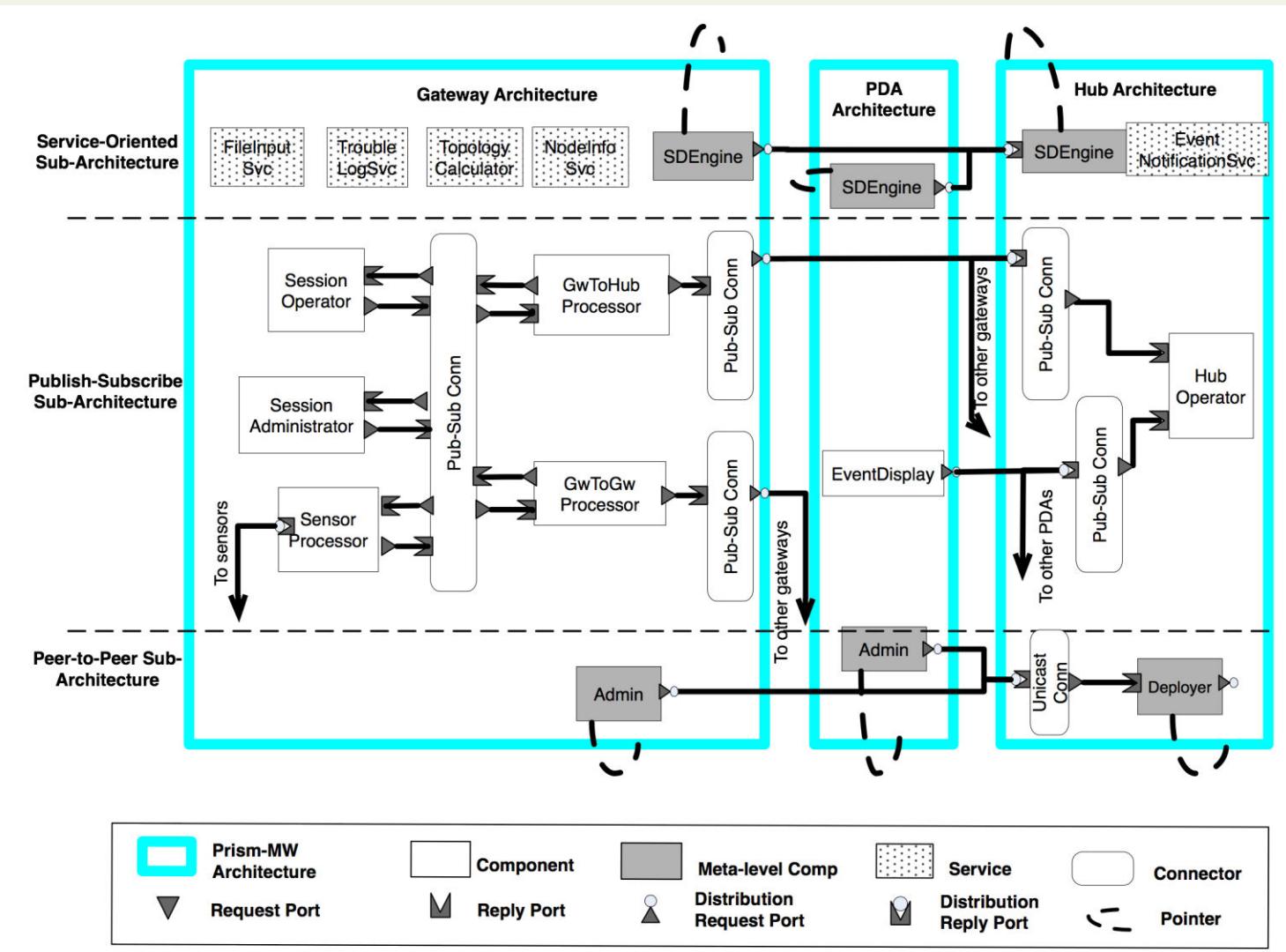
Robotics Subsumption Architecture



Robotics: Three-Layer



Wireless Sensor Networks



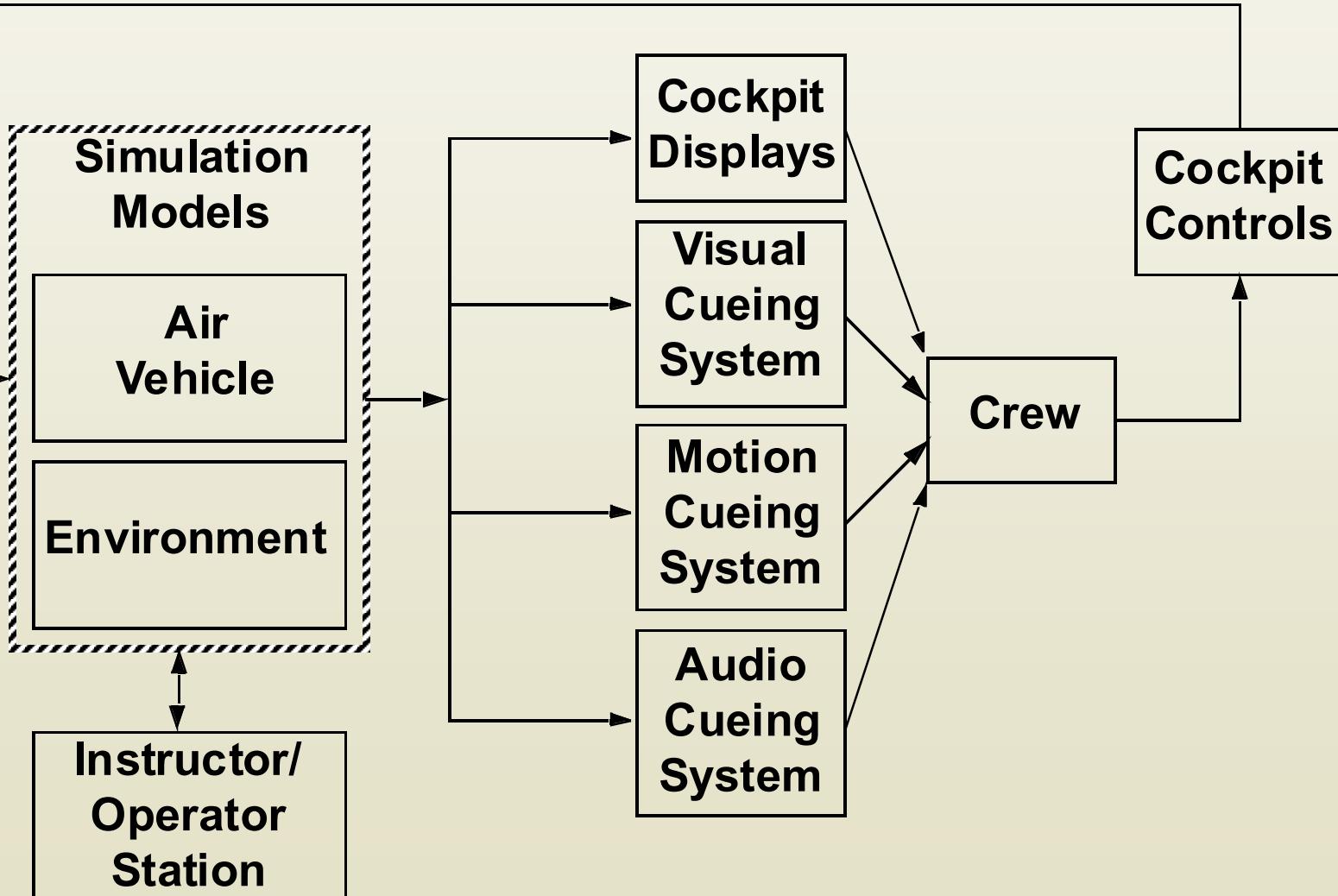
Flight Simulators: Key Characteristics

- Real-time performance constraints
 - ◆ Extremely high fidelity demands
 - Requires distributed computing
 - Must execute at high fixed frame rates
 - ◆ Often called harmonic frequencies
 - e.g. often 60Hz, 30Hz; some higher (100Hz)
 - ◆ Coordination across simulator components
 - All portions of simulator run at integral multiple of base rate
 - ◆ e.g. if base rate = 60Hz, then 30Hz, 15Hz, 12Hz, etc.
 - Every task must complete on time
 - ◆ Delays can cause simulator sickness

Flight Simulators: Key Characteristics (cont'd)

- Continuous evolution
- Very large & complex
 - ◆ Millions of SLOC
 - ◆ Exponential growth over lifetime of simulator
- Distributed development
 - ◆ Portions of work subcontracted to specialists
 - ◆ Long communication paths increase integration complexity
- Expensive verification & validation
 - ◆ Direct by-product of above characteristics
- Mapping of Simulation SW to HW components unclear
 - ◆ Efficiency muddies abstraction
 - ◆ Needed because of historical HW limitations

Functional Model



How Is the Complexity Managed?

- New style
 - ◆ “Structural Modeling”
 - ◆ Based on
 - Object-oriented design to model air vehicle's
 - ◆ Sub-systems
 - ◆ Components
 - Real-time scheduling to control execution order
- Goals of Style
 - ◆ Maintainability
 - ◆ Integrability
 - ◆ Scalability

How Is the Complexity Managed? (cont'd)

- Style principles
 - ◆ Pre-defined partitioning of functionality among SW elements
 - ◆ Restricted data- & control-flow
 - Data-flow through export areas only
 - ◆ Decoupling objects
 - ◆ Small number of element types
 - Results in replicated subsystems

How Is the Complexity Managed? (cont'd)

- Style principles (continued)
 - ◆ Objects fully encapsulate their own internal state
 - ◆ No side-effects of computations
 - ◆ Narrow set of system-wide coordination strategies
 - Employs pre-defined mechanisms for data & control passing
 - Enable component communication & synchronization

F-Sim In The Structural Modeling Style

- Five basic computational elements in two classes
 - ◆ Executive (or infrastructure)
 - Periodic sequencer
 - Event handler
 - Synchronizer
 - ◆ Application
 - Components
 - Subsystems
- Each of the five has
 - ◆ Small API
 - ◆ Encapsulated state
 - ◆ Severely limited external data upon which it relies

Level-0 Architecture

Executive Level

Event Handler

Periodic Sequencer

Synchronizer

coordination

Subsystem Level

Controller

Export Area

Data Gatherer

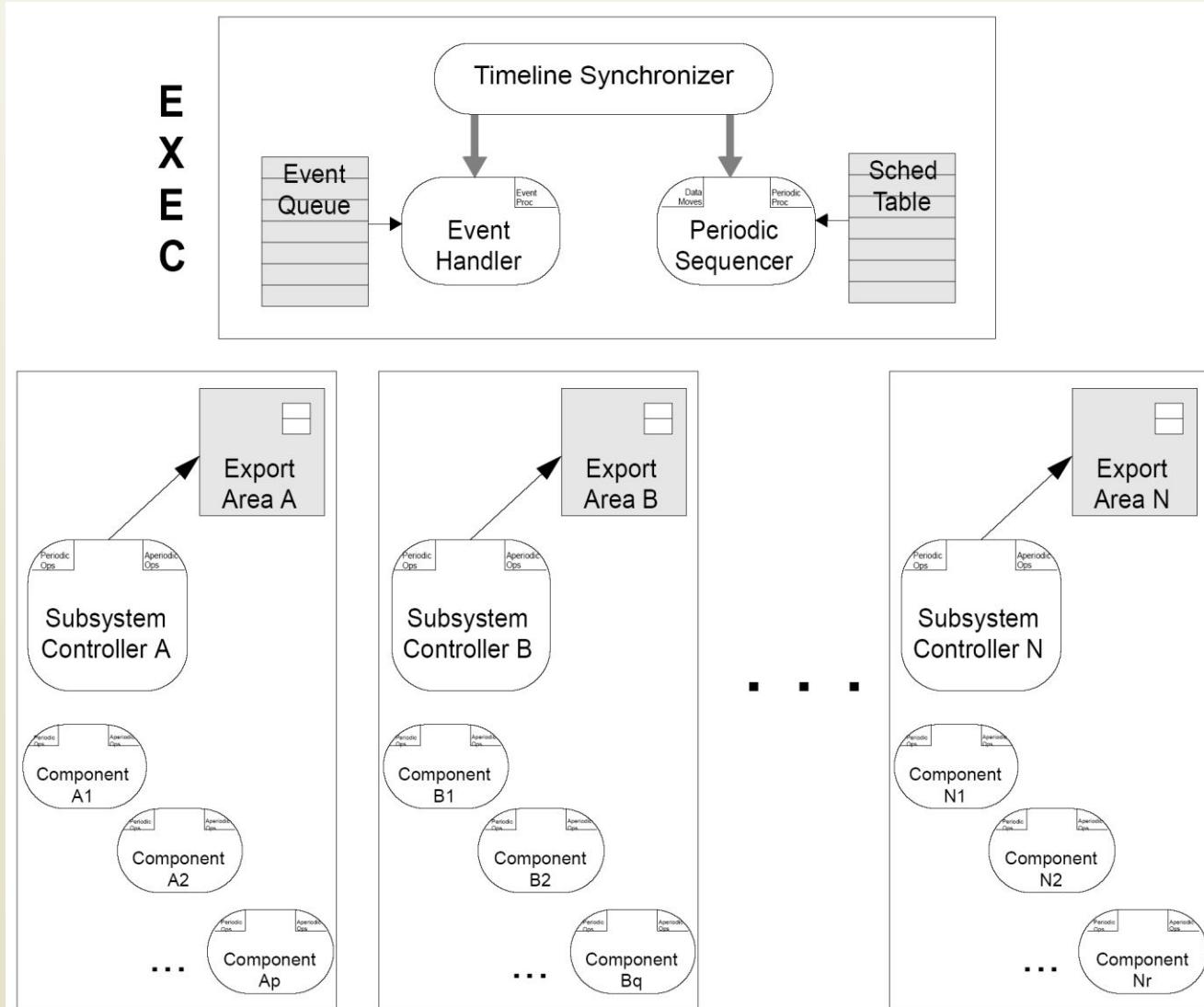
coordination and coupling

Component Level

Component

computation

Level-1 Architecture



Takeaways

- A great architecture is the ticket to runaway success
- A great architecture reflects deep understanding of the problem domain
- A great architecture probably combines aspects of several simpler architectures
- Develop a new architectural style with great care and caution. Most likely you don't need a new style.

Designing for NFPs

Software Architecture
Lecture 19

What Is an NFP?

- A software system's **non-functional property (NFP)** is a constraint on the manner in which the system implements and delivers its functionality
- Example NFPs
 - ◆ Efficiency
 - ◆ Complexity
 - ◆ Scalability
 - ◆ Heterogeneity
 - ◆ Adaptability
 - ◆ Dependability

Designing for FPs

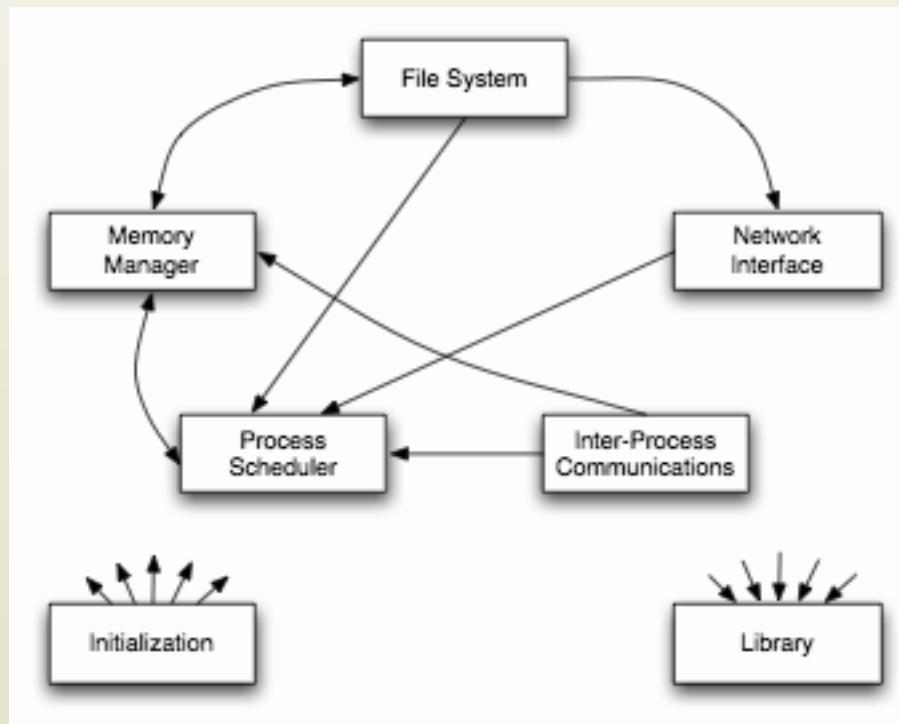
- Any engineering product is sold based on its functional properties (FPs)
 - ◆ TV set, DVD player, stereo, mobile telephone
- Providing the desired functionality is often quite challenging
 - ◆ Market demands
 - ◆ Competition
 - ◆ Strict deadlines
 - ◆ Limited budgets
- However, the system's success will ultimately rest on its NFPs
 - ◆ "This system is too slow!"
 - ◆ "It keeps crashing!"
 - ◆ "It has so many security holes!"
 - ◆ "Every time I change this feature I have to reboot!"
 - ◆ "I can't get it to work with my home theater!"

FPs vs. NFPs – An Example

- Microsoft Word 6.0
 - ◆ Released in the 1990s
 - ◆ Both for the PC and the Mac
 - ◆ Roughly the same functionality
 - ◆ It ran fine on the PC and was successful
 - ◆ It was extremely slow on the Mac
 - ◆ Microsoft “solved” the problem by charging customers for **downgrades**
 - ◆ A lot of bad publicity

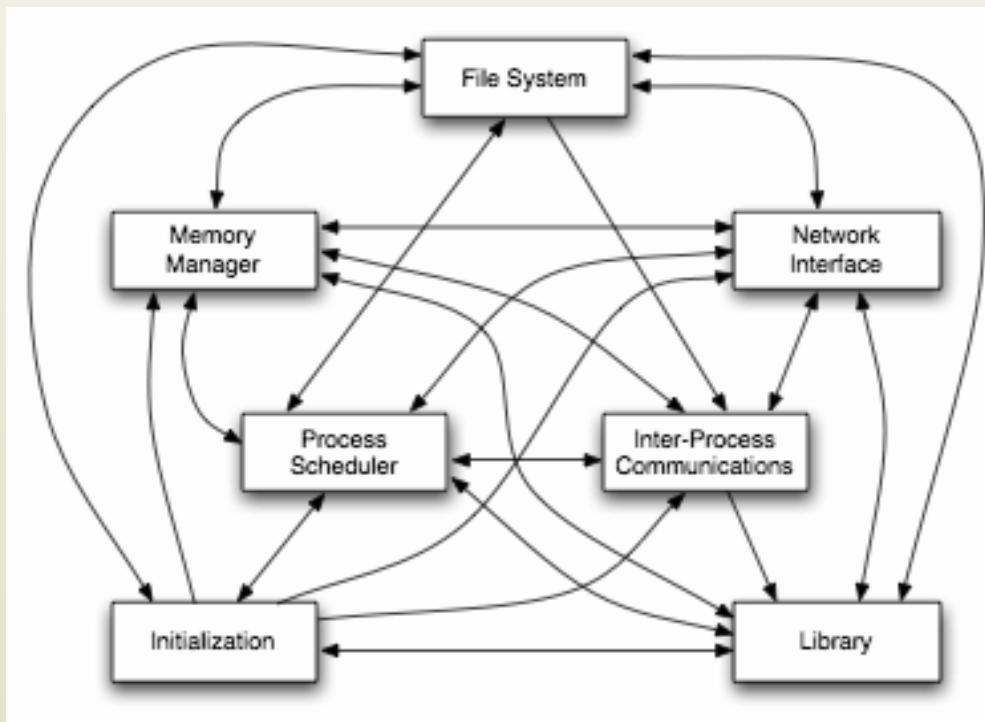
FPs vs. NFPs – Another Example

- Linux – “as-documented” architecture



FPs vs. NFPs – Another Example

- Linux – “as-implemented” architecture



Challenges of Designing for NFPs

- Only partially understood in many domains
 - ◆ E.g., MS Windows and security
- Qualitative vs. quantitative
- Frequently multi-dimensional
- Non-technical pressures
 - ◆ E.g., time-to-market or functional features

Design Guidelines for Ensuring NFPs

- Only guidelines, not laws or rules
- Promise but do not guarantee a given NFP
- Necessary but not sufficient for a given NFP
- Have many caveats and exceptions
- Many trade-offs are involved

Overarching Objective

- Ascertain the role of software architecture in ensuring various NFPs
 - ◆ At the level of major architectural building blocks
 - Components
 - Connectors
 - Configurations
 - ◆ As embodied in architectural style-level design guidelines

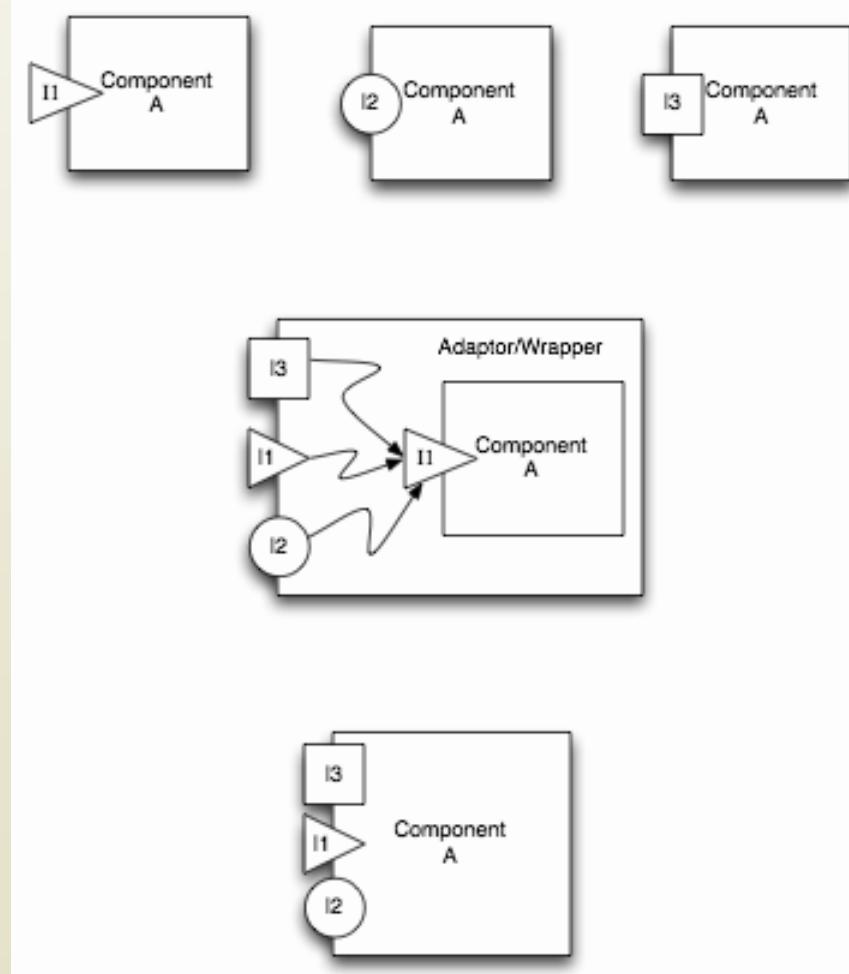
Efficiency

- **Efficiency** is a quality that reflects a software system's ability to meet its performance requirements while minimizing its usage of the resources in its computing environment
 - ◆ Efficiency is a measure of a system's resource usage *economy*
- What can software architecture say about efficiency?
 - ◆ Isn't efficiency an implementation-level property?
- *Efficiency starts at the architectural level!*

Software Components and Efficiency

- Keep the components “small” whenever possible
- Keep component interfaces simple and compact
- Allow multiple interfaces to the same functionality
- Separate data components from processing components
- Separate data from meta-data

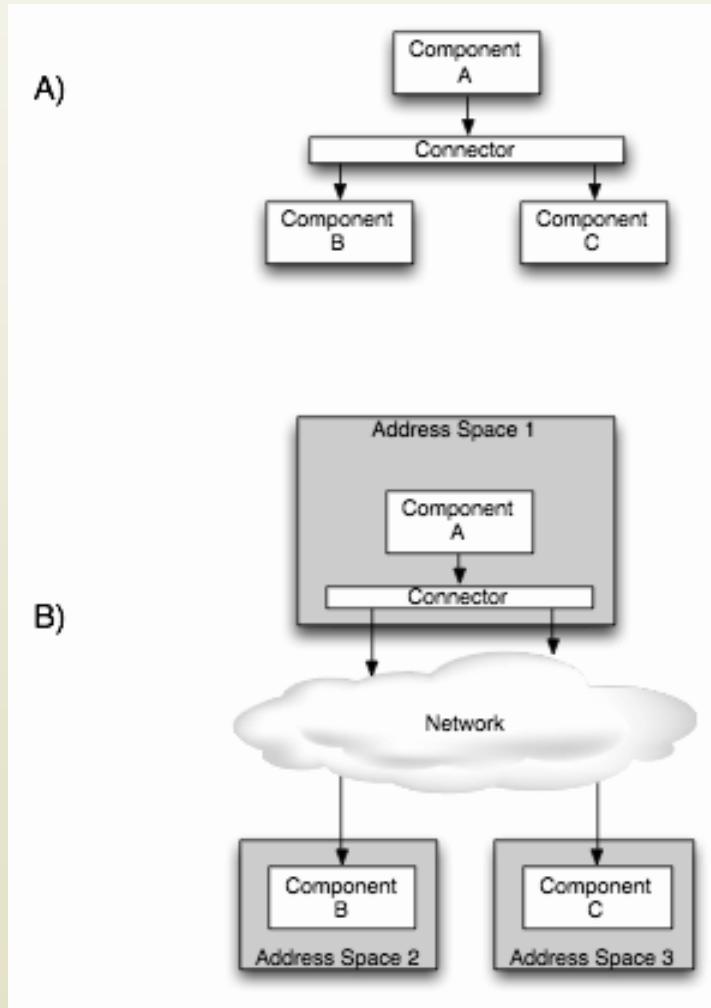
Multiple Interfaces to the Same Functionality



Software Connectors and Efficiency

- Carefully select connectors
- Use broadcast connectors with caution
- Make use of asynchronous interaction whenever possible
- Use location/distribution transparency judiciously

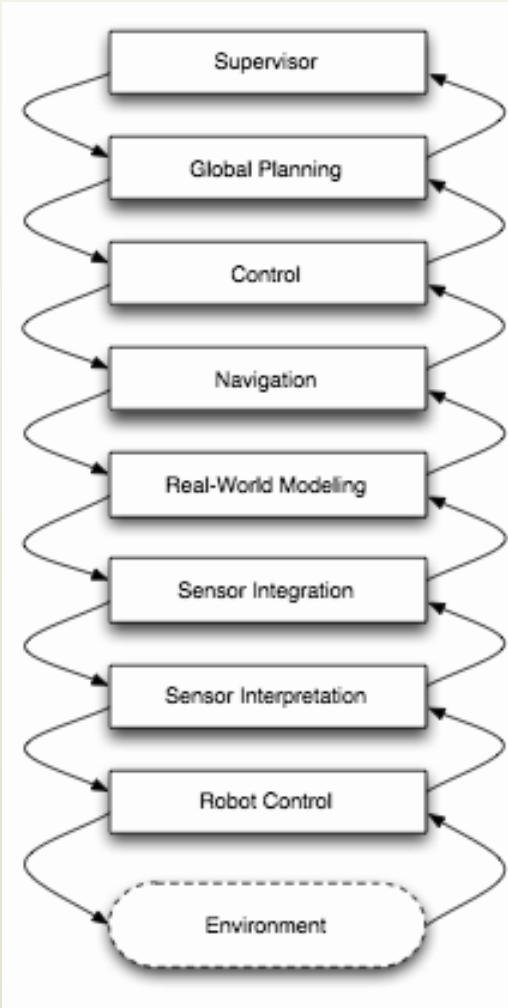
Distribution Transparency



Architectural Configurations and Efficiency

- Keep frequently interacting components “close”
- Carefully select and place connectors in the architecture
- Consider the efficiency impact of selected architectural styles and patterns

Performance Penalty Induced by Distance



NFP Design Techniques

**Software Architecture
Lecture 20**

Complexity

- IEEE Definition
 - ◆ **Complexity** is the degree to which a software system or one of its components has a design or implementation that is difficult to understand and verify
- **Complexity** is a software system's a property that is directly proportional to the size of the system, number of its constituent elements, their internal structure, and the number and nature of their interdependencies

Software Components and Complexity

- Separate concerns into different components
- Keep only the functionality inside components
 - ◆ Interaction goes inside connectors
- Keep components cohesive
- Be aware of the impact of off-the-shelf components on complexity
- Insulate processing components from changes in data format

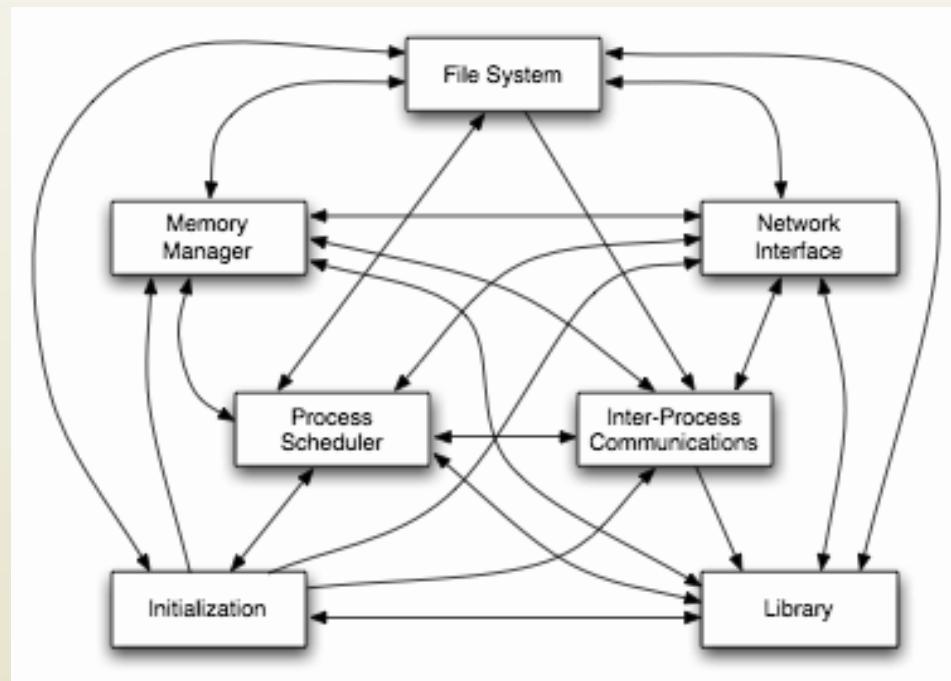
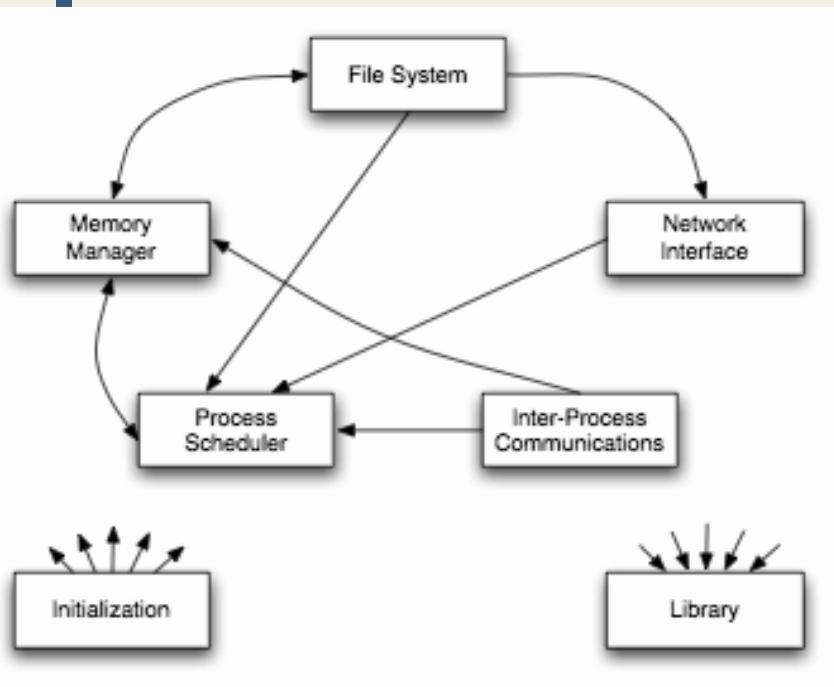
Software Connectors and Complexity

- Treat connectors explicitly
- Keep only interaction facilities inside connectors
- Separate interaction concerns into different connectors
- Restrict interactions facilitated by each connector
- Be aware of the impact of off-the-shelf connectors on complexity

Architectural Configurations and Complexity

- Eliminate unnecessary dependencies
- Manage all dependencies explicitly
- Use hierarchical (de)composition

Complexity in Linux



Scalability and Heterogeneity

- **Scalability** is the capability of a software system to be adapted to meet new requirements of size and scope
- **Heterogeneity** is the quality of a software system consisting of multiple disparate constituents or functioning in multiple disparate computing environments
 - ◆ **Heterogeneity** is a software system's ability to consist of multiple disparate constituents or function in multiple disparate computing environments
 - ◆ **Portability** is a software system's ability to execute on multiple platforms with minimal modifications and without significant degradation in functional or non-functional characteristics

Software Components and Scalability

- Give each component a single, clearly defined purpose
- Define each component to have a simple, understandable interface
- Do not burden components with interaction responsibilities
- Avoid unnecessary heterogeneity
 - ◆ Results in architectural mismatch
- Distribute the data sources
- Replicate data when necessary

Software Connectors and Scalability

- Use explicit connectors
- Give each connector a clearly defined responsibility
- Choose the simplest connector suited for the task
- Be aware of differences between direct and indirect dependencies
- Avoid placing application functionality inside connectors
 - ◆ Application functionality goes inside components
- Leverage explicit connectors to support data scalability

Architectural Configurations and Scalability

- Avoid system bottlenecks
- Make use of parallel processing capabilities
- Place the data sources close to the data consumers
- Try to make distribution transparent
- Use appropriate architectural styles

Adaptability

- **Adaptability** is a software system's ability to satisfy new requirements and adjust to new operating conditions during its lifetime

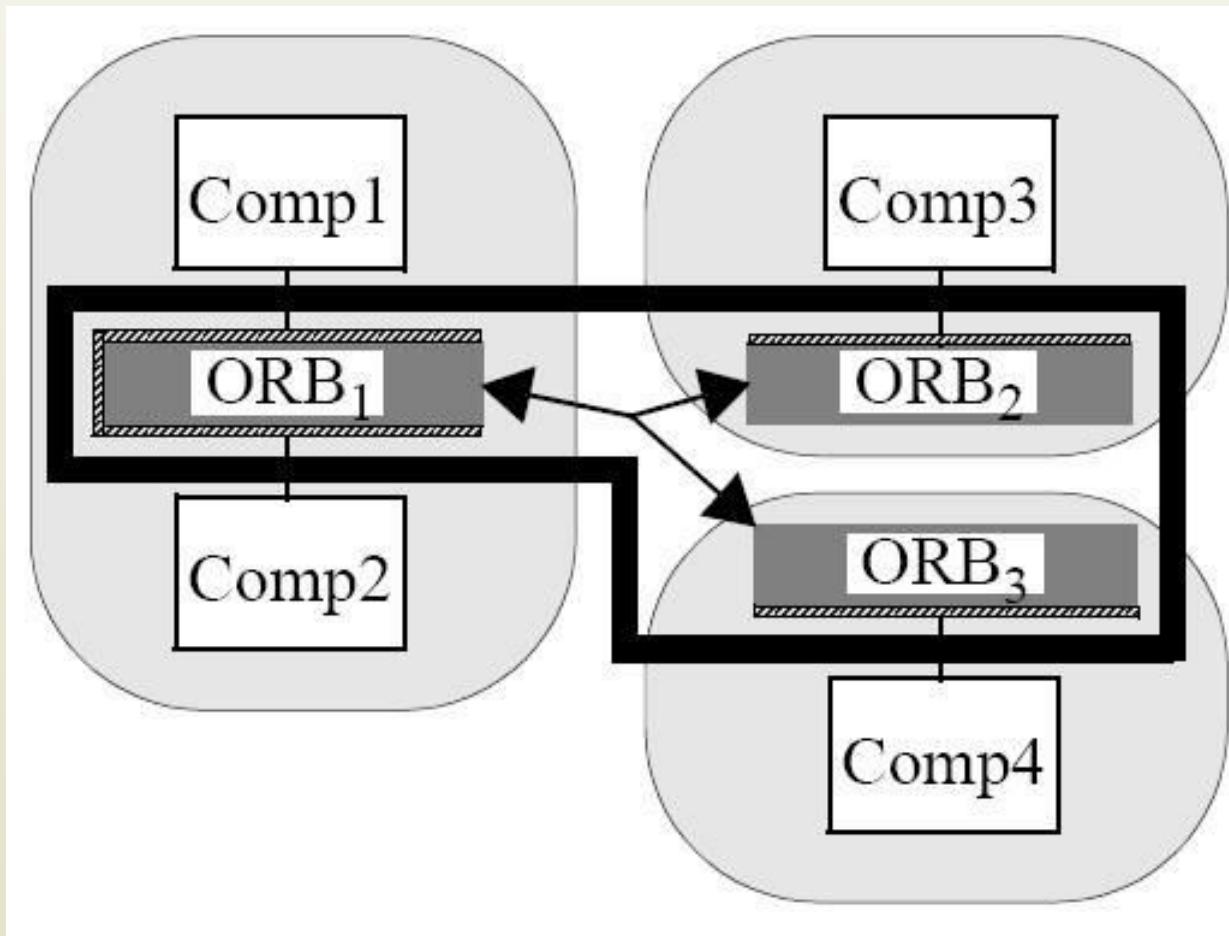
Software Components and Adaptability

- Give each component a single, clearly defined purpose
- Minimize component interdependencies
- Avoid burdening components with interaction responsibilities
- Separate processing from data
- Separate data from metadata

Software Connectors and Adaptability

- Give each connector a clearly defined responsibility
- Make the connectors flexible
- Support connector composability

Composable Connectors



Architectural Configurations and Adaptability

- Leverage explicit connectors
- Try to make distribution transparent
- Use appropriate architectural styles

Dependability

- Dependability is a collection of system properties that allows one to rely on a system functioning as required
 - ◆ **Reliability** is the probability that a system will perform its intended functionality under specified design limits, without failure, over a given time period
 - ◆ **Availability** is the probability that a system is operational at a particular time
 - ◆ **Robustness** is a system's ability to respond adequately to unanticipated runtime conditions
 - ◆ **Fault-tolerant** is a system's ability to respond gracefully to failures at runtime
 - ◆ **Survivability** is a system's ability to resist, recognize, recover from, and adapt to mission-compromising threats
 - ◆ **Safety** denotes the ability of a software system to avoid failures that will result in (1) loss of life, (2) injury, (3) significant damage to property, or (4) destruction of property

Software Components and Dependability

- Carefully control external component inter-dependencies
- Provide reflection capabilities in components
- Provide suitable exception handling mechanisms
- Specify the components' key state invariants

Software Connectors and Dependability

- Employ connectors that strictly control component dependencies
- Provide appropriate component interaction guarantees
- Support dependability techniques via advanced connectors

Architectural Configurations and Dependability

- Avoid single points of failure
- Provide back-ups of critical functionality and data
- Support non-intrusive system health monitoring
- Support dynamic adaptation

Security and Trust

Software Architecture
Lecture 21

Outline

- Security
- Design Principles
- Architectural Access Control
 - ◆ Access Control Models
 - ◆ Connector-Centric Architectural Access Control
- Trust
- Trust Model
 - ◆ Reputation-based Systems
 - ◆ Architectural Approach to Decentralized Trust Management

Security

- “The protection afforded to an automated information system in order to attain the applicable objectives of preserving the **integrity**, **availability** and **confidentiality** of information system resources (includes hardware, software, firmware, information/data, and telecommunications).”
 - ◆ National Institute of Standards and Technology

Confidentiality, Integrity, and Availability

- Confidentiality
 - ◆ Preserving the **confidentiality** of information means preventing unauthorized parties from accessing the information or perhaps even being aware of the existence of the information. I.e., secrecy.
- Integrity
 - ◆ Maintaining the **integrity** of information means that only authorized parties can manipulate the information and do so only in authorized ways.
- Availability
 - ◆ Resources are **available** if they are accessible by authorized parties on all appropriate occasions.

Design Principles for Computer Security

- **Least Privilege:** give each component only the privileges it requires
- **Fail-safe Defaults:** deny access if explicit permission is absent
- **Economy of Mechanism:** adopt simple security mechanisms
- **Complete Mediation:** ensure every access is permitted
- **Design:** do not rely on secrecy for security

Design Principles for Computer Security (cont'd)

- **Separation of Privilege**: introduce multiple parties to avoid exploitation of privileges
- **Least Common Mechanism**: limit critical resource sharing to only a few mechanisms
- **Psychological Acceptability**: make security mechanisms usable
- **Defense in Depth**: have multiple layers of countermeasures

Security for Microsoft IIS

POTENTIAL PROBLEM	PROTECTION MECHANISM	DESIGN PRINCIPLES
The underlying dll (ntdll.dll) was not vulnerable because...	Code was made more conservative during the Security Push.	Check precondition
Even if it were vulnerable...	Internet Information Services (IIS) 6.0 is not running by default on Windows Server 2003.	Secure by default
Even if it were running...	IIS 6.0 does not have WebDAV enabled by default.	Secure by default
Even if Web-based Distributed Authoring and Versioning (WebDAV) had been enabled...	The maximum URL length in IIS 6.0 is 16 Kbytes by default (> 64 Kbytes needed for the exploit).	Tighten precondition, secure by default
Even if the buffer were large enough...	The process halts rather than executes malicious code due to buffer-overrun detection code inserted by the compiler.	Tighten postcondition, check precondition
Even if there were an exploitable buffer overrun...	It would have occurred in w3wp.exe, which is running as a network service (rather than as administrator).	Least privilege (Data courtesy of David Aucsmith.)

--from [Wing, 2003]

Architectural Access Control Models

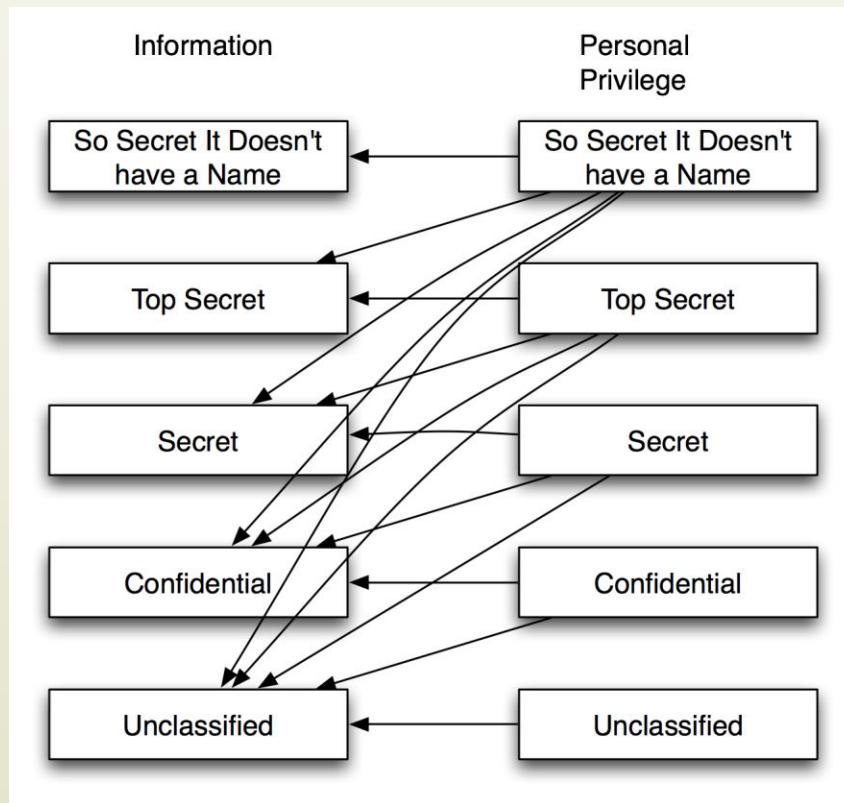
- Decide whether access to a protected resource should be granted or denied
- Discretionary access control
 - ◆ Based on the identity of the requestor, the resource, and whether the requestor has permission to access
- Mandatory access control
 - ◆ Policy based

Discretionary Access Control

	Database A	Component Q	Interface F
<i>Alice</i>	Read-Write; Always	Bend	Yes
<i>Bob</i>	Read-Write; Between 9 and 5	Fold	No
<i>Charles</i>	No access	Spindle	No
<i>Dave</i>	No access	Mutilate	Yes
<i>Eve</i>	Read-only; Always	None	No

Mandatory Access Control

- Bob: Secret
- Alice: Confidential
- Tom: Top Secret



Arrows show access (read/write) privileges
What about just appending?

Connector-Centric Architectural Access Control

- Decide what subjects the connected components are executing for
- Regulate whether components have sufficient privileges to communicate through the connectors
- Provide secure interaction between insecure components
- Propagate privileges in architectural access check
- Participate in deciding architectural connections
- Route messages according to established policies

Static analysis of architectures coupled with dynamic checking

Decentralization

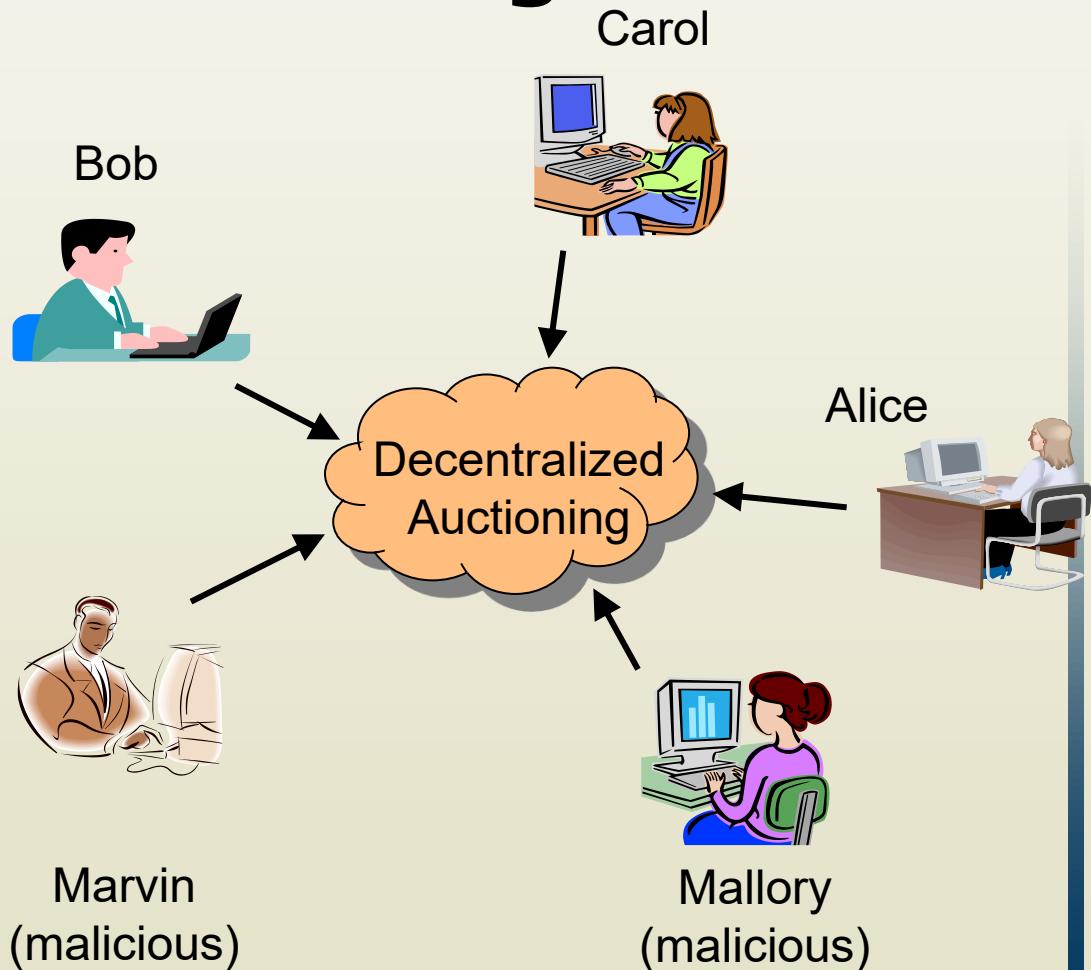
- No centralized authority to coordinate and control entities
- Independent peers, with possibly conflicting goals, interact with each other and make local autonomous decisions
- Presence of malicious peers in open decentralized applications
- Need for measures to protect peers against malicious attacks

Some Threats of Decentralization

- Impersonation: Mallory says she is Bob to Alice
- Fraudulent Actions: Mallory doesn't complete transactions
- Misrepresenting Trust: Mallory tells everyone Bob is evil
- Collusion: Mallory and Eve tell everyone Bob is evil
- Addition of Unknowns: Alice has never met Bob
- **Trust management can serve as a potential countermeasure**
 - ◆ **Trust relationships help peers establish confidence in other peers**

Decentralized Auctioning

- Open decentralized application
- Independent buyers/sellers
- Potentially malicious participants
- Need to counter threats



Impersonation

Bob



Bob is reliable and everyone has a good opinion about Bob

Alice

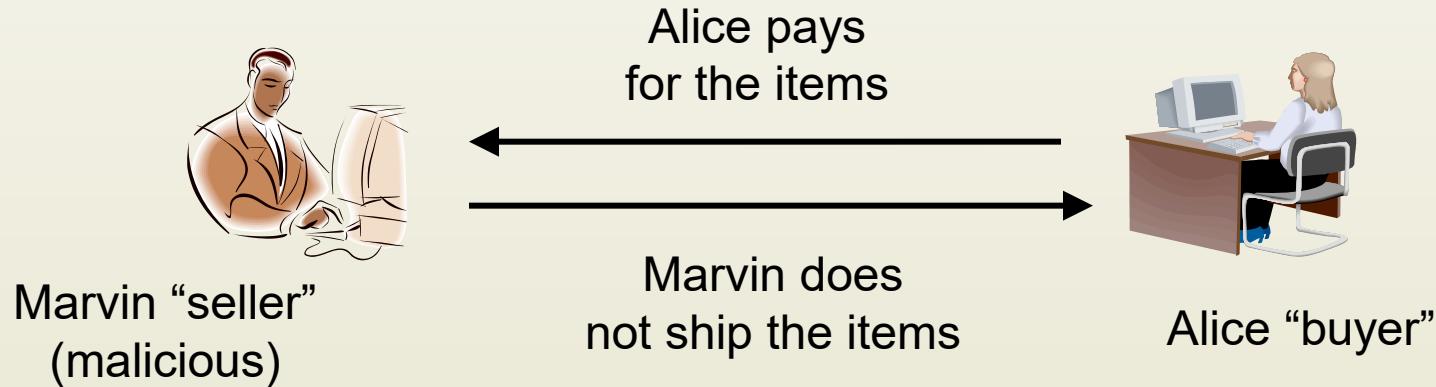


“I am Bob”



Mallory
(malicious)

Fraudulent Actions



Misrepresentation

Bob



Bob is reliable and everyone has a good opinion about Bob

Alice



“Bob is unreliable”



Mallory
(malicious)

Collusion

Bob



Bob is reliable and everyone has a good opinion about Bob

Alice



“Bob is unreliable”



Marvin
(malicious)



Mallory
(malicious)

Addition of Unknowns

Carol
(new entrant in the system)



Bob has no information about Carol; he is not sure whether to interact with Carol

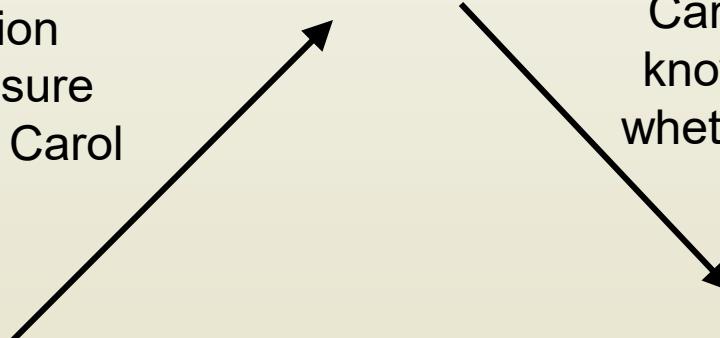


Bob

Carol is new and does not know Alice; she is not sure whether to interact with Alice



Alice



Background: Trust Management

- Trust
 - ◆ Trust is a particular level of the subjective probability with which an agent assesses that another agent will perform a particular action in a context that affects his actions [Gambetta, 1990]
- Reputation
 - ◆ Expectation about an entity's behavior based on past behavior [Abdul-Rahman, 2000]
 - ◆ May be used to determine trust
- Two types of trust management systems
 - ◆ Credential and Policy-based
 - ◆ Reputation-based

Role of Trust Management

- Each entity (peer) must protect itself against these threats
- Trust Management can serve as a potential countermeasure
 - ◆ Trust relationships between peers help establish confidence
- Two types of decentralized trust management systems
 - ◆ Credential and policy-based
 - ◆ Reputation-based

Architecture and Trust Management

- Decentralized trust management has received a lot of attention from researchers [Grandison and Sloman, 2000]
 - ◆ Primary focus has been on developing new models
- **But how does one build a trust-enabled decentralized application?**
 - ◆ **How do I pick a trust model for a given application?**
 - ◆ **And, how do I incorporate the trust model within each entity?**

Approach

- Select a suitable reputation-based trust model for a given application
- Describe this trust model precisely
- Incorporate the model within the structure (architecture) of an entity
 - ◆ Software architectural style for trust management (PACE)
- Result – entity architecture consisting of
 - ◆ components that encapsulate the trust model
 - ◆ additional trust technologies to counter threats

Key Insights

- Trust
 - ◆ Cannot be isolated to one component
 - ◆ Is a dominant concern in decentralized applications and should be considered early on during application development
 - ◆ Having an explicit architecture is one way to consistently address the cross-cutting concern of trust
- Architectural styles
 - ◆ Provide a foundation to reason about specific goals
 - ◆ Facilitate reuse of design knowledge
 - ◆ Allow known benefits to be leveraged and induce desirable properties

Design Guidelines: Approach

- Identify threats of decentralization
- Use the threats to identify guiding principles that help defend against the threats
- Incorporate these principles within an architectural style focused on decentralized trust management

Design Guidelines

Threats	Strategies
Impersonation	Digital identities, signature-based verification
Fraudulent Actions	Explicit trust, comparable trust
Misrepresentation	Explicit trust, comparable trust, separation of internal and external data
Collusion	Explicit trust, comparable trust, separation of internal and external data
Addition of unknowns	Implicit trust of user

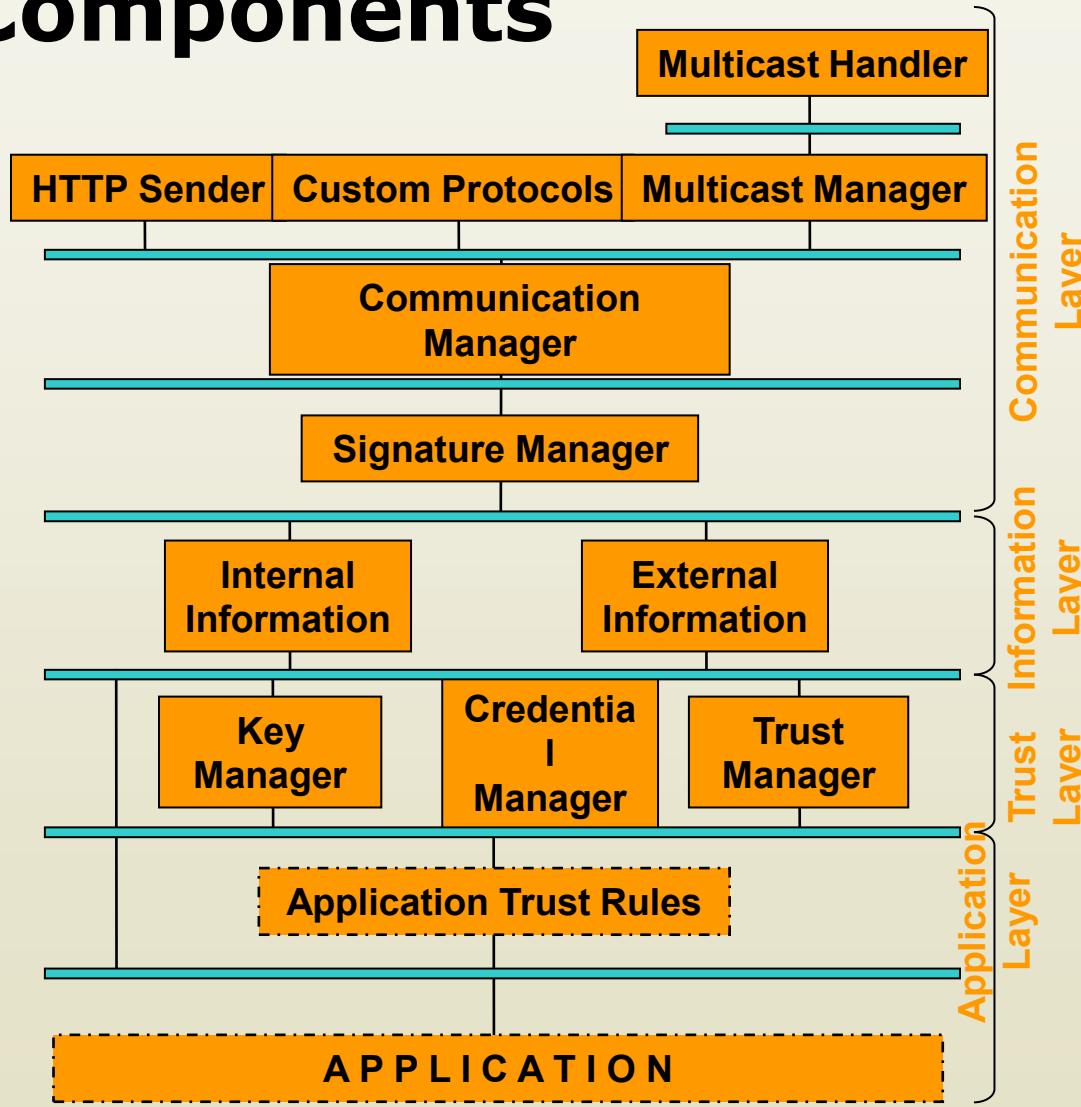
PACE Architectural Style

- Basis: C2, a layered event-based style
 - ◆ Allows the natural structuring of the four functional units according to their dependencies
 - ◆ Facilitates reuse
 - ◆ Extensive tool support
- The resultant architectural style is called PACE (Practical Architectural approach for Composing Egocentric trust)

Functional Units

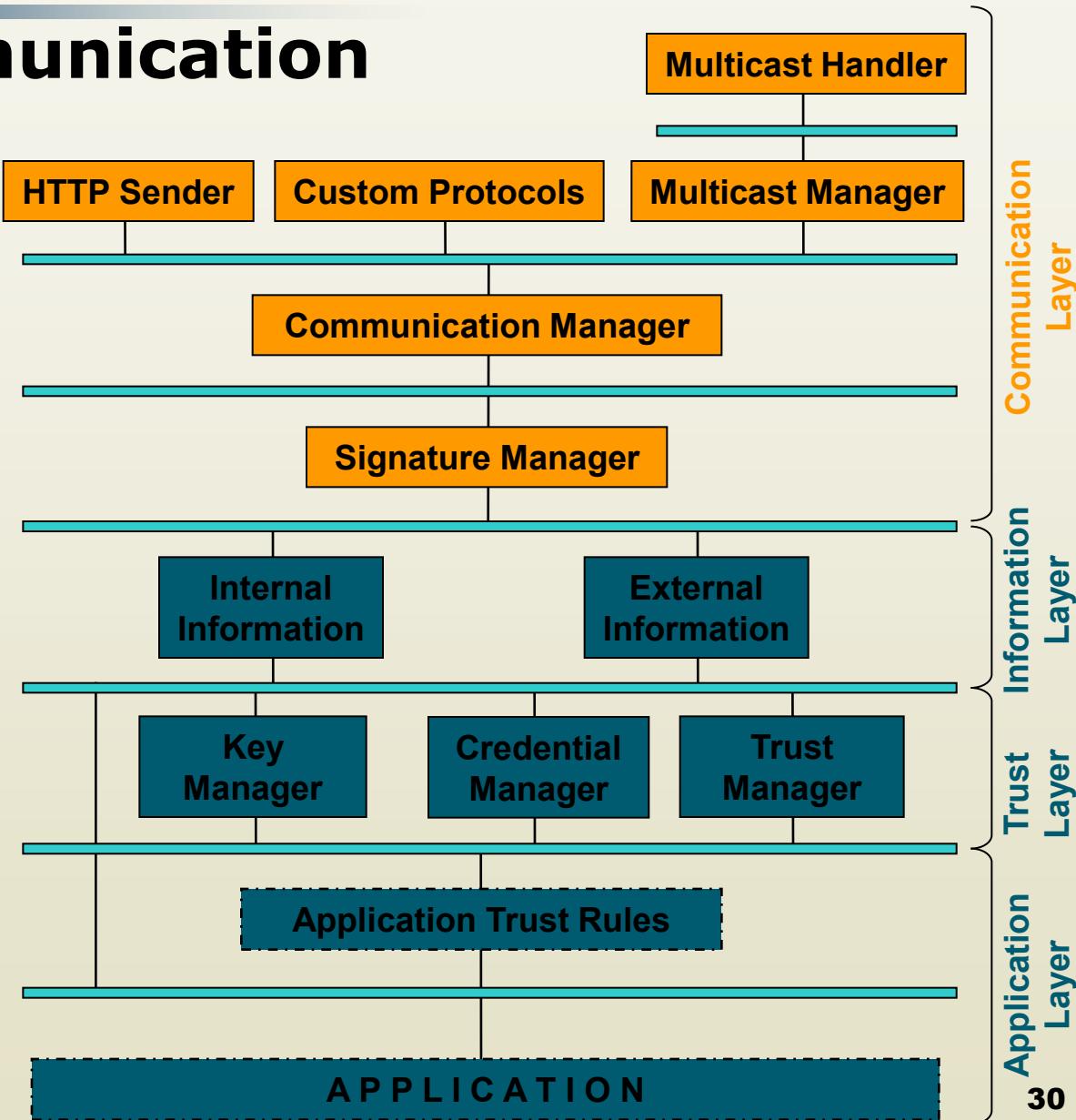
- Communication
 - ◆ Responsible for external interaction with other peers including data collection and transmission; does not depend upon data storage or analysis
- Information
 - ◆ Store all data including internal beliefs and reported information
- Trust
 - ◆ Responsible for trust computation and managing credentials; depends upon internal data for computation
- Application
 - ◆ Application-specific components including user interface; Builds upon services provided by the other three

PACE Components



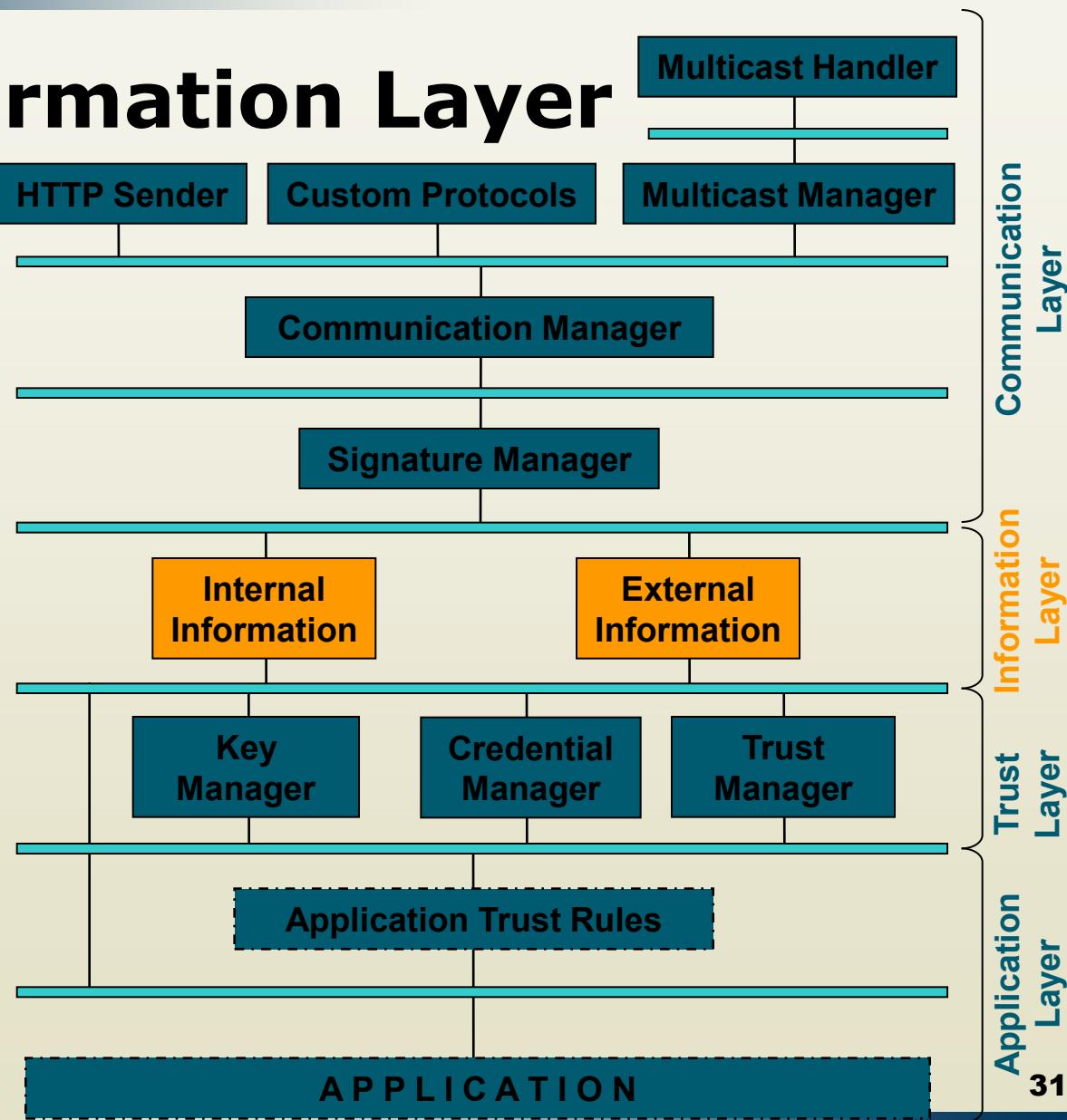
PACE: Communication Layer

- Multiple protocol handlers. Translate internal events into external messages and vice-versa
- Creates and manages protocol handlers
- Signs requests and verifies notifications



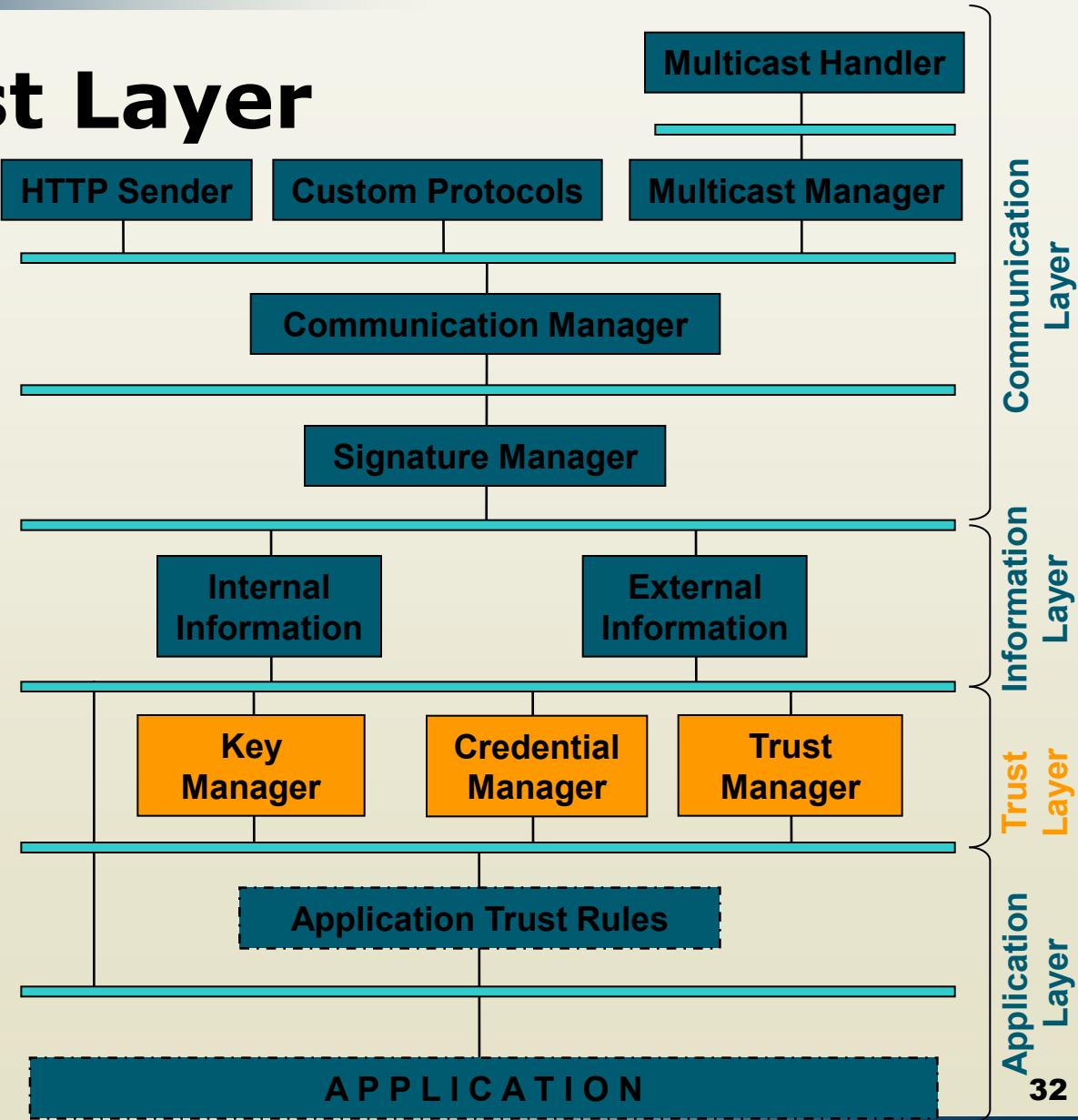
PACE: Information Layer

- Separates internal beliefs from reported information
- Stores internal beliefs persistently



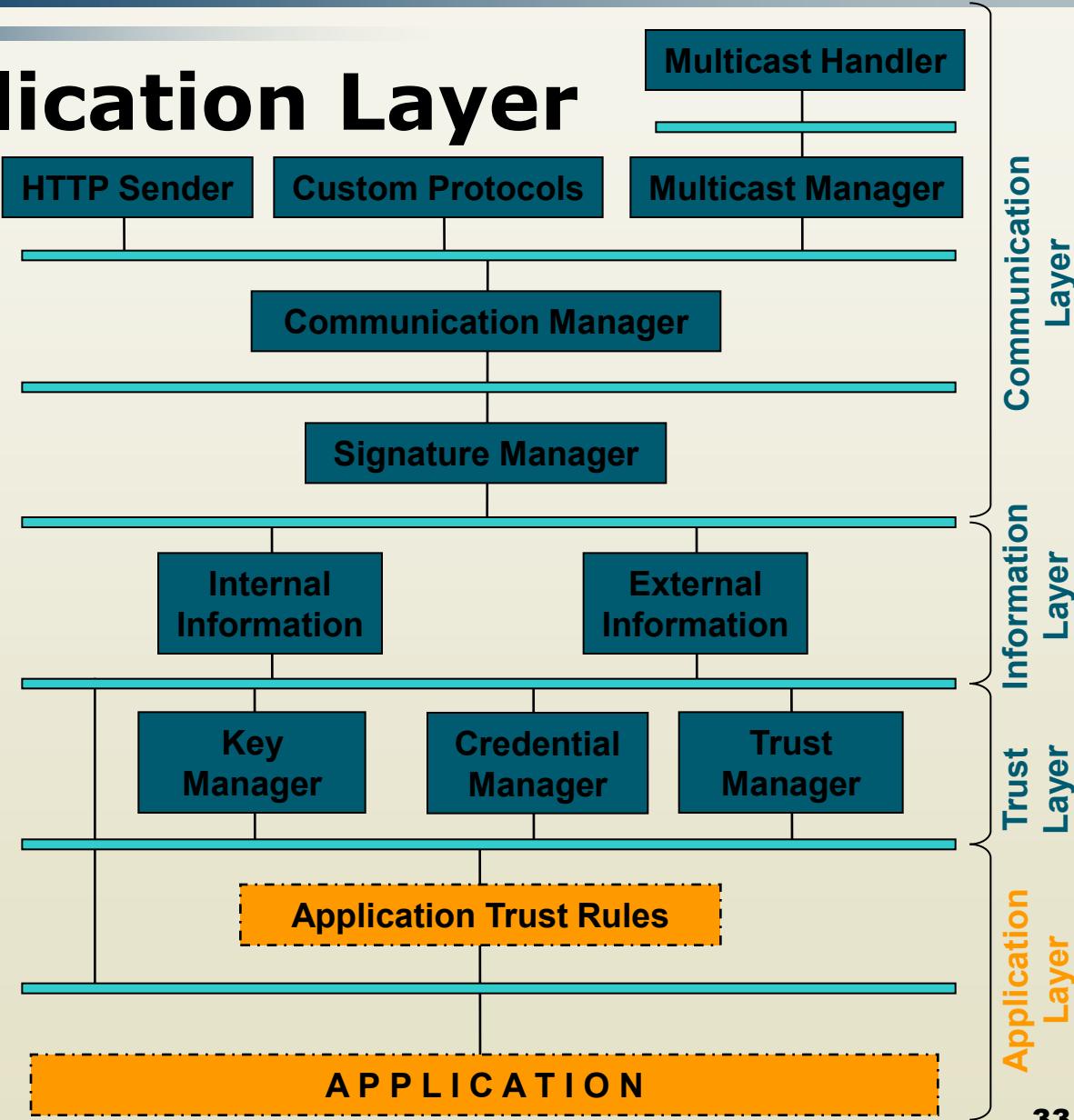
PACE: Trust Layer

- Incorporates different trust models and algorithms; can assign trust values to notifications received
- Generates unique public-private key pairs
- Maintains local cache of other peers' identities; requests public keys from peers and responds to revocations



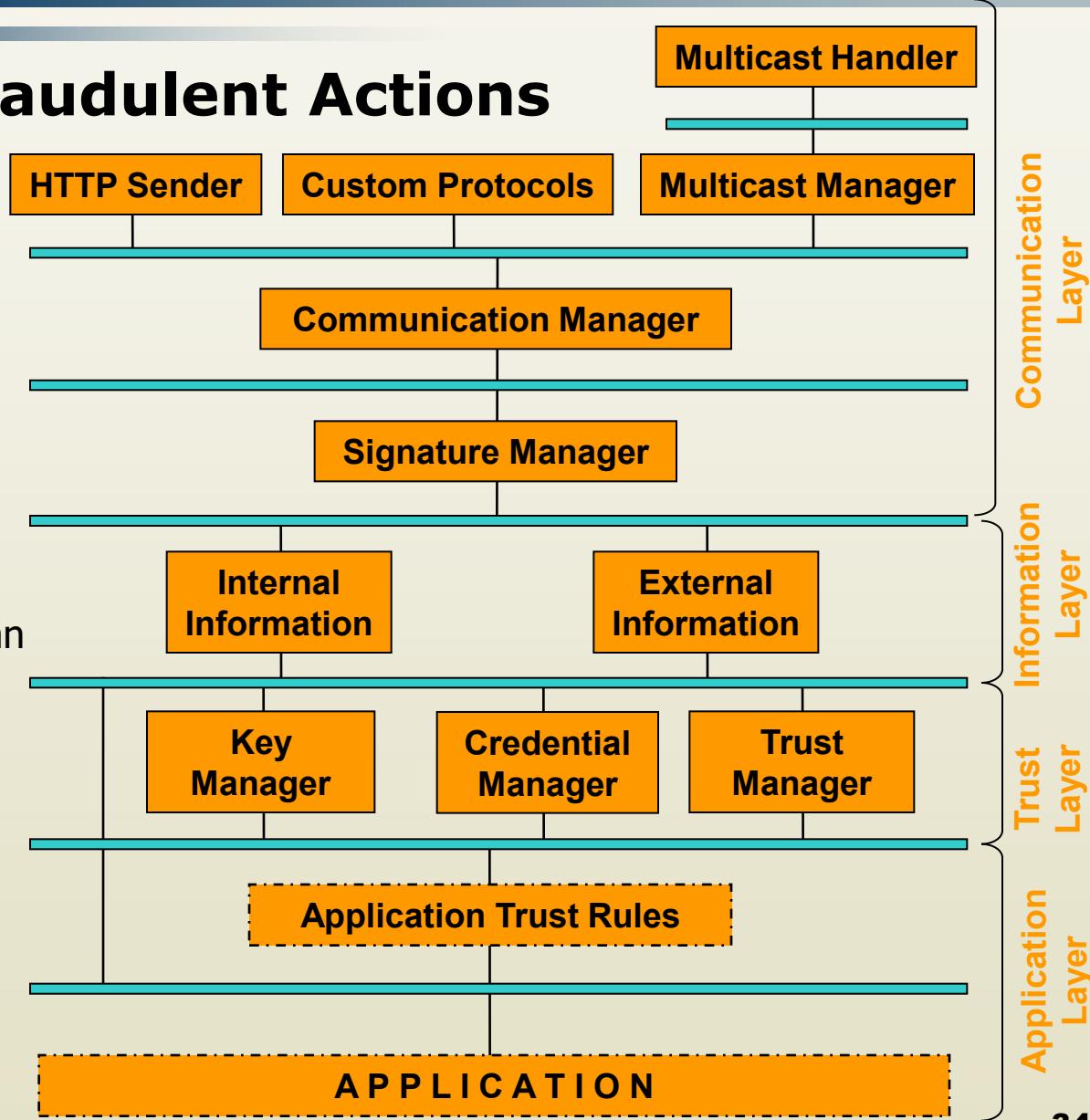
PACE: Application Layer

- Domain-specific trust rules; includes context of trust
- User-interface and application-specific components

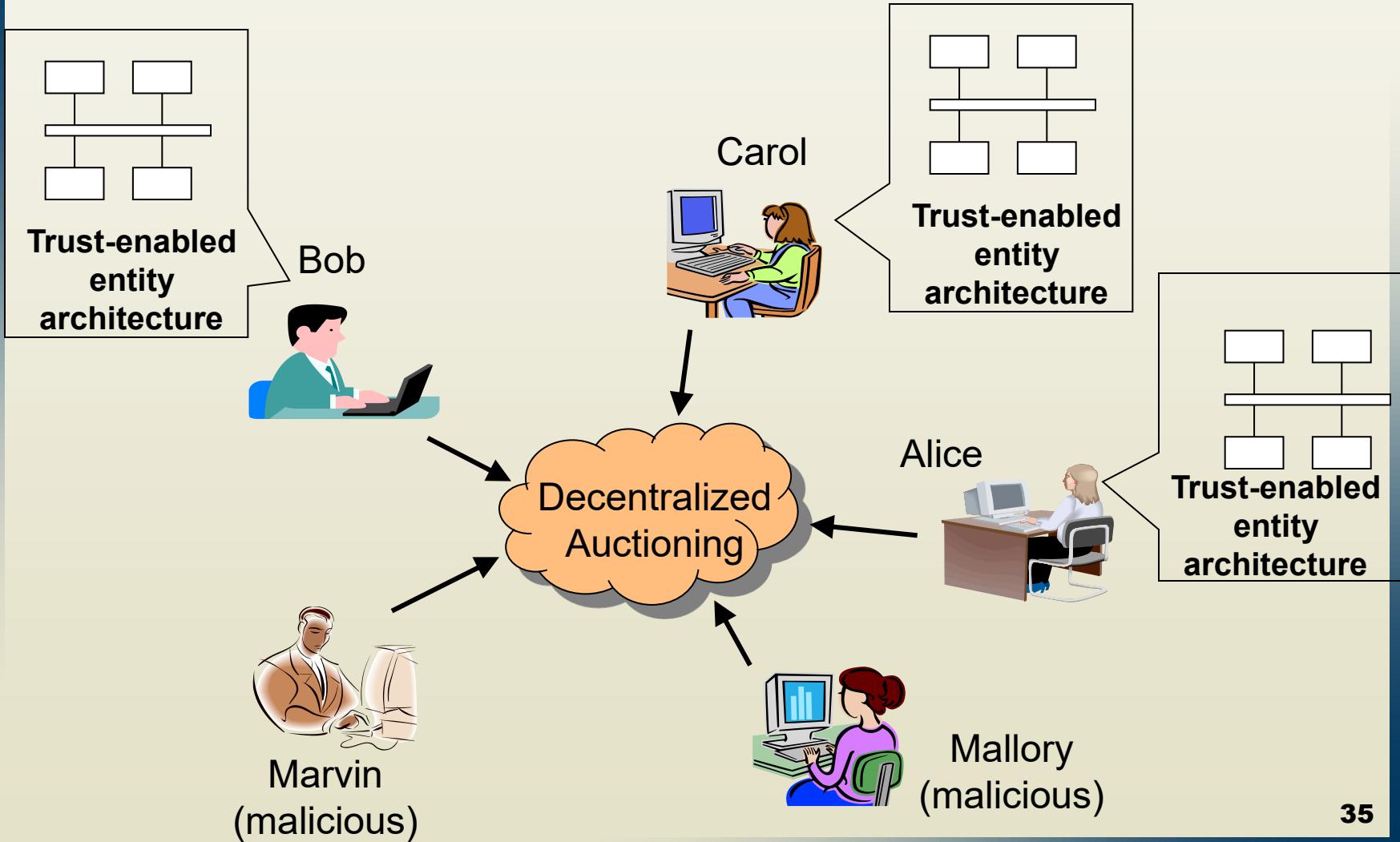


Counteracting Fraudulent Actions

- User sends request for trust information
- Others respond
- Responses are verified and tagged with trust values
- User sees these messages and makes an informed decision
- Post-interaction, user can change trust information



Result: Decentralized Auctioning



Deployment and Mobility

**Software Architecture
Lecture 22**

What Are Deployment and Mobility?

- *Deployment* is the process of placement of a system's software components on its hardware hosts
- Changing the deployment of a component during runtime is called *migration* or *redeployment*
- Migration or redeployment is a type of software system *mobility*
 - ◆ Mobility entails a superset of deployment issues

Deployment and Mobility Challenges

- Widely distributed target processors
- Target processors embedded inside heterogeneous devices
- Different software components may require different hardware configurations
- System lifespans may stretch over decades
- Software system evolution is continuous (requiring redeployment)
- Mobile code may mandate that *running, stateful* components be redeployed

Software Architecture and Deployment

- A system is deployed to a set of *hosts* or *sites*
- Each site provides a set of *resources* needed by the system's components
 - ◆ Hardware
 - ◆ Network
 - ◆ Peripheral devices
 - ◆ System software
 - ◆ Other application software
 - ◆ Data resources
- Resources may be *exclusive* or *sharable*

How Is Deployment Changing ?

Then

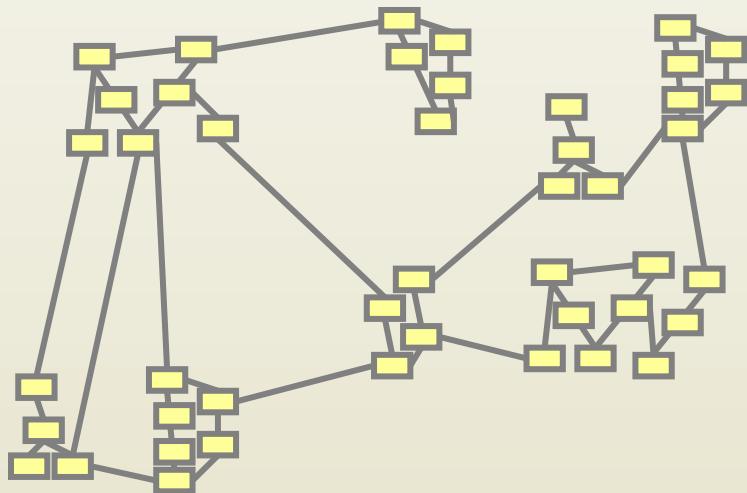
- Complete Installation procedure for software system on CD ROM
- Entire software system installation

Now

- Software producers and consumers cooperating and negotiating.
- “Update” of Software Systems

All this because of high connectivity

Deployment, Architecture, and Quality of Service



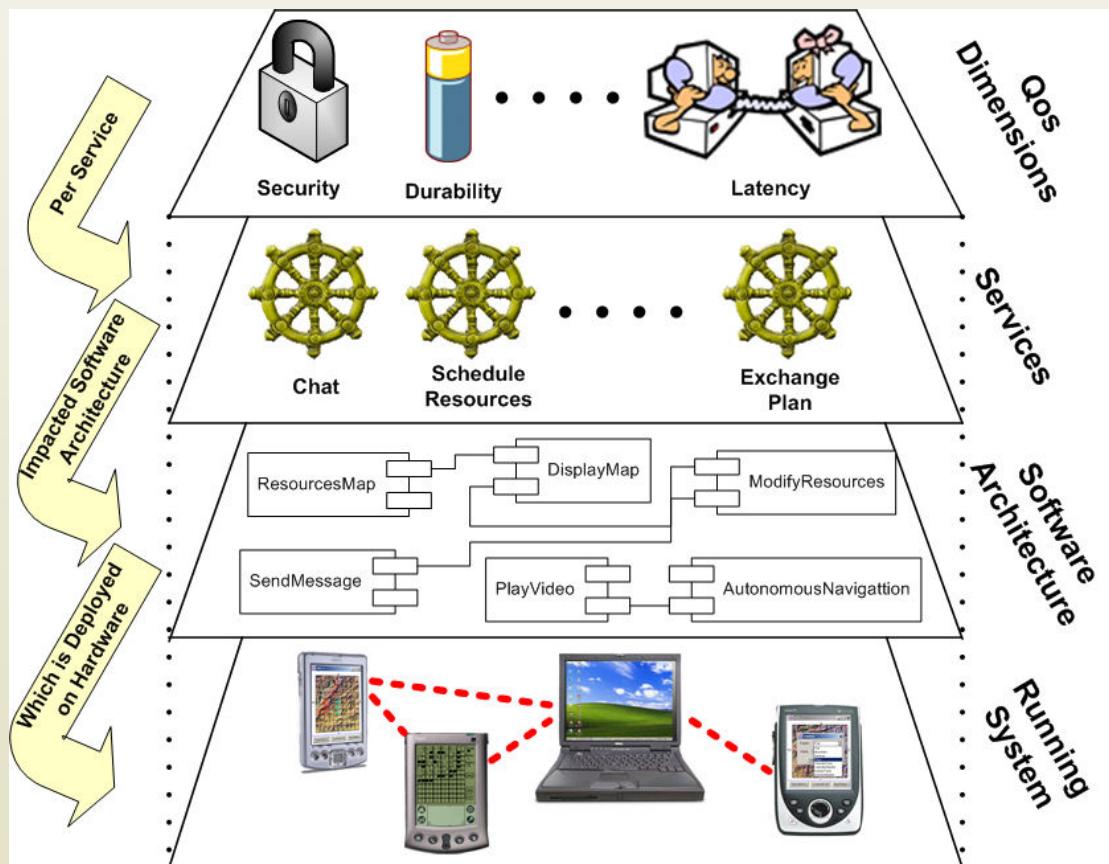
- *Deployment Architecture*: allocation of s/w components to h/w hosts
- h^c deployment architectures are possible for a given system
 - ◆ Many provide the same functionality
 - ◆ Provide different qualities of service (QoS)

Deployment Activities

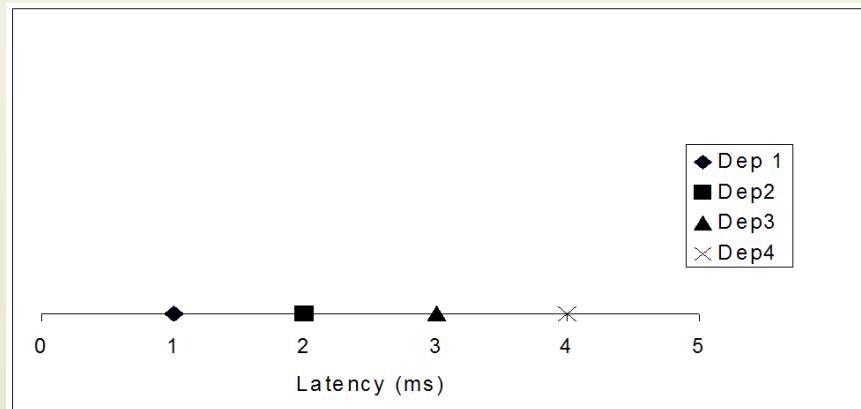
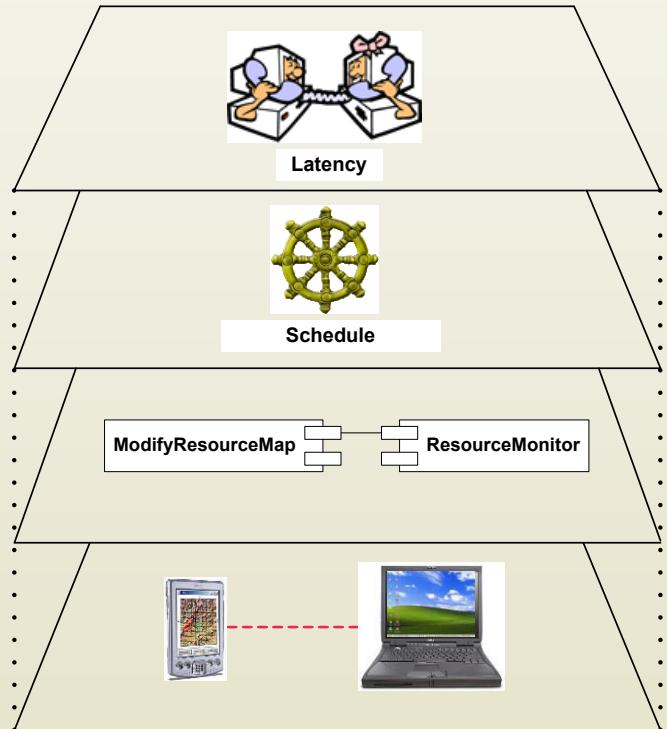
- Planning
- Modeling
- Analysis
- Implementation

Deployment Planning

How do we **find** and **effect** a deployment architecture that improves multiple QoS dimensions?

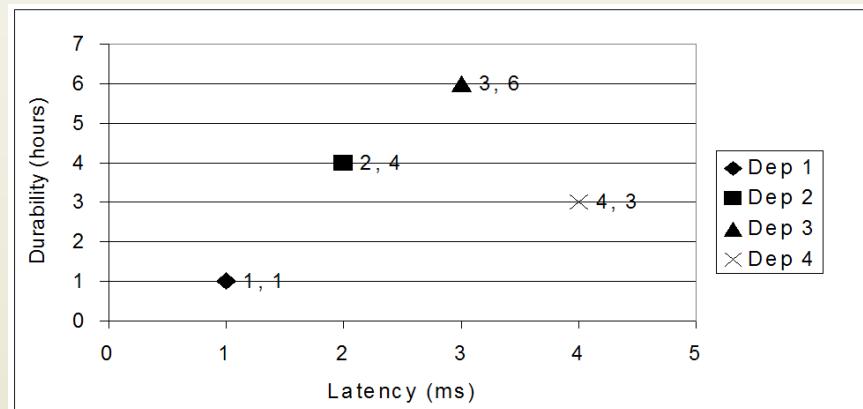
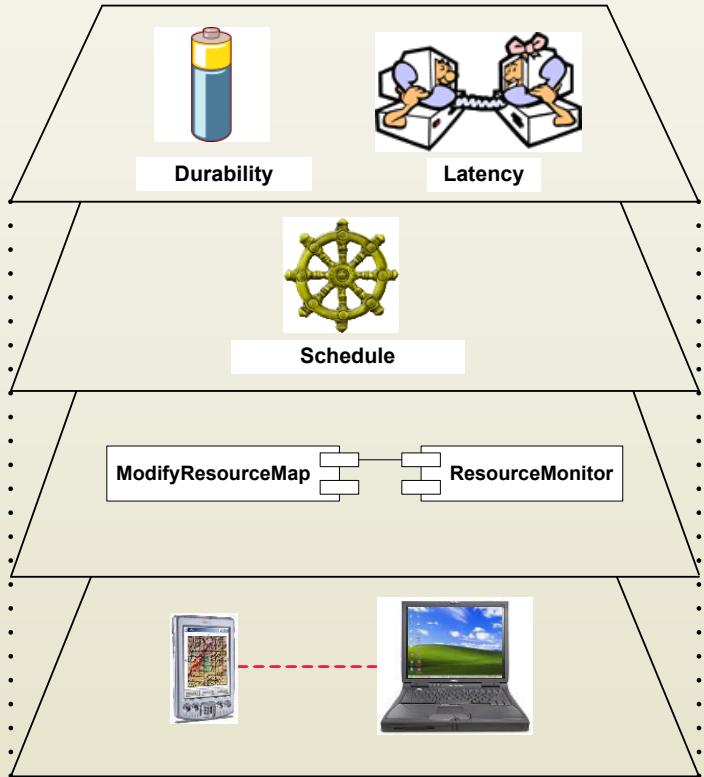


Scenario with a Single QoS Dimension



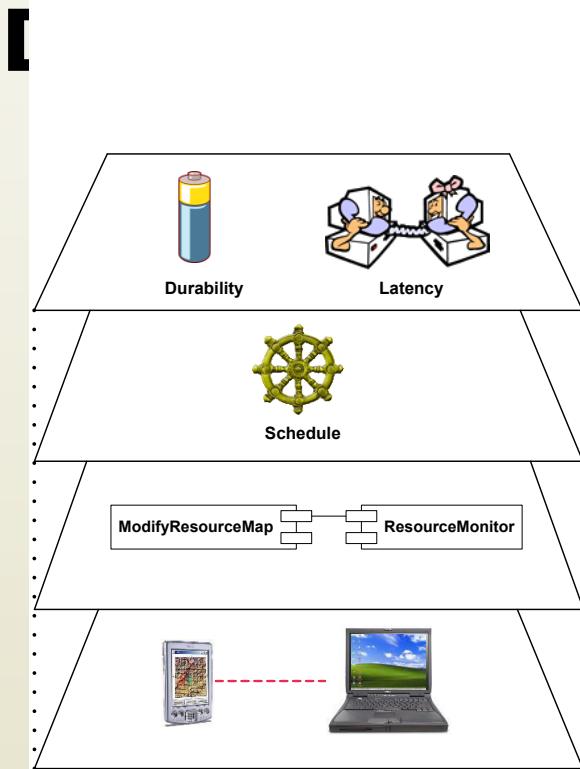
- Objective is to minimize latency
- The *optimal* deployment architecture is deployment 1

Conflicting QoS Dimensions



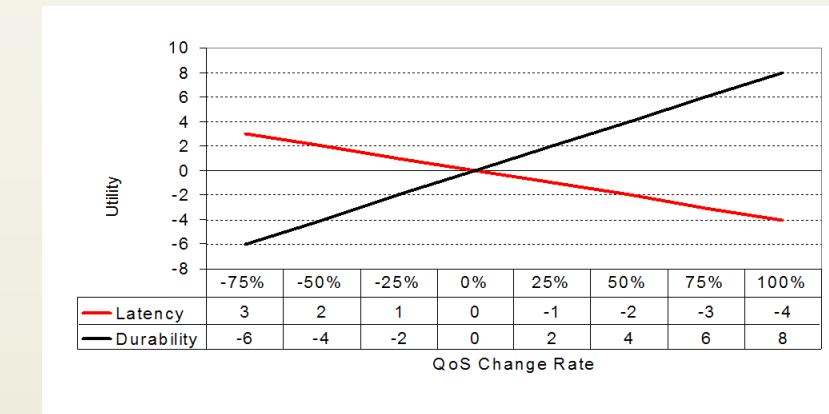
- Objective is to minimize latency and maximize durability
- There is no optimal deployment architecture!

Resolving Trade-Offs between QoS

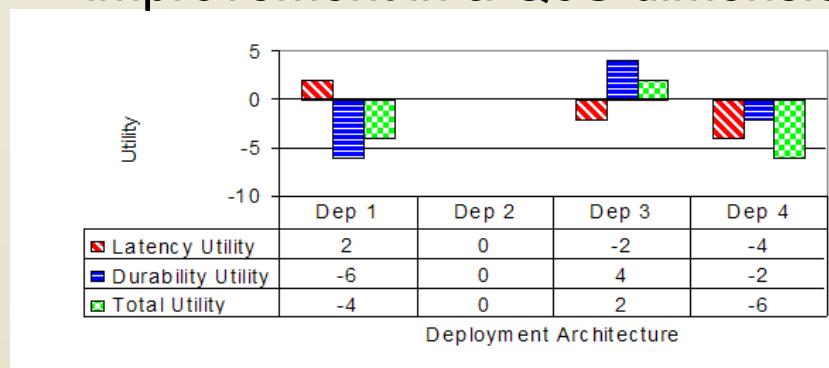


- Guiding Insight

- System users have varying QoS preferences for the system services they access

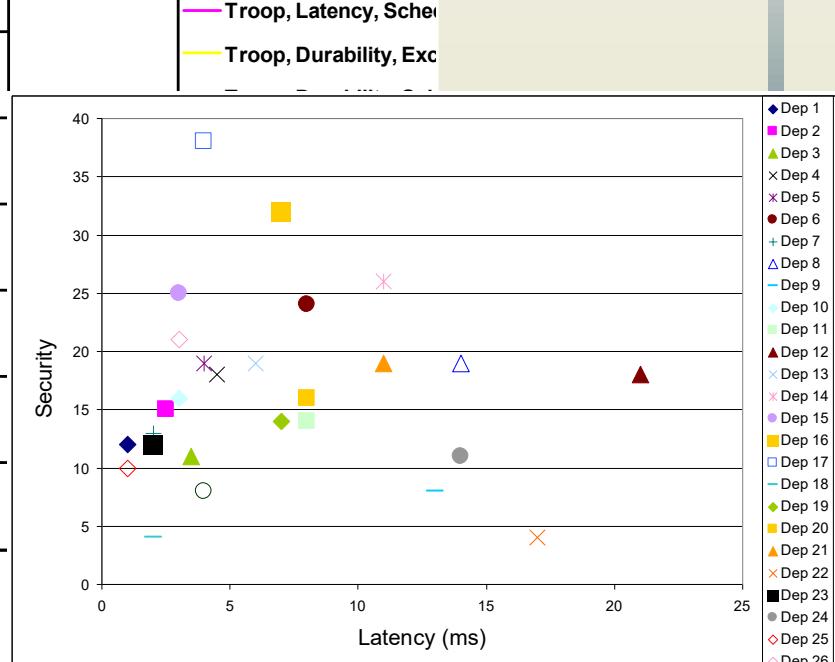
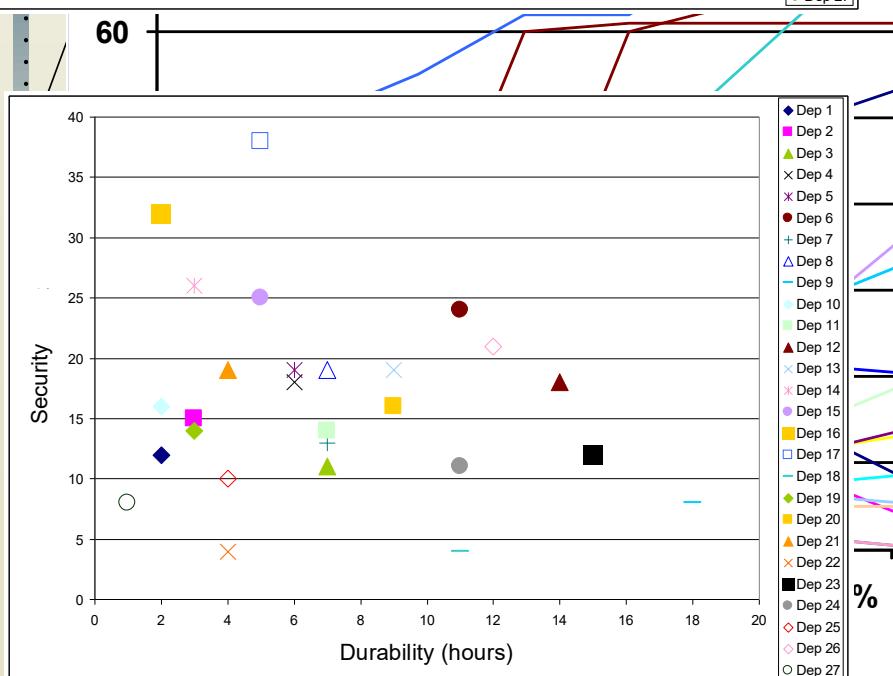
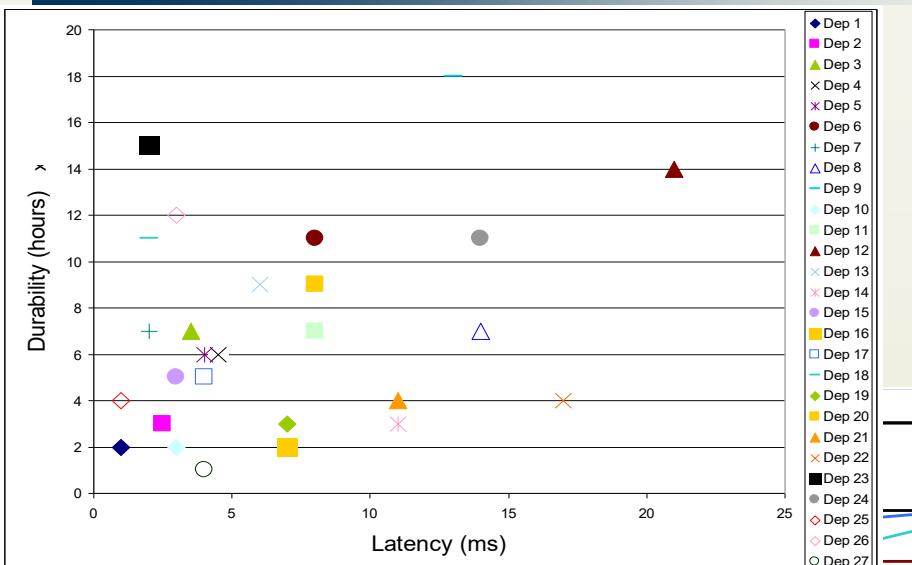


- A *utility function* denotes a user's preferences for a given rate of improvement in a QoS dimension

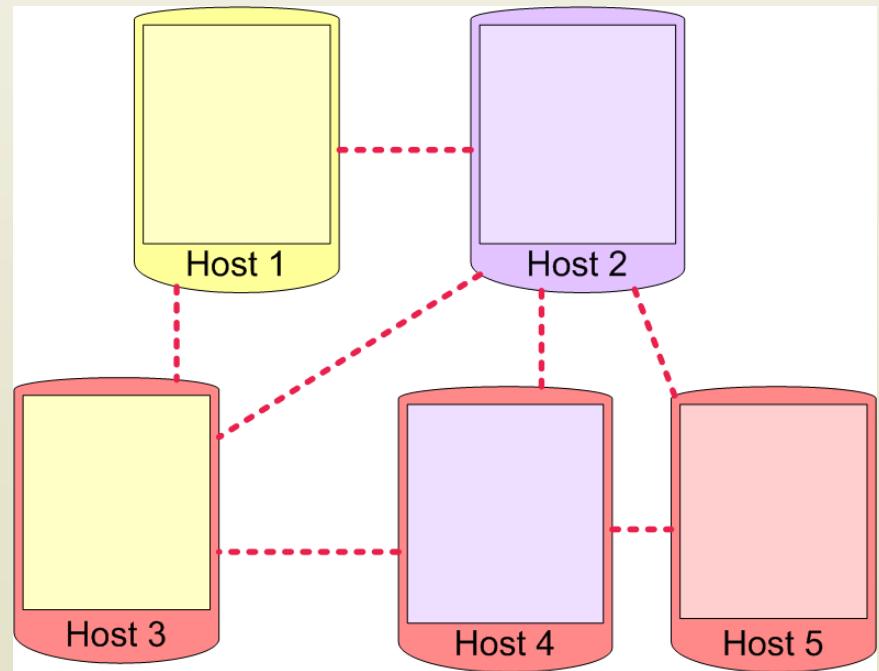
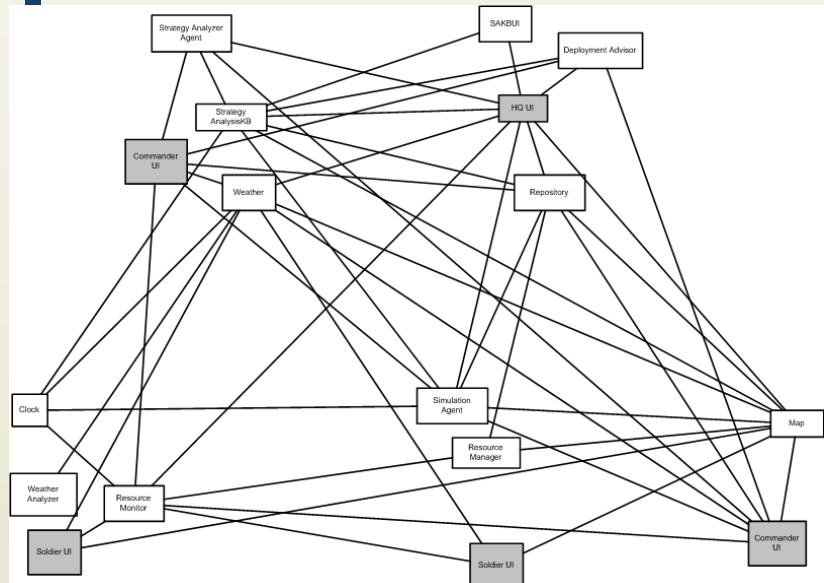


- Allows expression of optimization in terms of a single scalar value

A Slightly Larger Scenario

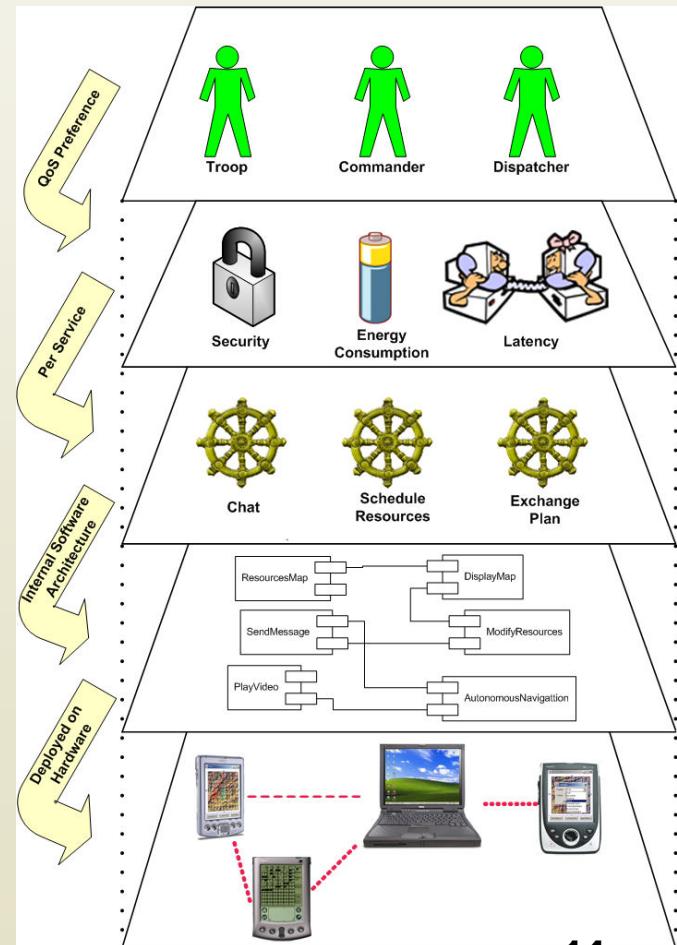


Deployment Modeling



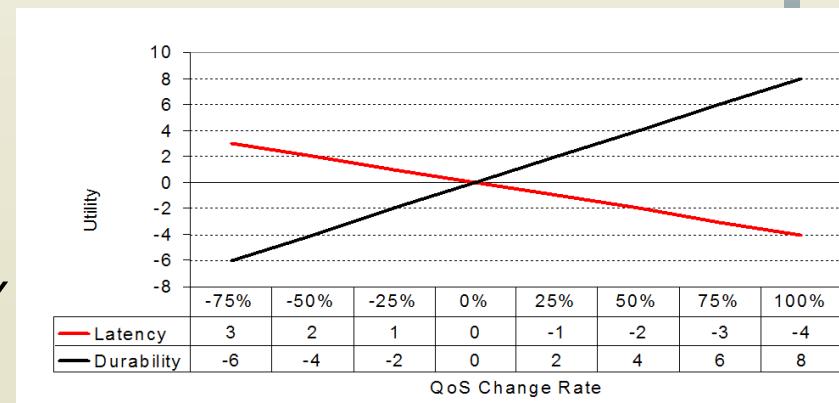
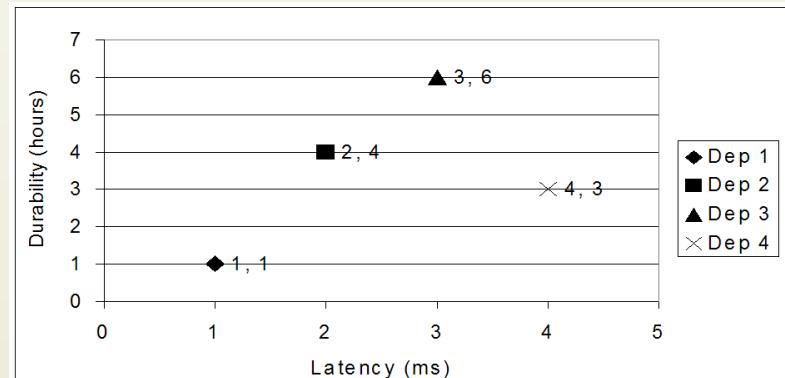
Modeling Architectural Elements

- Define sets that specify system elements and their properties
 - Set C of software components*
 - $C = \{ResourcesMap, SendMessage, Display, \dots\}$
 - Set CP of software component properties*
 - $CP = \{\text{size}, \text{reliability}, \dots\}$
 - Other sets*
 - H of hardware nodes, N of network links, I of logical links, S of services, Q of QoS, U of users
 - HP of hardware params, NP of network link params, CP of software component params, IP of logical link params
- Define functions that quantify system properties
 - Function $cParam: C \times CP \rightarrow R$
 - $cParam(\text{ResourcesMap}, \text{size}) = 150$
 - Other functions
 - $hParam, nParam, IParam, sParam$



Modeling QoS Dimensions

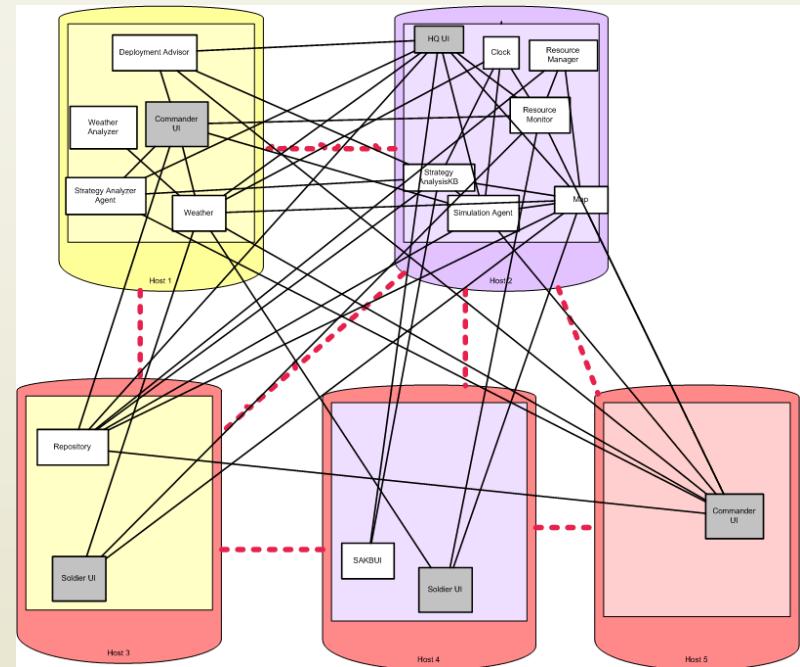
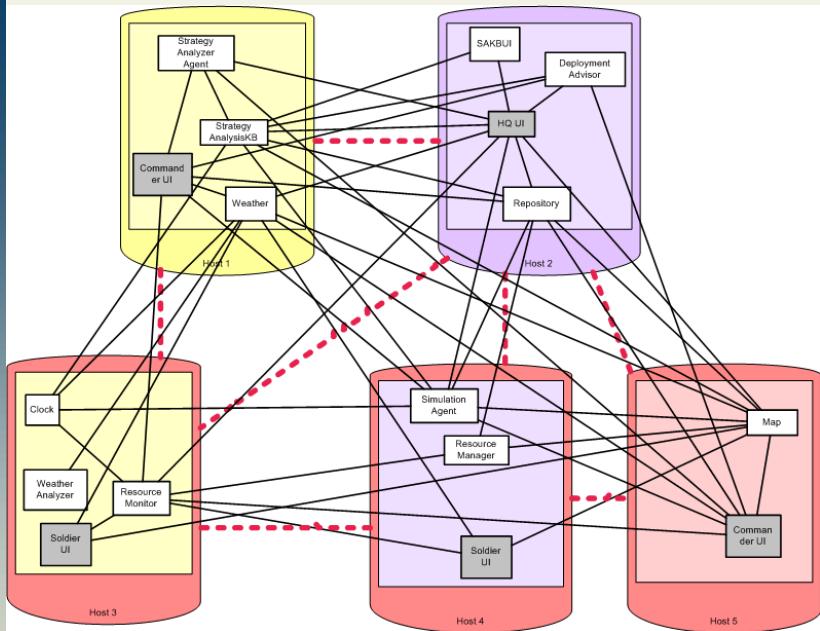
- Define QoS functions
 - ◆ $qValue: S \times Q \times DepSpace \rightarrow R$
 - quantifies the achieved level of QoS given a deployment
 - $qValue(Schedule, Latency, Dep 1) = 1ms$
- Define users' preferences in terms of utility
 - ◆ $qUtil: U \times S \times Q \times R \rightarrow [MinUtil, MaxUtil]$
 - represents the accrued utility for given rate of change
 - $qUtil(Commander, Schedule, Latency, 0.25) = -1$



Modeling System Constraints

- A set PC of parameter constraints
 - ◆ $PC = \{memory, bandwidth, \dots\}$
- A function $pcSatisfied: PC \times DepSpace \rightarrow [0,1]$
 - ◆ 1 if constraint is satisfied
 - ◆ 0 if constraint is not satisfied
- Functions that restrict locations of s/w components
 - ◆ $loc:C \times H \rightarrow [0,1]$
 - $loc(c,h)=1$ if c can be deployed on h
 - $loc(c,h)=0$ if c cannot be deployed on h
 - ◆ $colloc:C \times C \rightarrow [-1,1]$
 - $colloc(c1,c2)=1$ if $c1$ has to be on the same host as $c2$
 - $colloc(c1,c2)=-1$ if $c1$ cannot be on the same host as $c2$
 - $colloc(c1,c2)=0$ if there are no restrictions

Deployment Analysis



1. Are both deployments valid?
2. Which of the two deployments is “better”?
3. Does the selected deployment have required properties?

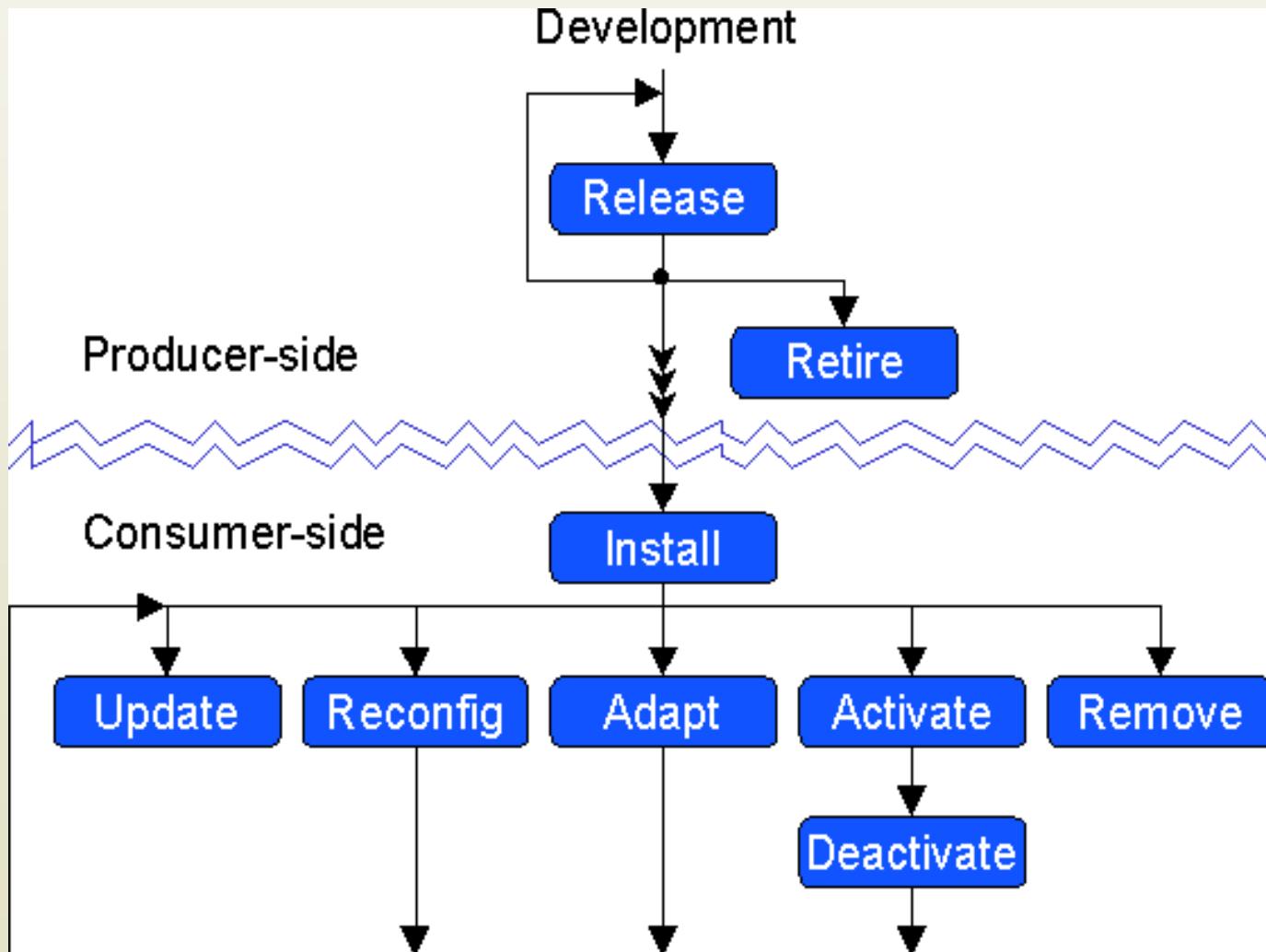
Deployment Analysis Strategies

- Different strategies with different strengths
 - Most of them offer approximate solutions
1. Mixed Integer Non-linear Programming (MINLP)
 2. Mixed Integer Linear Programming (MIP)
 3. Heuristic-based strategies
 - A. Greedy
 - B. Genetic
 - C. Decentralized

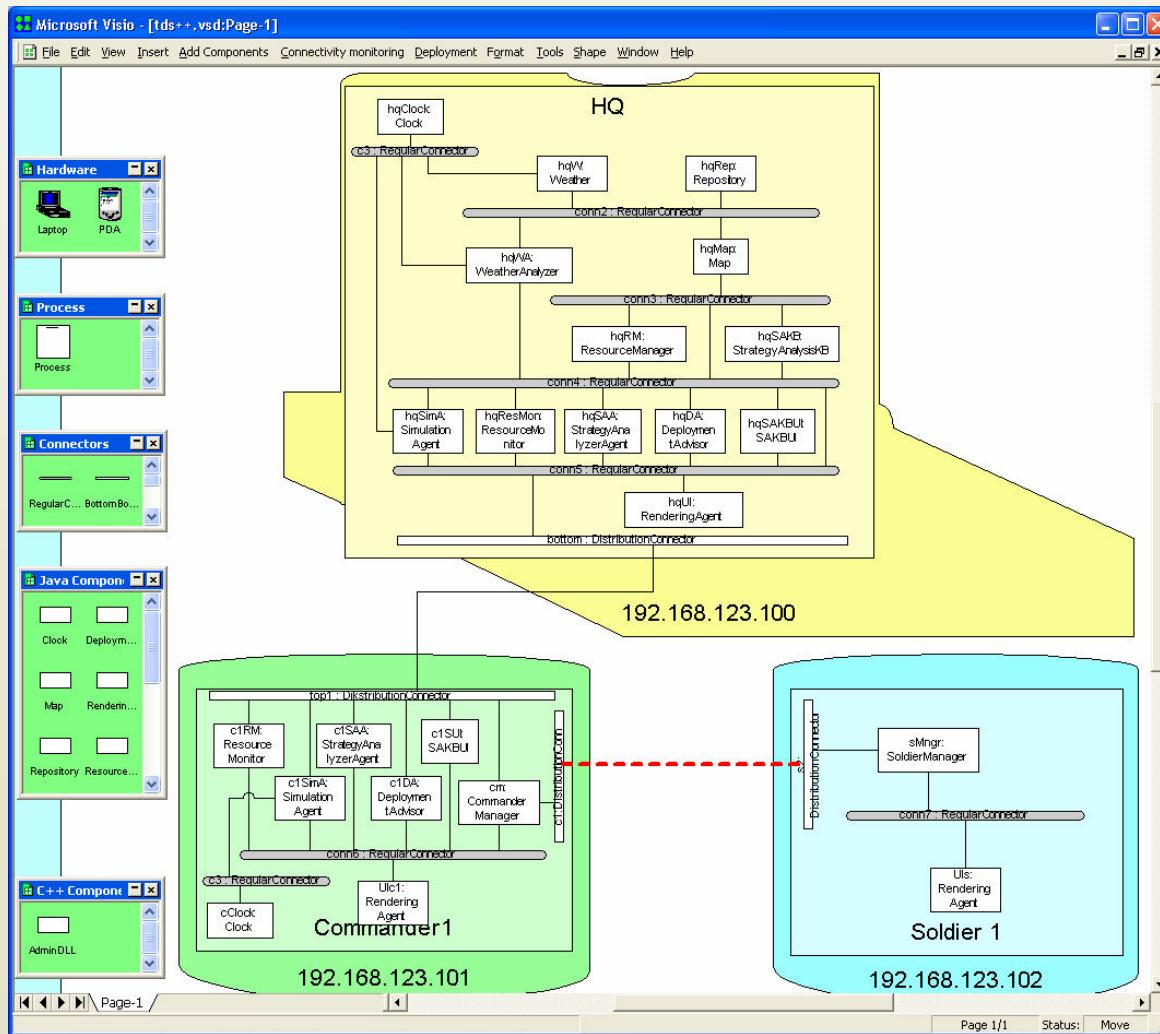
Deployment Implementation

- Release
- Install
- Activate
- Deactivate
- Update
- Adapt
- Reconfigure
- De-install or remove
- De-release or retire

Software Deployment Life Cycle



Deployment Tool Support – An Example



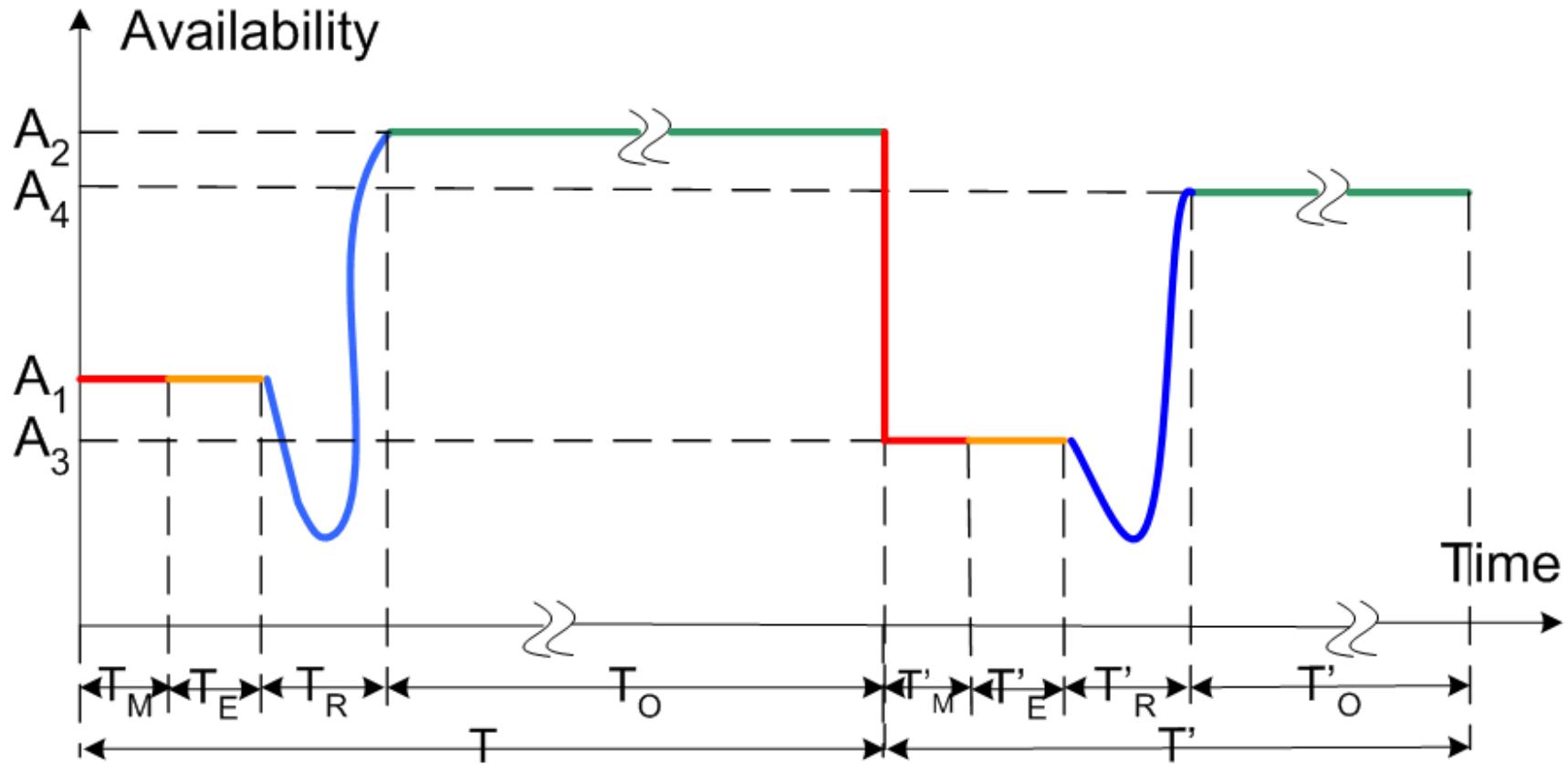
Software Mobility

- *Mobile computing* involves the movement of human users together with their hosts across different physical locations
 - ◆ This is also referred to as *physical mobility*
- Movement of software across hardware hosts during the system's execution, that action is referred to as *code mobility* or *logical mobility*
- If a software module that needs to be migrated contains runtime state, then the module's migration is known as *stateful mobility*
- If only the code needs to be migrated, that is known as *stateless mobility*

Mobility Paradigms

- Remote evaluation
 - ◆ Re-deploy needed component at runtime from a source host to a destination host
 - ◆ Install component on the destination host
 - ◆ Ensure that the system's architectural configuration and any architectural constraints are preserved
 - ◆ Activate the component
 - ◆ Executed the component to provide the desired service
 - ◆ Possibly de-activate and de-install
- Code-on-demand
 - ◆ Same as remote evaluation, but roles of target and destination hosts are reversed
- Mobile agent
 - ◆ Migration of a stateful software component that needs some remote resources to complete its task

Mobility and Quality of Service



Closing Thoughts

- It may be impractical or unacceptable to bring systems down for upgrades
 - ◆ (Re)deployment is thus necessary
- Architecture as a set of *principal* design decisions naturally encompasses (re)deployment
- Maintaining the relationship between architectural model and implementation stems degradation

Intro to Domain-Specific Software Engineering

Software Architecture
Lecture 23

Objectives

- Concepts
 - ◆ What is domain-specific software engineering (DSSE)
 - ◆ The “Three Lampposts” of DSSE: Domain, Business, and Technology
 - ◆ Domain Specific Software Architectures
- Product Lines
- Relationship between DSSAs, Product Lines, and Architectural Styles
- Examples of DSSE at work

Objectives

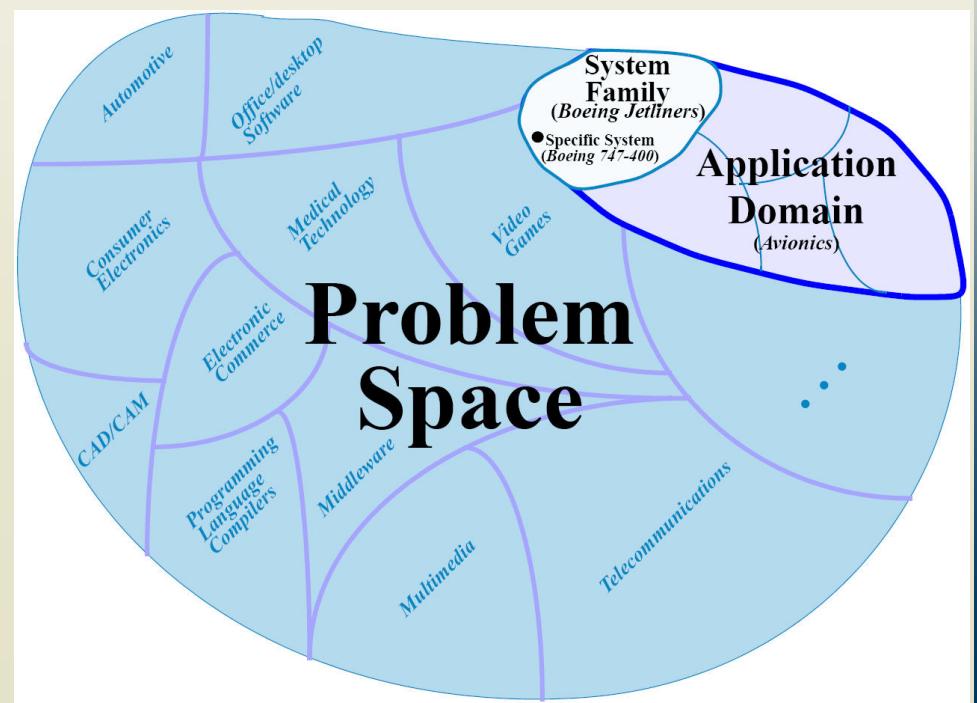
- Concepts
 - ◆ What is domain-specific software engineering (DSSE)
 - ◆ The Three Key Factors of DSSE: Domain, Business, and Technology
 - ◆ Domain Specific Software Architectures
- Product Lines
- Relationship between DSSAs, Product Lines, and Architectural Styles
- Examples of DSSE at work

Domain-Specific Software Engineering

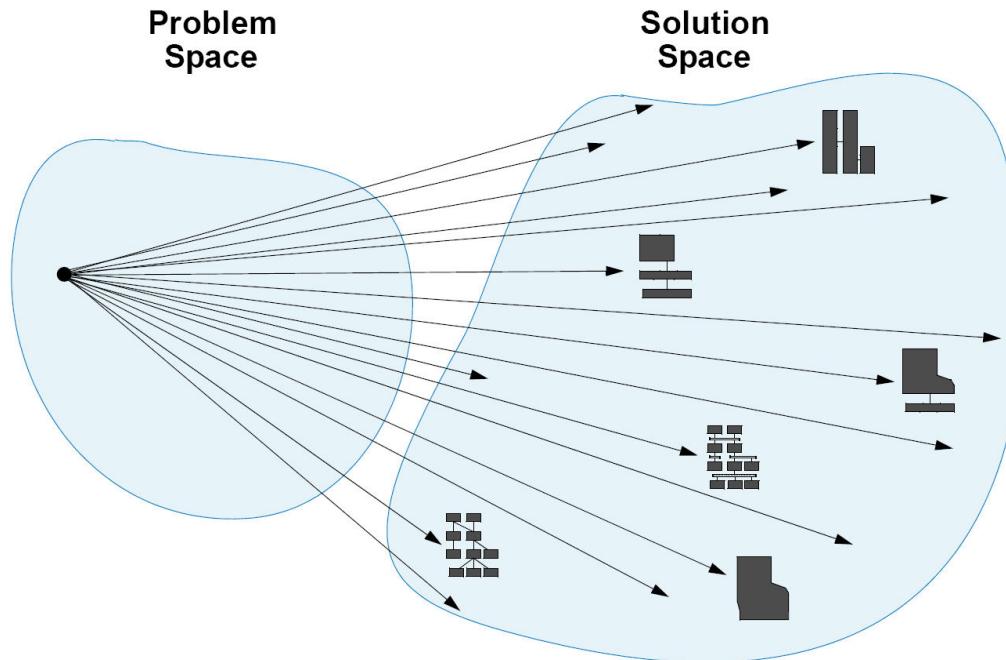
- The traditional view of software engineering shows us how to come up with solutions for problems *de novo*
- But starting from scratch every time is infeasible
 - ◆ This will involve re-inventing many wheels
- Once we have built a number of systems that do similar things, we gain critical knowledge that lets us exploit common solutions to common problems
 - ◆ In theory, we can simply build “the difference” between our new target system and systems that have come before

Examples of Domains

- Compilers for programming languages
- Consumer electronics
- Electronic commerce system/Web stores
- Video game
- Business applications
 - ◆ Basic/Standard/"Pro"
- We can subdivide, too:
 - ◆ Avionics systems
 - Boeing Jets
 - ◆ Boeing 747-400

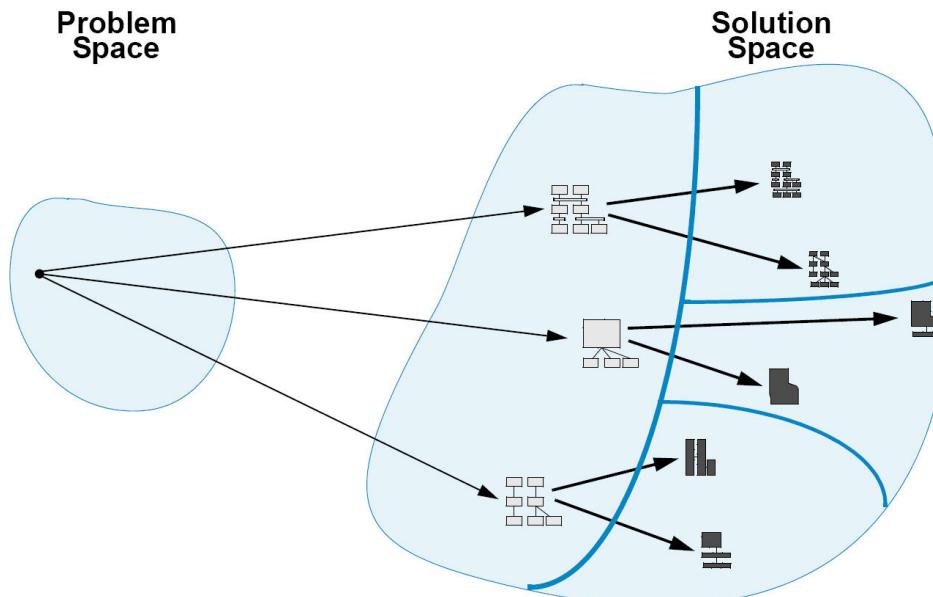


Traditional Software Engineering



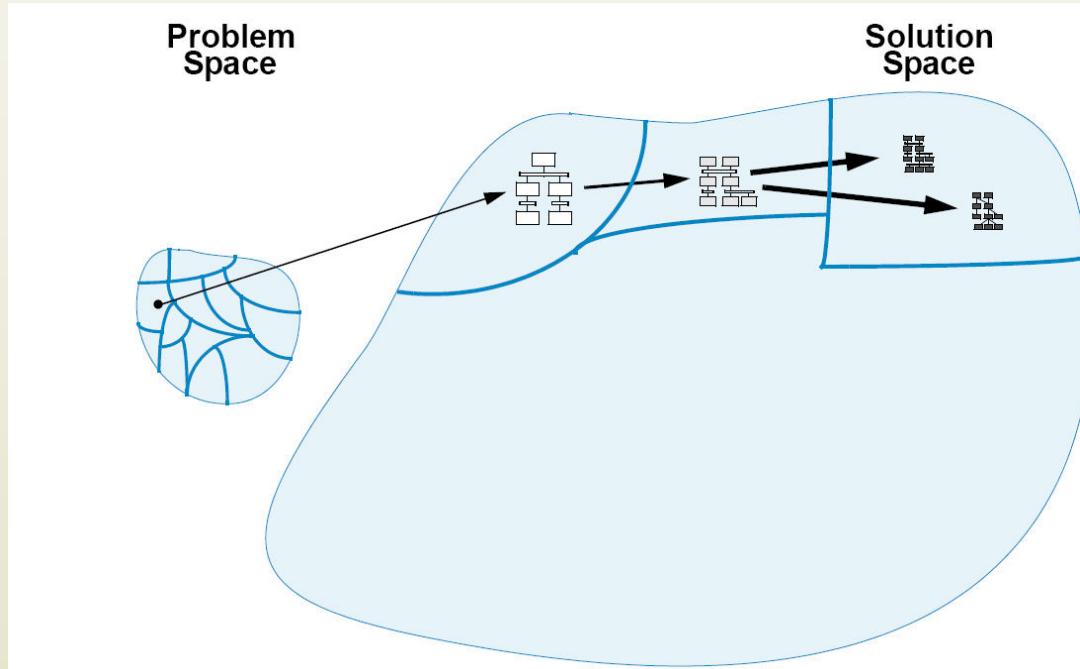
- One particular problem can be solved in innumerable ways

Architecture-Based Software Engineering



- Given a single problem, we select from a handful of potential architectural styles or architectures, and go from these into specific implementations

Domain-Specific Software Engineering



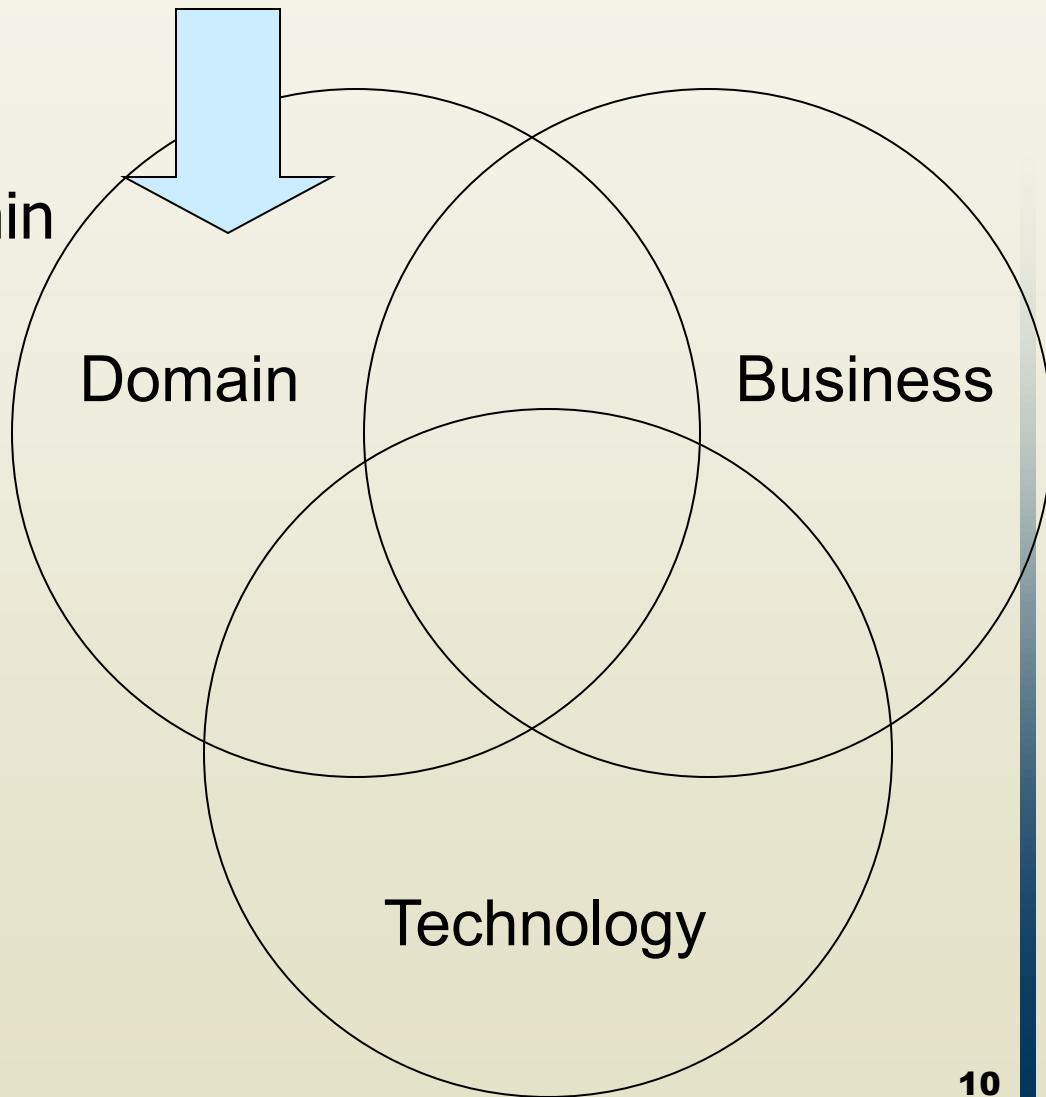
- We map regions of the problem space (domains) into domain-specific software architectures (DSSAs)
- These are specialized into application-specific architectures
- These are implemented

Three Key Factors of DSSE

- Domain
 - ◆ Must have a domain to constrain the problem space and focus development
- Technology
 - ◆ Must have a variety of technological solutions—tools, patterns, architectures & styles, legacy systems—to bring to bear on a domain
- Business
 - ◆ Business goals motivate the use of DSSE
 - Minimizing costs: reuse assets when possible
 - Maximize market: develop many related applications for different kinds of end users

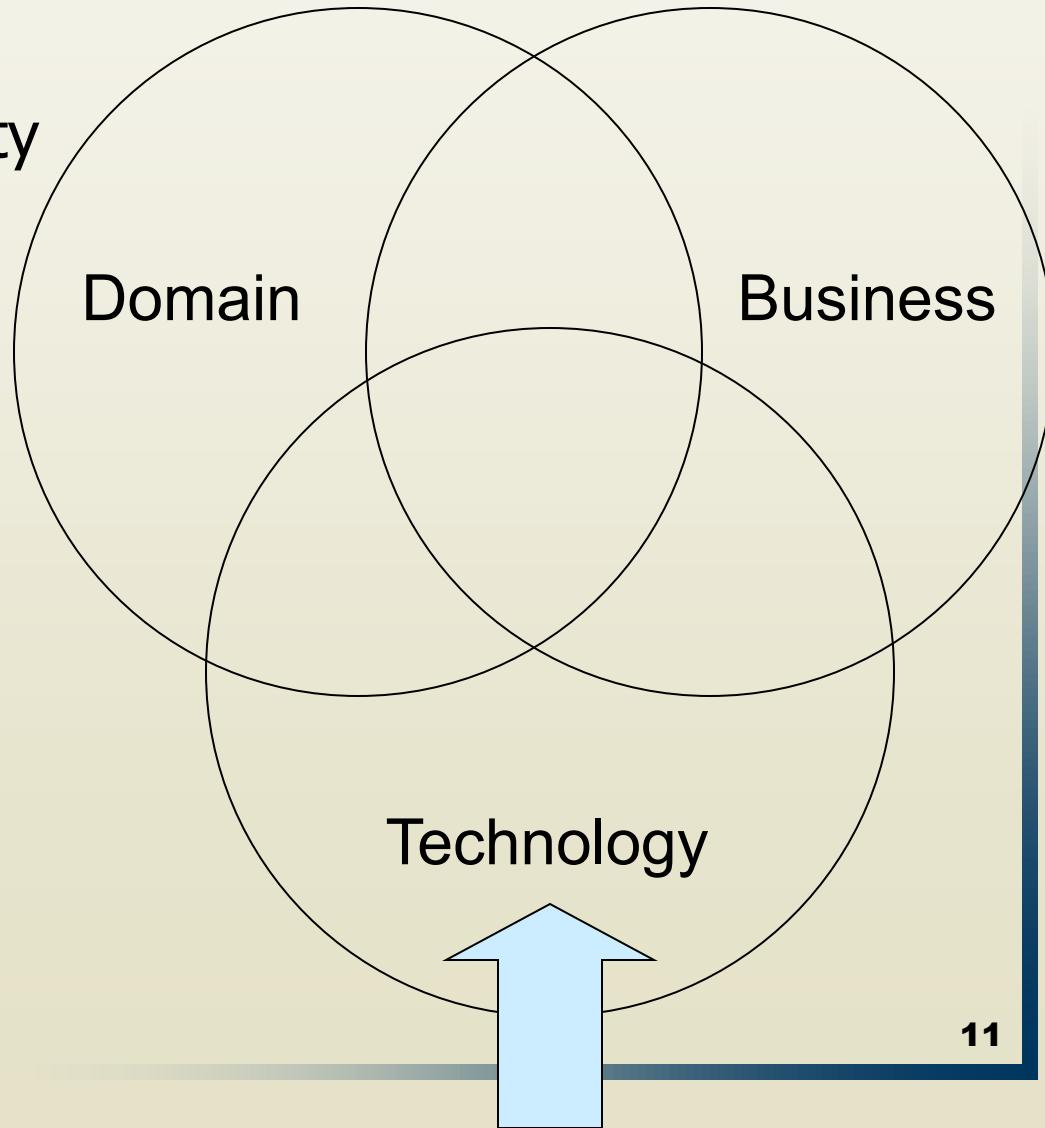
Three Key Factors

- Domain
 - ◆ Must have a domain to constrain the problem space and focus development



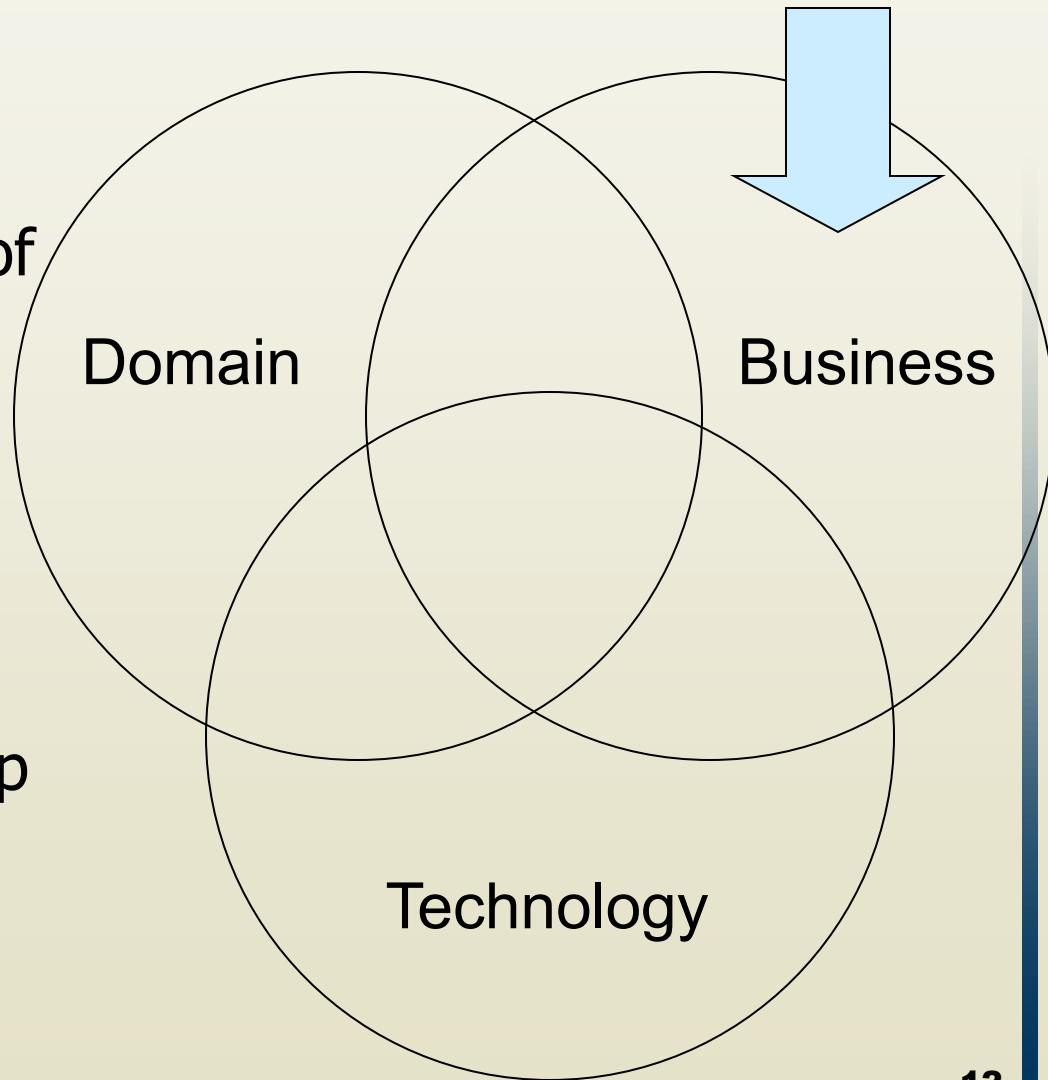
Three Key Factors

- Technology
 - ◆ Must have a variety of technological solutions—tools, patterns, architectures & styles, legacy systems—to bring to bear on a domain



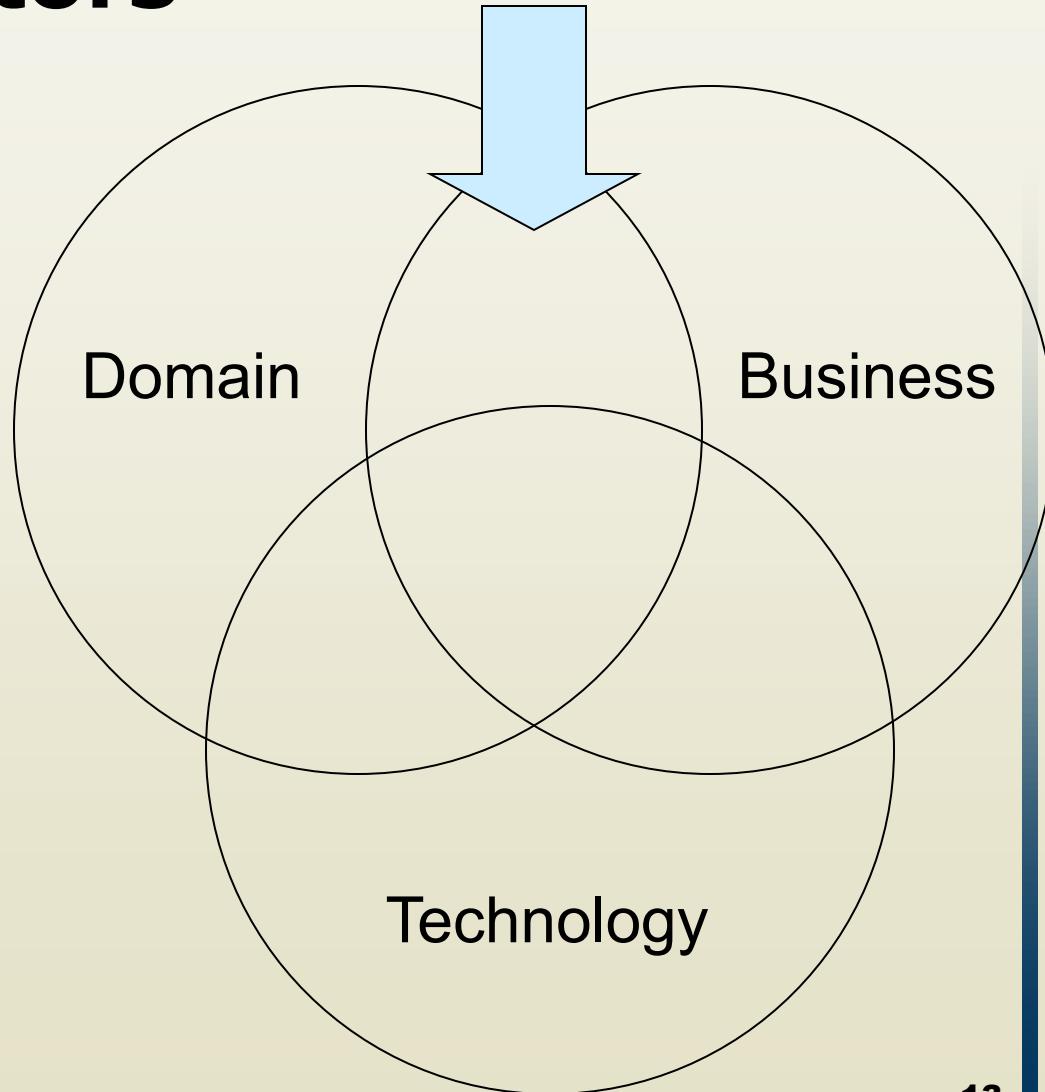
Three Key Factors

- Business
 - ◆ Business goals motivate the use of DSSE
 - Minimizing costs: reuse assets when possible
 - Maximize market: develop many related applications for different kinds of end users



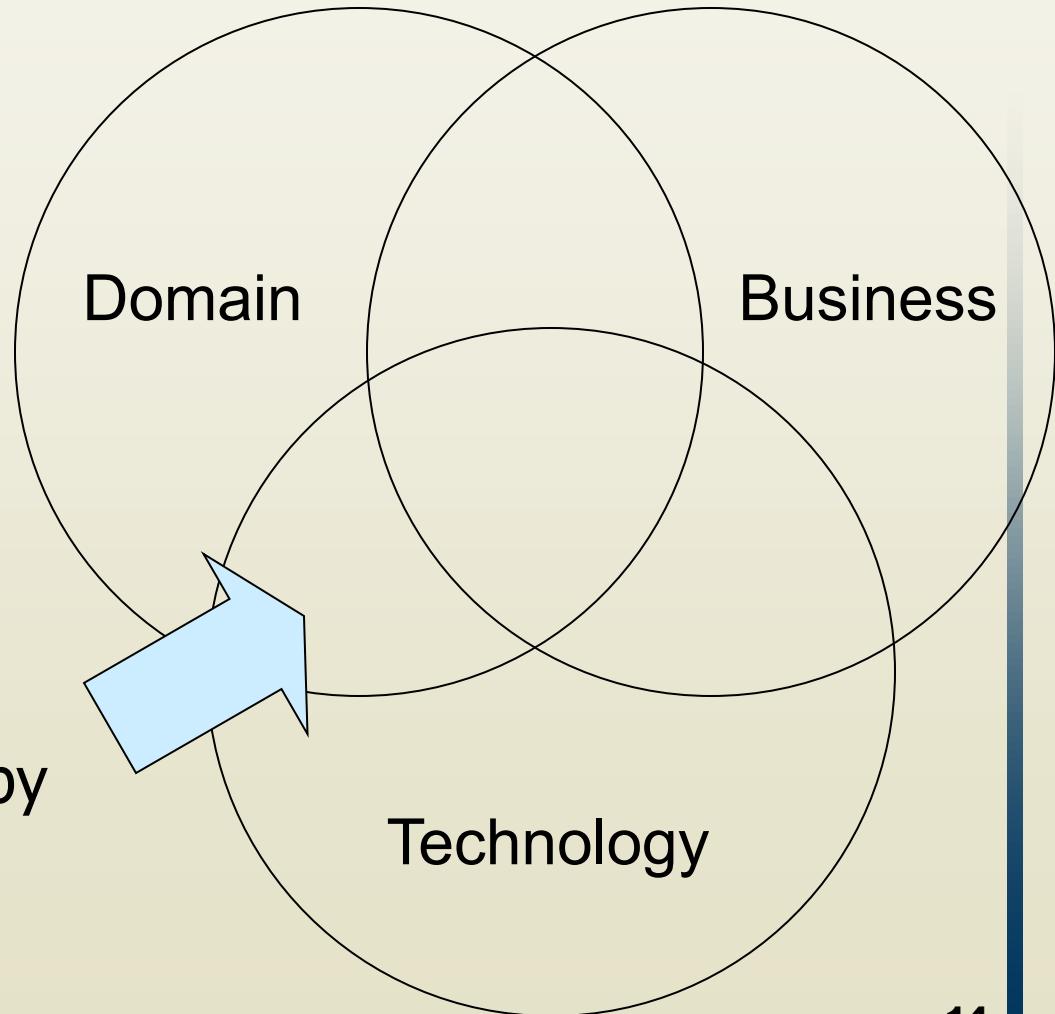
Three Key Factors

- Domain + Business
- “Corporate Core Competencies”
 - ◆ Domain expertise augmented by business acumen and knowledge of the market



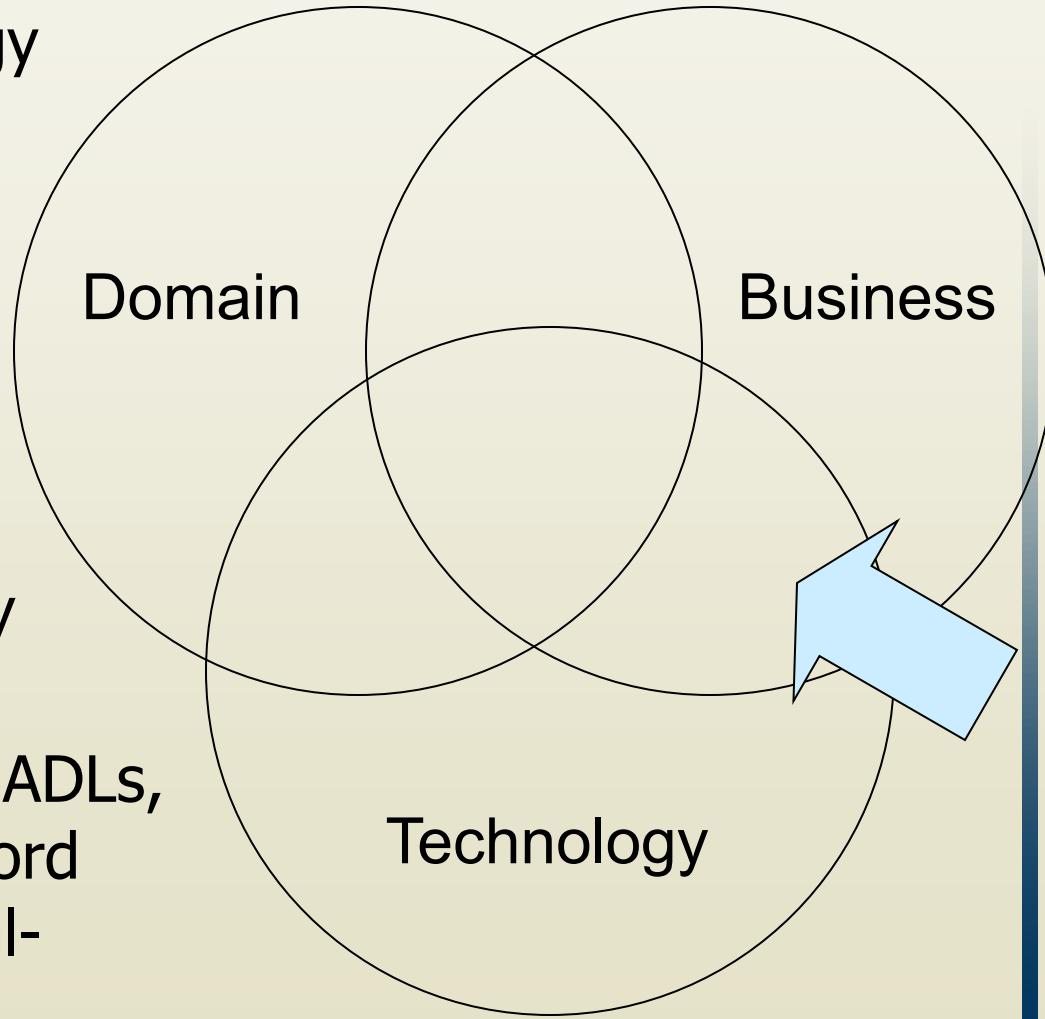
Three Key Factors

- Domain + Technology
- “Application Family Architectures”
 - ◆ All possible technological solutions to problems in a domain
 - ◆ Uninformed and *unconstrained* by business goals and knowledge



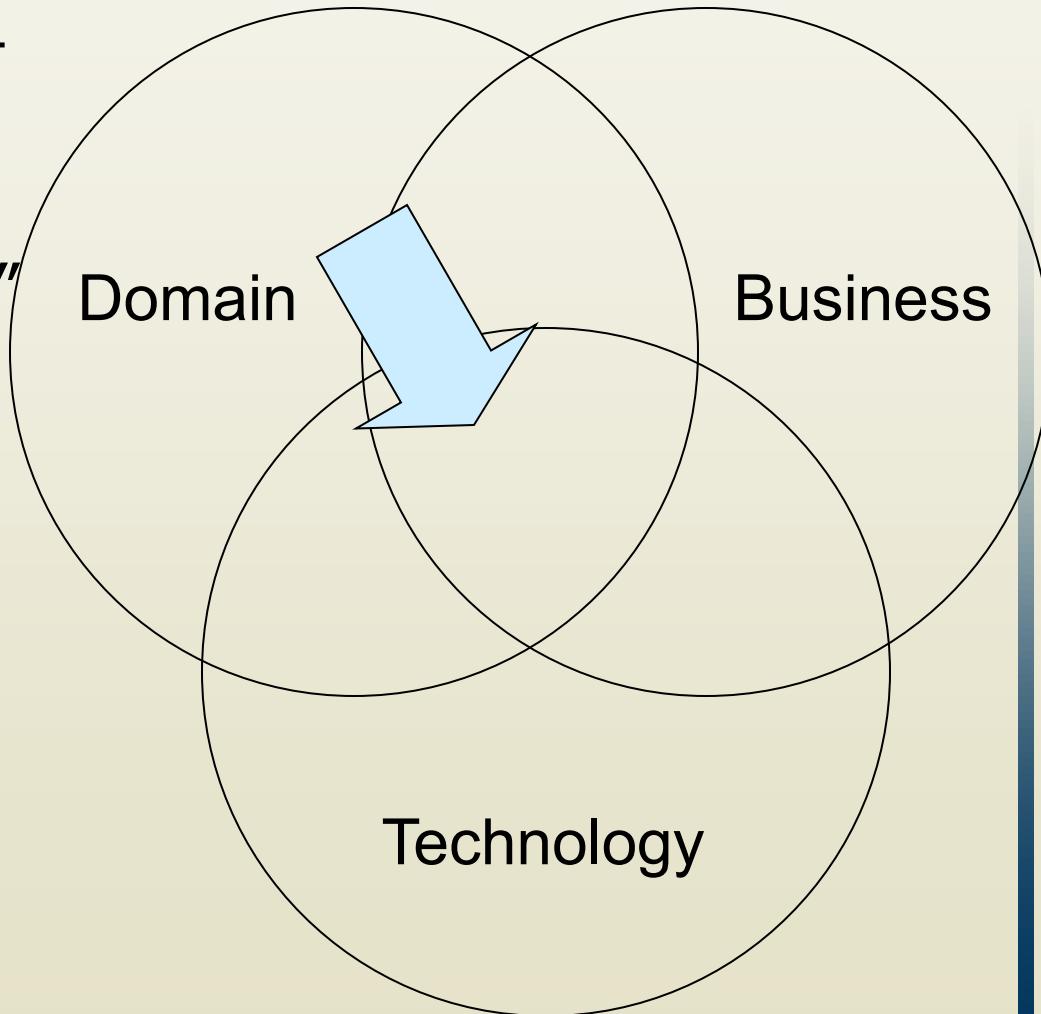
Three Key Factors

- Business + Technology
- “Domain independent infrastructure”
 - ◆ Tools and techniques for constructing systems independent of any particular domain
 - ◆ E.g., most generic ADLs, UML, compilers, word processors, general-purpose PCs



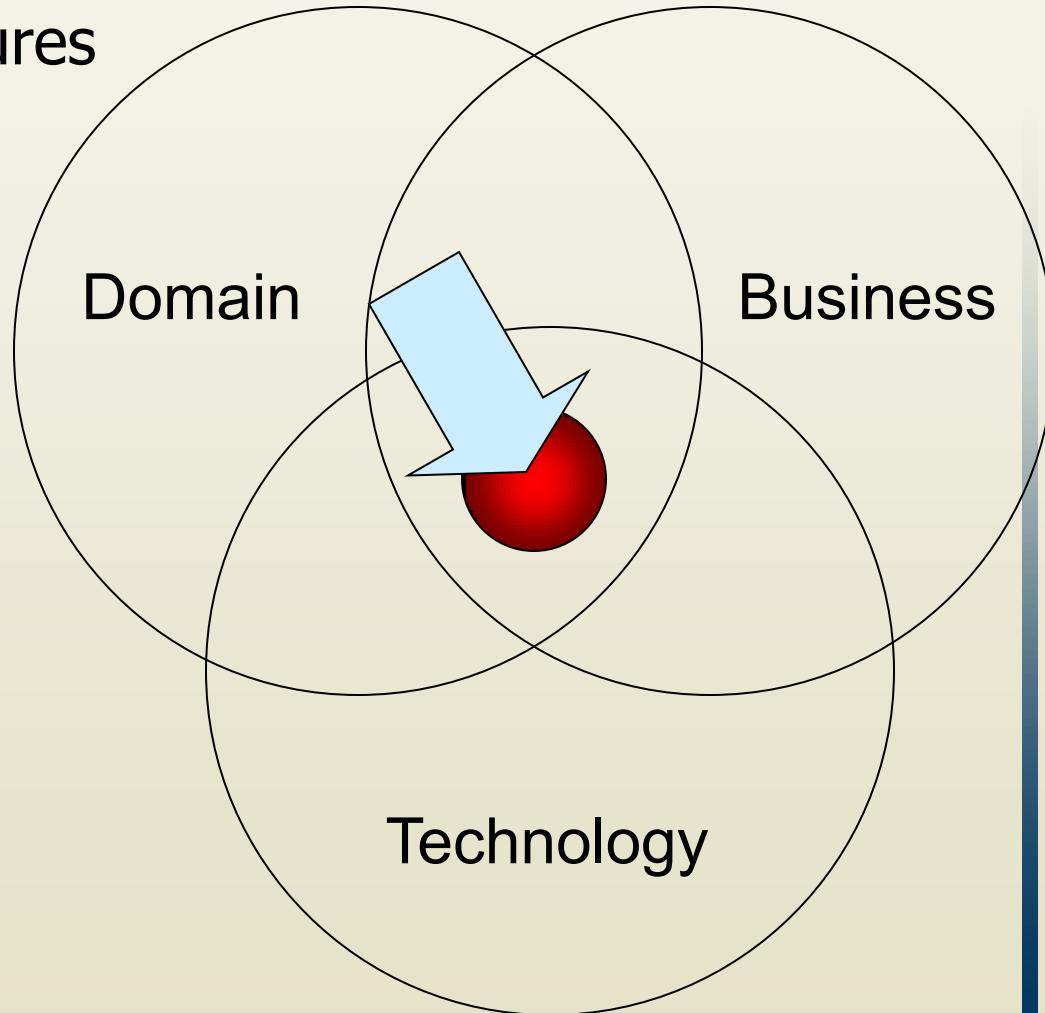
Three Key Factors

- Domain + Business + Technology
- “Domain-specific software engineering”
- Applies technology to domain-specific goals, tempered by business and market knowledge



Three Key Factors

- Product-Line Architectures
- A specific, related set of solutions within a broader DSSE
- More focus on commonalities and variability between individual solutions



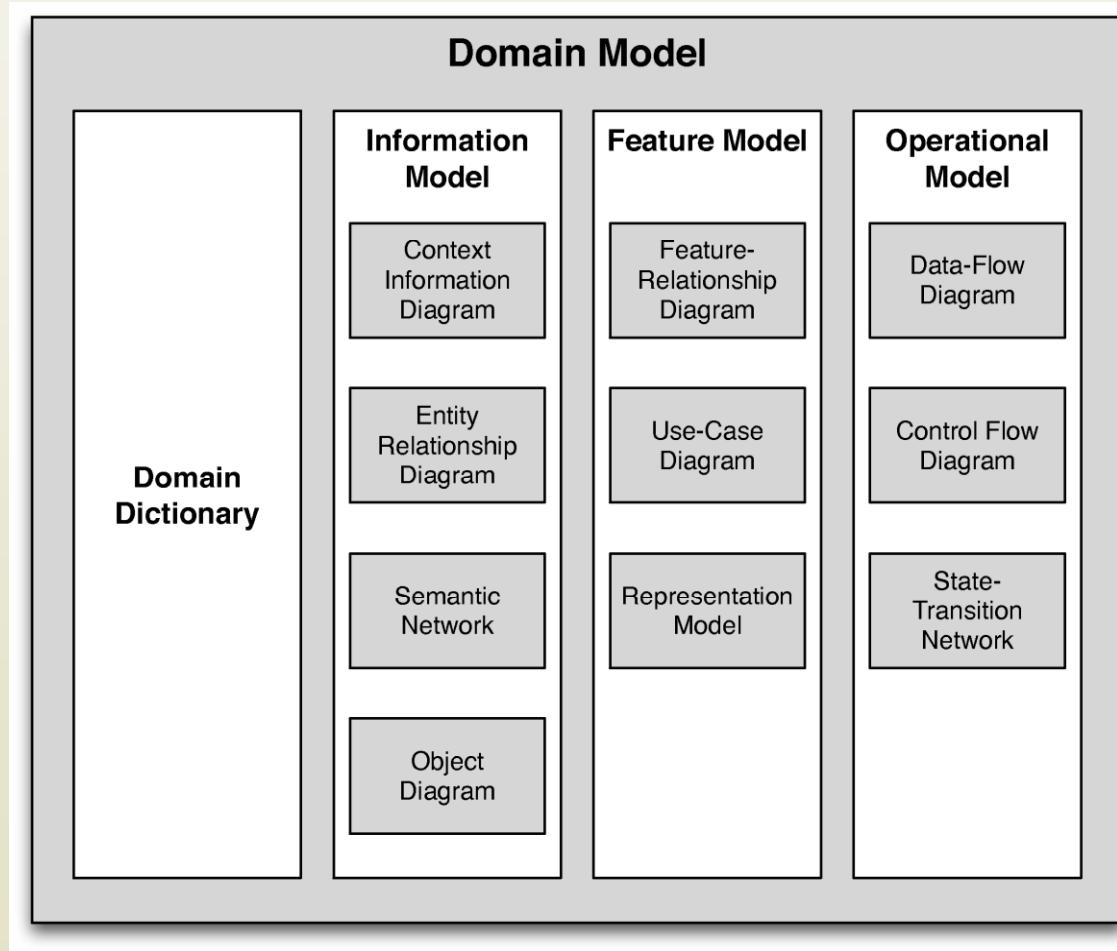
Becoming More Concrete

- Applying DSSE means developing a set of artifacts more specific than an ordinary software architecture
 - ◆ Focus on aspects of the domain
 - ◆ Focus on domain-specific solutions, techniques, and patterns
- These are
 - ◆ A *domain model* and
 - ◆ A domain-specific software architecture (DSSA)

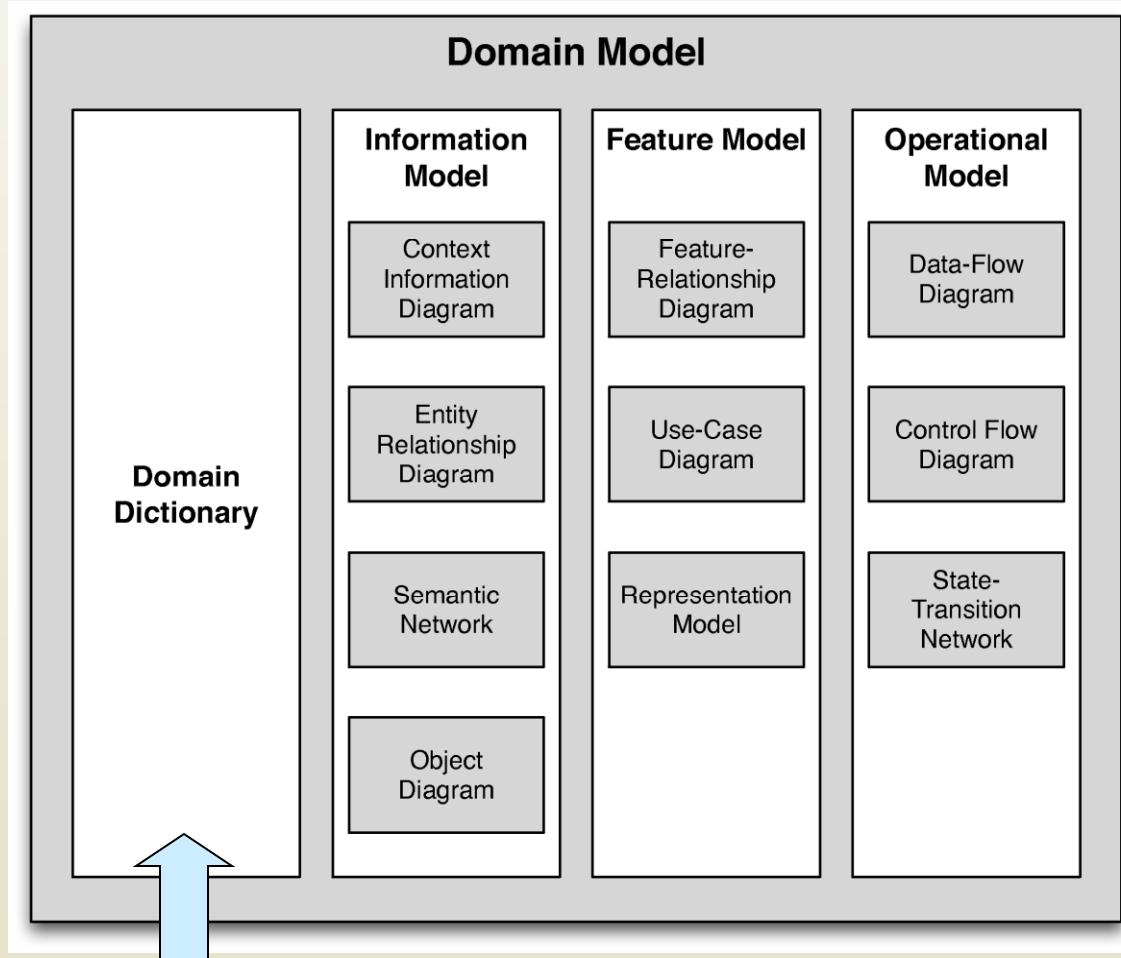
Domain Model

- A domain model is a set of artifacts that capture information about a domain
 - ◆ Functions performed
 - ◆ Objects (also known as entities) that perform the functions, and on which the functions are performed
 - ◆ Data and information that flows through the system
- Standardizes terminology and semantics
- Provides the basis for standardizing (or at least normalizing) descriptions of problems to be solved in the domain

Domain Model

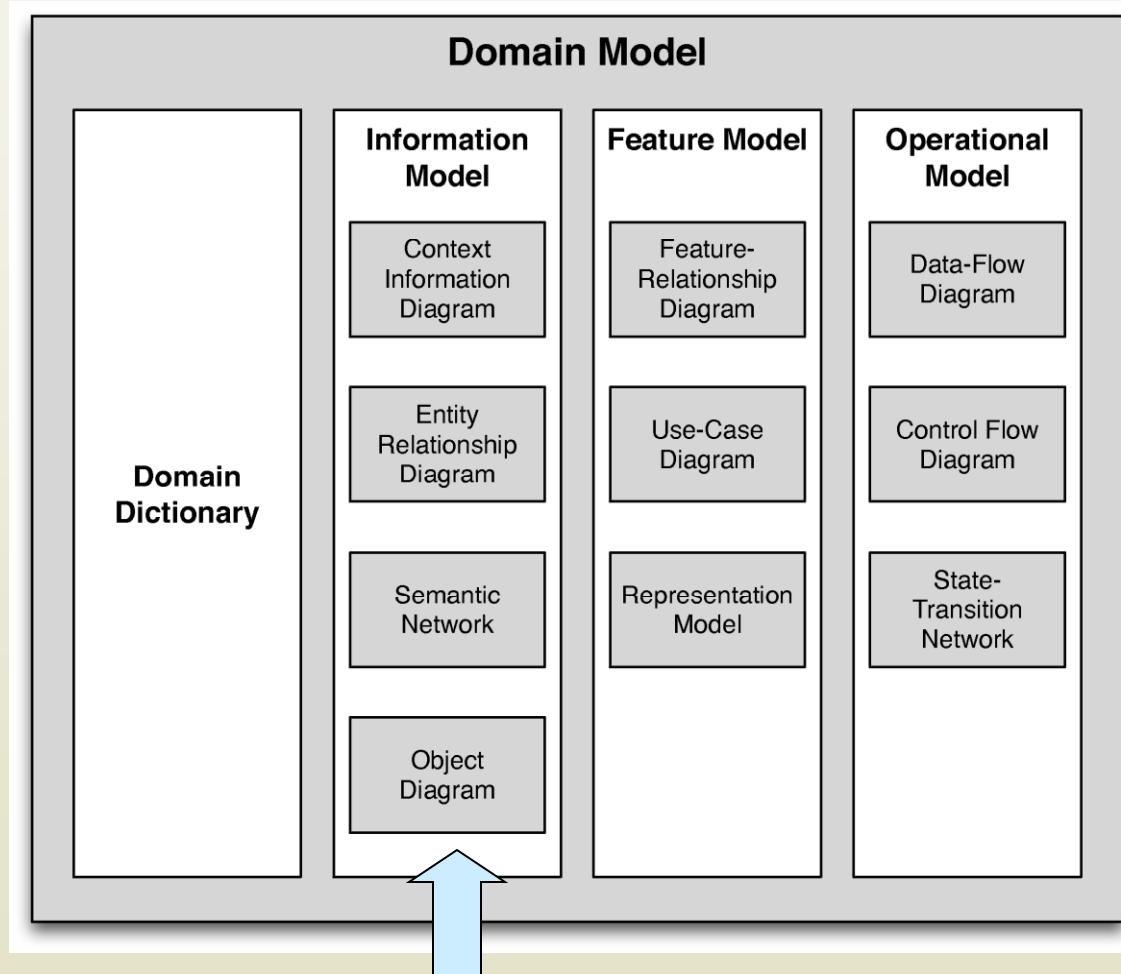


Domain Model



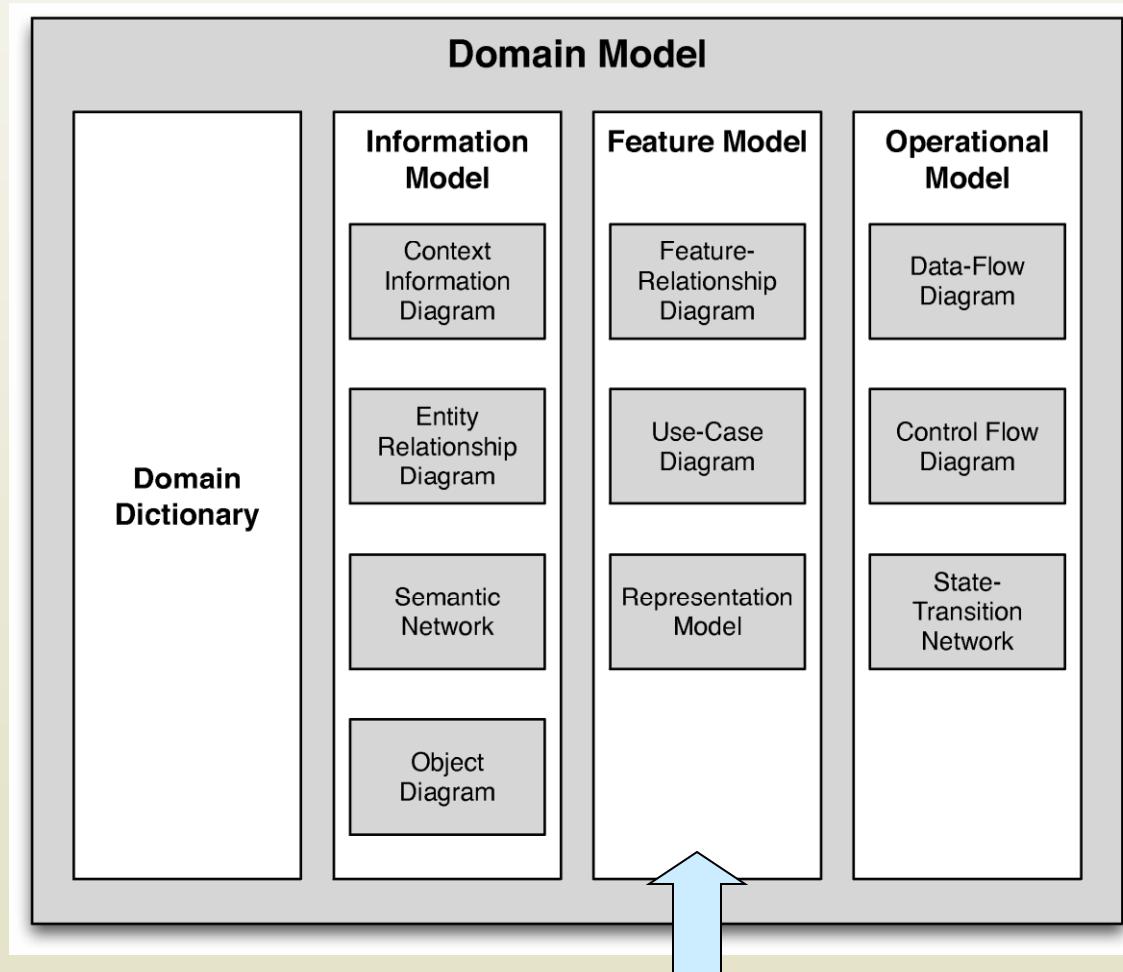
- Defines vocabulary for the domain

Domain Model



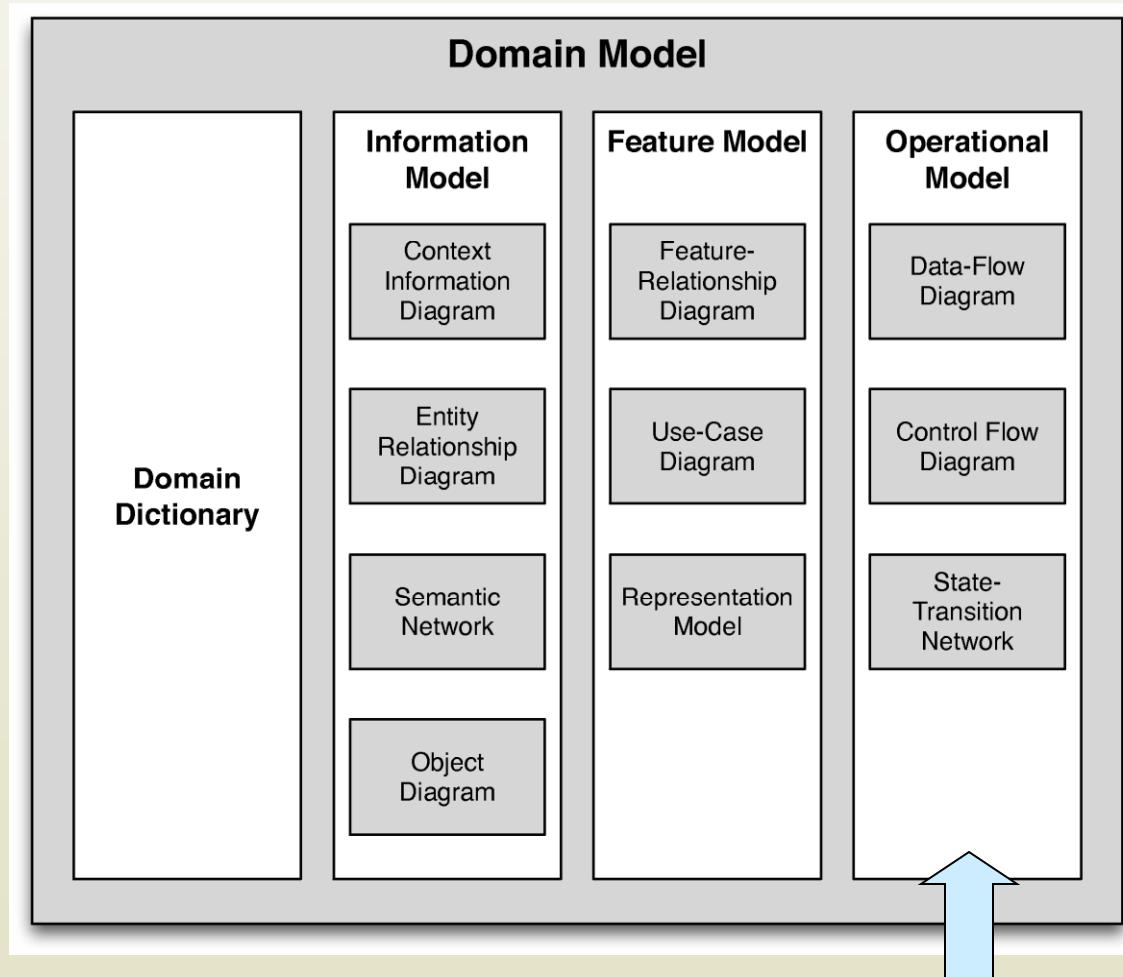
- Describes the entities and data in the domain

Domain Model



- Defines how entities and data combine to provide features 23

Domain Model



- Defines how data and control flow through entities

(Partial) Domain Dictionary

Lunar Module (LM): this is the portion of the spacecraft that lands on the moon. It consists of two main parts: the Ascent Stage (which holds the crew cabin) and the Descent Stage, which contains thrusters used for controlling the landing of the LM.

Reaction Control System (RCS): a system on the Lunar Module responsible for the stabilization during lunar surface ascent/descent and control of the spacecraft's orientation (attitude) and motion (translation) during maneuvers

Vertical velocity (see also One-dimensional motion):

For a free-falling object with no air resistance, ignoring the rotation of the lunar surface, the altitude is calculated as follows:

$$y = \frac{1}{2} * a * t^2 + v_i * t + y_i$$

y = altitude

a = constant acceleration due to gravity on a lunar body (see **Acceleration** for sample values)

t = time in seconds; v_i = initial velocity; y_i = initial altitude

When thrust is applied, the following equation is used:

$$y = \frac{1}{2} * (a_{burner} - a_{gravity}) * t^2 + v_i * t + y_i$$

y = altitude

a_{burner} = constant acceleration upward due to thrust

$a_{gravity}$ = constant acceleration due to gravity on a lunar
(see **Acceleration** for sample values)

t = time in seconds; v_i = initial velocity; y_i = initial altitude

Info Model: Context Info Diagram

- Defines high-level entities
- Defines what is considered inside and outside the domain (or subdomains)
- Defines relationships and high-level flows

Info Model: Entity-Relationship Diagram

- Defines entities and cardinal relationships between them

Info Model: Object Diagram

- Defines attributes and operations on entities
- Closely resembles class diagram in UML but may be more abstract

Object	Attributes	Operations
Landing Radar	Altitude Velocity	Sense Altitude (height) Sense Velocity
Descent Engine	Throttle Level Nozzle Direction	Set Throttle Level Change Nozzle Direction (Gimbal) Turn On / Off
Thrust Engines	Firing Mode Attitude Translation	Set Firing Mode to pulse or continuous Change Spacecraft Attitude Change Spacecraft Translation

Feature Model: Feature Relationship Diagram

Feature Relationship Diagram – Landing Phase

Mandatory: The Lunar Lander must continually read altitude from the Landing Radar and relay that data to Houston with less than 500 msec of latency. Astronauts must be able to control the descent of the Lunar Lander using manual control on the descent engine. The descent engine must respond to control commands in 250msec, with or without a functioning DSKY...

Optional/Variant: Lunar Lander provides the option to land automatically or allow the crew to manually steer the spacecraft.

Quality Requirements:

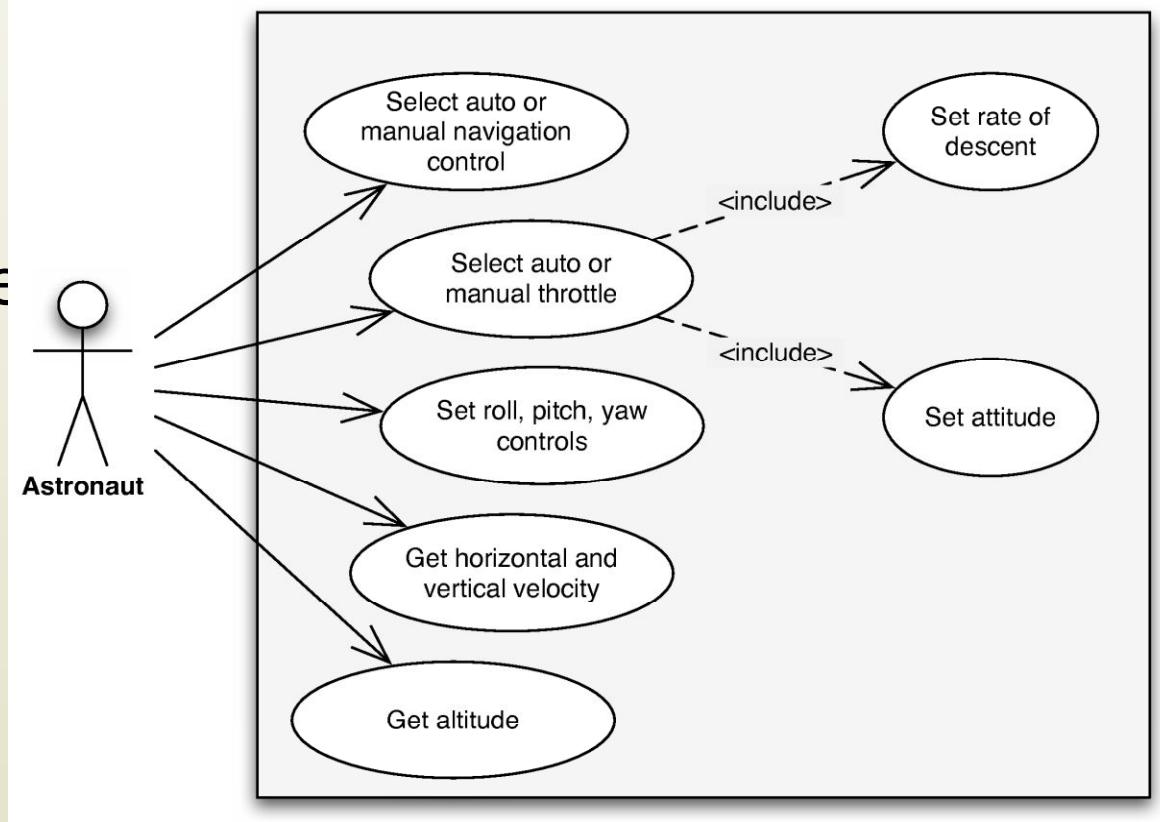
Real-time requirements: The thrusters and the descent engine must be able to respond to commands from the computer system in real-time.

Fault tolerance: Lunar Lander must be able to continue in its flight-path even when the main computer system (Primary Navigation Guidance & Control) goes down. Lunar Lander must be able to maintain system altitude even when one of the thrusters and propellant supplies goes down in the Reaction Control System.

- Describes overall mission operations of a system
- Describes major features and decomposition

Feature Model: Use Case Diagram

- Defines use cases within the domain
- Similar to use case models in UML



Feature Model: Representation Diagram

- Defines how information is presented to human users

Representation Diagram

DSKY Unit – Display and Keyboard Unit

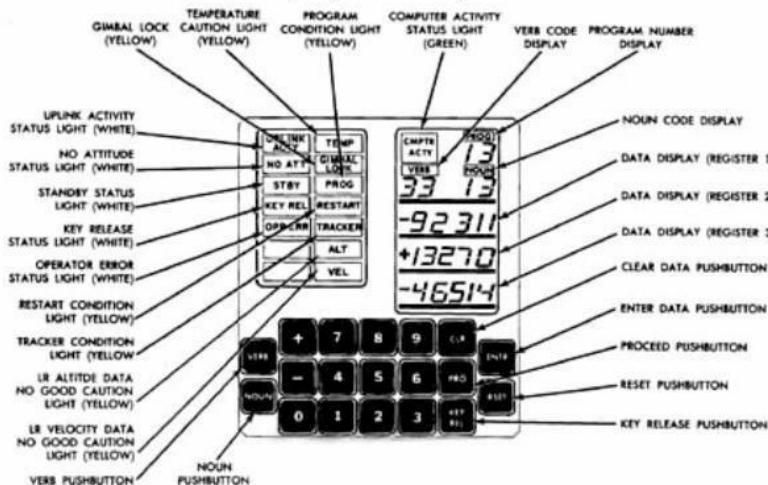


Figure 3: Lunar Module Display and Keyboard Unit (DSKY)

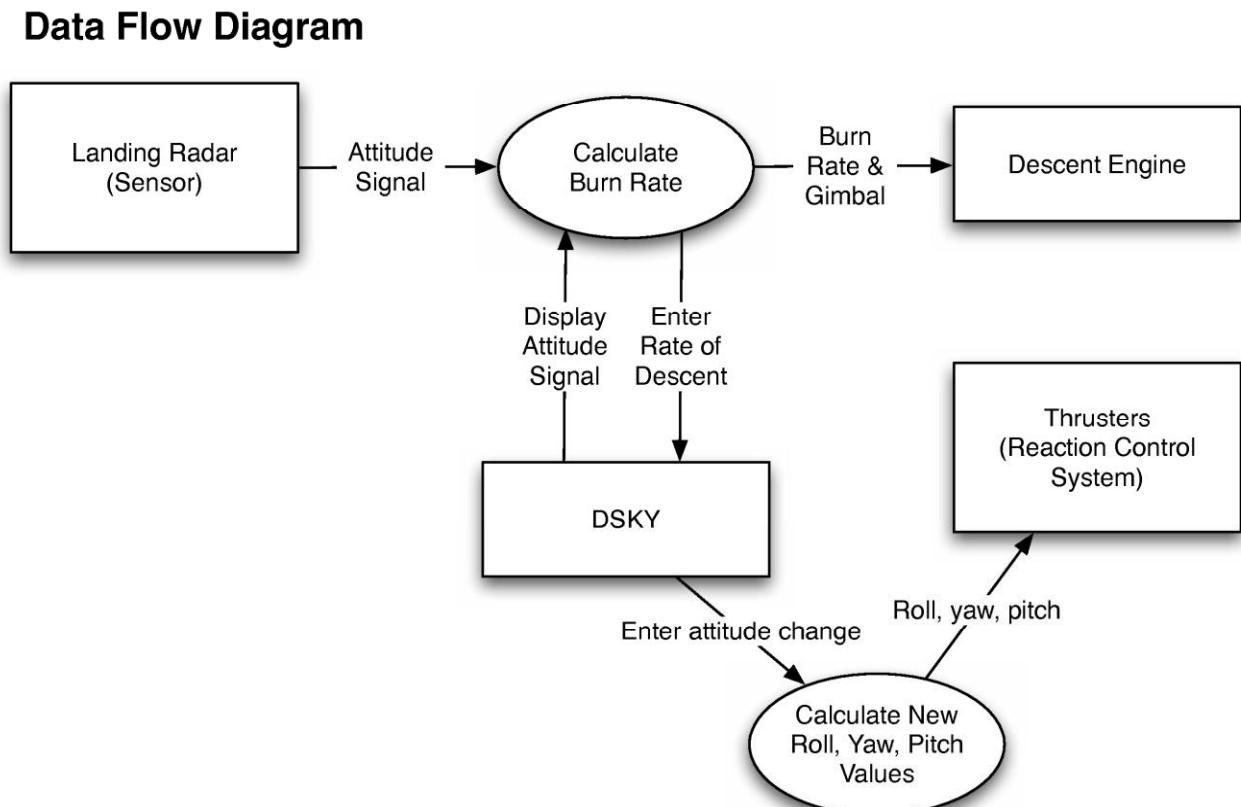
Source: TALES FROM THE LUNAR MODULE GUIDANCE COMPUTER – figure

Apollo 11 The NASA Mission Reports Vol 2 pp 166

- 3 five-digit registers – general purpose
- 3 two-digit registers – indicate phase for landing
- 19 keys
- Warning lights
- Issue commands via VERB & NOUN
 - VERB is the action
 - NOUN is the object to which the action is applied
 - Ex: VERB 6 NOUN 20
VERB 6 = Display in decimal
NOUN 20 = Angles
- 70 predefined PROGRAMS
 - Ex: PROGRAM for each descent phase executes trajectory
 - P63 – Braking Phase
 - P64 – Approach Phase
 - P65 – Landing Phase

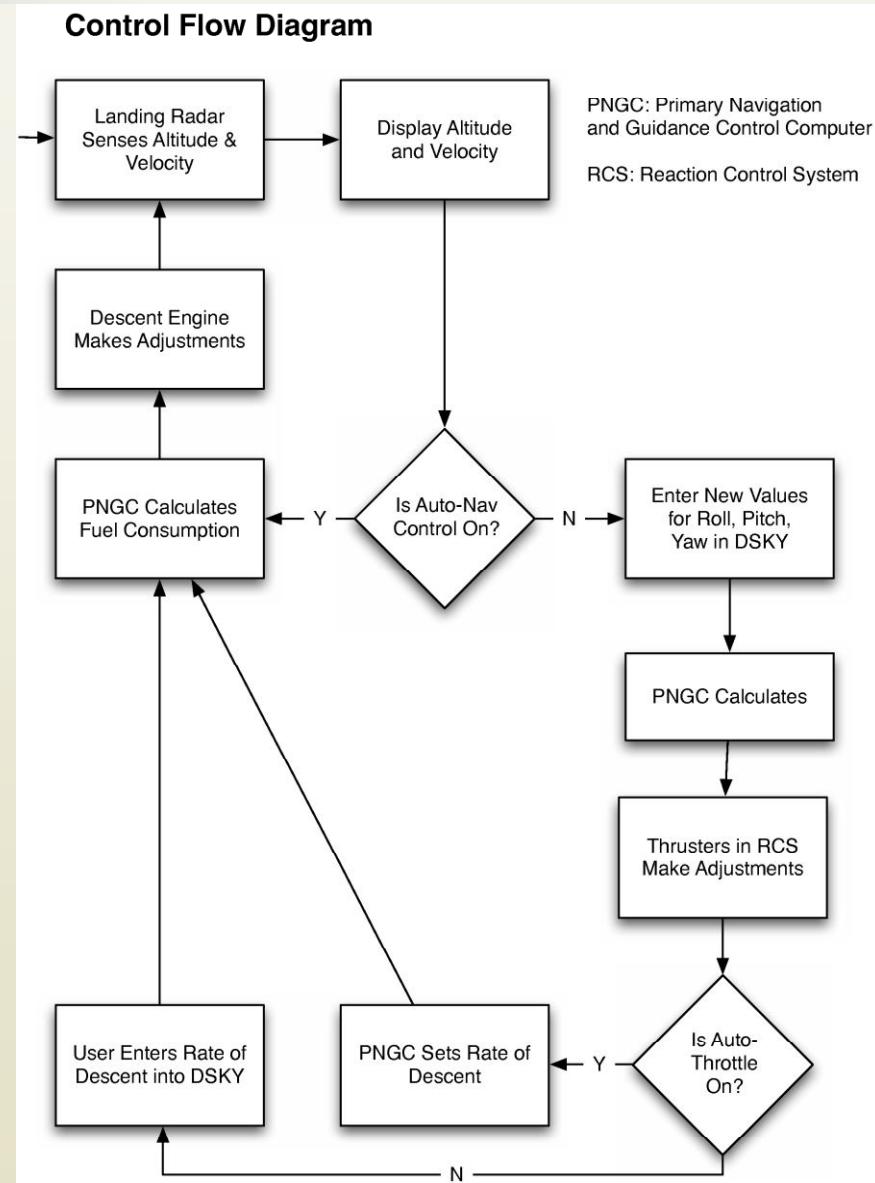
Operational Model: Data Flow Diagram

- Focuses on data flow between entities with no notion of control



Operational Model: Control Flow Diagram

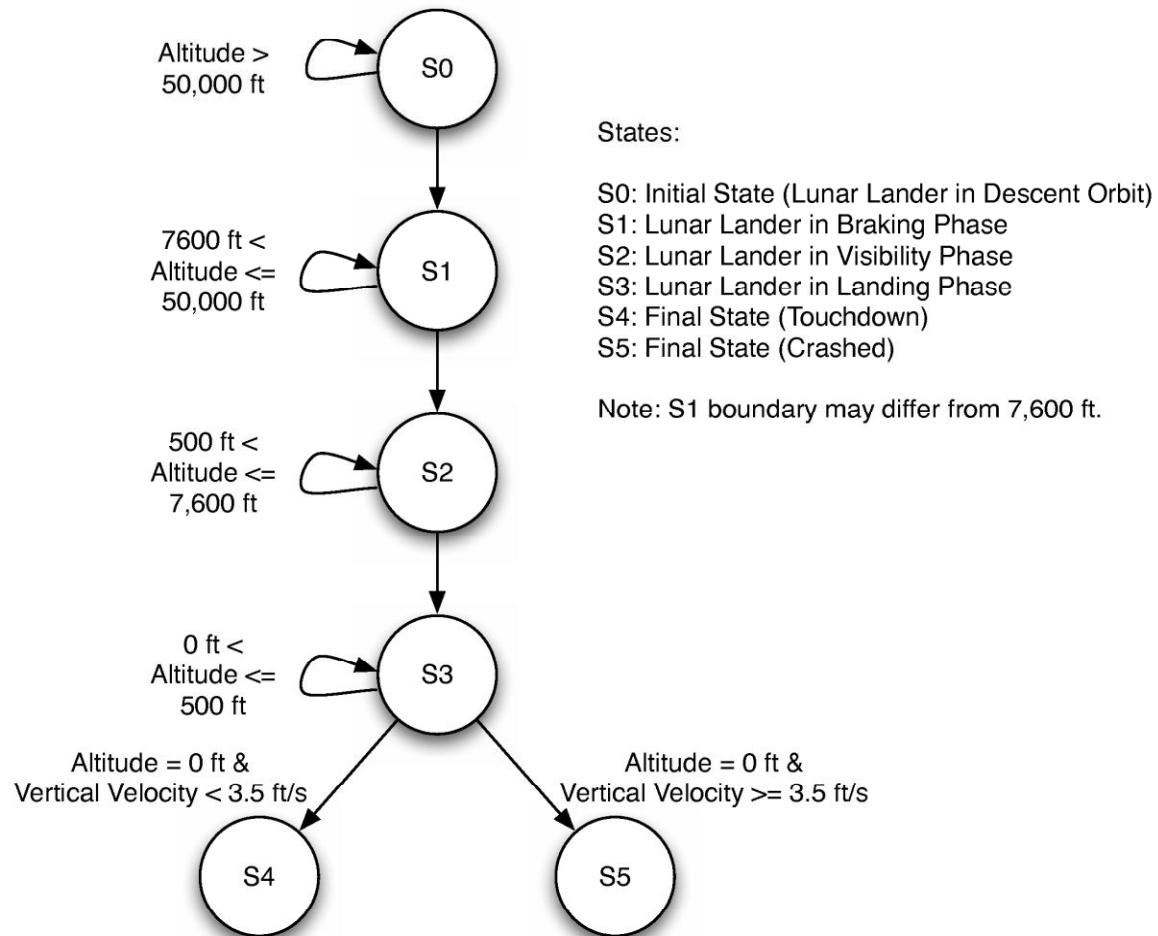
- Focuses on control flow between entities separate from data flow



Operational Model: State Transition Diagram

- Focuses on states of systems and transitions between them
- Resembles UML state diagrams

State Transition Diagram



Reference Requirements

- **Mandatory**
 - ◆ *Must display the current status of the Lunar Lander (horizontal and vertical velocities, altitude, remaining fuel)*
 - ◆ *Must indicate points earned by player based on quality of landing*
- **Optional**
 - ◆ *May display time elapsed*
- **Variant**
 - ◆ *May have different levels of difficulty based on pilot experience (novice, expert, etc)*
 - ◆ *May have different types of input depending on whether
 - Auto Navigation is enabled
 - Auto Throttle is enabled*
 - ◆ *May have to land on different celestial bodies
 - Moon
 - Mars
 - Jupiter's moons
 - Asteroid*

Domain-Specific Software Architecture

- **Definition:** Definition. A domain-specific software architecture (DSSA) comprises:
 - ◆ a reference architecture, which describes a general computational framework for a significant domain of applications;
 - ◆ a component library, which contains reusable chunks of domain expertise; and
 - ◆ an application configuration method for selecting and configuring components within the architecture to meet particular application requirements.

(Hayes-Roth)

Reference Architecture

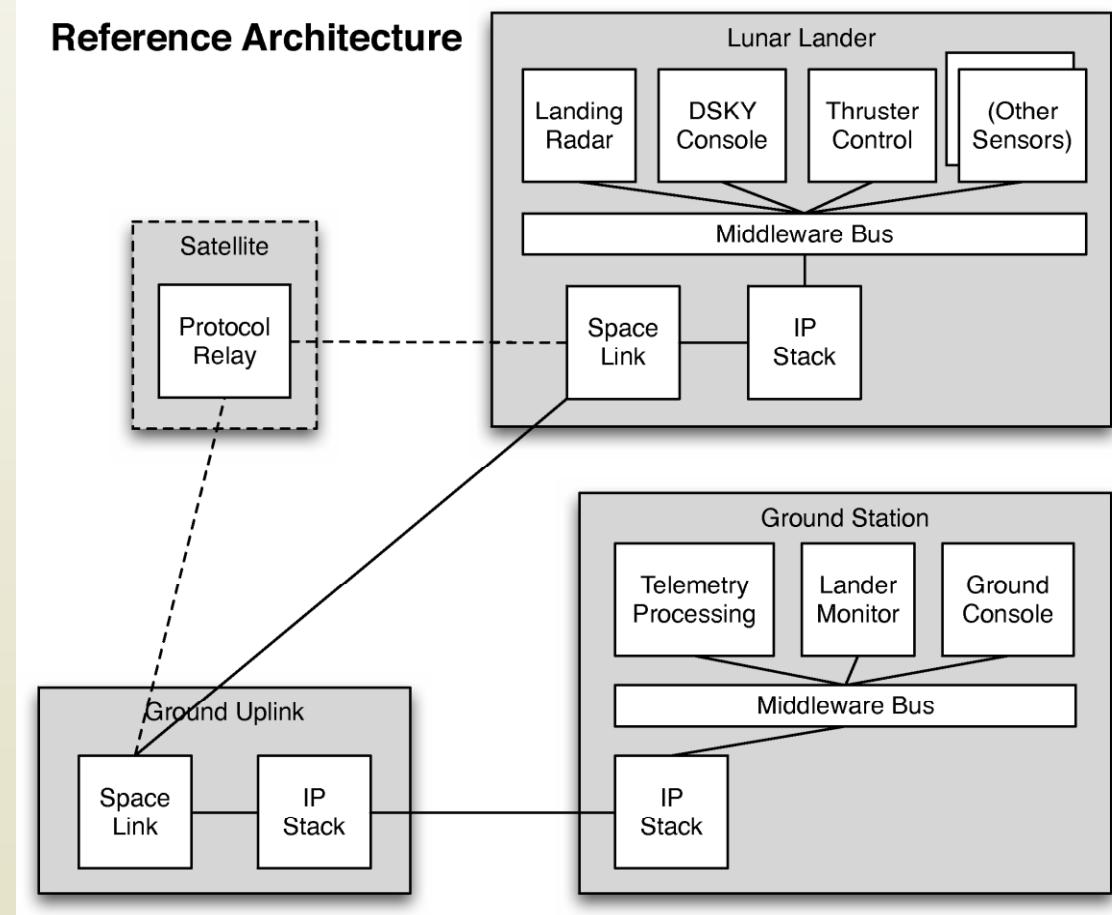
- **Definition.** *Reference architecture* is the set of principal design decisions that are simultaneously applicable to multiple related systems, typically within an application domain, with explicitly defined points of variation.
- Reference architectures are still architectures (since they are also sets of principal design decisions)
 - ◆ Distinguished by the presence of explicit points of variation (explicitly “unmade” decisions)

Different Kinds of Reference Architecture

- Complete single product architecture
 - ◆ A fully worked out exemplar of a system in a domain, with optional documentation as to how to diversify
 - Can be relatively weak due to lack of explicit guidance and possibility that example is a 'toy'
- Incomplete invariant architecture
 - ◆ Points of commonality as in ordinary architecture, points of variation are indicated but omitted
- Invariant architecture with explicit variation
 - ◆ Points of commonality as in ordinary architecture, specific variations indicated and enumerated

Example Reference Architecture

- Structural view of Lunar Lander DSSA
- Invariant with explicit points of variation
 - ◆ Satellite relay
 - ◆ Sensors



Domain-Specific Software Architecture and Product Lines

Software Architecture
Lecture 24

Objectives

- Concepts
 - ◆ What is domain-specific software engineering (DSSE)
 - ◆ The “Three Lampposts” of DSSE: Domain, Business, and Technology
 - ◆ Domain Specific Software Architectures
- Product Lines
- Relationship between DSSAs, Product Lines, and Architectural Styles
- Examples of DSSE at work

Objectives

- Concepts
 - ◆ What is domain-specific software engineering (DSSE)
 - ◆ The “Three Lampposts” of DSSE: Domain, Business, and Technology
 - ◆ Domain Specific Software Architectures
- Product Lines
- Relationship between DSSAs, Product Lines, and Architectural Styles
- Examples of DSSE at work

Product Lines

- A set of related products that have substantial commonality
 - ◆ In general, the commonality exists at the architecture level
- One potential ‘silver bullet’ of software engineering
 - ◆ Power through reuse of
 - Engineering knowledge
 - Existing product architectures, styles, patterns
 - Pre-existing software components and connectors

Domains vs. Product Lines

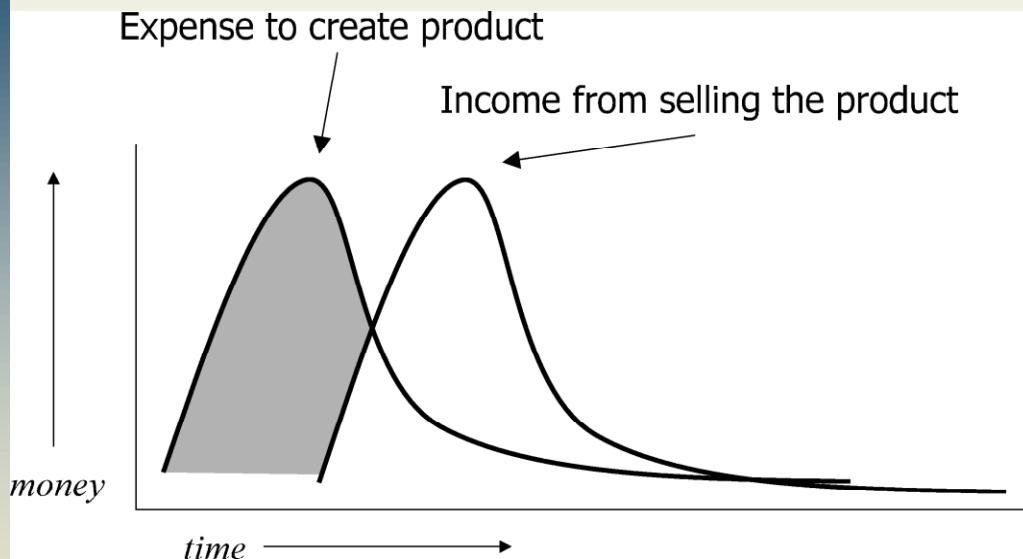
Domains are in the problem space, product lines are in the solution space

- Domain
 - ◆ Consumer electronics
 - ◆ Avionics
 - ◆ Compilers
 - ◆ Video games
- Product Line
 - ◆ Sony WEGA TVs
 - ◆ Boeing 747 Family
 - ◆ GNU compiler suite
 - ◆ Suite of games built on same engine

Business vs. Engineering Product Lines

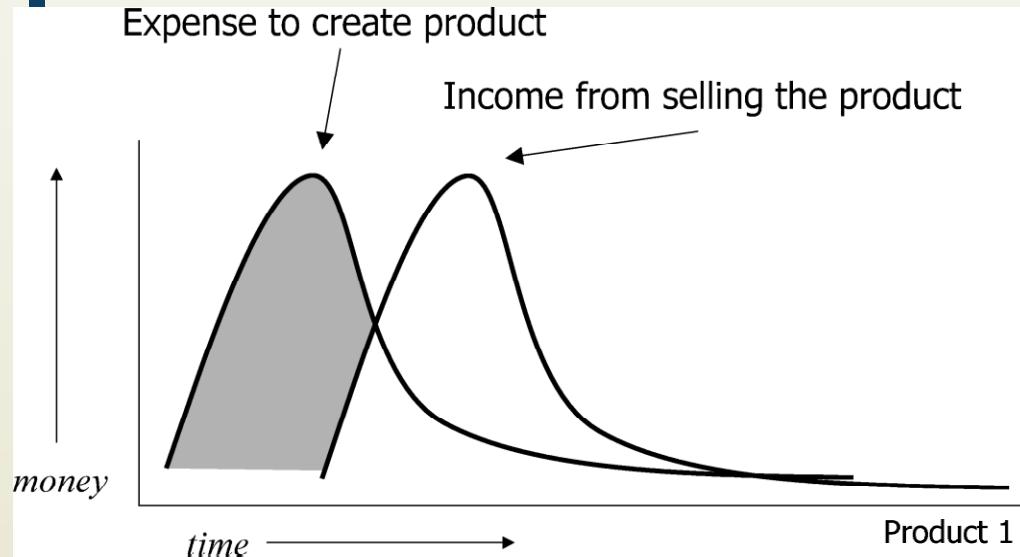
- Business Product Line
 - ◆ A set of products marketed under a common banner to increase sales and market penetration through bundling and integration
- Engineering Product Line
 - ◆ A set of products that have substantial commonality from a technical/engineering perspective
- Generally, business product lines *are* engineering product lines and vice-versa, but not always
 - ◆ Applications bundled after a company acquisition
 - ◆ Chrysler Crossfire & Mercedes SLK V6

Business Motivation for Product Lines

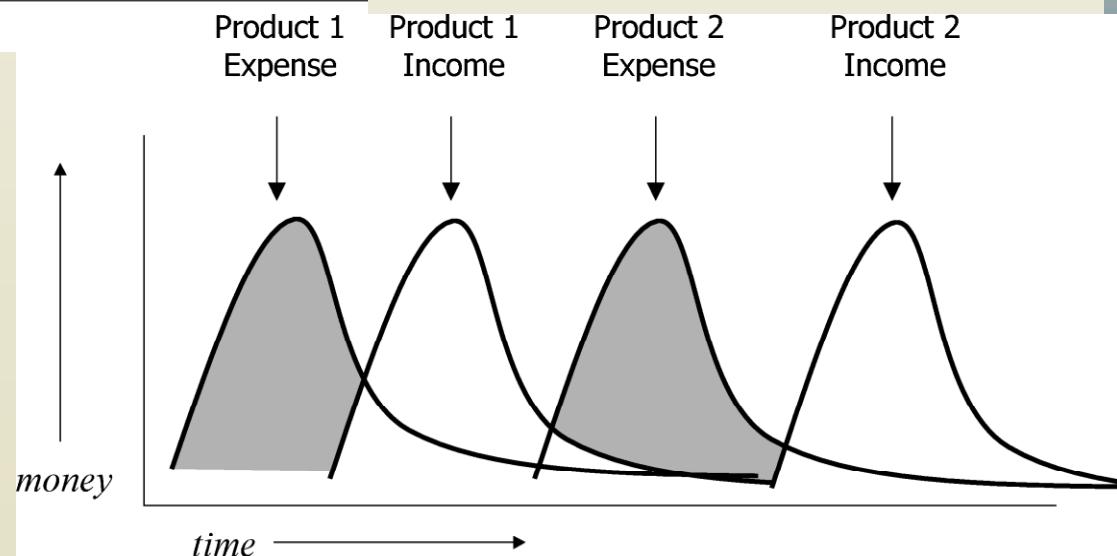


Traditional Software
Engineering

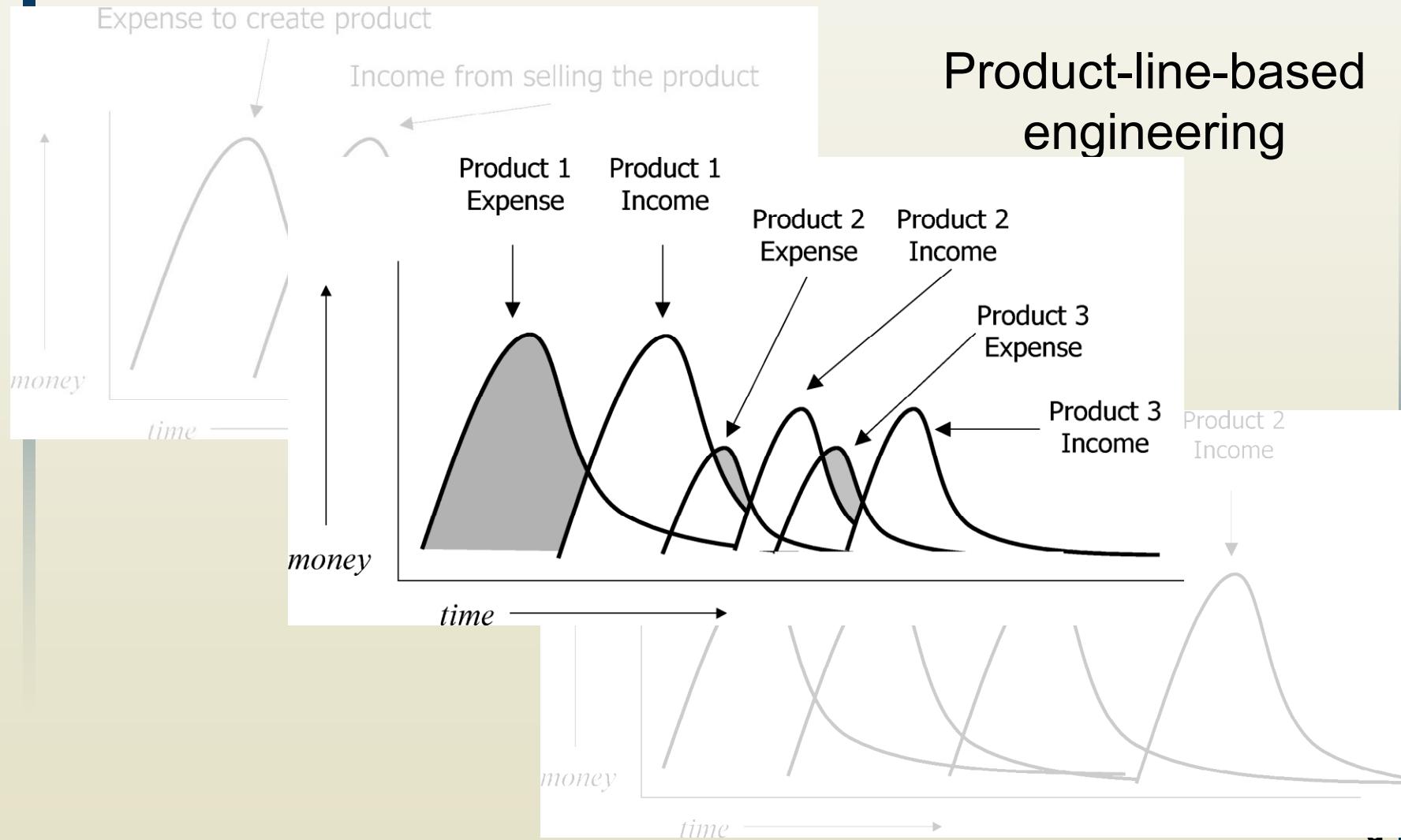
Business Motivation for Product Lines



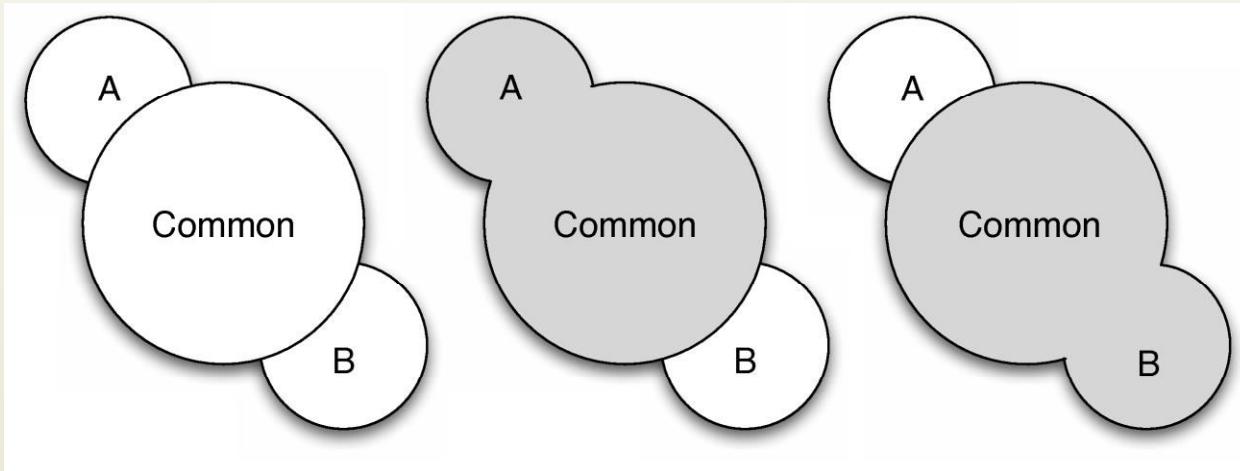
Traditional Software
Engineering



Business Motivation for Product Lines



Capturing Product Line Architectures

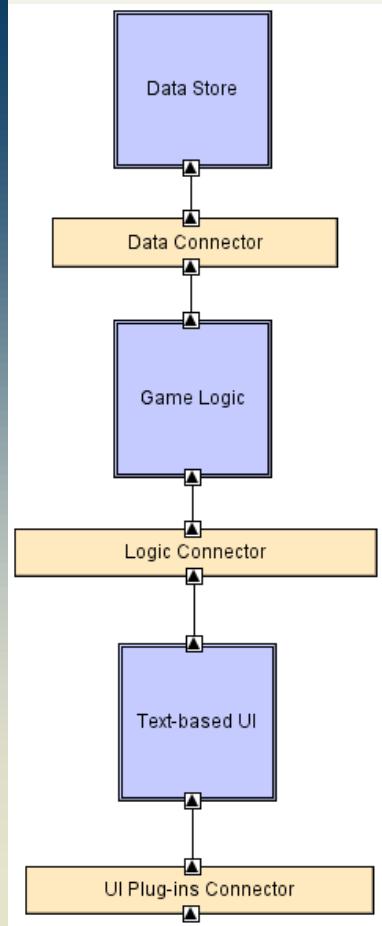


- Common: features common to all products
- A: features specific to product A
- B: features specific to product B
- Product A = Common + A
- Product B = Common + B

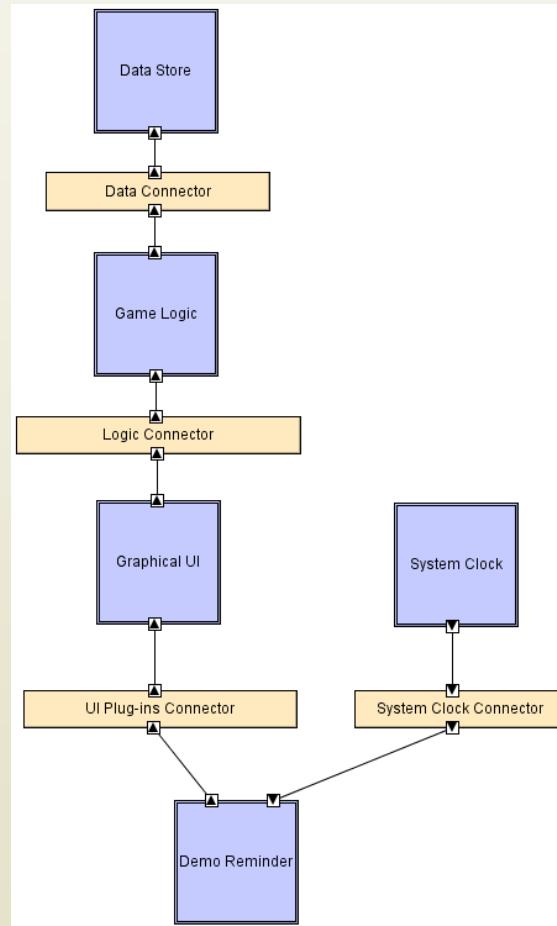
A Product-Line Architecture

- **Definition:** A product-line architecture captures the architectures of many related products simultaneously
- Generally employs explicit *variation points* in the architecture indicating where design decisions may diverge from product to product

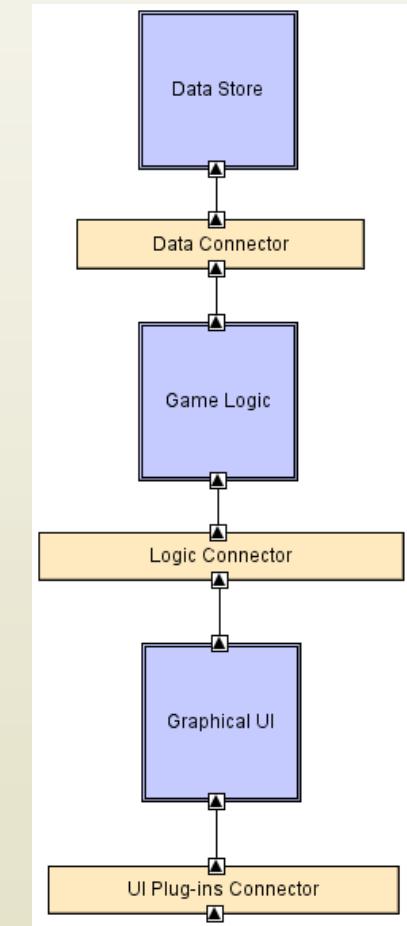
A Lunar Lander Product Line



“Lite”



“Demo”



“Pro”

Product Component Table

- Helps us decide whether creating a product line is viable or feasible

	Data Store	Data Store Connector	Game Logic	Game Logic Connector	Text-based UI	UI Plug-ins Connector	Graphical UI	System Clock	System Clock Connector	Demo Reminder
Lite	X	X	X	X	X	X				
Demo	X	X	X	X	X	X	X	X	X	X
Pro	X	X	X	X		X	X			

Group Components into Features

- Not a mechanical process
- Attempt to identify (mostly) orthogonal features, or features that would be beneficial in different products

	Data Store	Data Store Connector	Game Logic	Game Logic Connector	Text-based UI	UI Plug-ins Connector	Graphical UI	System Clock	System Clock Connector	Demo Reminder
<i>Core Elements</i>	X	X	X	X		X				
Text UI					X					
Graphical UI							X			
Time Limited								X	X	X

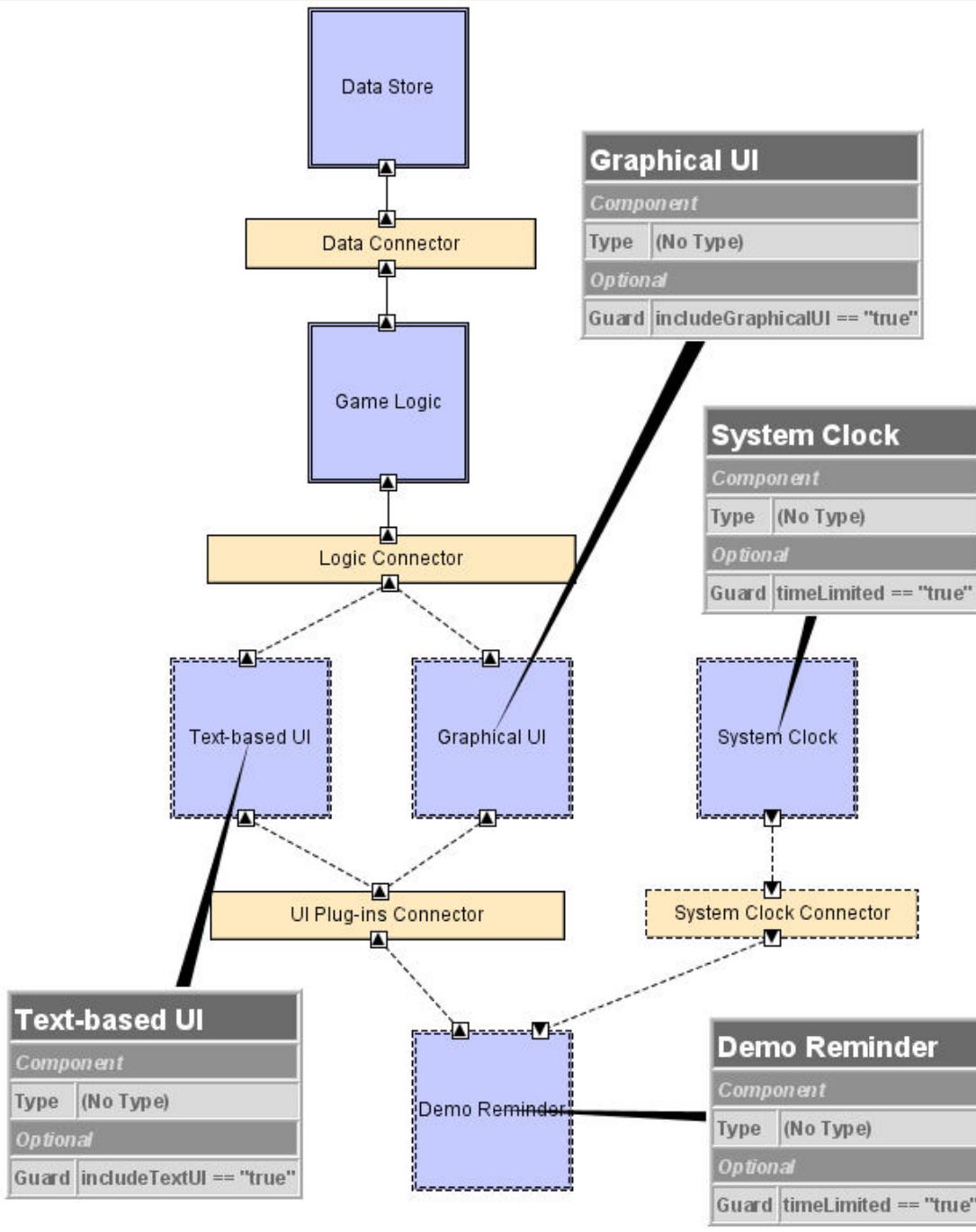
Reconstitute Products from Features

- Use technical and business knowledge to identify which combinations form feasible or marketable products that will be constructed

<i>Core Elements</i>	Text UI	Graphical UI	Time Limited
Lunar Lander Lite	X	X	
Lunar Lander Demo	X	X	X
Lunar Lander Pro	X	X	

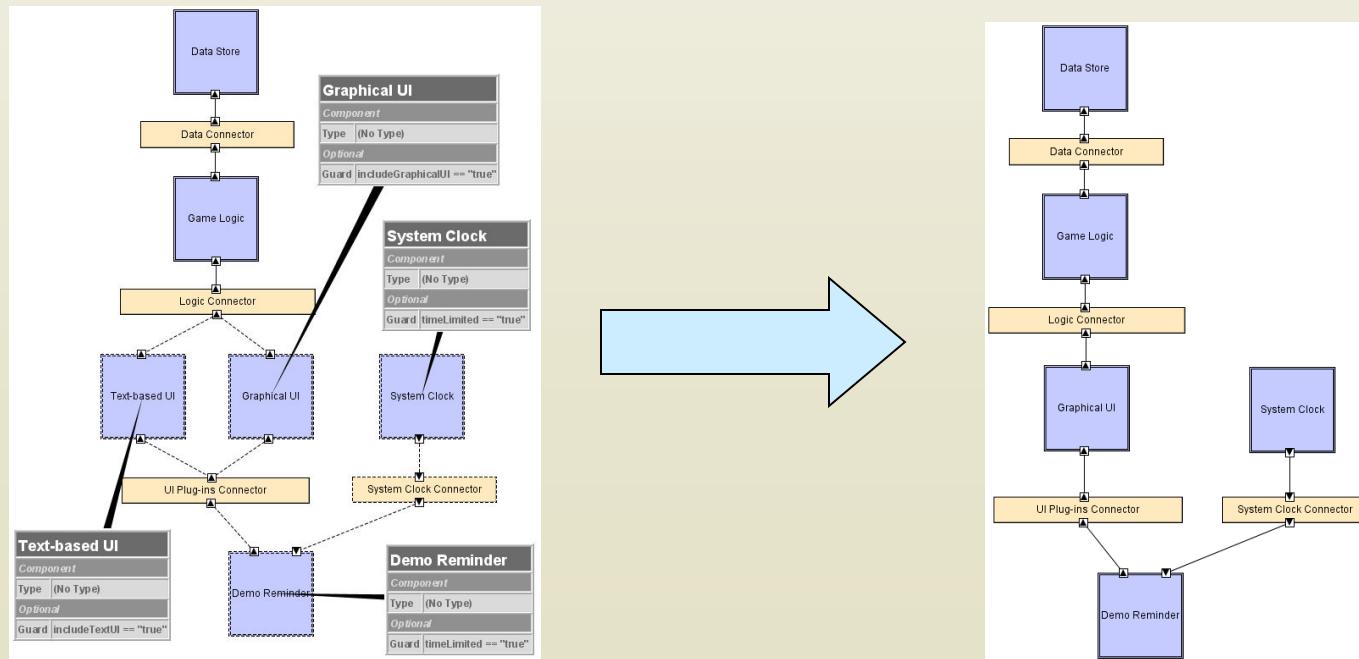
Modeling Product Line Architectures

- Architectural models need to be diversified with information about variation points and features
- Not all ADLs have good support for this
 - ◆ Exceptions include
 - Koala
 - xADL 2.0
 - ◆ These ADLs have explicit support for capturing variation points



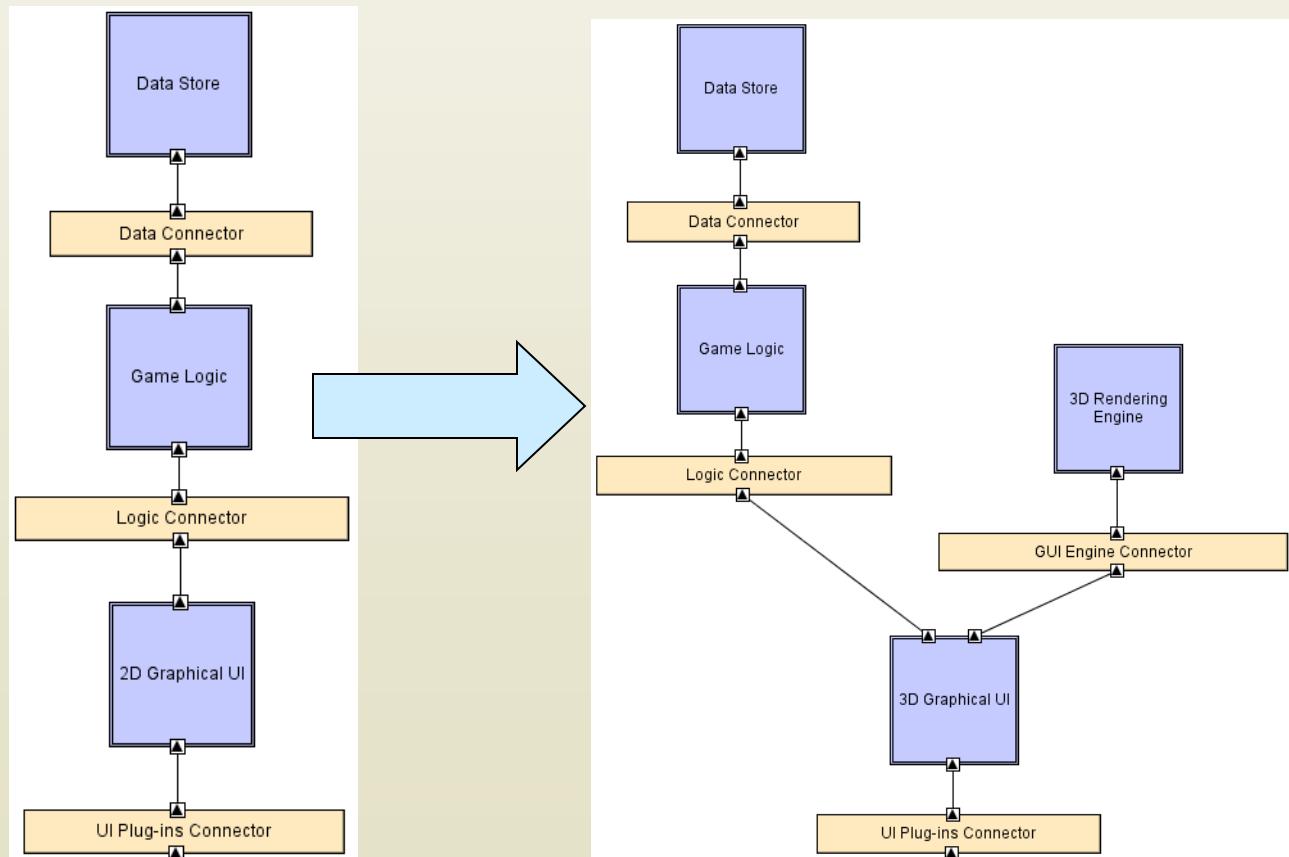
Selection

- Product-line selection is the process of extracting a single product architecture (or smaller product line) from an architectural model that contains explicit points of variation
- ADLs such as Koala and xADL 2.0 can do selection automatically with tools

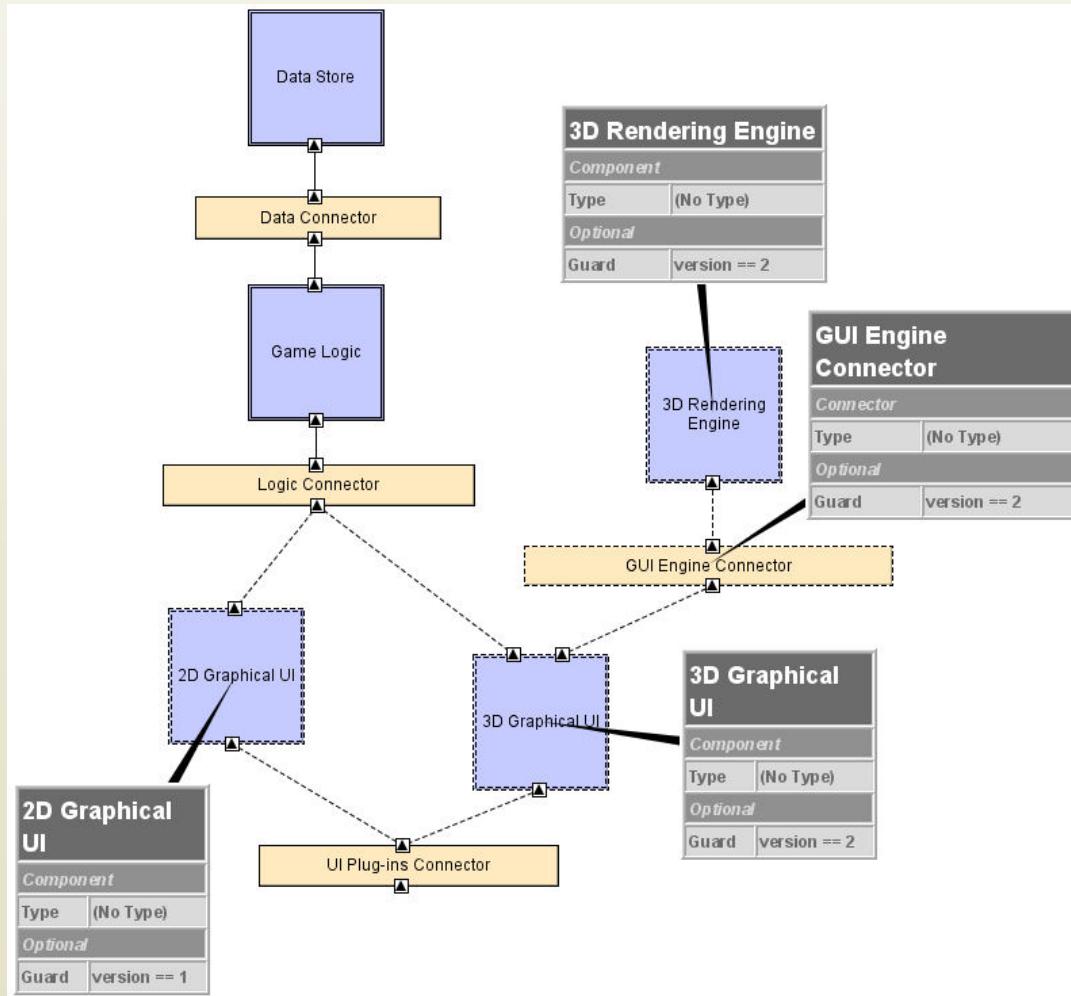


Product Lines for Evolution

- Products in a product line don't have to exclusively capture alternatives
 - ◆ They can also capture variation over time



Product Lines for Evolution

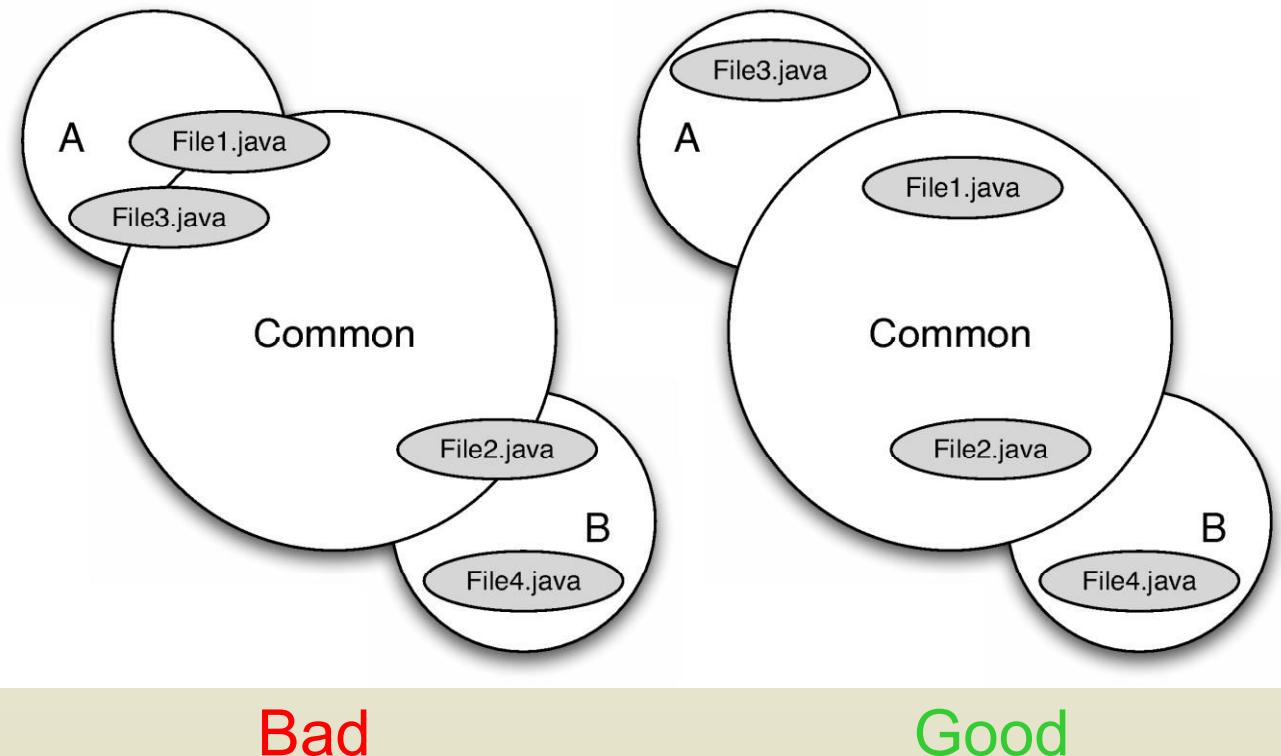


Product Lines for 'What-If' Analysis

- In addition to alternative products and different versions of the same product, product lines can capture different *potential* products
 - ◆ Selection can be used to quickly generate product architectures for potential products
 - ◆ These can be checked for certain properties or subjected to more rigorous analysis for feasibility or quality
 - ◆ Can also be used to generate new product ideas

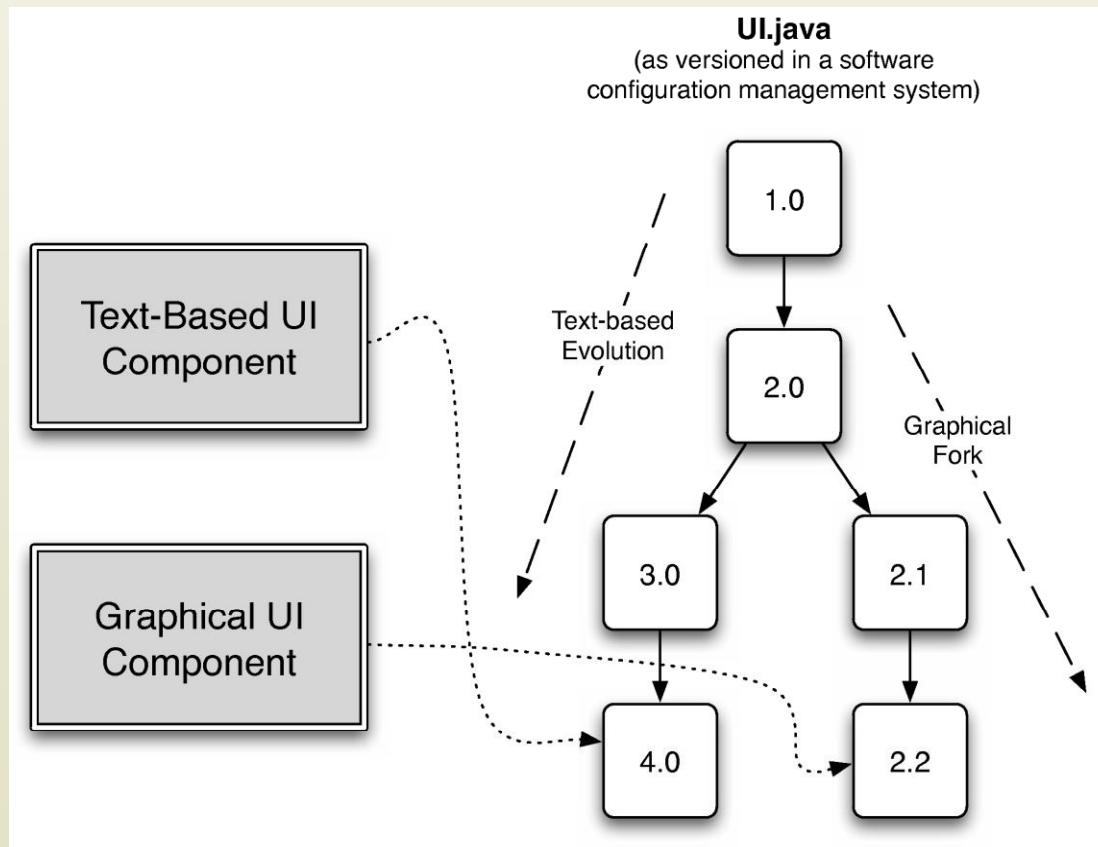
Implementation Issues

- Important to partition implementations along variation-point boundaries



Implementation Issues (cont'd)

- Keeping evolving architectures and version-controlled source repositories (e.g., RCS, CVS, Subversion) in sync



Unifying Products with Different Heritage

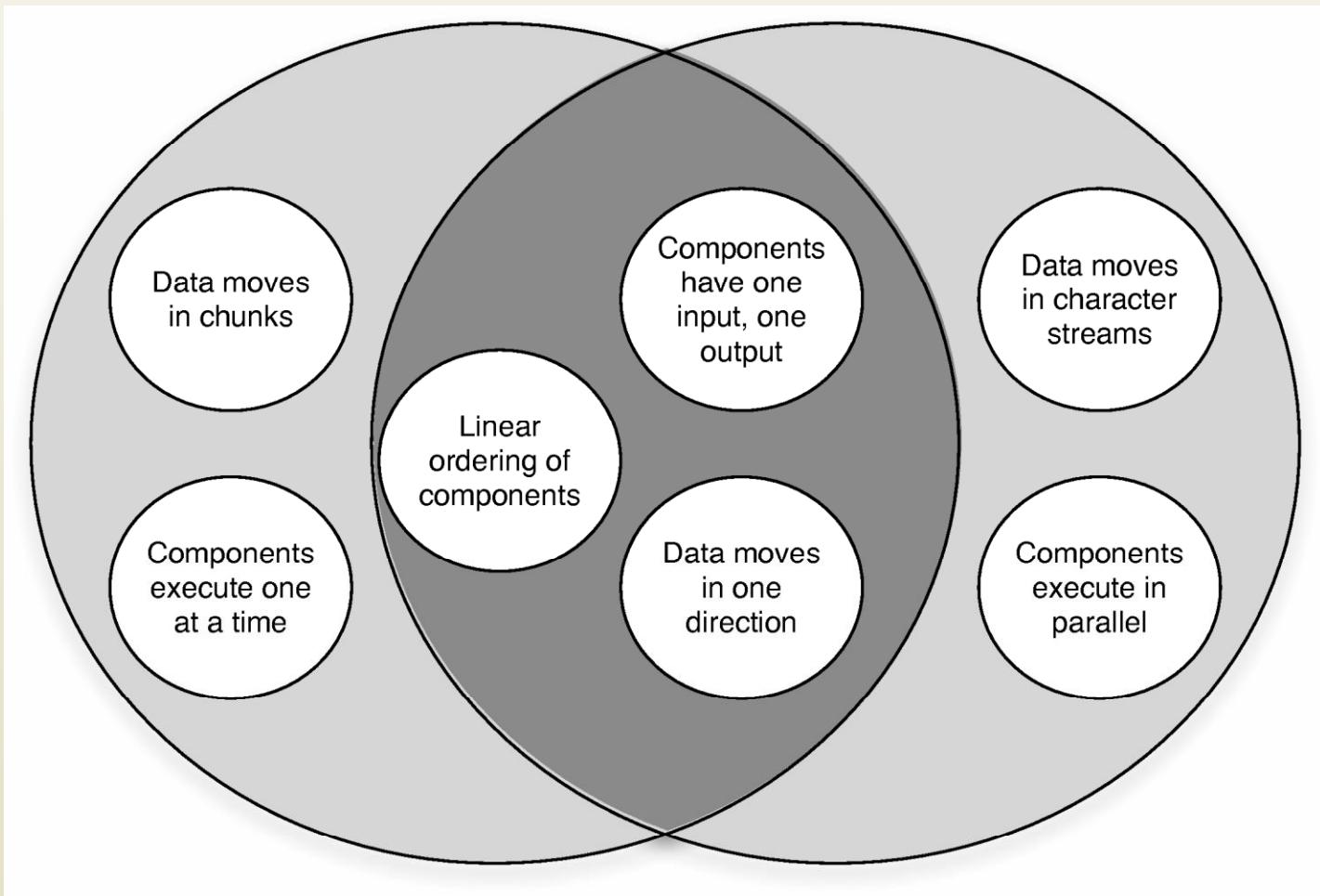
- Often, the idea to create a product line emerges *after* several products have been implemented and commonality is noticed
- Strategies include
 - ◆ No product line
 - It may be more expensive to create a product line or there may not be enough commonality
 - ◆ One master product
 - One product architecture becomes the basis for the product line
 - ◆ Hybrid
 - A new product line architecture emerges out of many products
 - Seems ideal but can be hard in practice

Architectural Styles, DSSAs, Product Lines

- Architectural Styles
 - ◆ Can be general or domain-specific
 - ◆ Provides general guidelines for diverse kinds of applications; focus on –ility development
- DSSAs
 - ◆ Domain specific; includes elaborate domain model and specific reference architecture
- Product lines
 - ◆ Explicit set of related products with common aspects



Families of Styles



Objectives

- Concepts
 - ◆ What is domain-specific software engineering (DSSE)
 - ◆ The “Three Lampposts” of DSSE: Domain, Business, and Technology
 - ◆ Domain Specific Software Architectures
- Product Lines
- Relationship between DSSAs, Product Lines, and Architectural Styles
- Examples of DSSE at work

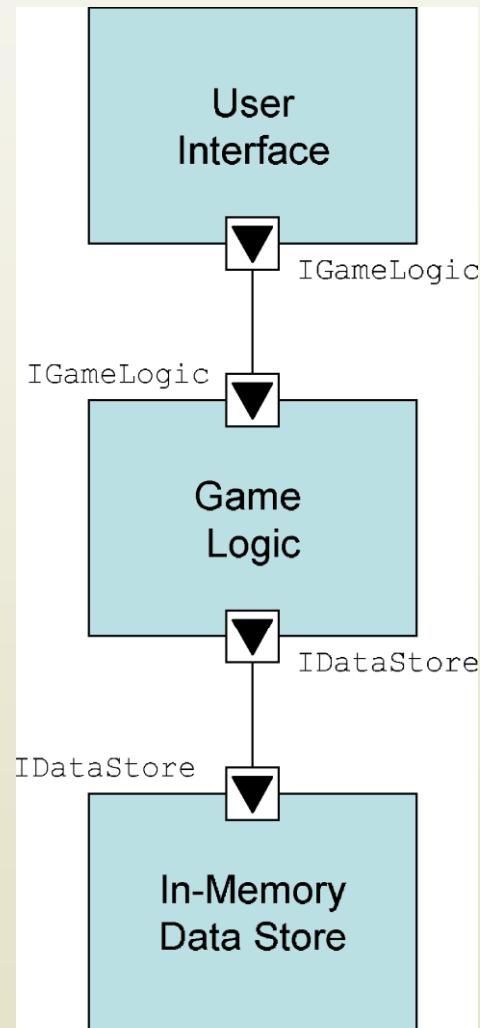
Koala

- An ADL and tool-set for developing product lines of consumer electronics
- A successor to Darwin in terms of structural modeling
- One of the first ADLs to capture points of variation explicitly
- Has tight bindings to implementations such that architecture descriptions can be “compiled” into application skeletons

Lunar Lander in Koala

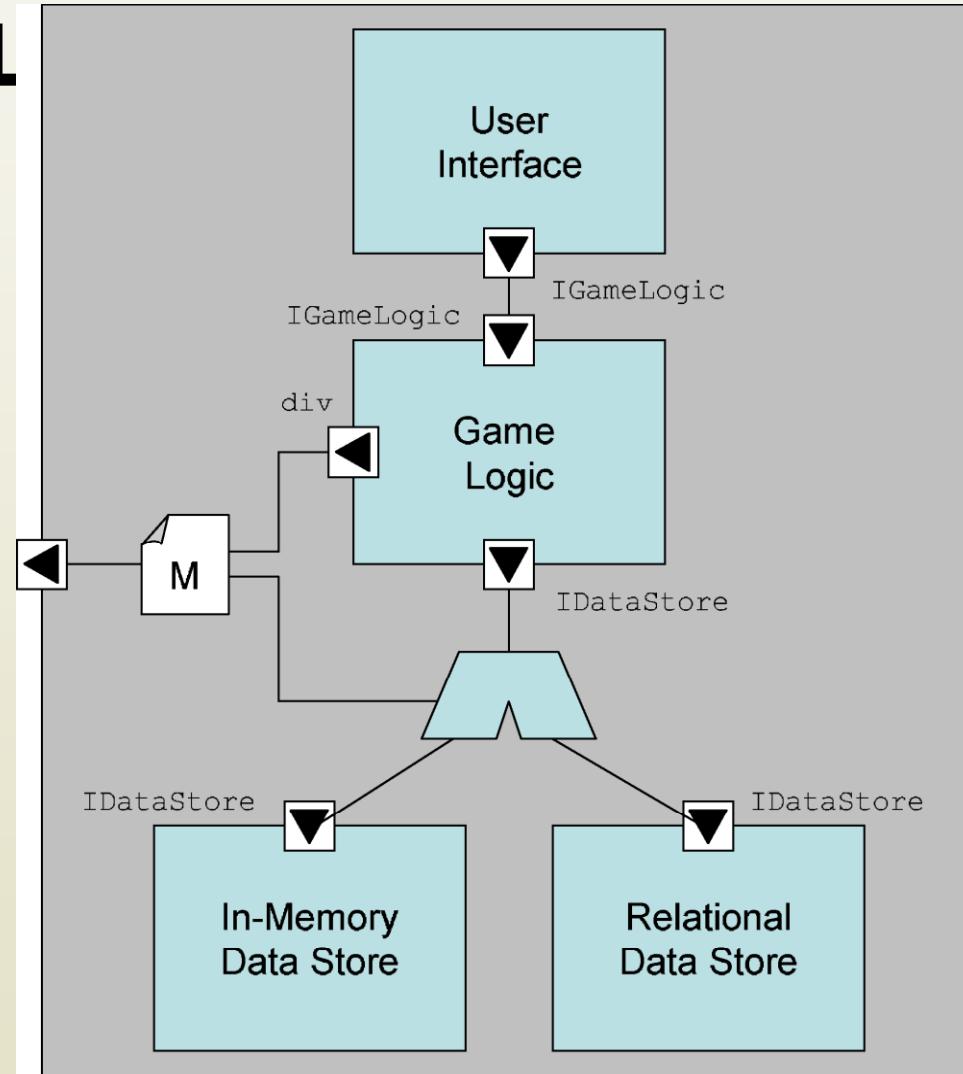
- No product line features yet
- Note similarities to Darwin, xADL 2.0
 - ◆ xADL 2.0 Archipelago visualization is derived from Koala

```
interface IDataStore{  
    void setAltitude(int altitudeInMeters);  
    int getAltitude();  
    void setBurnRate(int newBurnRate);  
    int getBurnRate();  
    ...  
}
```

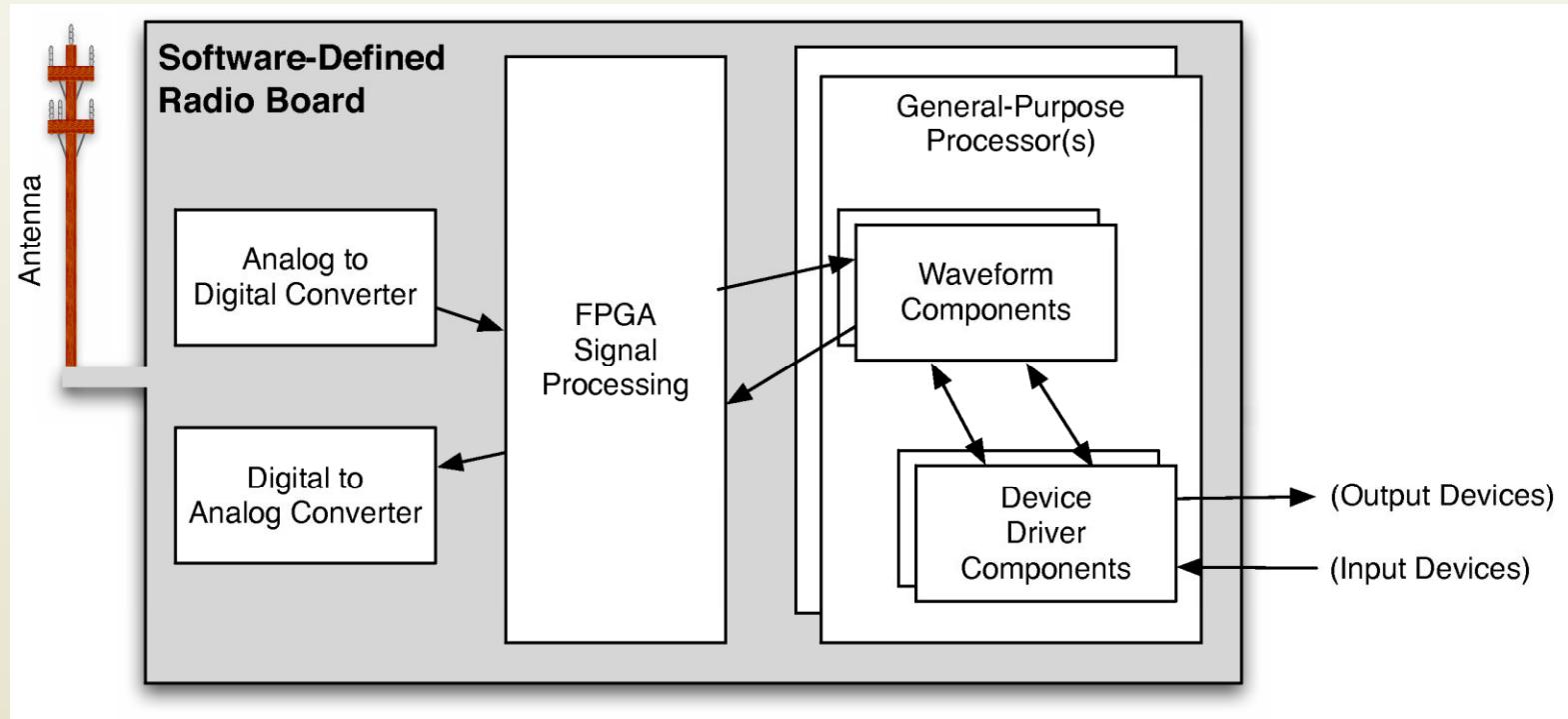


Lunar Lander PL

- Switch construct routes calls to one of two potential data stores
- ‘Diversity interface’ lets game logic component select callee from external config component
- Same IDataStore interface ensures call compatibility

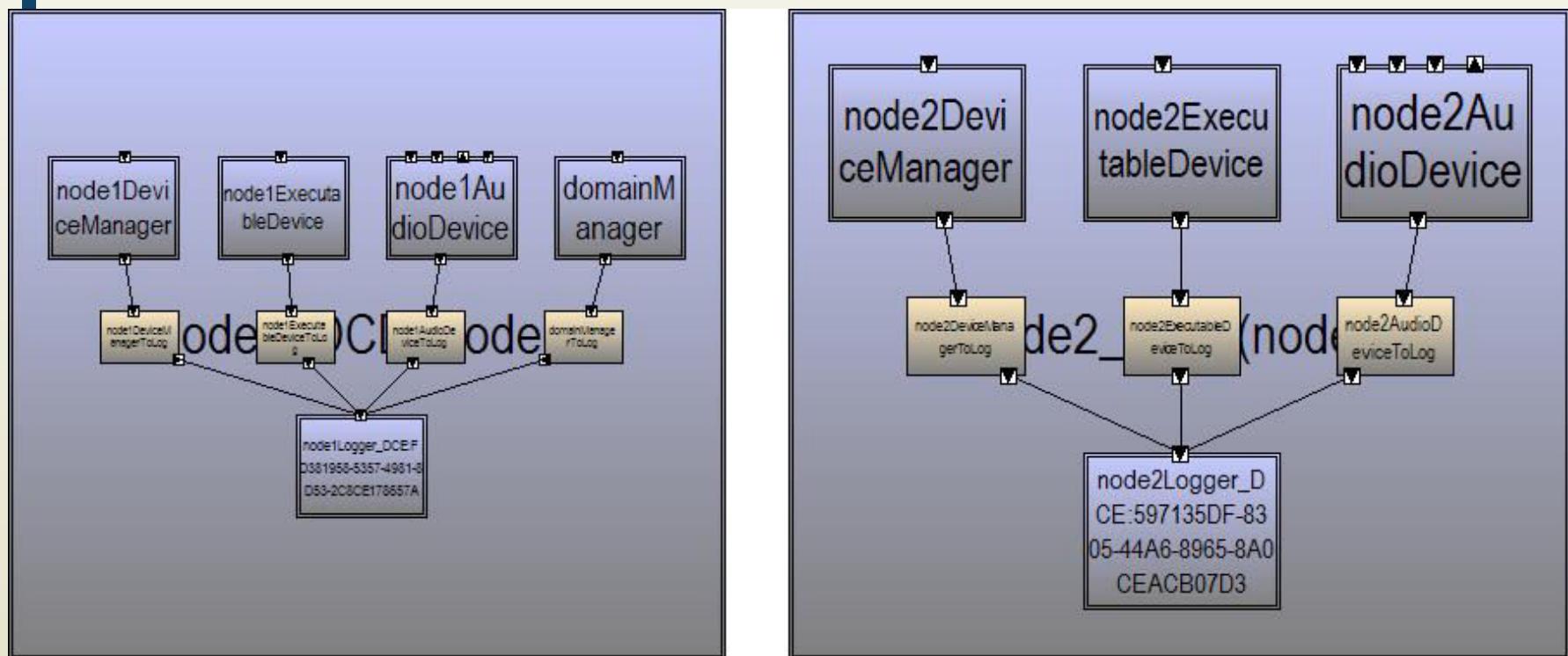


Software Defined Radios



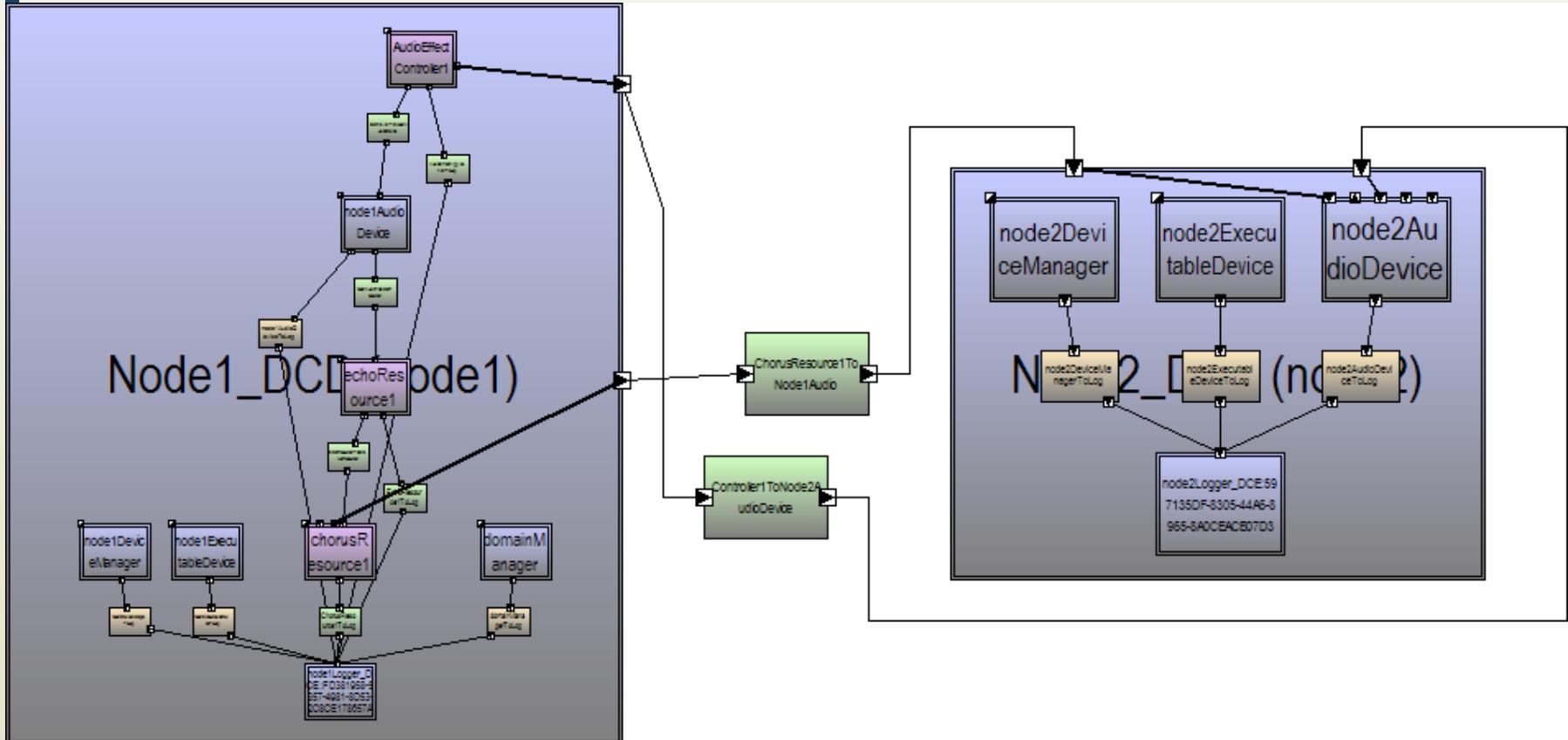
- Variation points in radio configuration, board configuration, software configuration

SDR in xADL 2.0



- Two-node “SCARI” SDR with just device drivers loaded

SDR in xADL 2.0



- Same radio with one waveform deployed

SDRs in xADL 2.0

- By applying product line techniques to SDRs
 - ◆ Can manage different configurations of the radio
 - Deploying components on alternative hosts
 - Deployments with
 - ◆ No waveforms
 - ◆ One waveform
 - ◆ Different combinations of waveforms
 - ◆ Can show radio in different states as radio starts up or transitions from one waveform to another

Architectural Adaptation

**Software Architecture
Lecture 25**

Adaptation

- Change is endemic to software
 - ◆ perceived and actual malleability of software induces stakeholders to initiate changes, e.g.:
 - Users want new features
 - Designer wants to improve performance
 - Application environment is changing
- Adaptation: modification of a software system to satisfy new requirements and changing circumstances

Goals of this Lecture

- Characterize adaptation, showing what changes, why, and who the players are
- Characterize the central role software architecture plays in system adaptation
- Present techniques for effectively supporting adaptation, based on an architecture-centric perspective

Sources and Motivations for Change

- Corrective Changes
 - ◆ Bug fixes
- Modification to the functional requirements
 - ◆ New features are needed
 - ◆ Existing ones modified
 - ◆ Perhaps some must be removed
- New or changed non-functional system properties
 - ◆ Anticipation of future change requests
- Changed operating environment
- Observation and analysis

Changes Arising from Product Line Forces

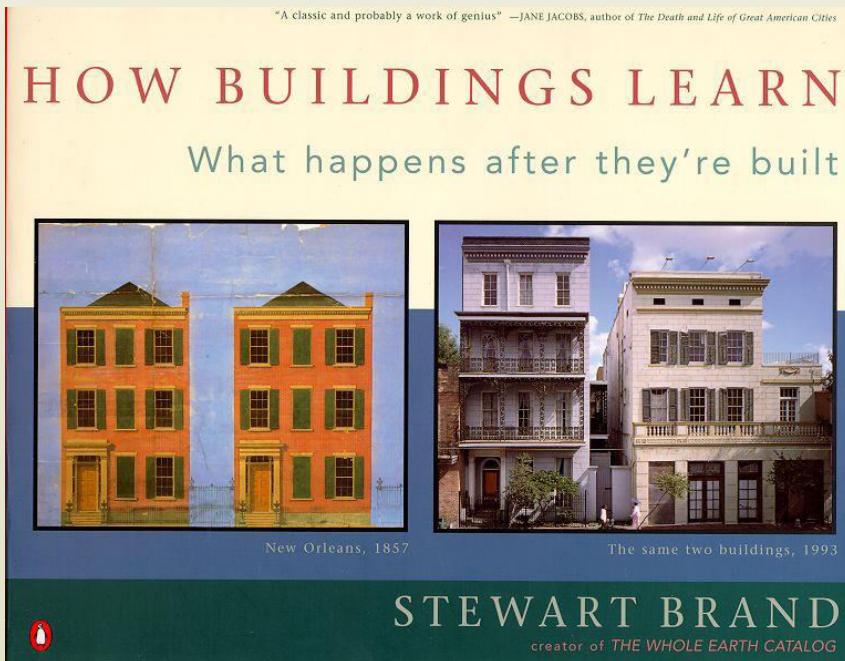
- Creating a new variant
 - ◆ Change at branch point
 - ◆ E.g.: Adding an integrated TV/DVD device to a TV product line
- Creation of a new branch point
- Merging product (sub)lines
 - ◆ Rationalizing their architectures

Motivation for Online Dynamic Change

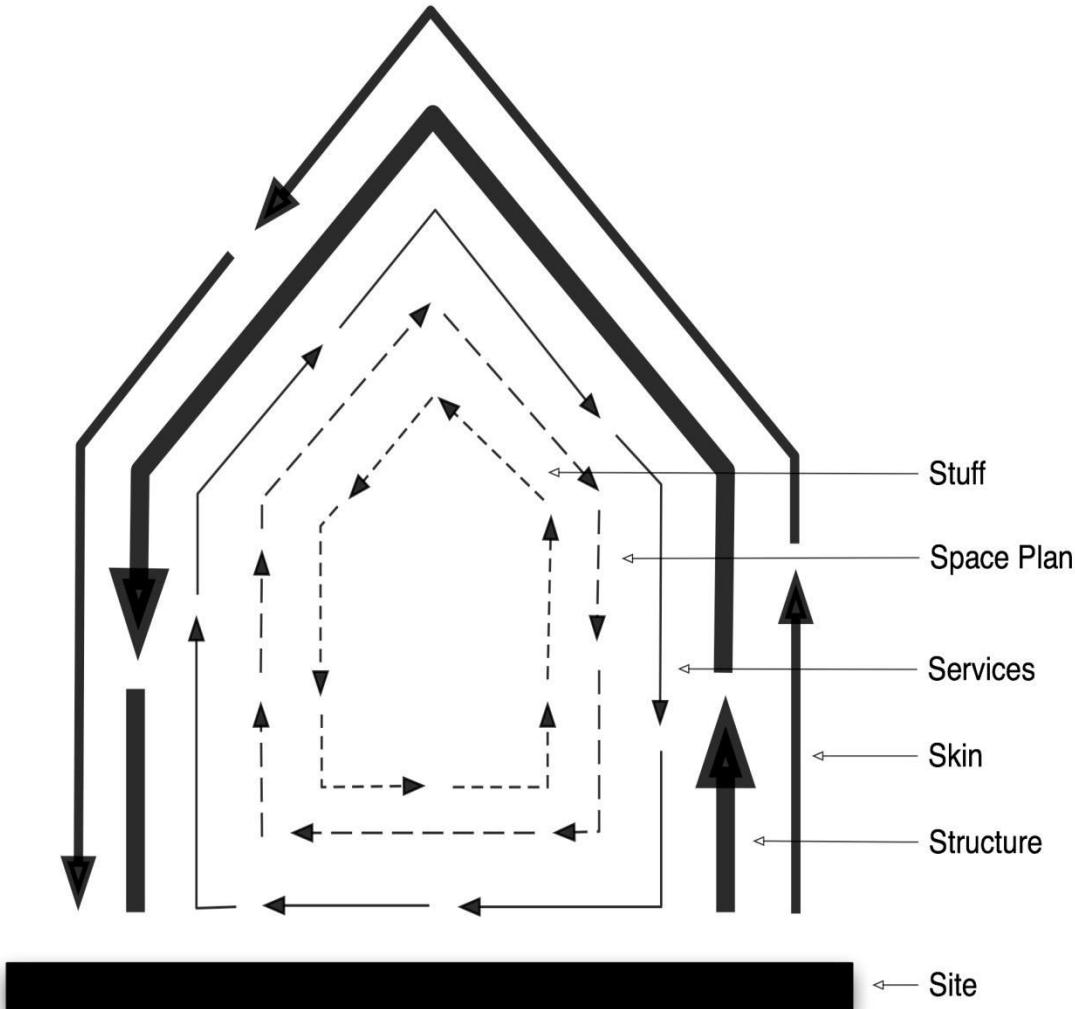
- Non-stop applications
 - ◆ software cannot be stopped because the “application” cannot be stopped
 - ◆ E.g., 24/7 systems
- Maintaining user or application state
 - ◆ stopping the software would cause the user to lose (mental) context
 - ◆ saving and/or recreating the software’s application state would be difficult or costly
- Re-installation difficulty
 - ◆ applications with complex installation properties
 - ◆ E.g., software in an automobile

Stewart Brand's Shearing Layers of Change

- “How Buildings Learn – What happens after they’re built” examines how and why buildings change over time
- Categorization of types of change according to the nature and cost of making a change



Shearing Layers in a Building



The Six Shearing Layers

- **Site:** the geographical setting, the urban location, and the legally defined lot
 - ◆ its boundaries and context outlast generations of ephemeral buildings.
- **Structure (“the building”):** the foundation and load-bearing elements
 - ◆ perilous and expensive to change, so people don’t
- **Skin:** exterior surfaces
 - ◆ change every ~20 years, to keep up with fashion, technology, or for repair

The Six Shearing Layers (cont'd)

- **Services**: working guts of a building: communications wiring, electrical wiring, plumbing, sprinkler systems, etc.
- **Space Plan**: interior layout – where walls, ceilings, floors, and doors go
- **Stuff**: chairs, desks, phones, pictures, kitchen appliances, lamps, hair brushes
 - ◆ things that switch around daily to monthly

To Shear or Not to Shear – Pompidou Center



The Six Shearing Layers

- Site
- Structure
- Skin
- Services
- Space Plan
- Stuff

How do these relate to software architecture?

Changing Component Interiors

- A component's performance may be improved by a change to the algorithm that it uses internally
- Capabilities that facilitate component adaptation
 - ◆ Knowledge of self and exposure of this knowledge to external entities
 - ◆ Knowledge of the component's role in the larger architecture
 - ◆ Pro-active engagement of other elements of a system in order to adapt

Change of Component Interface

- In many cases adaptation to meet modified functional properties entails changing a component's interface
- Adaptors/wrappers are a popular technique to mitigate “ripple effect”
 - ◆ but, subsequent changes to previously unmodified methods become even more complex

Connector Change

- Typically changes to connectors are motivated by
 - ◆ The desire to alter non-functional properties
 - such as distribution of components, fault-tolerance, efficiency, modifiability, etc.
 - increased independence of sub-architectures in the face of potential failures
 - improved performance
- The more powerful the connector the easier architectural change
 - ◆ E.g., connectors supporting event-based communication
 - ◆ What is the downside?

Change in the Configuration

- Changes to the configuration of components and connectors represents fundamental change to a system's architecture
- Effectively supporting such a modification requires working from an explicit model of the architecture
- Many dependencies between components will exist, and the architectural model is the basis for managing and preserving such relationships

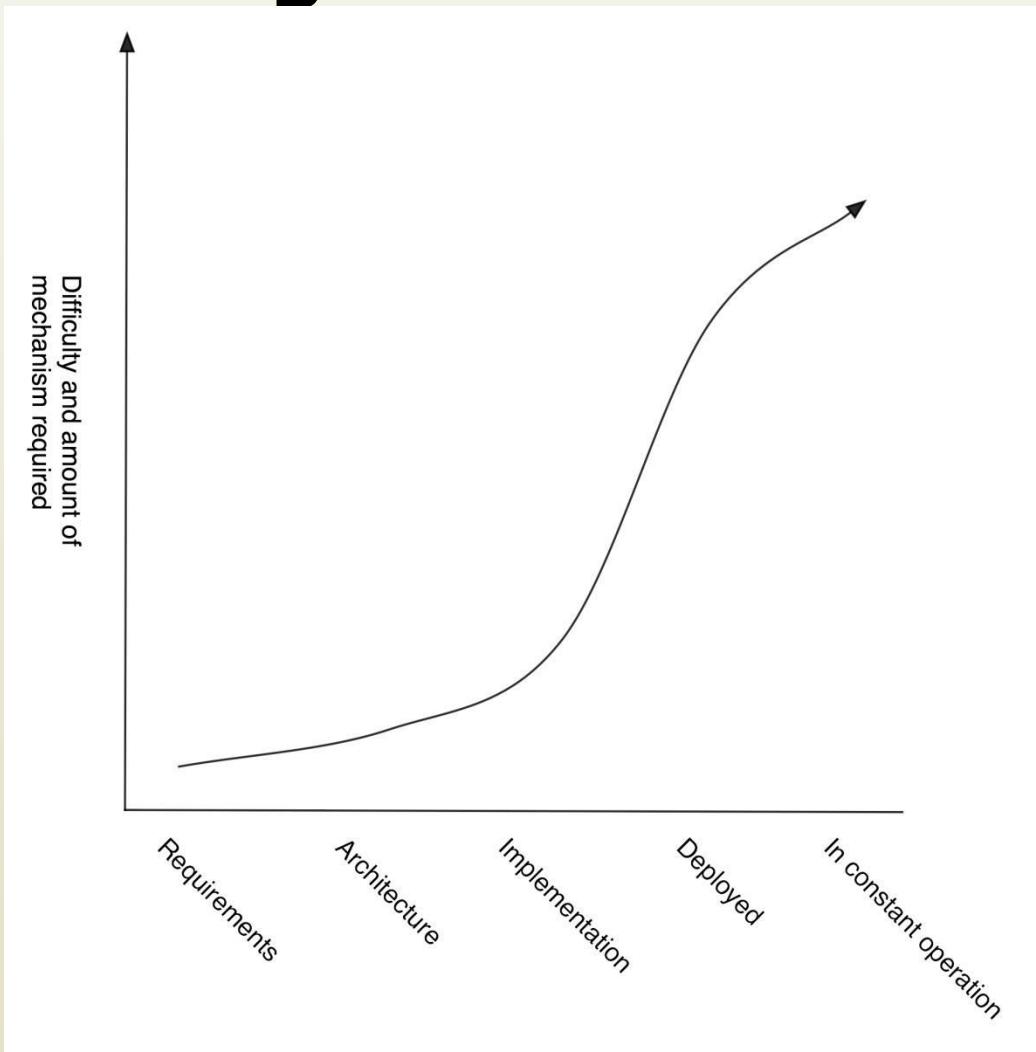
Change Agents and Context (1)

- Change Agents – Identity and Location
 - ◆ The processes that carry out adaptation may be performed by
 - human
 - automated agents
 - a combination thereof
 - ◆ If an adaptation agent is part of a deployed application from the outset, the potential is present for an effective adaptation process
 - ◆ Agent may have access to contextual information
 - ◆ E.g., periodically a user of a desktop OS is notified when an OS or application upgrade is available

Change Agents and Context (2)

- Knowledge
 - ◆ Agent might need knowledge to mitigate adaptation risk
 - ◆ Agent has to know the constraints that must be retained in the modified system
 - ◆ If the system's architectural design decisions have been violated, architectural recovery will be required
- Degree of Freedom
 - ◆ Freedom that the engineer has in designing the changes
 - ◆ Greater freedom → large solution space
 - ◆ By learning the constraints of the system, the coherence of the architecture can be retained

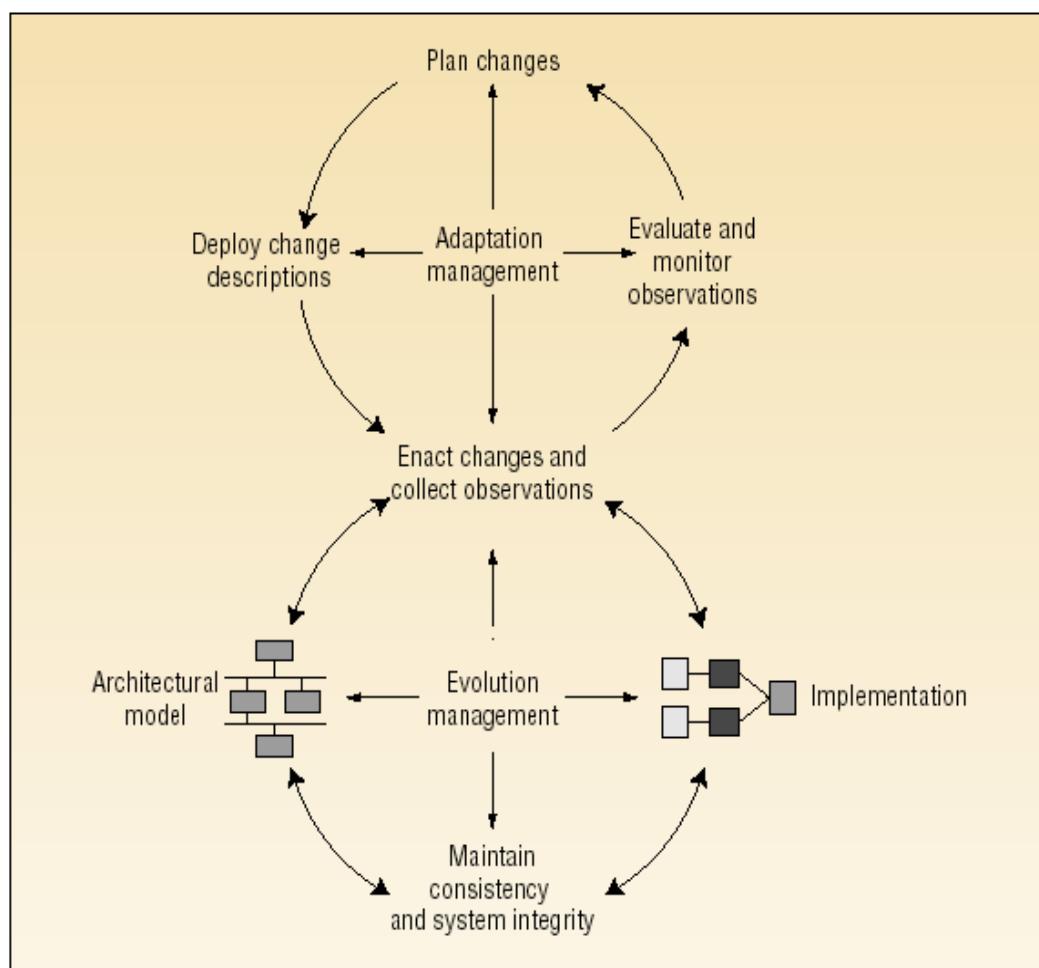
Time of Change



Architecture-Centric Adaptation

- In the absence of explicit architecture the engineer is left to reason about adaptation
 - ◆ from memory
 - ◆ from source code
- Architecture can serve as the primary focus of reasoning about potential adaptations
 - ◆ Descriptive architecture is the reference point

Conceptual Architecture for Adaptation

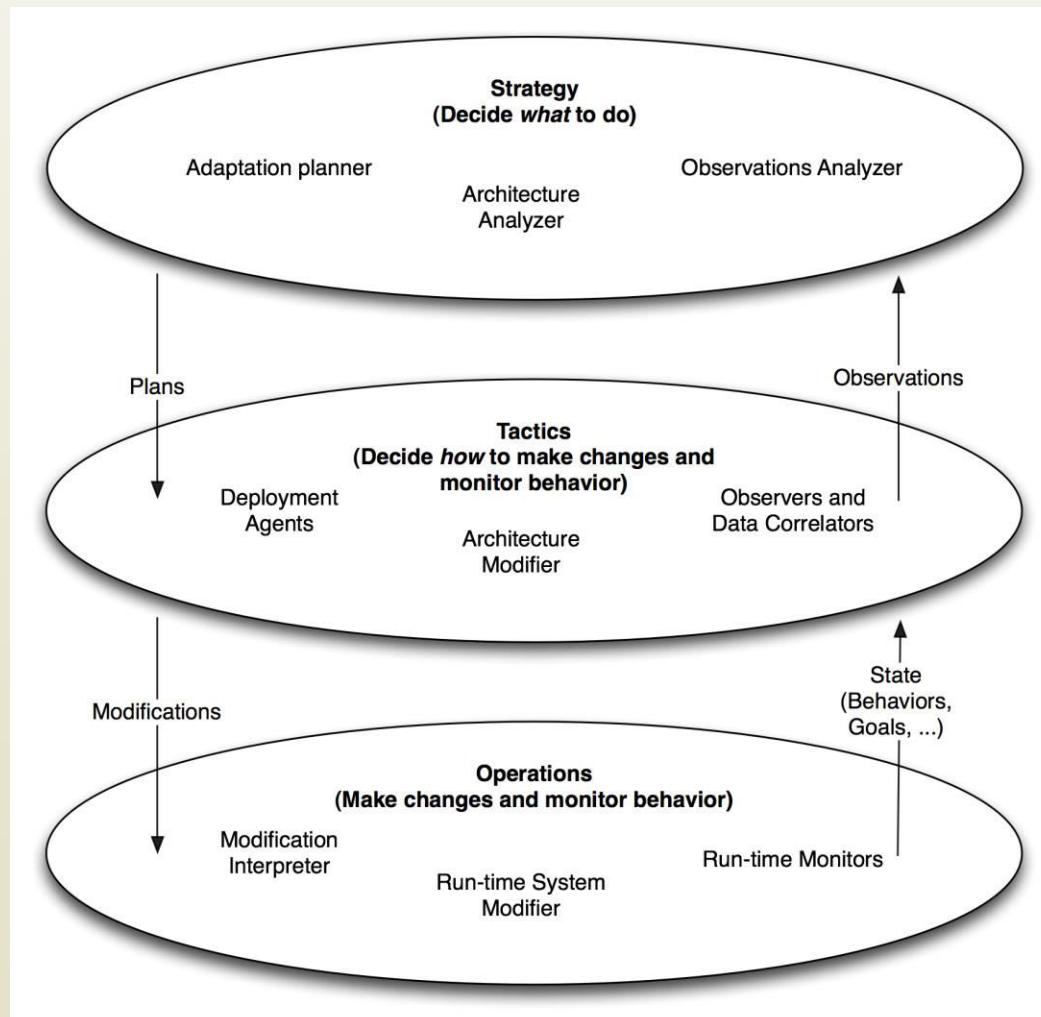


From: "An Architecture-based Approach to Self-Adaptive Software", Oreizy, et.al. IEEE Intelligent Systems, 14 (3), 1999

Activities, Agents, and Entities

- Adaptation management and evolution management can be separated into three types of activities
 - ◆ **Strategic** refers to determining what to do
 - ◆ **Tactical** refers to developing detailed plans for achieving the strategic goals
 - ◆ **Operations** refers to the nuts-and-bolts of carrying out the detailed plans

Strategy, Tactics, and Operations



Categories of Techniques

- Observing and collecting state
- Analyzing data and planning change
- Describing change descriptions
- Deploying change descriptions
- Effecting the change

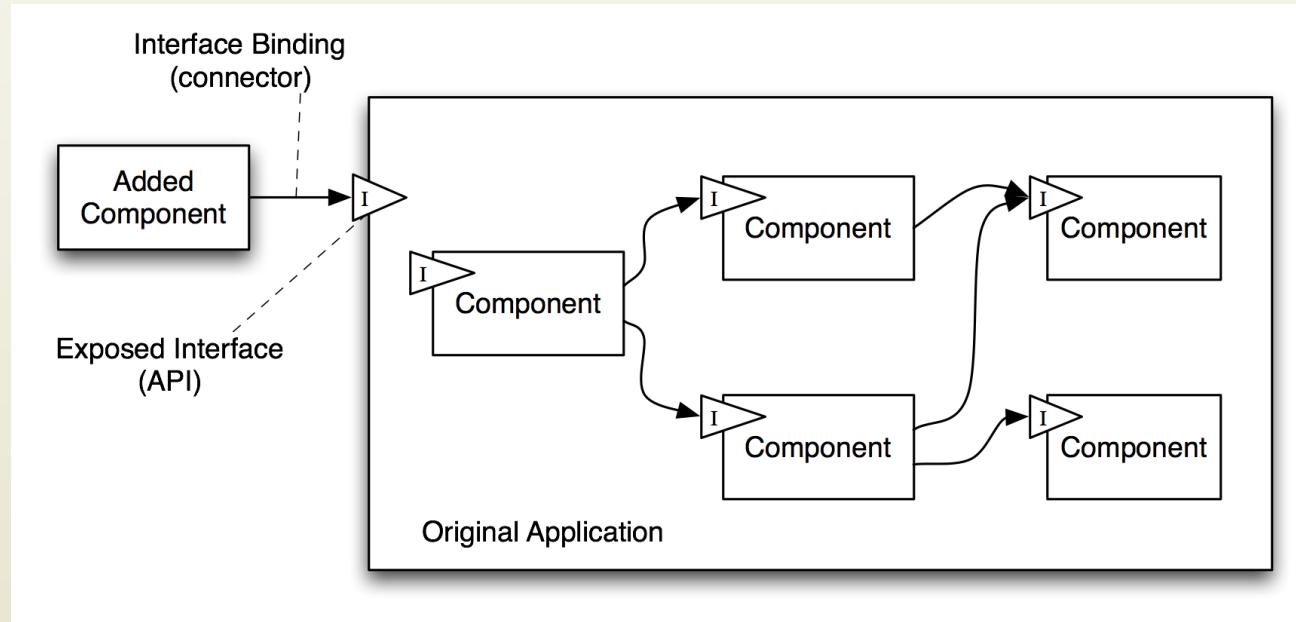
Architectures/Styles that Support Adaptation

- Explicit models
- First-class connectors
- Adaptable connectors
- Message-based communication
- ...others?

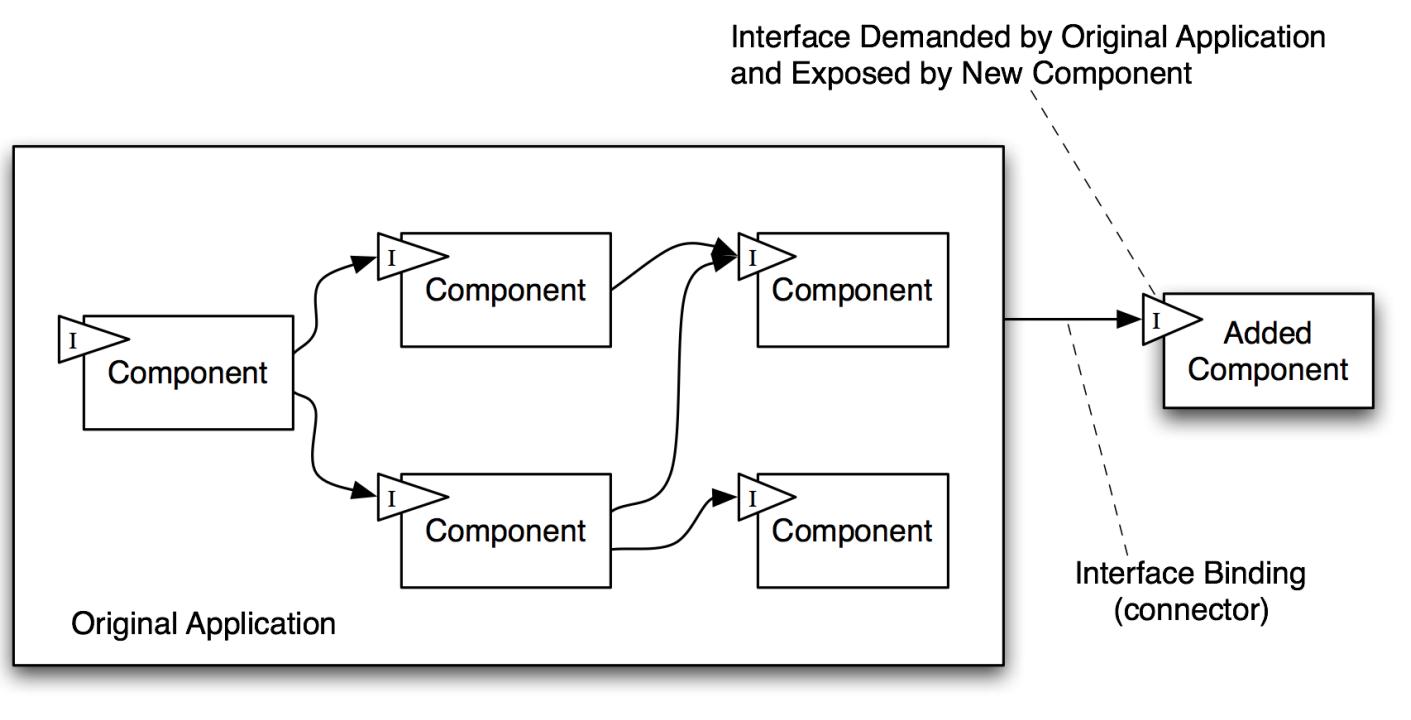
Architectures/Styles Supporting Adaptation

- Application programming interfaces (API)
- Scripting languages
- Plug-ins
- Component/object architectures
- Event interfaces

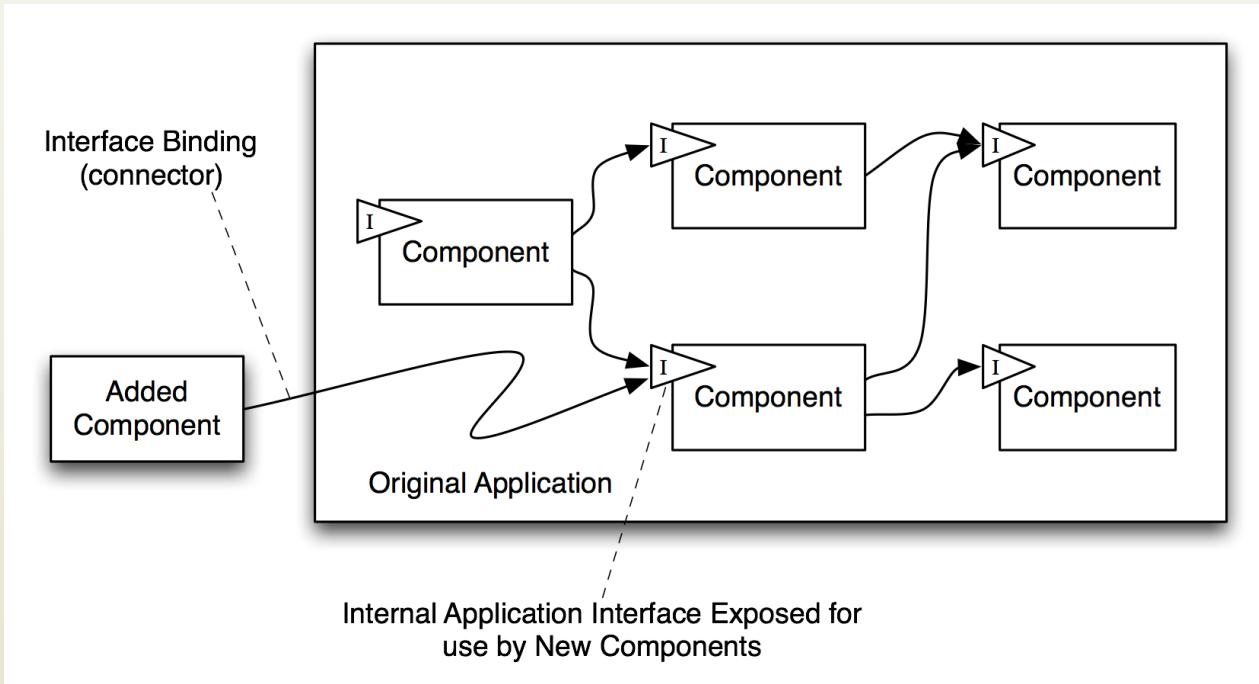
API-Based Extension



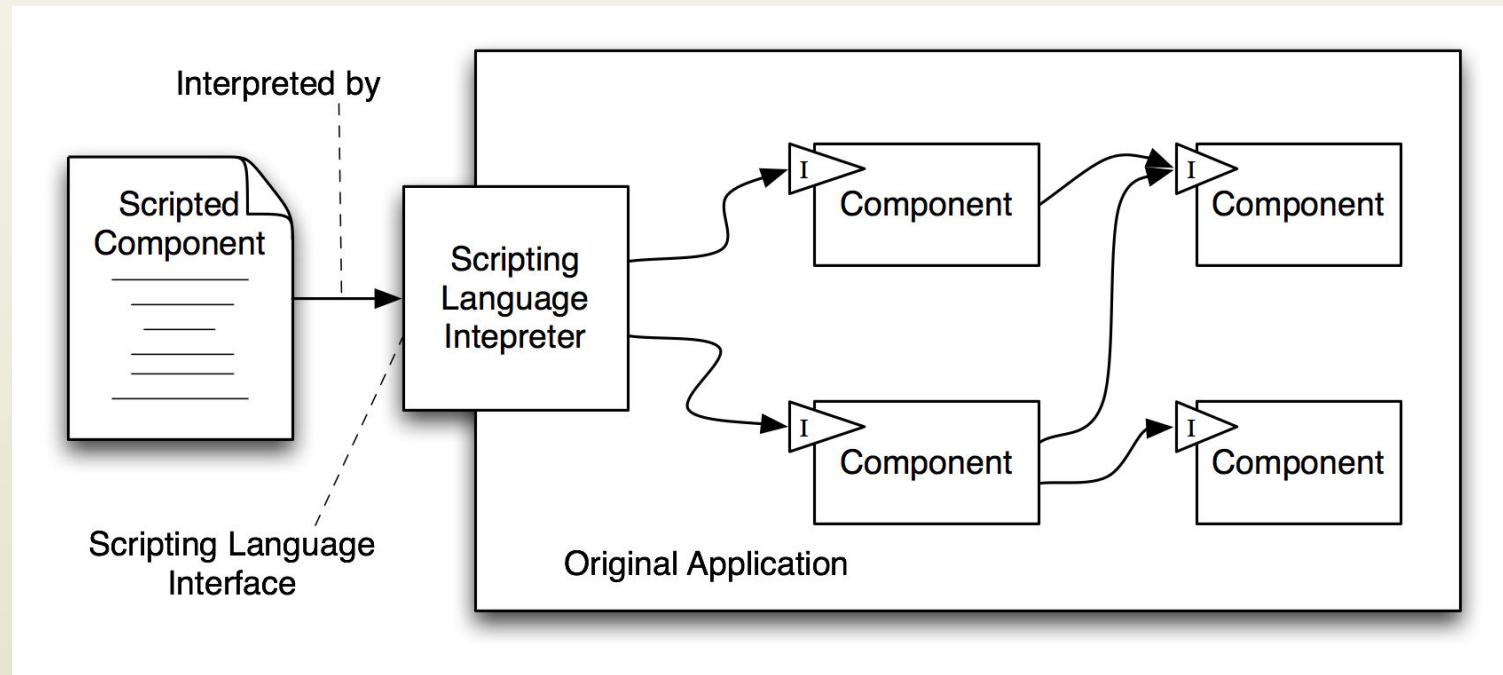
Plug-In Based Extension



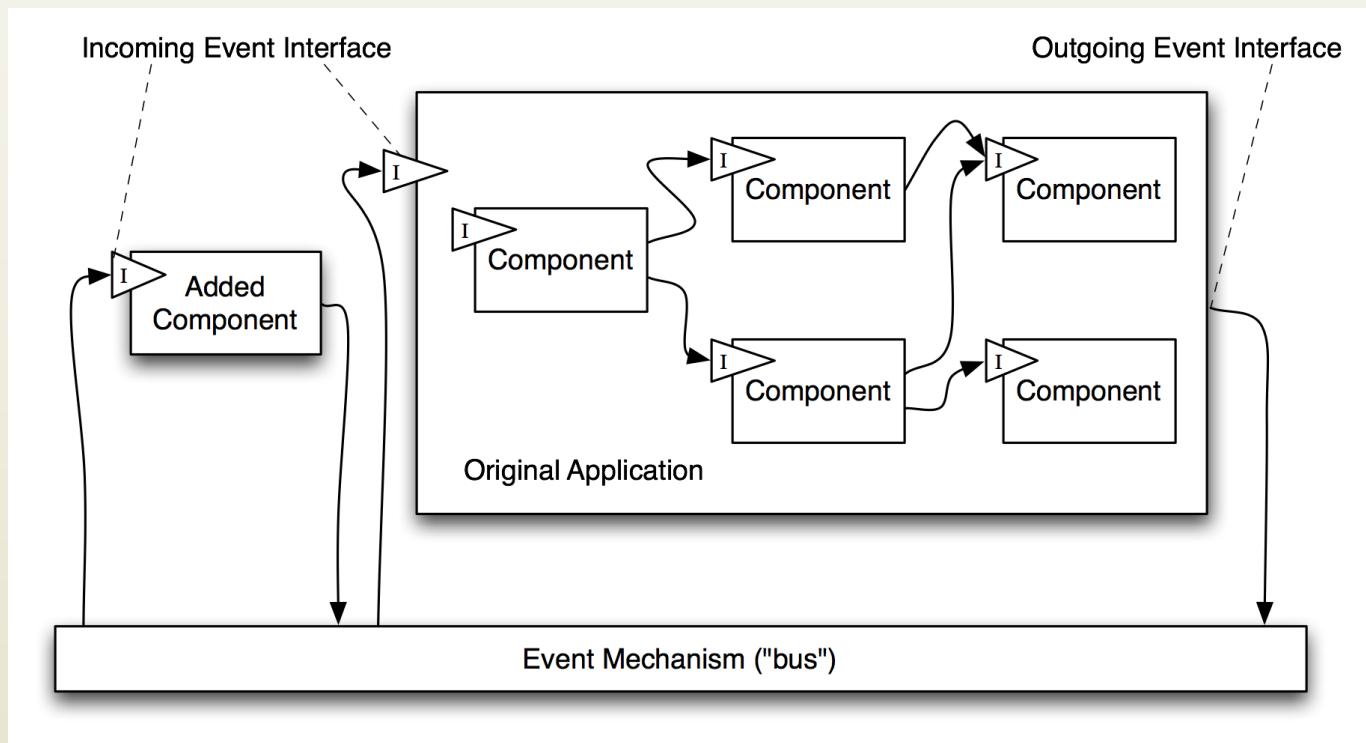
Component-Object Approach



Scripting-Based Extension



Event-Based Extension



The Special Problems of On-the-fly Change

- The principle of quiescence
 - ◆ A node in the active state can initiate and respond to transactions.
 - ◆ The state identified as necessary for reconfiguration is the passive state,
 - node must respond to transaction
 - not currently engaged in a transaction that it initiated
 - it will not initiate new transactions

Standards

Software Architecture
Lecture 26

Objectives

- Concepts
 - ◆ What are standards?
 - ◆ Why use standards?
 - And why not? (drawbacks)
 - ◆ Deciding when to adopt a standard
- Prevalent Architectural Standards
 - ◆ Conceptual standards
 - ◆ Notational standards
 - ◆ Standard tools
 - ◆ Process standards

Objectives

- Concepts
 - ◆ What are standards?
 - ◆ Why use standards?
 - And why not? (drawbacks)
 - ◆ Deciding when to adopt a standard
- Prevalent Architectural Standards
 - ◆ Conceptual standards
 - ◆ Notational standards
 - ◆ Standard tools
 - ◆ Process standards

What are standards?

- **Definition:** a *standard* is a form of agreement between parties
- Many kinds of standards
 - ◆ For notations, tools, processes, organizations, domains
- There is a prevalent view that complying to standard 'X' ensures that a constructed system has high quality
 - ◆ This is almost never strictly true
 - ◆ But that doesn't mean standards are worthless!
 - ◆ Here, we will attempt to put standards in perspective

***De jure* and *de facto* standards**

- Some standards are controlled by a body considered authoritative
 - ◆ ANSI, ISO, ECMA, W3C, IETF
- These standards are called *de jure* ("from law")
- *De jure* standards usually
 - ◆ are formally defined and documented
 - ◆ are evolved through a rigorous, well-known process
 - ◆ are managed by an independent body, governmental agency, or multi-organizational coalition rather than a single individual or company

***De jure* and *de facto* standards (cont'd)**

- Some standards emerge through widespread awareness and use
- These standards are called *de facto* ("in practice")
- *De facto* standards usually
 - ◆ are created by a single individual organization to address a particular need
 - ◆ are adopted due to technical superiority or market dominance of the creating organization
 - ◆ evolve through an emergent, market-driven process
 - ◆ are managed by the creating organization or the users themselves, rather than through a formal custodial body

Examples of *de jure* and *de facto*

- *De jure* standards
 - ◆ UML (managed by OMG)
 - ◆ CORBA (also managed by OMG)
 - ◆ HTTP protocol (managed by IETF)
- *De facto* standards
 - ◆ PDF format (managed by Adobe)
 - May become *de jure* through ISO
 - ◆ Windows (managed by Microsoft)
- There is a substantial gray area between these two

Gray-area Standards

- HTML
 - ◆ Officially standardized by W3C, indicating *de jure*
 - ◆ Flavors and browser-specific extensions developed by Microsoft, Netscape, and others, creating *de facto* variants
 - ◆ None of these has power to force users to use standard
- JavaScript
 - ◆ Developed by Netscape; copied (as JScript) by Microsoft
 - ◆ After substantial adoption and possibly under threat of forking/splintering, Netscape submits it to ECMA
 - ◆ Now standardized as ECMAScript (*de jure*)
 - ◆ JavaScript and variants continue to be developed as compatible extensions of ECMAScript

Another spectrum

- Standards (whether *de jure* or *de facto*) can be:
 - ◆ Open
 - Allow public participation in the standardization process
 - Anyone can submit ideas or changes for review
 - ◆ Closed (a.k.a. proprietary)
 - Only the custodians of a standard can participate in its evolution

Open vs. closed standards

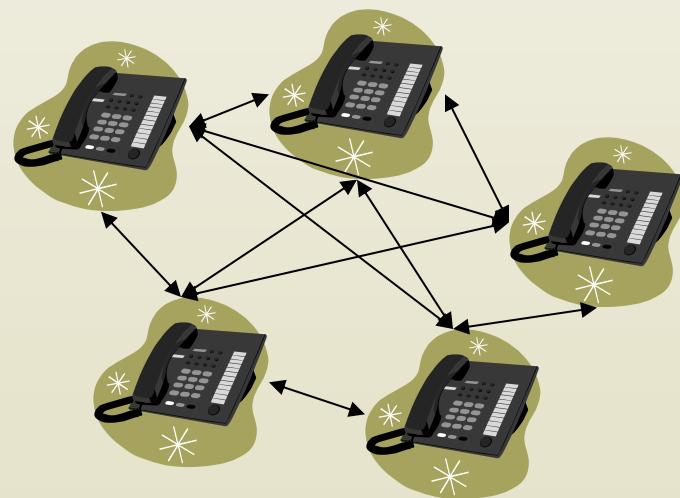
- Another spectrum with a gray area
 - ◆ Some standards bodies have high barriers to entry (e.g., steep membership fees, vote of existing membership)
 - ◆ Some standards (e.g., Java) have aspects of both
 - Sun Microsystems is effectively in control of Java as a *de facto* standard
 - There is an open “community process” by which external parties can participate in a limited way

Why use standards?

- Standards are an excellent way to create and exploit *network effects*
- A network effect exists if the value of participation increases as the number of users of the standard increases



← versus →



- Other network effects:
 - ◆ TCP/IP, HTTP & HTML, UML...

Why use standards? (cont'd)

- To ensure interoperability between products developed by different organizations
 - ◆ Usually in the interest of fostering a network effect
- To carry hard-won engineering knowledge from one project to another
 - ◆ To take advantage of hard-won engineering knowledge created by others
- As an effort to attract tool vendors
 - ◆ To create economies of scale in tools
- To attempt to control the standard's evolution in your favor

Drawbacks of standards

- Limits your agility
 - ◆ Remember that doing 'good' architecture-based development means identifying what is important in *your* project
- Standards often attempt to apply the same techniques to a too-broad variety of situations
- The most widely adopted standards are often the most general

Overspecification vs. underspecification

- A perennial tension in standards use and development
- Overspecification
 - ◆ A standard prescribes too much and therefore limits its applicability too much
- Underspecification
 - ◆ A standard prescribes too little and therefore doesn't provide enough guidance
 - Possibly in an effort to broaden adoption

Two different kinds of underspecification

- Two compromises often made in negotiation when disagreements occur
 - ◆ Leave the disagreeable part of the standard unspecified or purposefully ambiguous
 - ◆ Include both opinions in the standard but make them both optional
- Both of these weaken the standard's value
 - ◆ Consider the different kinds of reduction in interoperability imposed by these strategies
- Although they may improve adoption!

When to adopt a standard?

- Early adoption
 - ◆ Benefits
 - Improved ability to influence the standard
 - ◆ Get your own goals incorporated; exclude competitors
 - Early to market
 - ◆ If standard becomes successful, early marketers will profit
 - Early experience
 - ◆ Leverage enhanced experience to your benefit

When to adopt a standard? (cont'd)

- Early adoption
 - ◆ Drawbacks
 - Risk of failure
 - ◆ Standard may not be successful for reasons out of your control
 - Moving target
 - ◆ Early standards tend to evolve and 'churn' more than mature ones, and may be 'buggy'
 - Lack of support
 - ◆ Early standards tend to have immature (or no) support from tool and solution vendors

When to adopt a standard? (cont'd)

- Late adoption
 - ◆ Benefits
 - Maturity of standard
 - Better support
 - ◆ Drawbacks
 - Inability to influence the standard
 - Restriction of innovation

Objectives

- Concepts
 - ◆ What are standards?
 - ◆ Why use standards?
 - And why not? (drawbacks)
 - ◆ Deciding when to adopt a standard
- Prevalent Architectural Standards
 - ◆ Conceptual standards
 - ◆ Notational standards
 - ◆ Standard tools
 - ◆ Process standards

IEEE 1471

- Recommended practice for architecture description
 - ◆ Often mandated for use in government projects
- Scope is limited to architecture descriptions (as opposed to processes, etc.)
- Does not prescribe a particular notation for models
 - ◆ Does prescribe a minimal amount of content that should be contained in models
- Identifies the importance of stakeholders and advocates models that are tailored to stakeholder needs
- A notion of views and viewpoints similar to the ones used in this course

IEEE 1471 (cont'd)

- Very high level
 - ◆ Purposefully light on specification
 - ◆ Does not advocate any specific notation or process
- Useful as a starting point for thinking about architecture
 - ◆ Defines key terms
 - ◆ Advocates focus on stakeholders
- Being compliant does NOT ensure that you are doing good architecture-centric development

Department of Defense Architecture Framework

- DoDAF, evolved from C4ISR
 - ◆ Has some other international analogs (MoDAF)
 - ◆ 'Framework' here refers to a process or set of viewpoints that should be used in capturing an architecture
 - Not necessarily an architecture implementation framework
- Identifies specific viewpoints that should be captured
 - ◆ Includes what kinds of information should be captured
 - ◆ Does not prescribe a particular notation for doing the capture

DoDAF (cont'd)

Concept	Our Term	DoDAF Term
A set of perspectives from which descriptions are developed	Viewpoint set	View
A perspective from which descriptions are developed	Viewpoint	(Kind of) product
An artifact describing a system from a particular perspective	View	Product

- Some vocabulary inconsistency with our terms (and IEEE 1471 among others)

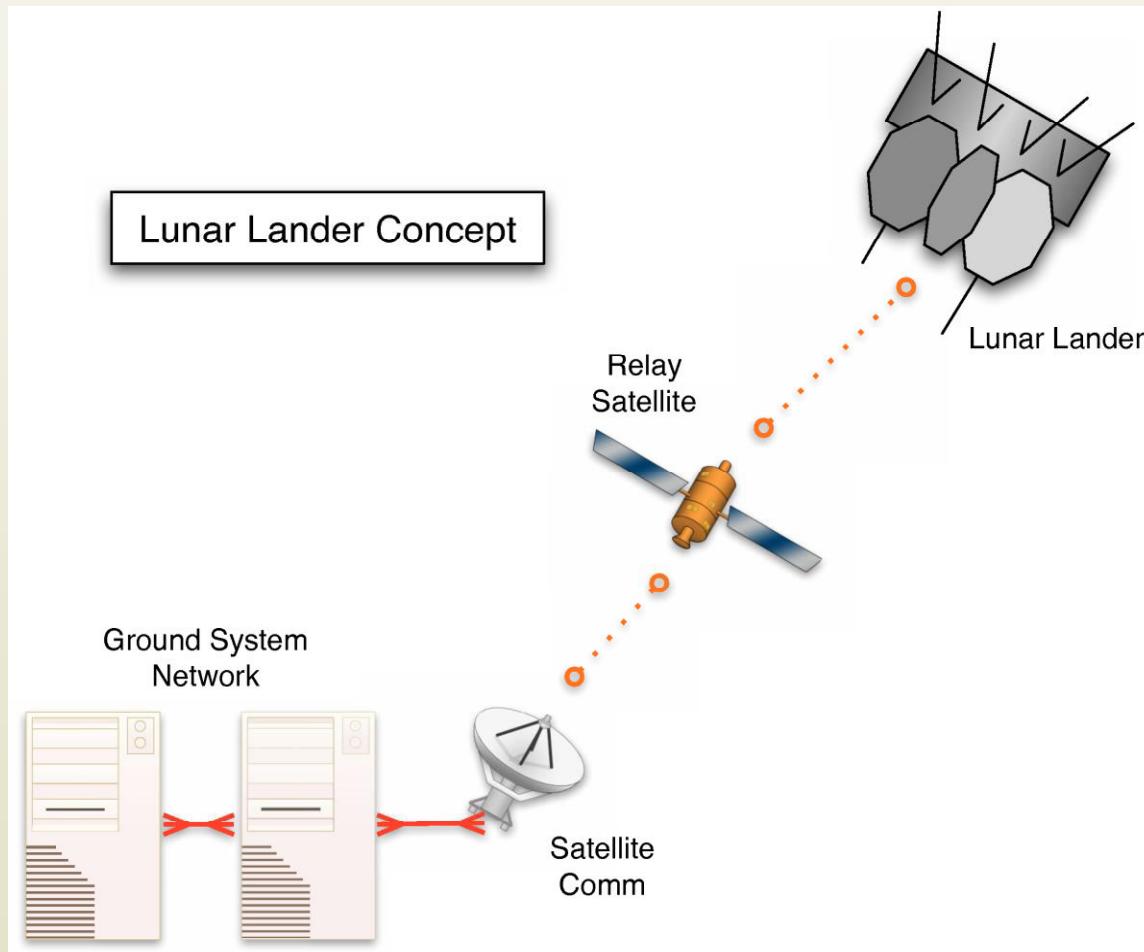
DoDAF (cont'd)

- Three views (in our terms: viewpoint sets)
 - ◆ Operational View (OV)
 - “Identifies what needs to be accomplished, and who does it”
 - Defines processes and activities, the operational elements that participate in those activities, and the information exchanges that occur between the elements
 - ◆ Systems View (SV)
 - Describe the systems that provide or support operational functions
 - and the interconnections between them
 - Systems in SV associated with elements in OV

DoDAF (cont'd)

- Three views (in our terms: viewpoint sets)
 - ◆ Technical Standards View (TV)
 - Identify standards, (engineering) guidelines, rules, conventions, and other documents
 - To ensure that implemented systems meet their requirements and are consistent with respect to the fact that they are implemented according to a common set of rules
- Also a few products address cross cutting concerns that affect All Views (AV)
 - ◆ E.g., dictionary of terms

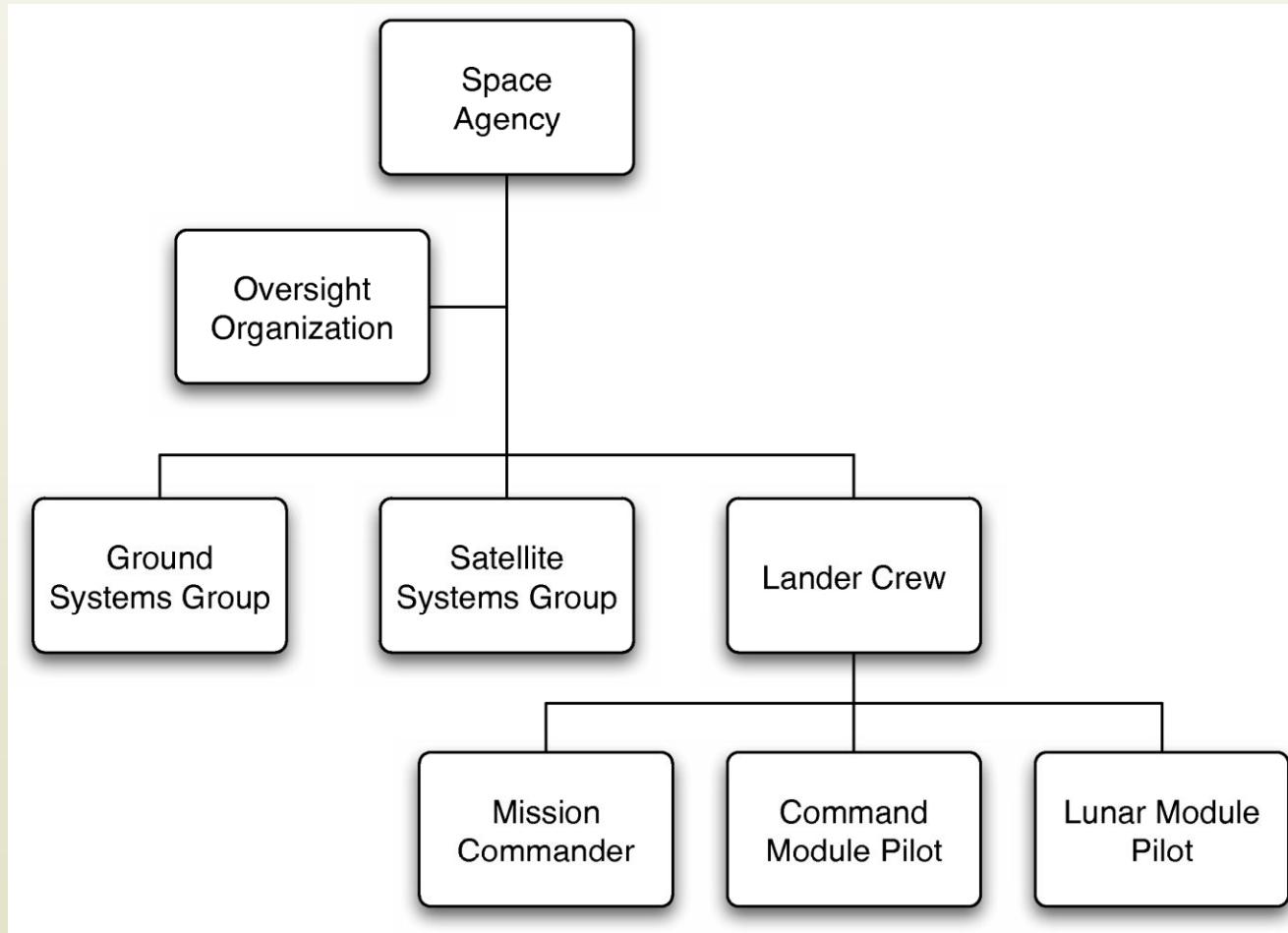
DoDAF Examples



OV-1

“High-Level Operational Concept Graphic”

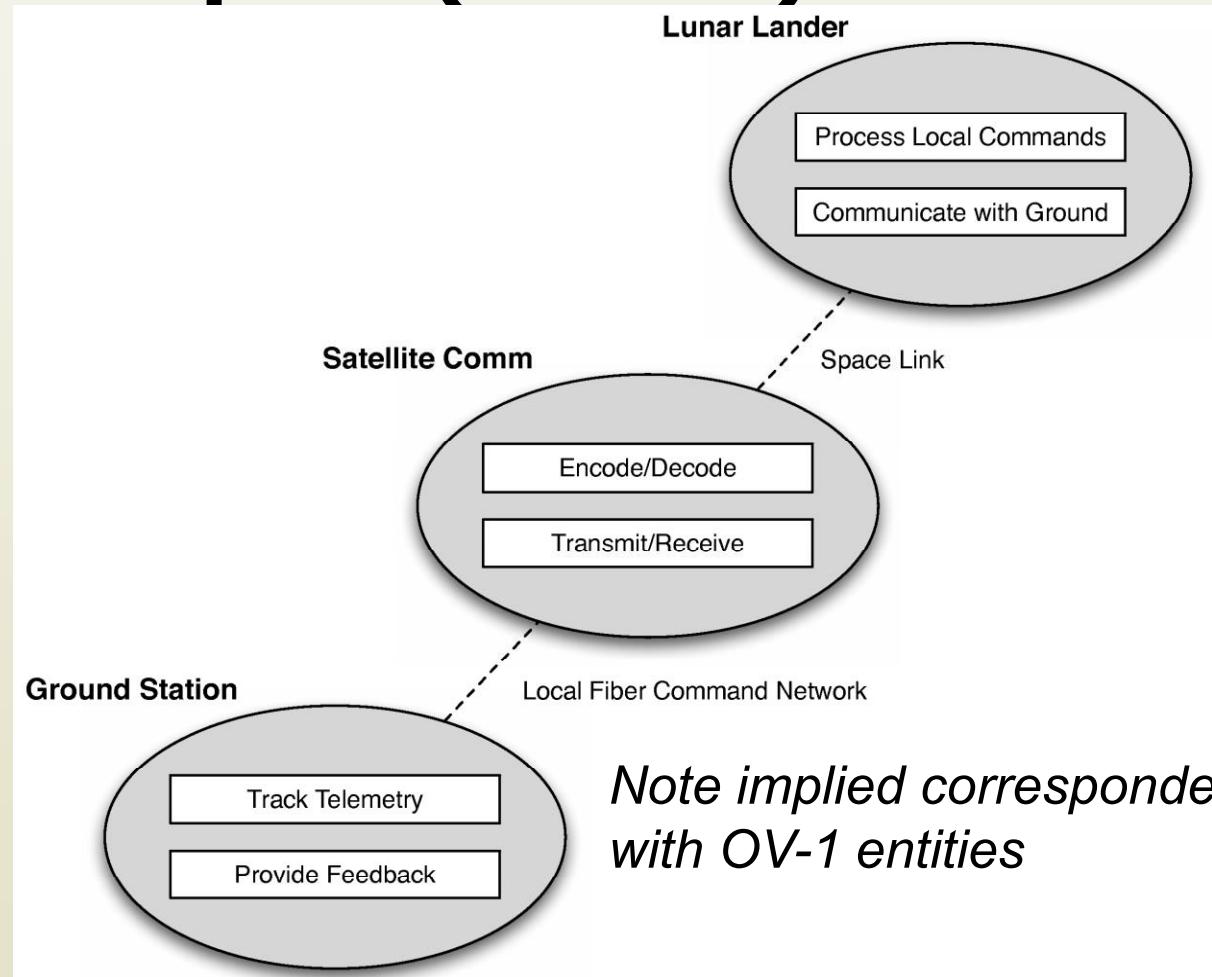
DoDAF Examples (cont'd)



OV-4

“Organizational Relationships”

DoDAF Examples (cont'd)



SV-1

“Systems Interface Description”

DoDAF Examples (cont'd)

from to	Ground Station	Satellite Comm	Lunar Lander
Ground Station		Ground Feedback (TCP/IP)	
Satellite Comm	Lander Transmissions (TCP/IP)		Ground Feedback (Space Protocol)
Lunar Lander		Lander Transmissions (Space Protocol)	

SV-3
“Systems-Systems Matrix”

One of several “N²” views in DoDAF

DoDAF Examples (cont'd)

Standards for SV-1 Systems			Ground Station	Satellite Comm	Lunar Lander
Service	Service Area	Standard			
Information Technology Standards	Operating System Standard	ISO/IEC 9945-1:1996, Information Technology - Portable Operating System Interface (POSIX) - Part 1: System Application Program Interface (API)	Baseline: 1 January 2007	Baseline	Baseline + 3 mos
Information Transfer Standards	Data Flow Network	Extensible Markup Language (XML) 1.0 (Fourth Edition) W3C Recommendation 16 August 2006	Baseline + 6 mos	Baseline + 6 mos	
	Physical Layer	FDDI / ANSI X3.148-1988, Physical Layer Protocol (PHY) -- also ISO 9314-1	Baseline + 3 mos	Baseline + 3 mos	

TV-1

“Technical Standards Profile ”

DoDAF Takeaways

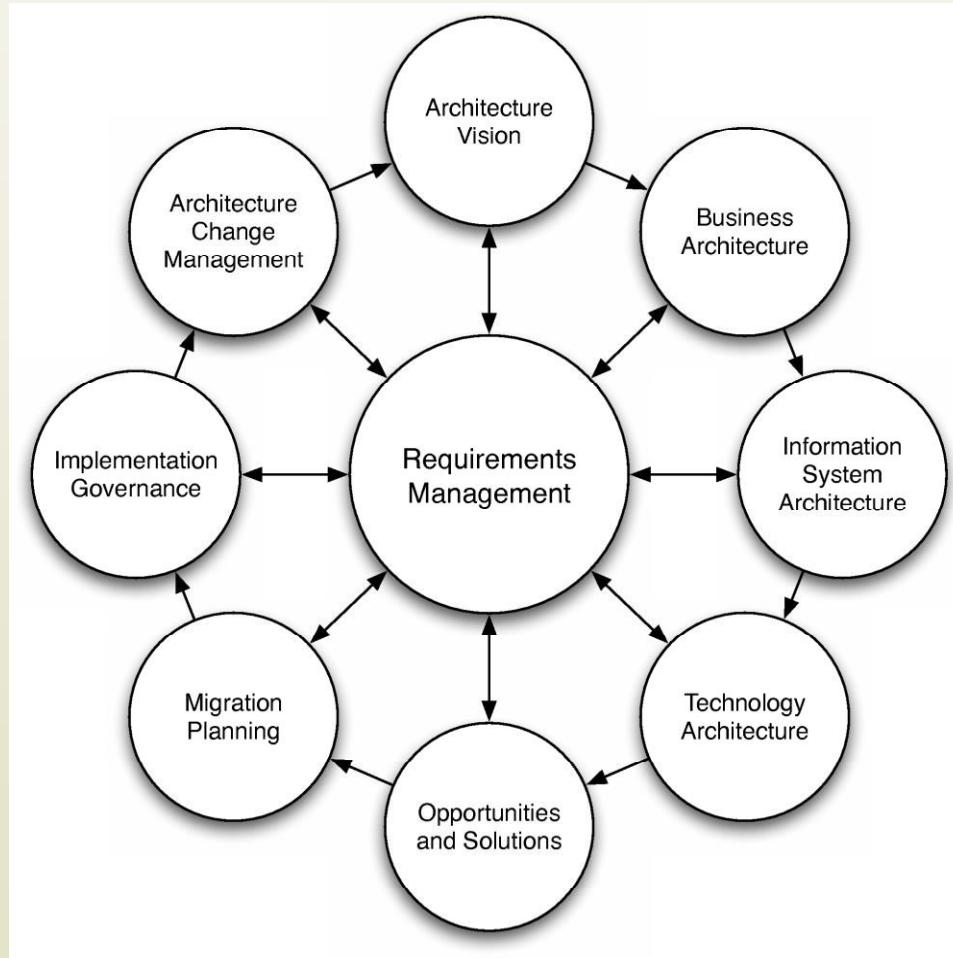
- Extremely comprehensive standard advocating capture of many views
 - ◆ Takes a high-level organizational perspective
 - ◆ OV views tend to deal with human and systems organizations
 - ◆ SV views tend to deal with technical aspects of systems (most like the architectural descriptions we have been talking about)
 - ◆ TV views tend to deal with practical issues of reuse and leveraging existing technology
- Tells us a lot about WHAT to model, but nearly nothing about HOW to model it

The Open Group Architecture Framework

- TOGAF – an “enterprise architecture” framework
 - ◆ Focuses beyond hardware/software
 - ◆ How can enterprises build systems to achieve business goals?
- Four key areas addressed
 - ◆ Business concerns, which address business strategies, organizations, and processes;
 - ◆ Application concerns, which address applications to be deployed, their interactions, and their relationships to business processes;
 - ◆ Data concerns, which address the structure of physical and logical data assets of an organization and the resources that manage these assets; and
 - ◆ Technology concerns, which address issues of infrastructure and middleware.

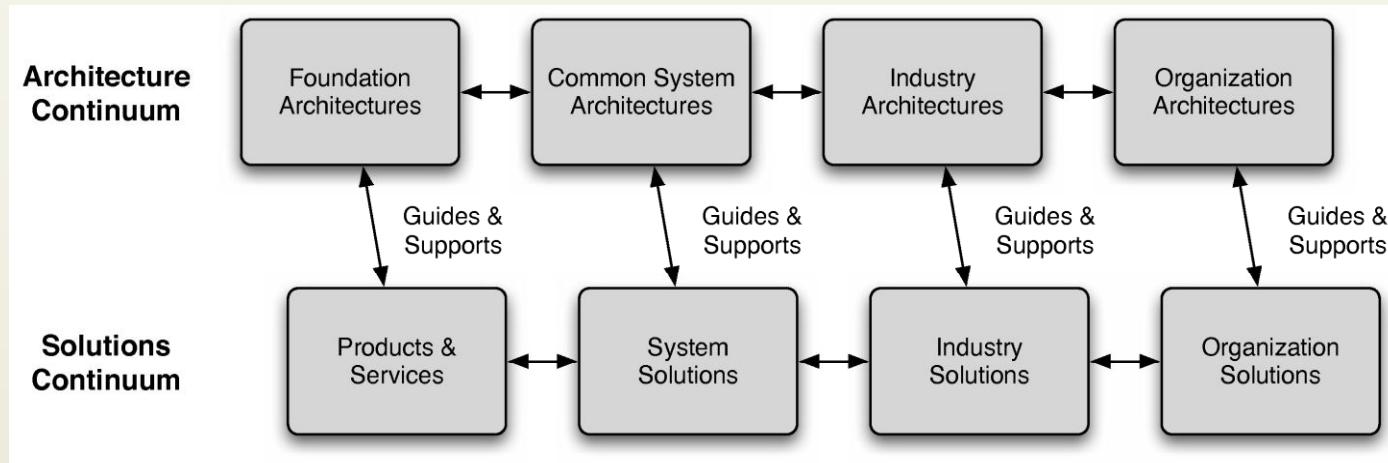
TOGAF Part 1: ADM

- An iterative process for architecture-centric development
- Each step in the processes associated with views to be captured
- Early phases focus on conceptual issues; later phases move toward reduction to practice



Redrawn from the TOGAF Specification
33

TOGAF Part 2: Enterprise Continuum



Redrawn from the TOGAF Specification

- Taxonomizes different kinds of architectures and the solutions that are supported by those
- Left side is more technical and concrete
- Right side is more organizational
- TOGAF Technical Reference Model and Standards Information Base identifies and taxonomizes many solution elements

TOGAF Part 3: TOGAF Resource Base

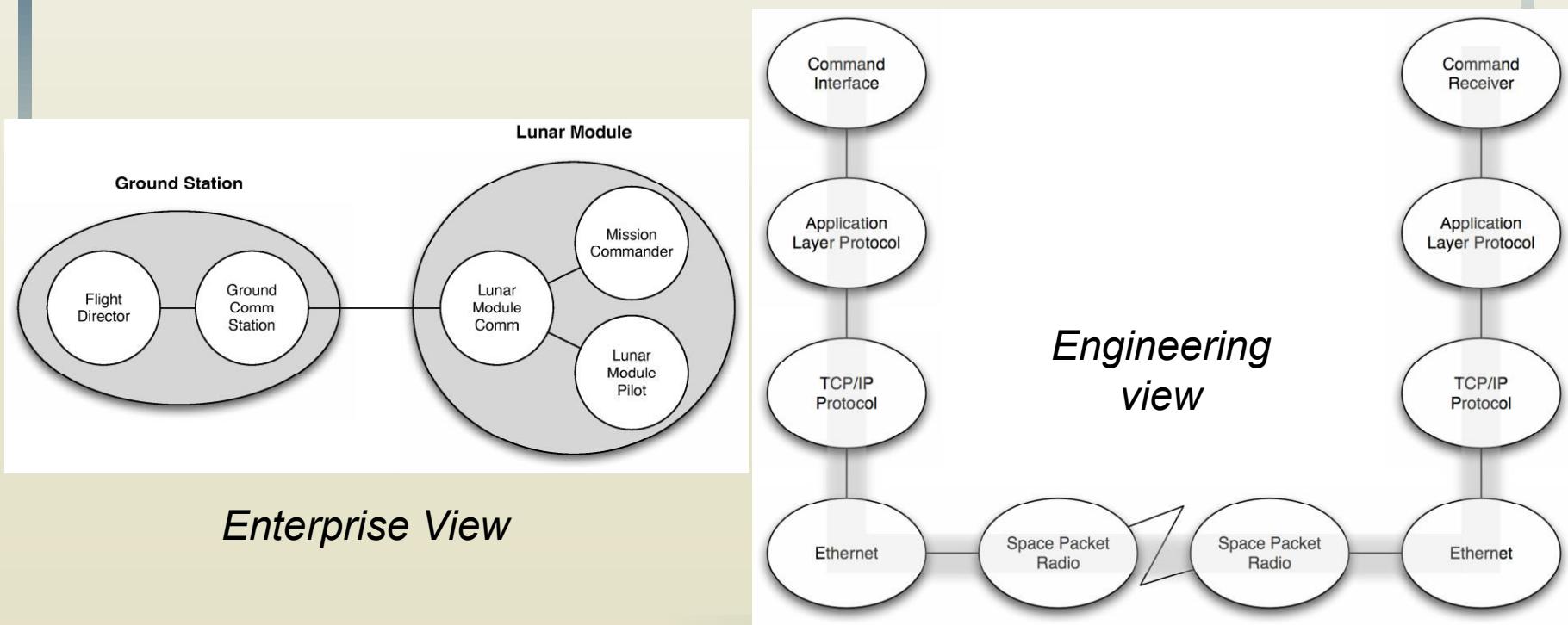
- A collection of useful information and resources that can be employed in following the ADM process
- Includes
 - ◆ advice on how to set up boards and contracts for managing architecture
 - ◆ checklists for various phases of the ADM process
 - ◆ a catalog of different models that exist for evaluating architectures
 - ◆ how to identify and prioritize different skills needed to develop architectures
 - ◆

TOGAF Takeaways

- Large size and broad scope looks at systems development from an enterprise perspective
- More suited to developing entire organizational information systems rather than individual applications
- A collection and clearinghouse for IT “best practices” of all sorts

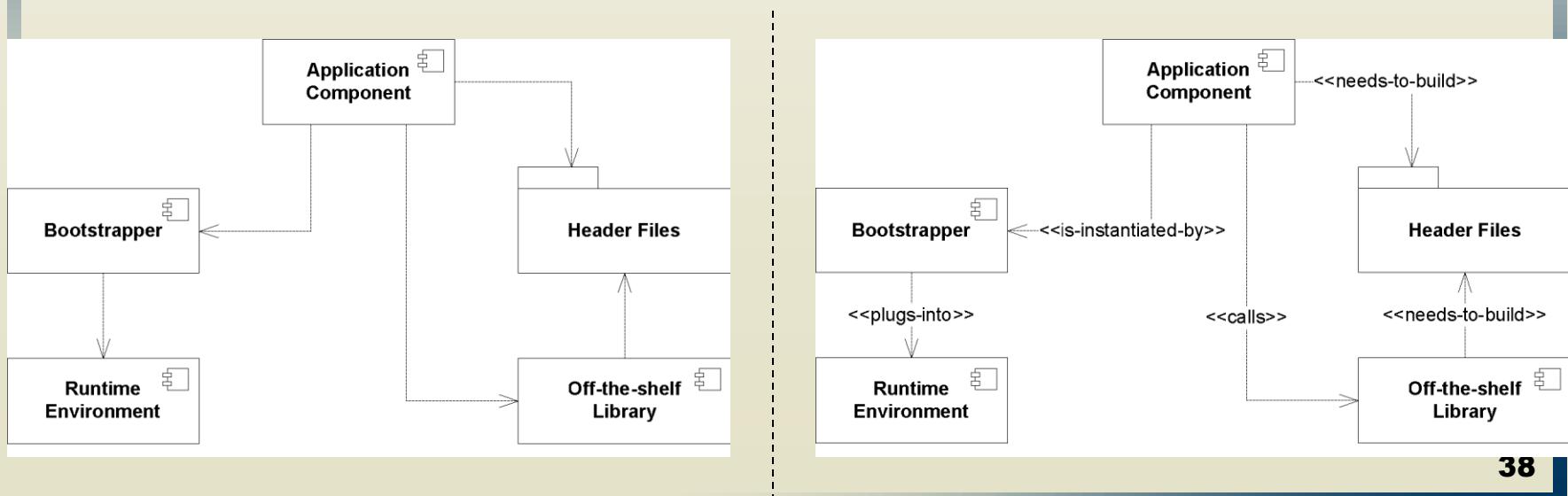
RM-ODP

- Another standard for viewpoints, similar to DoDAF but more limited in scope; resemble DoDAF SV
 - ◆ Prescribes 5 viewpoints for distributed systems



UML

- Discussed extensively already in this course
- As a standard, primarily prescribes a syntax
- Some semantics with purposeful ambiguity
- Encourages specialization of the standard through the use of profiles, which are mini-standards



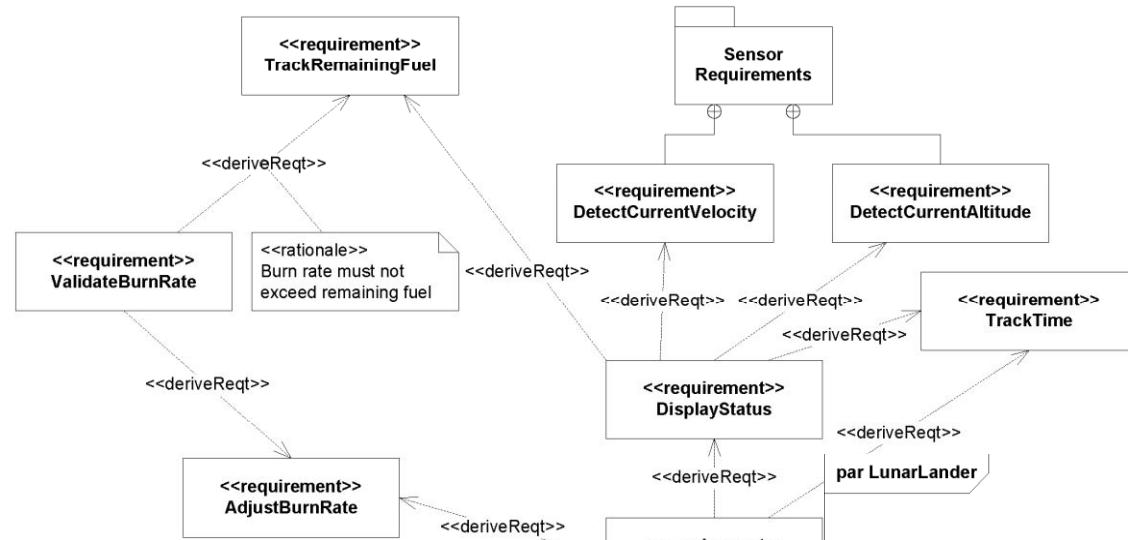
UML Takeaways

- Provide a common syntactic framework to express many common types of design decisions
- Profiles are needed to improve rigor
 - ◆ But profiles can only specialize existing UML diagram types, not create new ones
- Documenting a system in UML does not ensure overall system quality
 - ◆ You can document a bad architecture in UML as easily as a good one

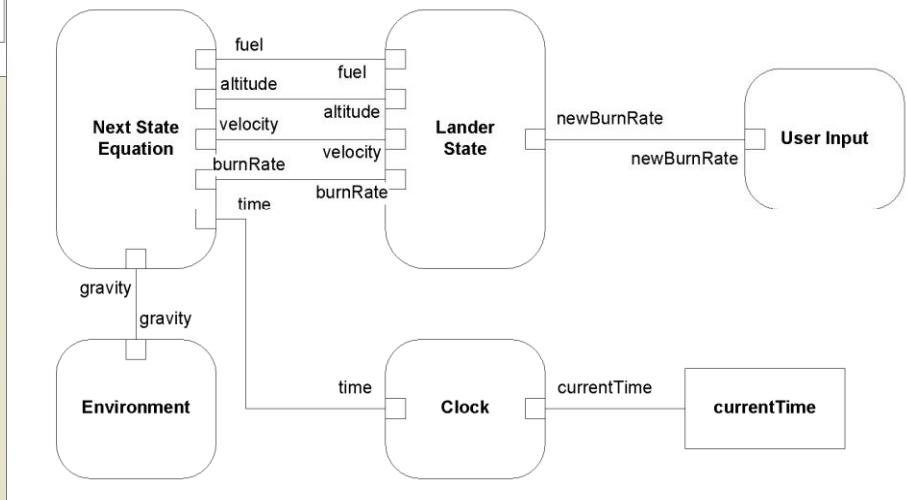
SysML

- An extended version of UML
- Developed by a large consortium of organizations (mainly large system integrators and developers)
- Intended to mitigate UML's "software bias"
- SysML group found UML standard insufficient and profiles not enough to resolve this
 - ◆ Developed new diagram types to capture system-engineering specific views
 - ◆ Limited momentum among tool vendors; focus shifting to more heavily use UML profiles

Req [package] LunarLanderRequirements [Requirement Derivation]



SysML Requirement Diagram

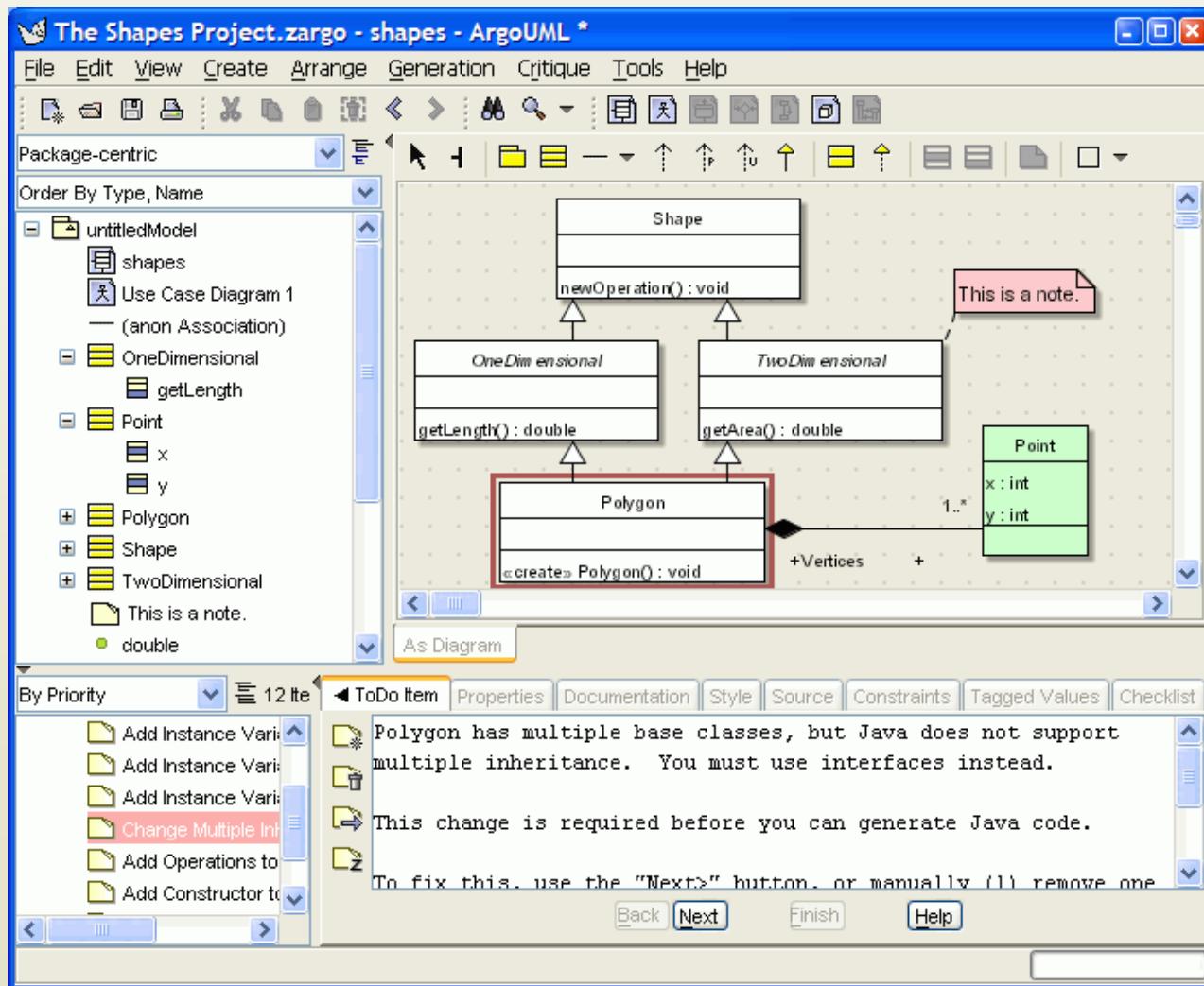


SysML Parametric Diagram

Standard UML Tools

- E.g., Rational Rose, ArgoUML, Microsoft Visio
- These are *de facto* standards
- All support drawing UML diagrams
- Vary along several dimensions
 - ◆ Support for built-in UML extension mechanisms
 - Profiles, stereotypes, tagged values, constraints
 - ◆ Support for UML consistency checking
 - ◆ Ability to generate other artifacts
 - ◆ Generation of UML from other artifacts
 - ◆ Traceability to other systems
 - ◆ Support for capturing non-UML information

ArgoUML – a UML tool

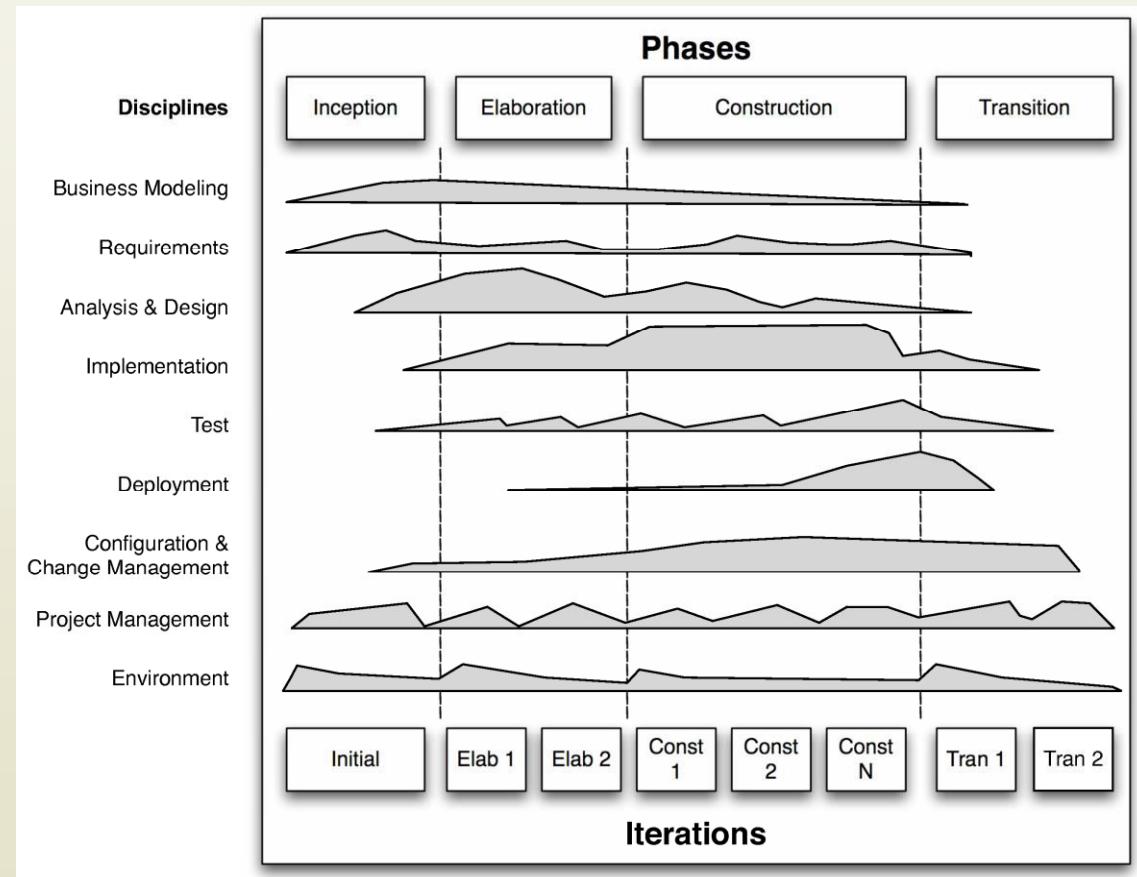


Telelogic System Architect

- Formerly Popkin System Architect; popular among architects
 - ◆ Supports 50+ different diagram types
 - UML, IDEF, OMT, generic flowcharting, even GUI design
 - Variants for DoDAF, service-oriented architectures, enterprise resource planning
- Effectively generic diagram editor specialized for many different diagram types with different symbols, connections
 - ◆ Very little understanding of diagram semantics
 - ◆ Specialized variants have some understanding of semantics but generally less than notation-specific editors

Rational Unified Process

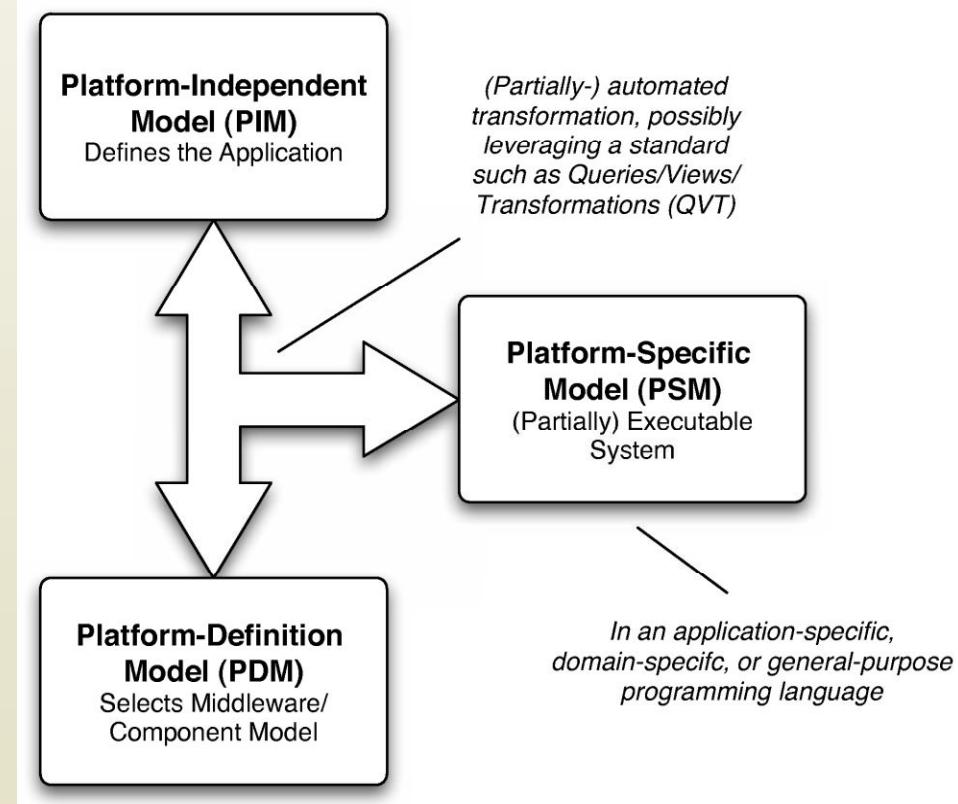
- Phased iterative process framework/meta-process
- Like spiral model, focus on iteration and risk management
- Tends to view architecture as an artifact rather than a pervasive discipline



Redrawn from the RUP documentation

Model-Driven Architecture

- Also known as MDA
- Core idea: specify your architecture in detailed enough terms that implementations can be auto-generated entirely from models
- This vision is hard to achieve in general
 - ◆ May be more successful in a strong DSSE context



Redrawn from the MDA documentation

Overall Takeaways

- Standards confer many benefits
 - ◆ Network effects, reusable engineering knowledge, interoperability, common vocabulary and understanding
- But are not a panacea!
 - ◆ Knowledge of a breadth of standards is needed to be a good architect, but it is critical to maintain perspective
- Caveats
 - ◆ This has been a very quick tour through complex standards; many standards are hundreds of pages and can't adequately be explained in five minutes
 - ◆ Most available online – investigate yourself!

People, Roles, and Teams

**Software Architecture
Lecture 27**

The Need

- The greatest architectures are the product of
 - ◆ A single mind or
 - ◆ A very small, carefully structured team
 - Rechtin, *Systems Architecting: Creating & Building Complex Systems*, 1991, p21
- Every project should have exactly 1 identifiable architect
 - ◆ For larger projects, principal architect should be backed up by architect team of modest size
 - Booch, *Object Solutions*, 1996

Software Architects

- Architect is “jack of all trades”
- Maintainer of system’s conceptual integrity
- Part of team
 - ◆ Set of people with complementary skills
 - ◆ Committed to common
 - Purpose
 - Performance goals
 - Approach
 - ◆ Hold each other accountable
- Life of architect is long series of locally suboptimal decisions made partly in the dark
 - ◆ Sometimes painful

Desired Skill Set

- Software development expertise
- Domain expertise
- Communicator
- Strategist
- Consultant
- Leader
- Technologist
- Cost estimator
- Cheerleader
- Politician
- Salesperson

Blending the Skill Set

- May need different people & skills based on
 - ◆ Characteristics of project & domain
 - ◆ Lifecycle “phase”
 - ◆ Type of architecture
 - Enterprise vs. product-line vs. product
 - ◆ Distinction between junior & senior architects
- Each architect should possess some subset of above skills
- What architects are usually *not* in a project
 - ◆ Developers – though they may prototype their ideas
 - ◆ Managers – except in small organizations

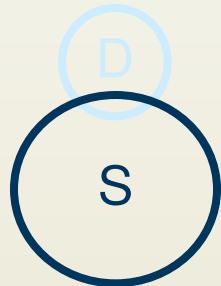
Architects As Software Development Experts

- Must understand nuances of software development
 - ◆ Principles
 - ◆ Methods & techniques
 - ◆ Methodologies
 - ◆ Tools
- Need not be world-class software programmers
- Should understand ramifications of architectural choices
 - ◆ Do not live in ivory tower
 - ◆ Some architectural choices constrain implementation options
 - ◆ Some implementation-level techniques & tools constrain architectural choices

Architects As Domain Experts

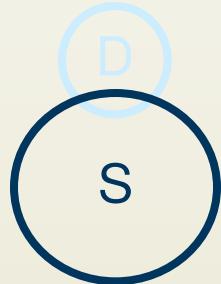
- Software engineering expertise is not enough
- Problem domain nuances
 - ◆ Maturity
 - ◆ Stability
 - ◆ System user profile
- May greatly affect selected & developed architectural solutions
 - ◆ Distribution
 - ◆ Scale
 - ◆ Evolvability
- Requires artifacts that model problem space
 - ◆ Not solution space

Team Needs Balance & Shared Vocabulary

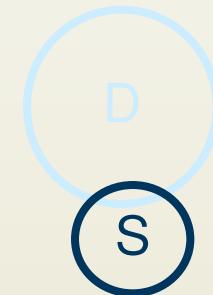


Too little domain
knowledge

Team Needs Balance & Shared Vocabulary

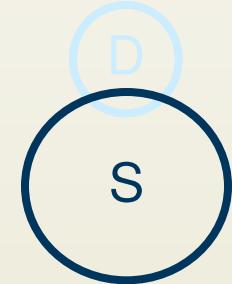


Too little domain
knowledge

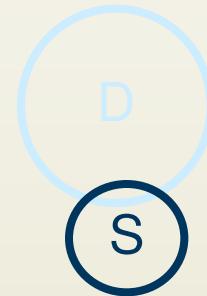


Too little SWE
knowledge

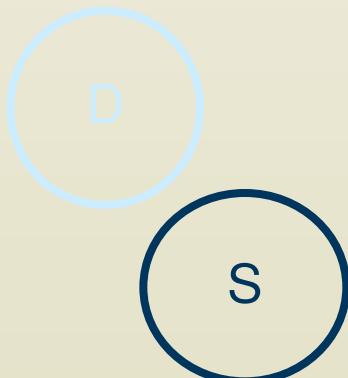
Team Needs Balance & Shared Vocabulary



Too little domain knowledge



Too little SWE knowledge



No Shared Vocabulary

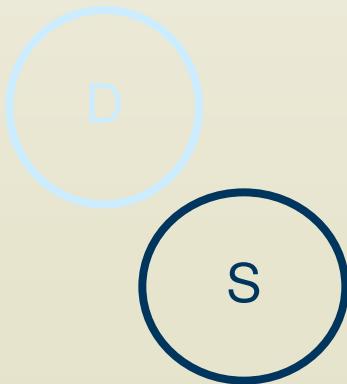
Team Needs Balance & Shared Vocabulary



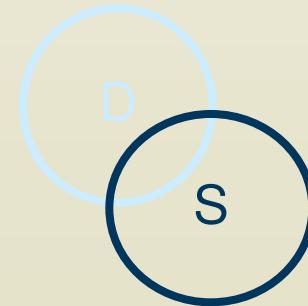
Too little domain knowledge



Too little SWE knowledge



No Shared Vocabulary



Good!

Architects As Communicators

- At least ½ of the job
- Must
 - ◆ Listen to stakeholder concerns
 - ◆ Explain the architecture
 - ◆ Negotiate compromises
- Need good communication skills
 - ◆ Writing
 - ◆ Speaking
 - ◆ Presenting

Architects Communicate With

- Managers
 - ◆ Must relay key messages
 - Architecture is useful & important
 - Ensure support throughout project
 - ◆ Must listen to concerns
 - Cost
 - Schedule
- Developers
 - ◆ Convince them that architecture is effective
 - ◆ Justify local suboptimal choices
 - ◆ Listen to problems
 - Tools
 - Methods
 - Design choices
- Other software architects
 - ◆ Ensure conceptual integrity
 - ◆ Ensure desired system properties & evolution

Architects Also Communicate With

- System engineers
 - ◆ Coordinate requirements & solutions
 - ◆ Explain how architecture addresses key concerns
- Customers
 - ◆ Determine needs
 - ◆ Explain how architecture addresses requirements
- Users
 - ◆ Determine needs
 - ◆ Explain how architecture addresses those needs
 - ◆ Listen to problems
- Marketers
 - ◆ Get/help set goals & directions
 - ◆ Explain how architecture addresses marketing objectives

Architects As Strategists

- Developing elegant architecture is not enough
 - ◆ Technology is only part of picture
 - ◆ Architecture must be right for organization
- Must fit organization's
 - ◆ Business strategy
 - ◆ Rationale behind it
 - ◆ Business practices
 - ◆ Planning cycles
 - ◆ Decision making processes
- Must also be aware of competitors'
 - ◆ Products
 - ◆ Strategies
 - ◆ Processes

Architects As Consultants

- Architects must recognize developers are their primary “customer”
- Developers
 - ◆ Goals do not match architects’
 - ◆ Not focused on making architecture successful
 - ◆ Focused on
 - Satisfying functional, quality, and scheduling requirements
 - Subsystems for which they are responsible

Architects As Consultants (cont.)

- Developers must be convinced to
 - ◆ Learn, adhere to, & effectively leverage architecture
 - Architects need to make these tasks reasonably easy
 - ◆ Document & report architecturally-relevant modifications
 - Architects need to make clear what's architecturally-relevant
 - ◆ "Where are load bearing walls?"

Architects As Leaders

- Must be technical leader
 - ◆ Based on knowledge & achievement
 - ◆ Command respect by ideas, expertise, words, & actions
 - ◆ Cannot rely on position in org chart
 - Must do so without managerial authority
- Ensures that design decisions, guidelines, and rules are followed
- To improve productivity & quality, injects
 - ◆ New ideas, solutions, & techniques
 - ◆ Mentor newcomers & junior people
- Make decisions & help assure their implementation
 - ◆ Enlist help of others in doing so

Architects As Technologists

- Understand software development approaches
 - ◆ E.g., object-oriented (OO) & component-based
- Understand fundamental technologies
 - ◆ Operating system/networking
 - ◆ Middleware
 - ◆ Security
 - ◆ Databases
 - ◆ Graphical user interface (GUI) toolkits
- Keep on top of trends
 - ◆ E.g., CORBA, COM/DCOM, JavaBeans, UML, XML
- Demonstrated expertise in
 - ◆ System modeling
 - ◆ Architectural trade-off analysis
 - ◆ Tying architectural solutions to system requirements

Architects As Cost Estimators

- Must understand financial ramifications of architectural choices
 - ◆ Green-field vs. Brown-field development
 - ◆ Cost of COTS adoption
 - ◆ Cost of development for reuse
 - ◆ Company's financial stability & position in marketplace
- Technologically superior solution is not always most appropriate one
 - ◆ Impact on cost & schedule
- Quick, approximate cost estimations are often sufficient
 - ◆ Detailed cost estimation techniques can be applied once set of candidate solutions is narrowed down

Architects As Cheerleaders

- Especially needed on long, large, complex projects
 - ◆ Development teams work in trenches on small subsets of project
 - ◆ Managers lose sight of overall project goals
 - ◆ Customers get impatient from long wait
- Must
 - ◆ Maintain high-level vision with necessary details sprinkled in
 - ◆ Convince different stakeholders of architecture's
 - Beauty
 - Utility
 - Adaptability
 - Technological impact
 - Financial impact
 - ◆ Keep the troops' morale high

Architects As Politicians

- Must get key organization players committed to architecture
- Must do a lot of influencing themselves
 - ◆ Find out who the key players are
 - ◆ Find out what they want
 - ◆ Find out the organization behind the organization
- Architects must continuously
 - ◆ Listen
 - ◆ Network
 - ◆ Articulate
 - ◆ Sell vision
 - ◆ See problem from multiple viewpoints

Architects as Salespeople

- For many of the above reasons, architects must sell
 - ◆ Overall vision
 - ◆ Technological solutions
 - ◆ Key properties of architecture
 - ◆ Key properties of eventual system that architecture will directly enable
 - ◆ Cost/schedule profile
 - ◆ Importance of sticking to architecture
 - ◆ Penalties of deviating from it

Software Architecture Team

- Collection of software architects
- Typically stratified
- Team size fluctuates during life of project
 - ◆ 1 architect per 10 developers during project inception
 - ◆ 1 architect per 12-15 developers in later stages
- Architects may
 - ◆ Become subsystem development leads
 - Maintainers of grand vision on development team
 - Bridges to “central” architecture team for duration of project
 - ◆ Be shifted to other projects
 - After project’s architecture is sufficiently stable

Role of Architecture Team

- Define software architecture
- Maintain architectural integrity of software
- Assess technical risks associated with design
- Propose order & contents of development iterations
- Coordinate & coexist with other teams
- Assist in project management decisions
- Assist marketing in future product definition

Defining Software Architecture

- The architecture team must define
 - ◆ Major design major elements
 - ◆ System organization/structure
 - ◆ The way major elements interact
- Works with system engineers & development teams

Maintaining Architectural Integrity

- The architecture team develops & maintains guidelines for
 - ◆ Design
 - ◆ Programming
- Helps with reviews
- Approves
 - ◆ Changes to interfaces of major components
 - ◆ Departures from guidelines
- Final arbiter on aesthetics
- Assists change control board with resolving software problems

Assessing Technical Risks

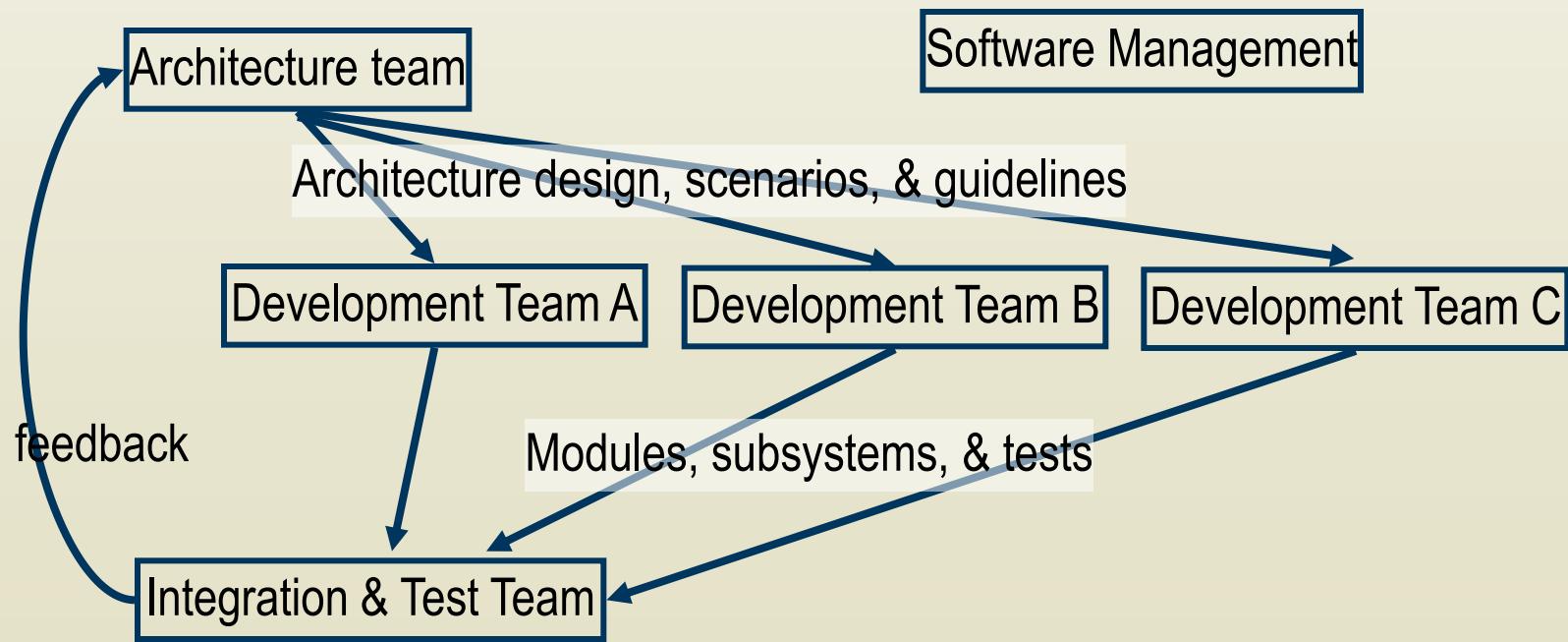
- The architecture team maintains lists of perceived risks
- May propose exploratory studies or prototypes

Ordering & Content Of Development Iterations

- The architecture team determines order & contents of iterations based on
 - ◆ Selected scenarios
 - ◆ Services to be studied & implemented
- Helps development teams transition from architectural design to more detailed design

Coordinate & Coexist with Other Teams

- No structural difference between architecture team & other teams
- Just focused on higher level issues



Pitfalls of Software Architect Teams

- Imbalance of skills
 - ◆ Lack of software development experience
 - ◆ Lack of domain expertise
- Lack of authority
 - ◆ Team acts as committee
- Life in ivory tower
- Confusing tools/techniques/methodologies with architectures
- Procrastination

Pitfall: Lack Of Authority

- Problem
 - ◆ What incentives are there for group leaders to
 - Follow recommendations of architecture team?
 - Report progress or problems to architecture team?
 - ◆ Architect team
 - Frequently has no explicit authority
 - ◆ Architects are not managers
 - Just another team in organization
 - ◆ Problem compounded when external architect or architecture team is hired
- Solution:
 - ◆ Must influence based on skills & experience
 - ◆ Must communicate

Pitfall: Life In Ivory Tower

- Problem
 - ◆ Developers & managers must be aware of architecture team's existence & role
- Solution
 - ◆ Team must continuously communicate with rest of personnel
 - ◆ Team must be co-located with rest of project personnel
 - ◆ Do not use team as retirement home for ageing developers
 - ◆ Architecture team must recognize & adjust to organizational realities
 - Technological base
 - Personnel issues
 - Organizational politics
 - Market pressures

Pitfall: Imbalance Of Skills

- Problem
 - ◆ Predominant expertise in one area creates imbalance
 - Database
 - GUI
 - Networking
 - Systems
 - ◆ Imbalance may affect how architecture is
 - Designed
 - Communicated
 - Evolved
- Solution
 - ◆ Balanced architecture team appropriate to
 - Project type & size
 - Problem domain
 - Personnel

Pitfall: Confusing Tools With Architectures

- Problem
 - ◆ Common pitfall
 - ◆ Usual culprits
 - Databases
 - GUI
 - Case tools
 - ◆ More recently, the culprit is middleware
 - “Our architecture is CORBA”
 - ◆ Tools tend to influence architecture
- Solution
 - ◆ Balanced architecture team

Pitfall: Procrastination

- Problem
 - ◆ Waiting until the team is convinced it is able to make the “right” decision
 - ◆ Incomplete & changing information yields indecision
 - ◆ Architects’ indecision impacts other teams
 - Domino effect, may paralyze entire project
- Solution
 - ◆ Often better to make a decision than suspend the project
 - Make educated guesses
 - Document rationale for decision
 - Document known consequences
 - Change decision if/when better alternatives present themselves
 - Be decisive
 - ◆ Being an effective architect demands rapidly making tactical decisions & living with resulting anxiety
 - Suboptimal decisions based on incomplete information

Summary

- Designate architect or assemble architecture team to be creators & proponents of common system goal/vision
- Architects must be experienced at least in problem domain & software development
- Software architect is a full-time job
- Charter of software architecture team should
 - ◆ Clearly define its roles & responsibilities
 - ◆ Clearly specify its authority
- Do not isolate software architecture team from the rest of project personnel
- Avoid pitfalls