

The Big Idea

Software Architecture
Lecture 1

The Origins

- Software Engineers have always employed software architectures
 - ◆ Very often without realizing it!
- Address issues identified by researchers and practitioners
 - ◆ Essential software engineering difficulties
 - ◆ Unique characteristics of programming-in-the-large
 - ◆ Need for software reuse
- Many ideas originated in other (non-computing) domains

Software Engineering Difficulties

- Software engineers deal with unique set of problems
 - ◆ Young field with tremendous expectations
 - ◆ Building of vastly complex, but intangible systems
 - ◆ Software is not useful on its own e.g., unlike a car, thus
 - ◆ It must conform to changes in other engineering areas
- Some problems can be eliminated
 - ◆ These are Brooks' "accidental difficulties"
- Other problems can be lessened, but not eliminated
 - ◆ These are Brooks' "essential difficulties"

Accidental Difficulties

- Solutions exist
 - ◆ Possibly waiting to be discovered
- Past productivity increases result of overcoming
 - ◆ Inadequate programming constructs & abstractions
 - Remedied by high-level programming languages
 - Increased productivity by factor of five
 - Complexity was never inherent in program at all

Accidental Difficulties (cont'd)

- Past productivity increases result of overcoming (cont'd)
 - ◆ Viewing results of programming decisions took long time
 - Remedied by time-sharing
 - Turnaround time approaching limit of human perception
 - ◆ Difficulty of using heterogeneous programs
 - Addressed by integrated software development environments
 - Support task that was conceptually always possible

Essential Difficulties

- Only partial solutions exist for them, if any
- Cannot be abstracted away
 - ◆ Complexity
 - ◆ Conformity
 - ◆ Changeability
 - ◆ Intangibility

Complexity

- No two software parts are alike
 - ◆ If they are, they are abstracted away into one
- Complexity grows non-linearly with size
 - ◆ E.g., it is impossible to enumerate all states of program
 - ◆ Except perhaps “toy” programs

Conformity

- Software is required to conform to its
 - ◆ Operating environment
 - ◆ Hardware
- Often “last kid on block”
- Perceived as most conformable

Changeability

- Change originates with
 - ◆ New applications, users, machines, standards, laws
 - ◆ Hardware problems
- Software is viewed as infinitely malleable

Intangibility

- Software is not embedded in space
 - ◆ Often no constraining physical laws
- No obvious representation
 - ◆ E.g., familiar geometric shapes

Pewter Bullets

- Ada, C++, Java and other high-level languages
- Object-oriented design/analysis/programming
- Artificial Intelligence
- Automatic Programming
- Graphical Programming
- Program Verification
- Environments & tools
- Workstations

Promising Attacks On Complexity (In 1987)

- Buy vs. Build
- Requirements refinement & rapid prototyping
 - ◆ Hardest part is deciding what to build (or buy?)
 - ◆ Must show product to customer to get complete spec.
 - ◆ Need for iterative feedback

Promising Attacks On Complexity (cont'd)

- Incremental/Evolutionary/Spiral Development
 - ◆ Grow systems, don't build them
 - ◆ Good for morale
 - ◆ Easy backtracking
 - ◆ Early prototypes
- Great designers
 - ◆ Good design can be taught; great design cannot
 - ◆ Nurture great designers

Primacy of Design

- Software engineers collect requirements, code, test, integrate, configure, etc.
- An architecture-centric approach to software engineering places an emphasis on design
 - ◆ Design pervades the engineering activity from the very beginning
- But how do we go about the task of architectural design?

Analogy: Architecture of Buildings

- We all live in them
- (We think) We know how they are built
 - ◆ Requirements
 - ◆ Design (blueprints)
 - ◆ Construction
 - ◆ Use
- This is similar (though not identical) to how we build software

Some Obvious Parallels

- Satisfaction of customers' needs
- Specialization of labor
- Multiple perspectives of the final product
- Intermediate points where plans and progress are reviewed

Deeper Parallels

- Architecture is different from, but linked with the product/structure
- Properties of structures are induced by the design of the architecture
- The architect has a distinctive role and character

Deeper Parallels (cont'd)

- Process is not as important as architecture
 - ◆ Design and resulting qualities are at the forefront
 - ◆ Process is a means, not an end
- Architecture has matured over time into a discipline
 - ◆ Architectural styles as sets of constraints
 - ◆ Styles also as wide range of solutions, techniques and palettes of compatible materials, colors, and sizes

More about the Architect

- A distinctive role and character in a project
- Very broad training
- Amasses and leverages extensive experience
- A keen sense of aesthetics
- Deep understanding of the domain
 - ◆ Properties of structures, materials, and environments
 - ◆ Needs of customers

More about the Architect (cont'd)

- Even first-rate programming skills are insufficient for the creation of complex software applications
 - ◆ But are they even necessary?

Limitations of the Analogy...

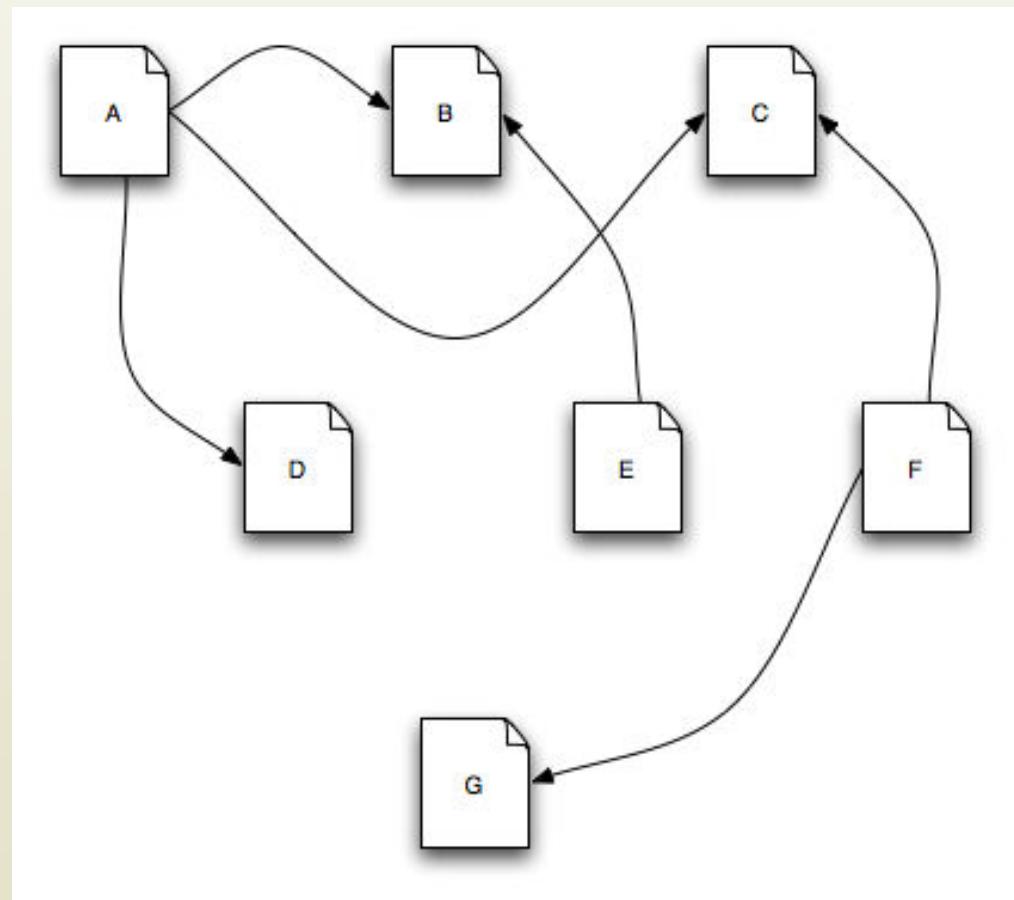
- We know a lot about buildings, much less about software
- The nature of software is different from that of building architecture
- Software is much more malleable than physical materials
- The two “construction industries” are very different
- Software deployment has no counterpart in building architecture
- Software is a machine; a building is not

...But Still Very Real Power of Architecture

- Giving preeminence to architecture offers the potential for
 - ◆ Intellectual control
 - ◆ Conceptual integrity
 - ◆ Effective basis for knowledge reuse
 - ◆ Realizing experience, designs, and code
 - ◆ Effective project communication
 - ◆ Management of a set of variant systems
- Limited-term focus on architecture will not yield significant benefits!

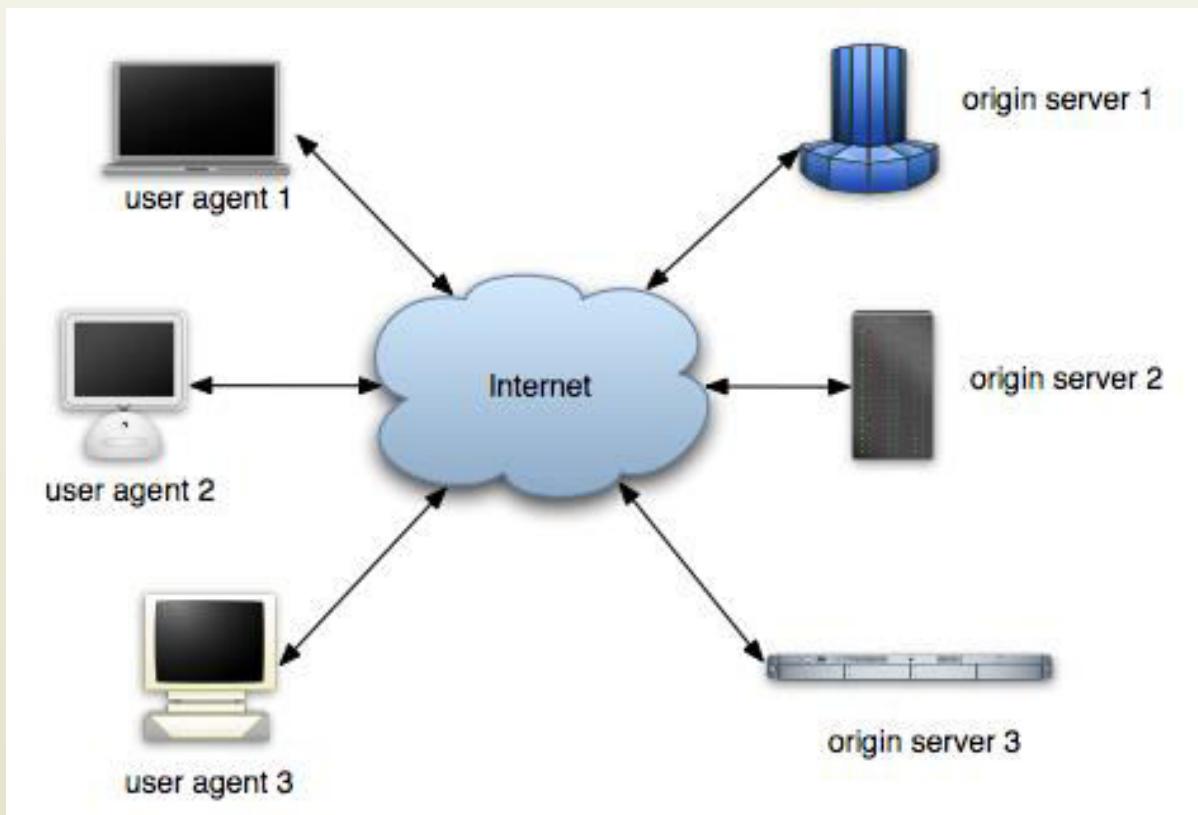
Architecture in Action: WWW

- This is the Web



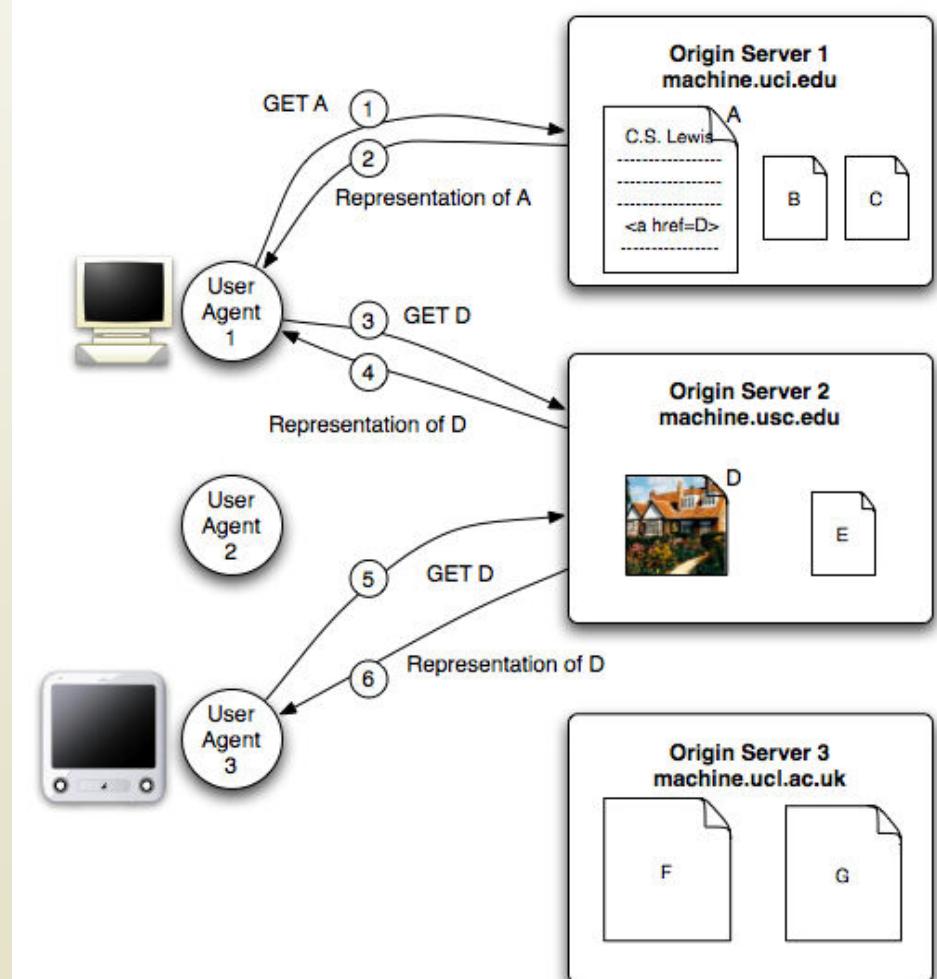
Architecture in Action: WWW

- So is this



Architecture in Action: WWW

- And this



www in a (Big) Nutshell

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

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

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

WWW's Architecture

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

WWW's Architecture (cont'd)

- Stylistic constraints of the Web's architectural style are not apparent in the code
 - ◆ The effects of the constraints are evident in the Web
- One of the world's most successful applications is only understood adequately from an architectural vantage point.

Architecture in Action: Desktop

- Remember pipes and filters in Unix?
 - ◆ ls invoices | grep –e august | sort
- Application architecture can be understood based on very few rules
- Applications can be composed by non-programmers
 - ◆ Akin to Lego blocks
- A simple architectural concept that can be comprehended and applied by a broad audience

Architecture in Action: Product Line

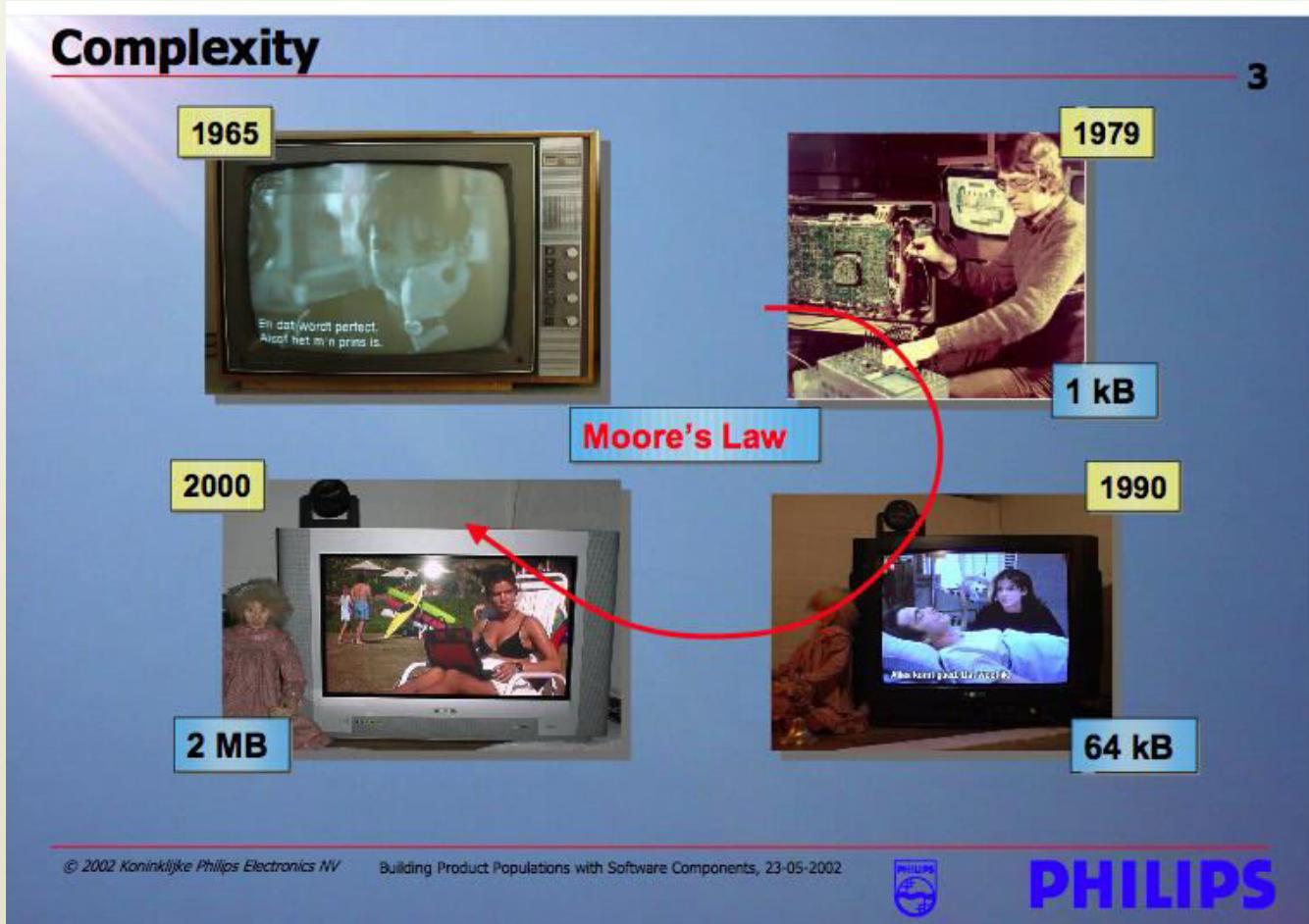
- Motivating example
 - ◆ A consumer is interested in a 35-inch HDTV with a built-in DVD player for the North American market.

Such a device might contain upwards of a million lines of embedded software.

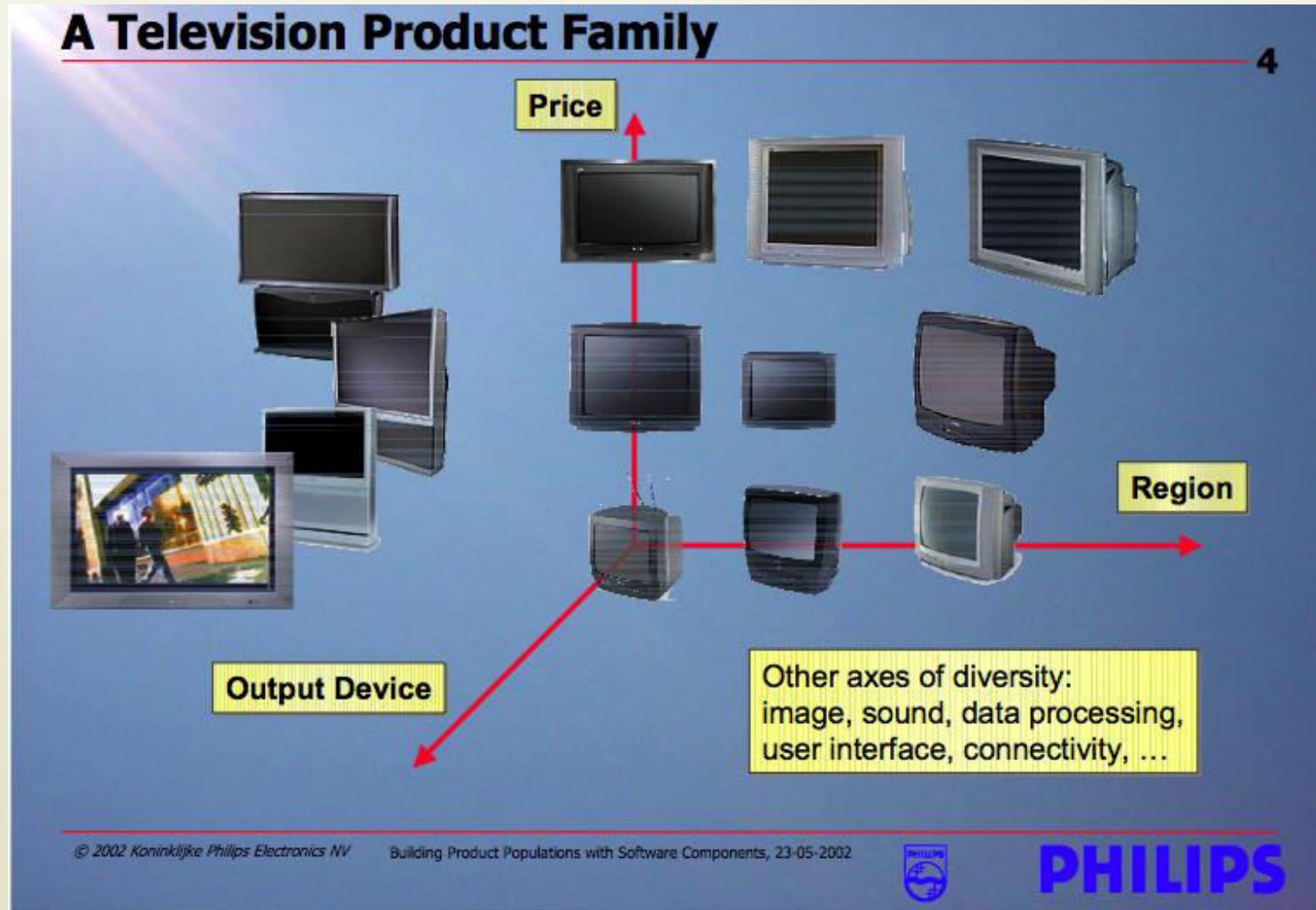
This particular television/DVD player will be very similar to a 35-inch HDTV without the DVD player, and also to a 35-inch HDTV with a built-in DVD player for the European market, where the TV must be able to handle PAL or SECAM encoded broadcasts, rather than North America's NTSC format.

These closely related televisions will similarly each have a million or more lines of code embedded within them.

Growing Sophistication of Consumer Devices



Families of Related Products



The Necessity and Benefit of PLs

- Building each of these TVs from scratch would likely put Philips out of business
- Reusing structure, behaviors, and component implementations is increasingly important to successful business practice
 - ◆ It simplifies the software development task
 - ◆ It reduces the development time and cost
 - ◆ it improves the overall system reliability
- Recognizing and exploiting commonality and variability across products

Reuse as the Big Win

- Architecture: reuse of
 - ◆ Ideas
 - ◆ Knowledge
 - ◆ Patterns
 - ◆ engineering guidance
 - ◆ Well-worn experience
- Product families: reuse of
 - ◆ Structure
 - ◆ Behaviors
 - ◆ Implementations
 - ◆ Test suites...

Added Benefit – Product Populations

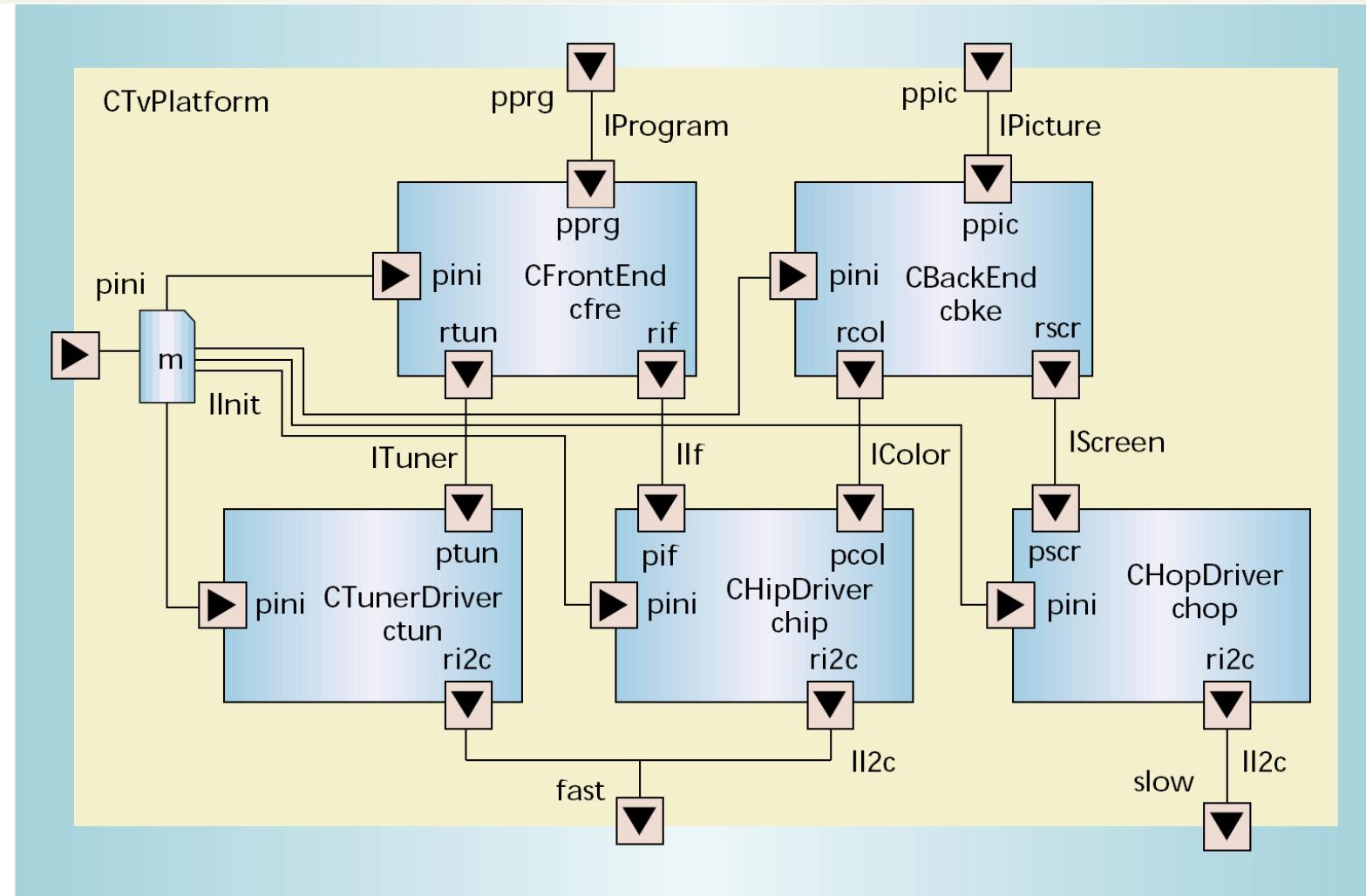
Convergence 6

TV	+	VCR	=	TVCR	
TV	+	DVD	=	TV-DVD	
TV	+	HD	=	Tivo	
TV	+	STB	=	Digital TV	
TV	+	Audio	=	Home Theater	

© 2002 Koninklijke Philips Electronics NV Building Product Populations with Software Components, 23-05-2002

 **PHILIPS**

The Centerpiece – Architecture



Summary

- Software is complex
- So are buildings
 - ◆ And other engineering artifacts
 - ◆ Building architectures are an attractive source of analogy
- Software engineers can learn from other domains
- They also need to develop—and have developed—a rich body of their own architectural knowledge and experience

Architectures in Context

**Software Architecture
Lecture 2**

Fundamental Understanding

- Architecture is a set of principal design decisions about a software system
- Three fundamental understandings of software architecture
 - ◆ Every application has an architecture
 - ◆ Every application has at least one architect
 - ◆ Architecture is not a phase of development

Wrong View: Architecture as a Phase

- ◆ Treating architecture as a phase denies its foundational role in software development
- ◆ More than “high-level design”
- ◆ Architecture is also represented, e.g., by object code, source code, ...

Context of Software Architecture

- Requirements
- Design
- Implementation
- Analysis and Testing
- Evolution
- Development Process

Requirements Analysis

- Traditional SE suggests requirements analysis should remain unsullied by any consideration for a design
- However, without reference to existing architectures it becomes difficult to assess practicality, schedules, or costs
 - ◆ In building architecture we talk about specific rooms...
 - ◆ ...rather than the abstract concept “means for providing shelter”
- In engineering new products come from the observation of existing solution and their limitations

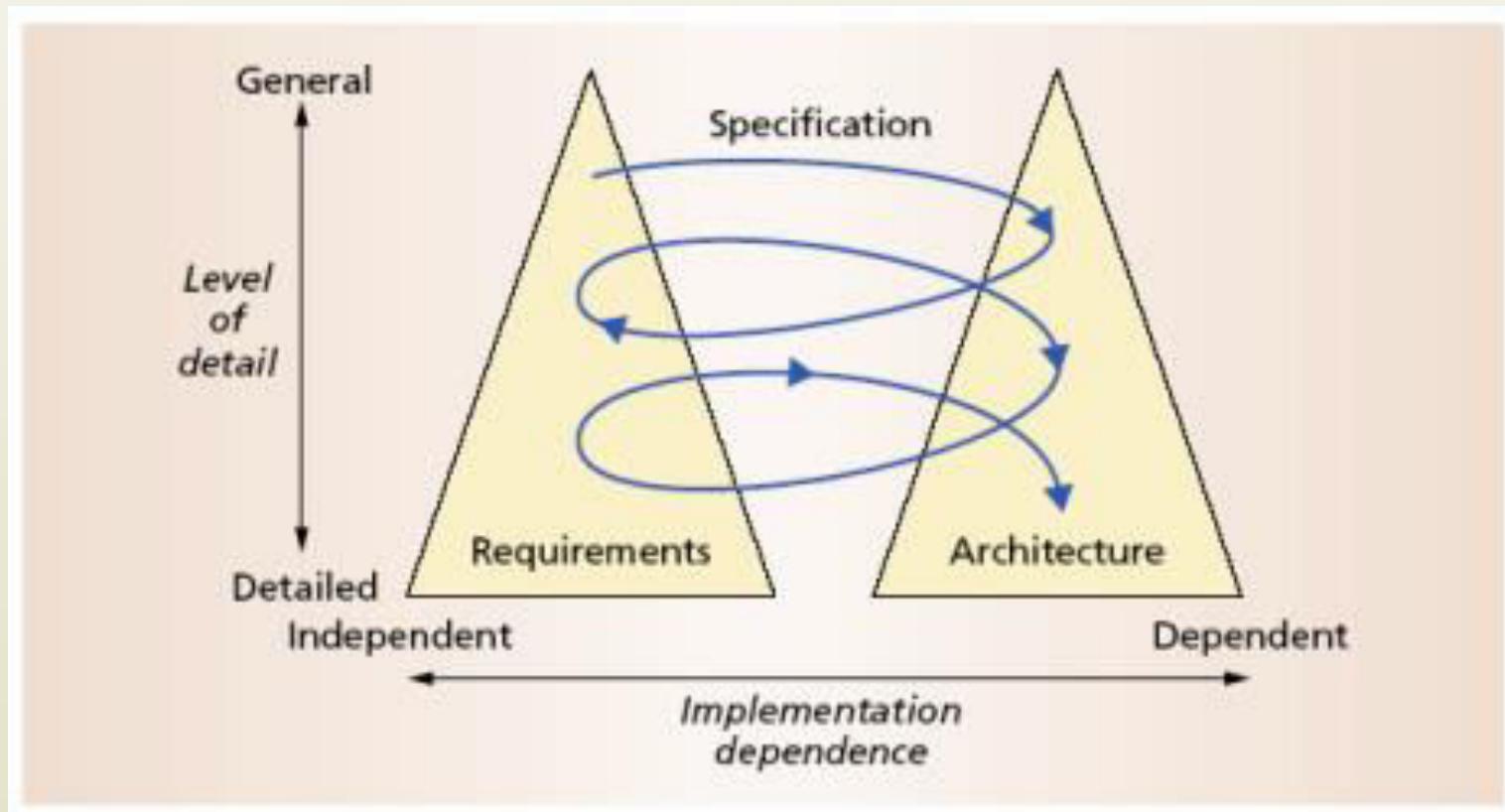
New Perspective on Requirements Analysis

- Existing designs and architectures provide the solution vocabulary
- Our understanding of what works now, and how it works, affects our wants and perceived needs
- The insights from our experiences with existing systems
 - ◆ helps us imagine what might work and
 - ◆ enables us to assess development time and costs
- → Requirements analysis and consideration of design must be pursued at the same time

Non-Functional Properties (NFP)

- NFPs are the result of architectural choices
- NFP questions are raised as the result of architectural choices
- Specification of NFP might require an architectural framework to even enable their statement
- An architectural framework will be required for assessment of whether the properties are achievable

The Twin Peaks Model



Design and Architecture

- Design is an activity that pervades software development
- It is an activity that creates part of a system's architecture
- Typically in the traditional Design Phase decisions concern
 - ◆ A system's structure
 - ◆ Identification of its primary components
 - ◆ Their interconnections
- Architecture denotes the set of principal design decisions about a system
 - ◆ That is more than just structure

Architecture-Centric Design

- Traditional design phase suggests translating the requirements into algorithms, so a programmer can implement them
- Architecture-centric design
 - ◆ stakeholder issues
 - ◆ decision about use of COTS component
 - ◆ overarching style and structure
 - ◆ package and primary class structure
 - ◆ deployment issues
 - ◆ post implementation/deployment issues

Design Techniques

- Basic conceptual tools
 - ◆ Separation of concerns
 - ◆ Abstraction
 - ◆ Modularity
- Two illustrative widely adapted strategies
 - ◆ Object-oriented design
 - ◆ Domain-specific software architectures (DSSA)

Object-Oriented Design (OOD)

- Objects
 - ◆ Main abstraction entity in OOD
 - ◆ Encapsulations of state with functions for accessing and manipulating that state

Pros and Cons of OOD

- Pros
 - ◆ UML modeling notation
 - ◆ Design patterns
- Cons
 - ◆ Provides only
 - One level of encapsulation (the object)
 - One notion of interface
 - One type of explicit connector (procedure call)
 - ◆ Even message passing is realized via procedure calls
 - ◆ OO programming language might dictate important design decisions
 - ◆ OOD assumes a shared address space

DSSA

- Capturing and characterizing the best solutions and best practices from past projects within a domain
- Production of new applications can focus on the points of novel variation
- Reuse applicable parts of the architecture and implementation
- Applicable for product lines
 - ◆ →Recall the Philips Koala example discussed in the previous lecture

Implementation

- The objective is to create machine-executable source code
 - ◆ That code should be faithful to the architecture
 - Alternatively, it may adapt the architecture
 - How much adaptation is allowed?
 - Architecturally-relevant vs. -unimportant adaptations
 - ◆ It must fully develop all outstanding details of the application

Faithful Implementation

- All of the structural elements found in the architecture are implemented in the source code
- Source code must not utilize major new computational elements that have no corresponding elements in the architecture
- Source code must not contain new connections between architectural elements that are not found in the architecture
- Is this realistic?
Overly constraining?
What if we deviate from this?

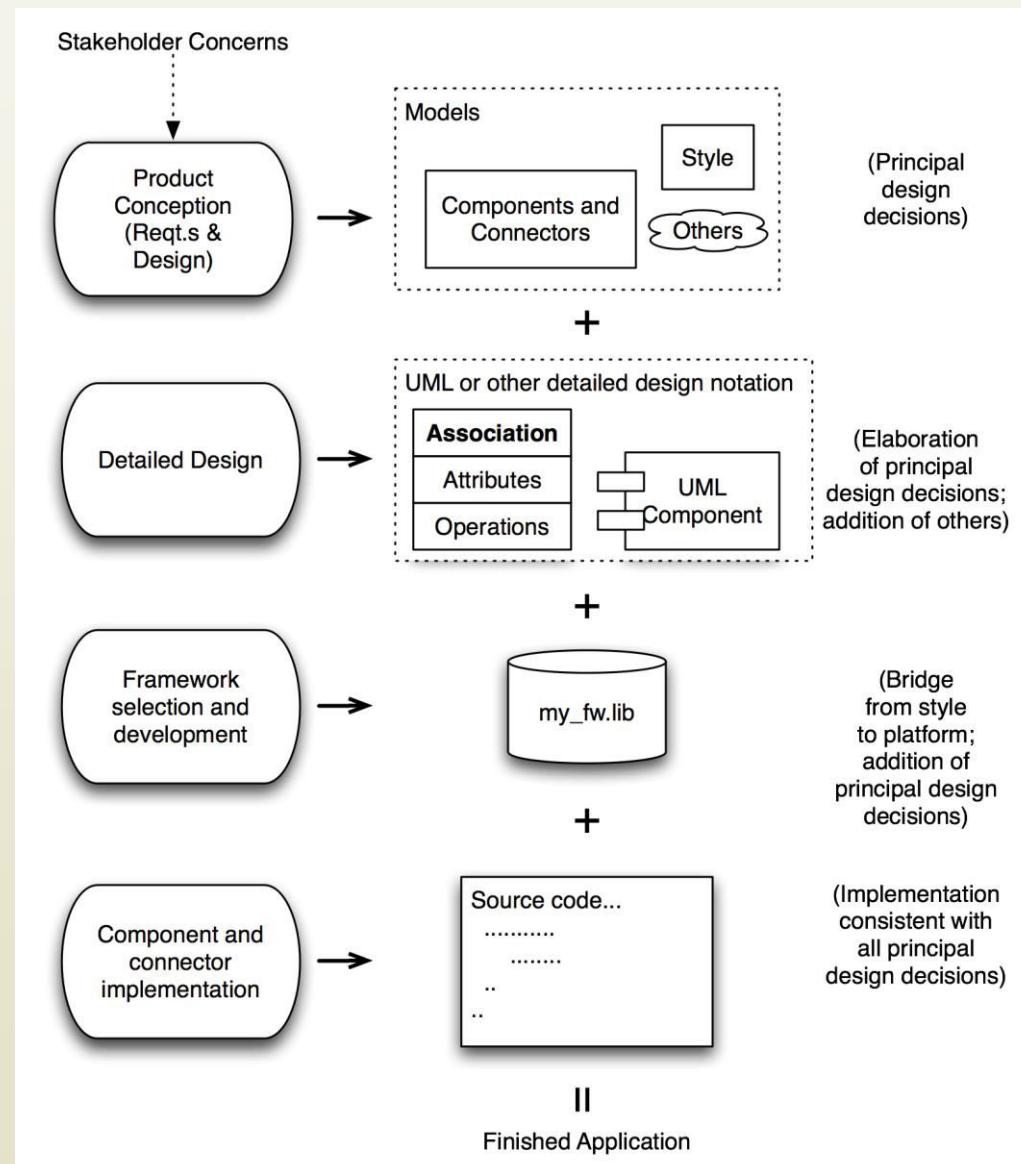
Unfaithful Implementation

- The implementation does have an architecture
 - ◆ It is latent, as opposed to what is documented.
- Failure to recognize the distinction between planned and implemented architecture
 - ◆ robs one of the ability to reason about the application's architecture in the future
 - ◆ misleads all stakeholders regarding what they believe they have as opposed to what they really have
 - ◆ makes any development or evolution strategy that is based on the documented (but inaccurate) architecture doomed to failure

Implementation Strategies

- Generative techniques
 - ◆ e.g. parser generators
- Frameworks
 - ◆ collections of source code with identified places where the engineer must “fill in the blanks”
- Middleware
 - ◆ CORBA, DCOM, RPC, ...
- Reuse-based techniques
 - ◆ COTS, open-source, in-house
- Writing all code manually

How It All Fits Together



Analysis and Testing

- Analysis and testing are activities undertaken to assess the qualities of an artifact
- The earlier an error is detected and corrected the lower the aggregate cost
- Rigorous representations are required for analysis, so precise questions can be asked and answered

Analysis of Architectural Models

- Formal architectural model can be examined for internal consistency and correctness
- An analysis on a formal model can reveal
 - ◆ Component mismatch
 - ◆ Incomplete specifications
 - ◆ Undesired communication patterns
 - ◆ Deadlocks
 - ◆ Security flaws
- It can be used for size and development time estimations

Analysis of Architectural Models (cont'd)

- Architectural model
 - ◆ may be examined for consistency with requirements
 - ◆ may be used in determining analysis and testing strategies for source code
 - ◆ may be used to check if an implementation is faithful

Evolution and Maintenance

- All activities that chronologically follow the release of an application
- Software will evolve
 - ◆ Regardless of whether one is using an architecture-centric development process or not
- The traditional software engineering approach to maintenance is largely ad hoc
 - ◆ Risk of architectural decay and overall quality degradation
- Architecture-centric approach
 - ◆ Sustained focus on an explicit, substantive, modifiable, faithful architectural model

Architecture-Centric Evolution Process

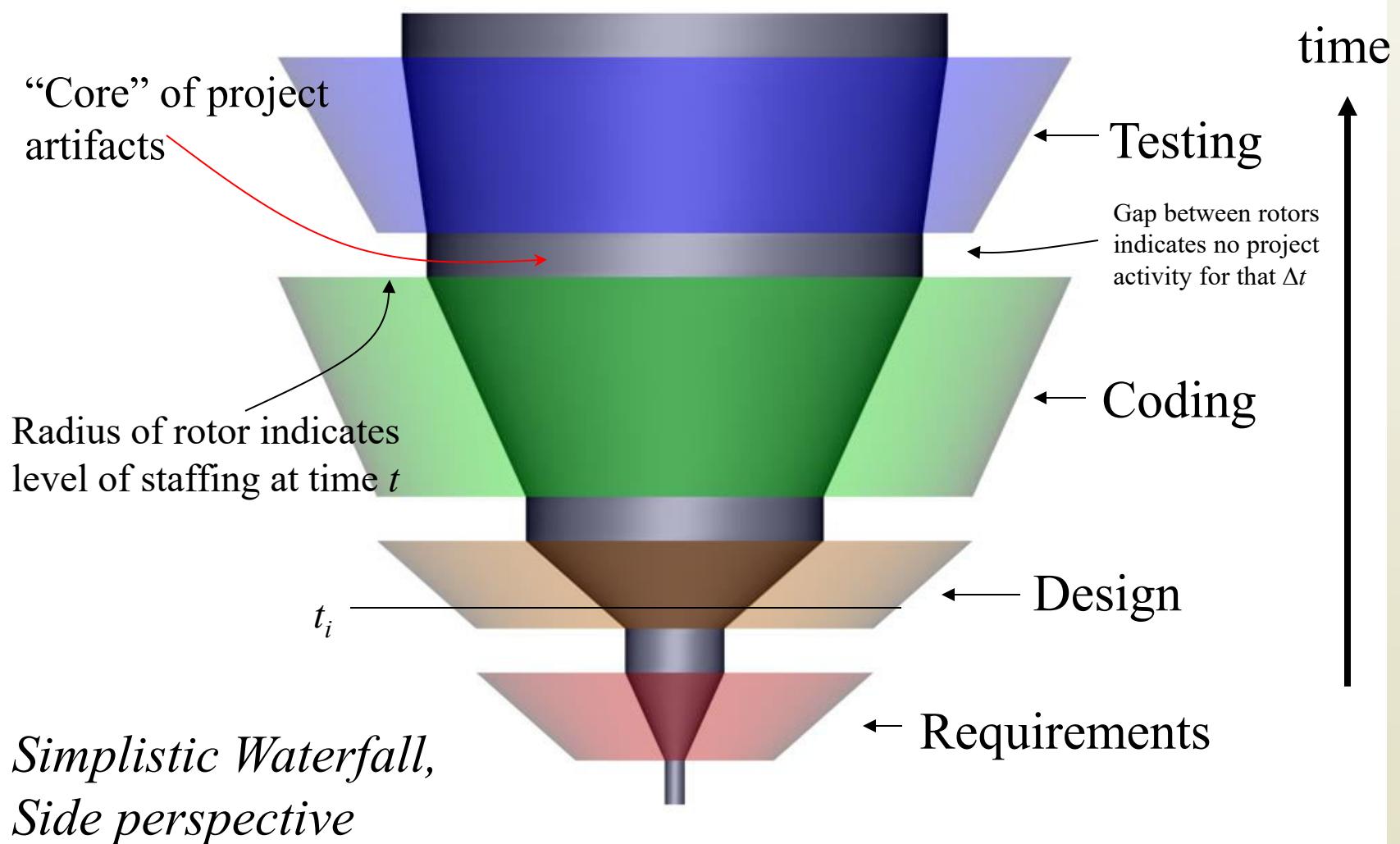
- Motivation
- Evaluation or assessment
- Design and choice of approach
- Action
 - ◆ includes preparation for the next round of adaptation

Processes

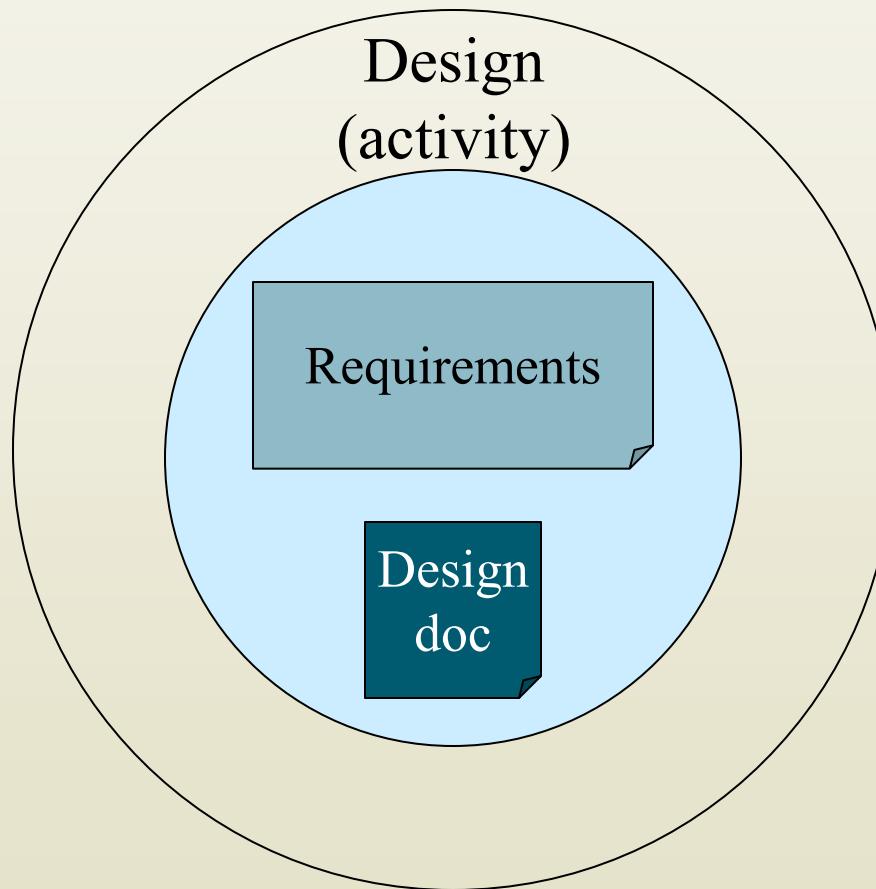
- Traditional software process discussions make the process activities the focal point
- In architecture-centric software engineering the product becomes the focal point
- No single “right” software process for architecture-centric software engineering exists

Turbine – A New Visualization Model

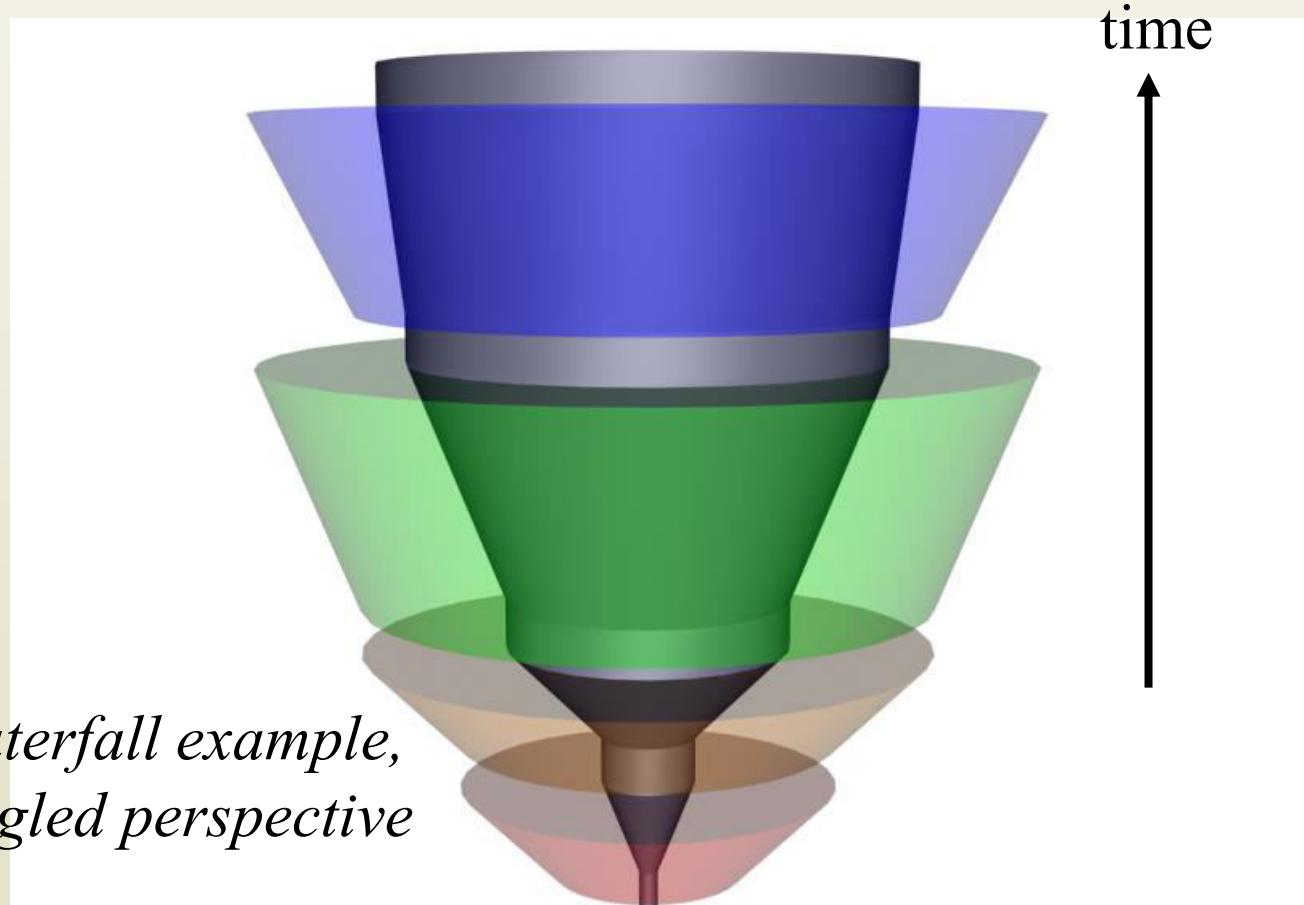
- Goals of the visualization
 - ◆ Provide an intuitive sense of
 - Project activities at any given time
 - ◆ Including concurrency of types of development activities
 - The “information space” of the project
 - ◆ Show centrality of the products
 - (Hopefully) Growing body of artifacts
 - Allow for the centrality of architecture
 - ◆ But work equally well for other approaches, including “dysfunctional” ones
 - ◆ Effective for indicating time, gaps, duration of activities
 - ◆ Investment (cost) indicators



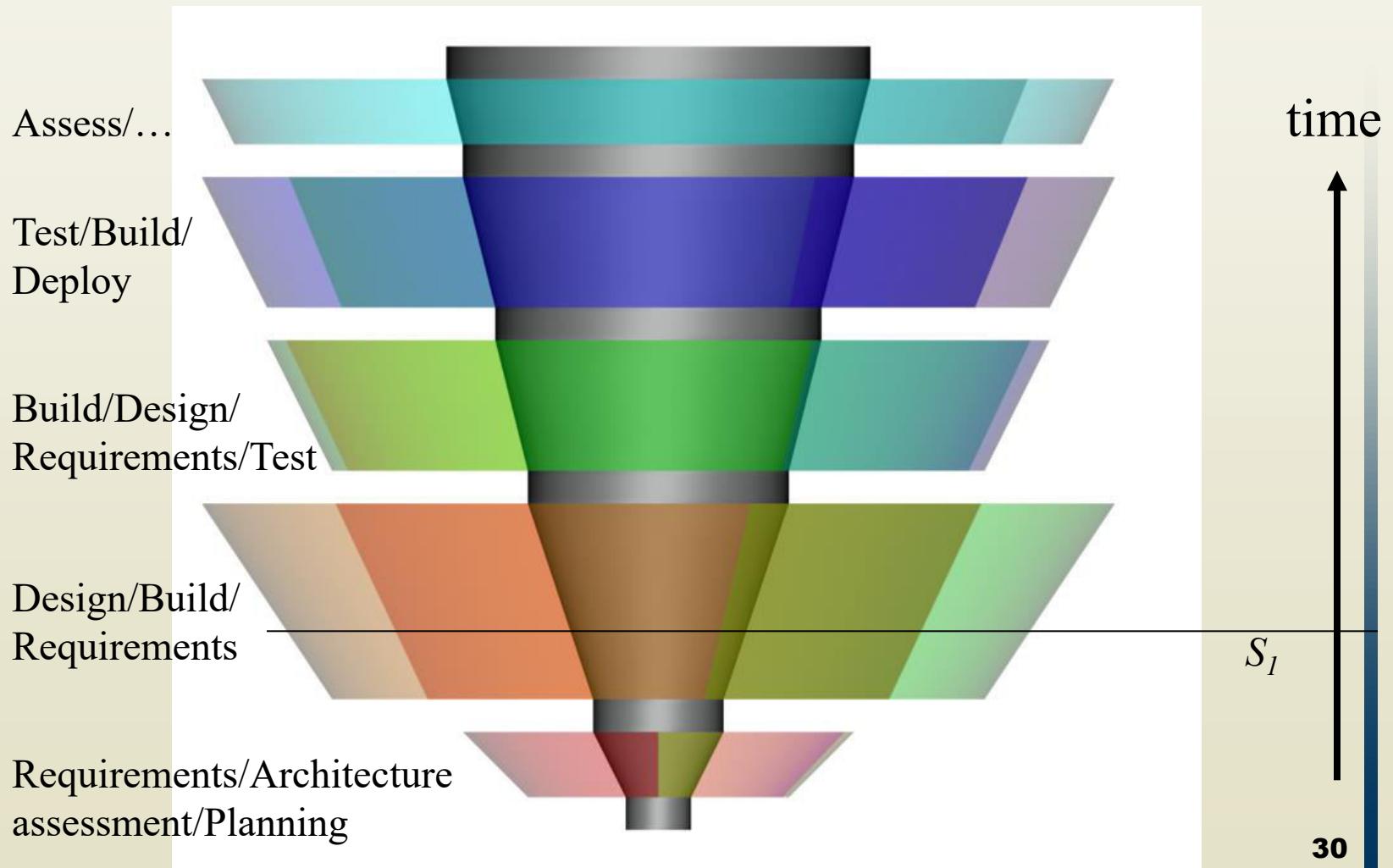
Cross-section at time t_i



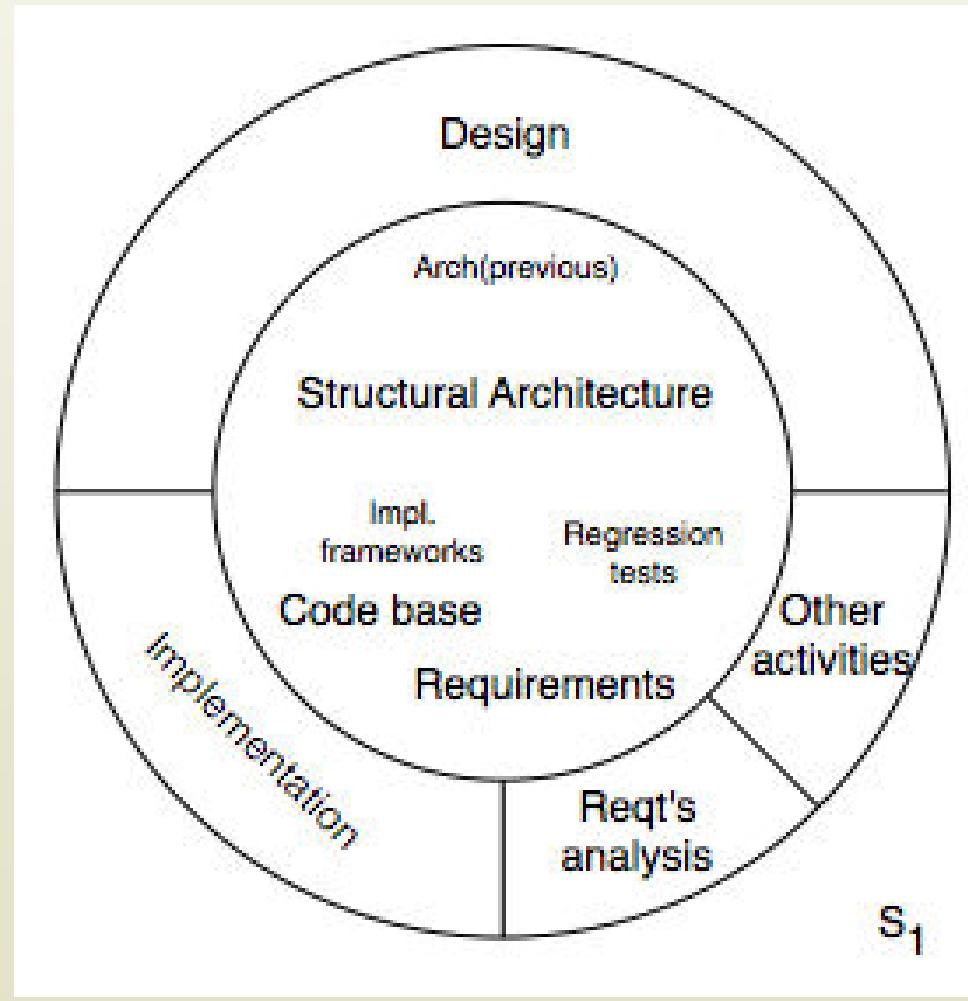
The Turbine Model



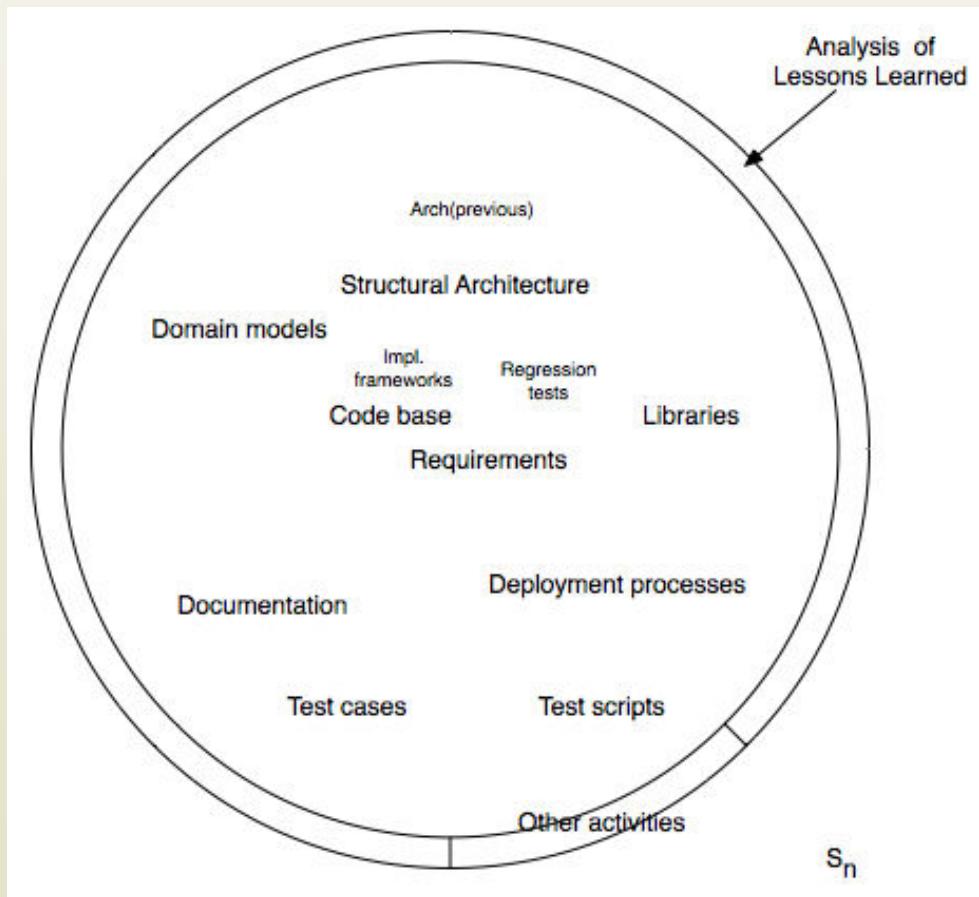
A Richer Example



A Sample Cross-Section



A Cross-Section at Project End



Volume Indicates Where Time was Spent

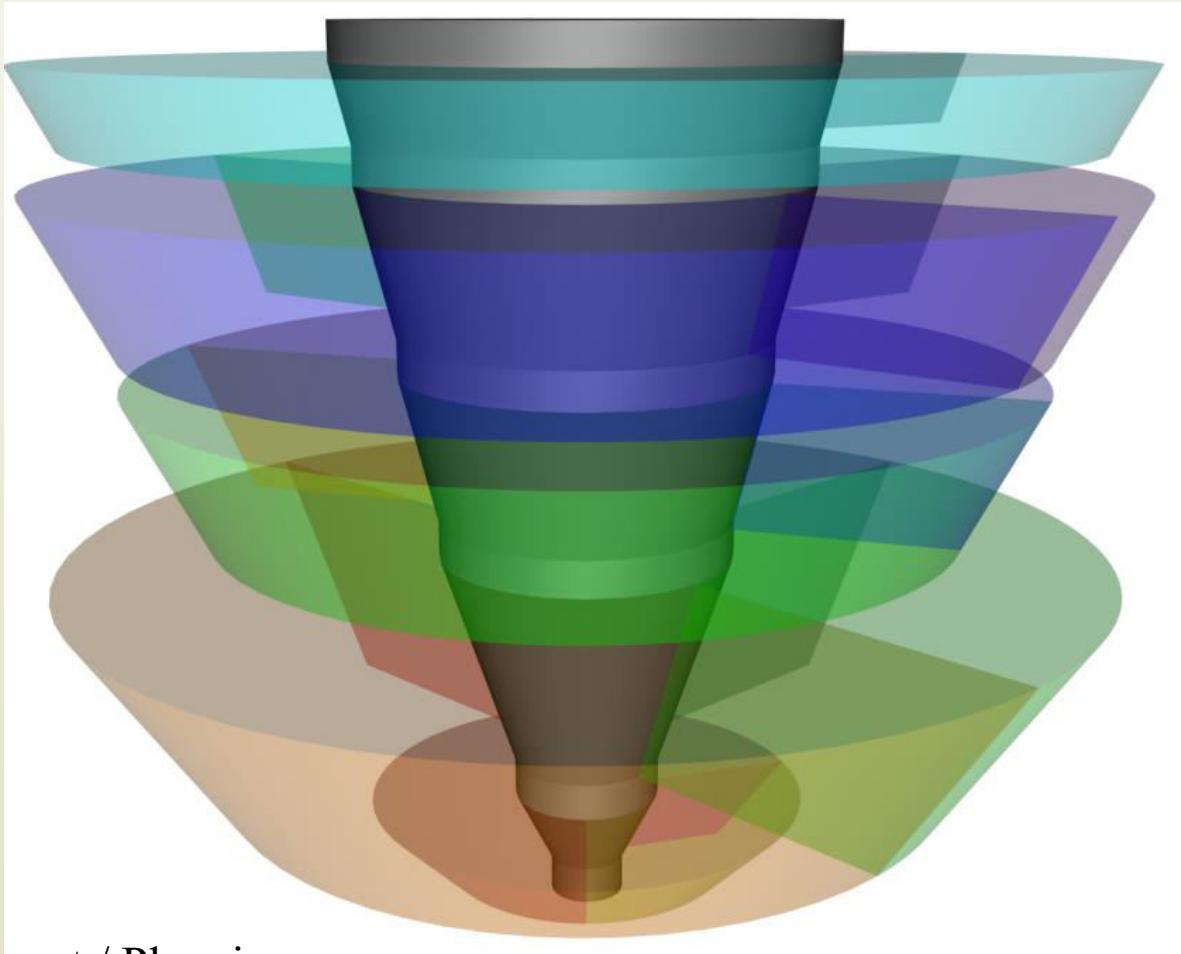
Assess/...

Test/Build/
Deploy

Build/Design/
Requirements/Test

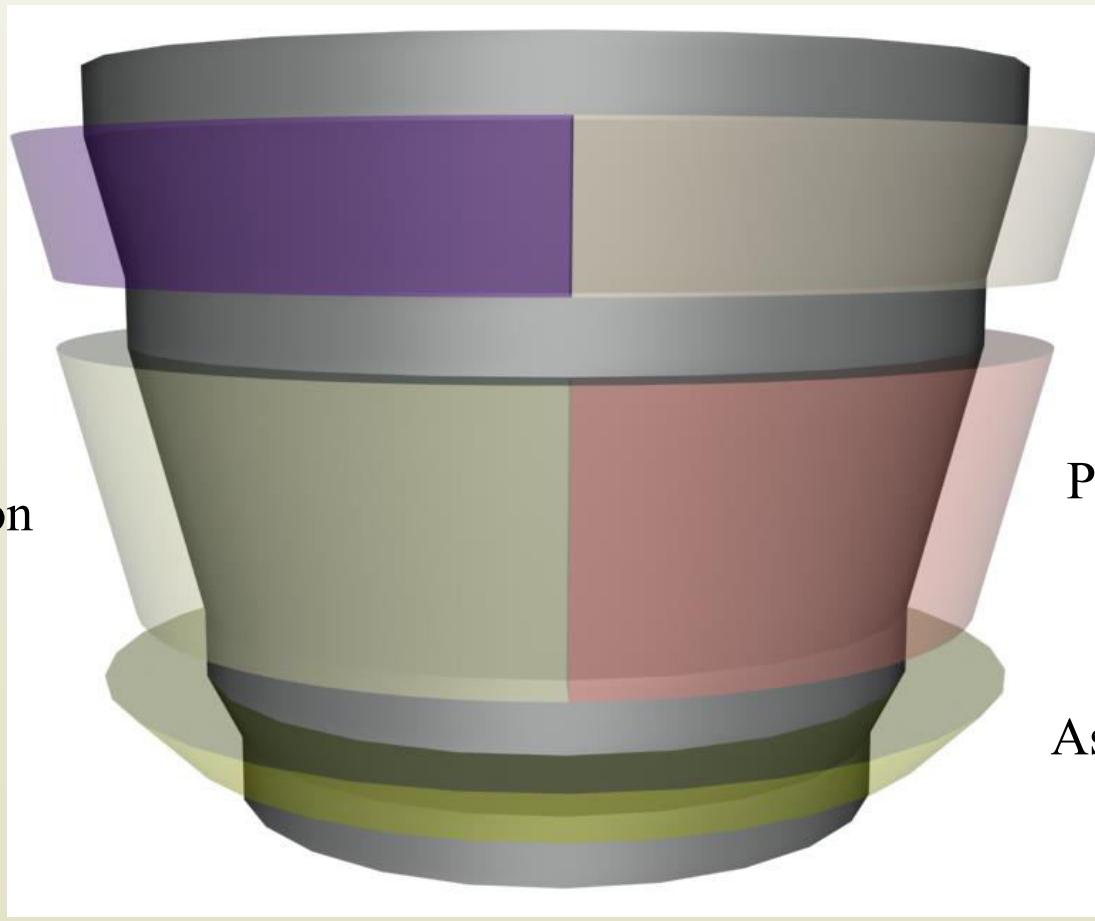
Design/Build/
Requirements

Requirements/
Architecture Assessment / Planning



A Technically Strong Product-Line Project

Customization



Deployment
Capture of new work
Other

Parameterization

Assessment

Visualization Summary

- It is illustrative, not prescriptive
- It is an aid to thinking about what's going on in a project
- Can be automatically generated based on input of monitored project data
- Can be extended to illustrate development of the information space (artifacts)
 - ◆ The preceding slides have focused primarily on the development activities

Processes Possible in this Model

- Traditional, straight-line waterfall
- Architecture-centric development
- DSSA-based project
- Agile development
- Dysfunctional process

Summary (1)

- A proper view of software architecture affects every aspect of the classical software engineering activities
- The requirements activity is a co-equal partner with design activities
- The design activity is enriched by techniques that exploit knowledge gained in previous product developments
- The implementation activity
 - ◆ is centered on creating a faithful implementation of the architecture
 - ◆ utilizes a variety of techniques to achieve this in a cost-effective manner

Summary (2)

- Analysis and testing activities can be focused on and guided by the architecture
- Evolution activities revolve around the product's architecture.
- An equal focus on process and product results from a proper understanding of the role of software architecture

Basic Concepts

Software Architecture
Lecture 3

What is Software Architecture?

- **Definition:**
 - ◆ A software system's architecture is the set of *principal design decisions* about the system
- Software architecture is the blueprint for a software system's construction and evolution
- Design decisions encompass every facet of the system under development
 - ◆ Structure
 - ◆ Behavior
 - ◆ Interaction
 - ◆ Non-functional properties

What is “Principal”?

- “Principal” implies a degree of importance that grants a design decision “architectural status”
 - ◆ It implies that not all design decisions are architectural
 - ◆ That is, they do not necessarily impact a system’s architecture
- How one defines “principal” will depend on what the stakeholders define as the system goals

Other Definitions of Software Architecture

- Perry and Wolf
 - ◆ Software Architecture = { Elements, Form, Rationale }
what how why
- Shaw and Garlan
 - ◆ Software architecture [is a level of design that] involves
 - the description of elements from which systems are built,
 - interactions among those elements,
 - patterns that guide their composition, and
 - constraints on these patterns.
- Kruchten
 - ◆ Software architecture deals with the design and implementation of the high-level structure of software.
 - ◆ Architecture deals with abstraction, decomposition, composition, style, and *aesthetics*.

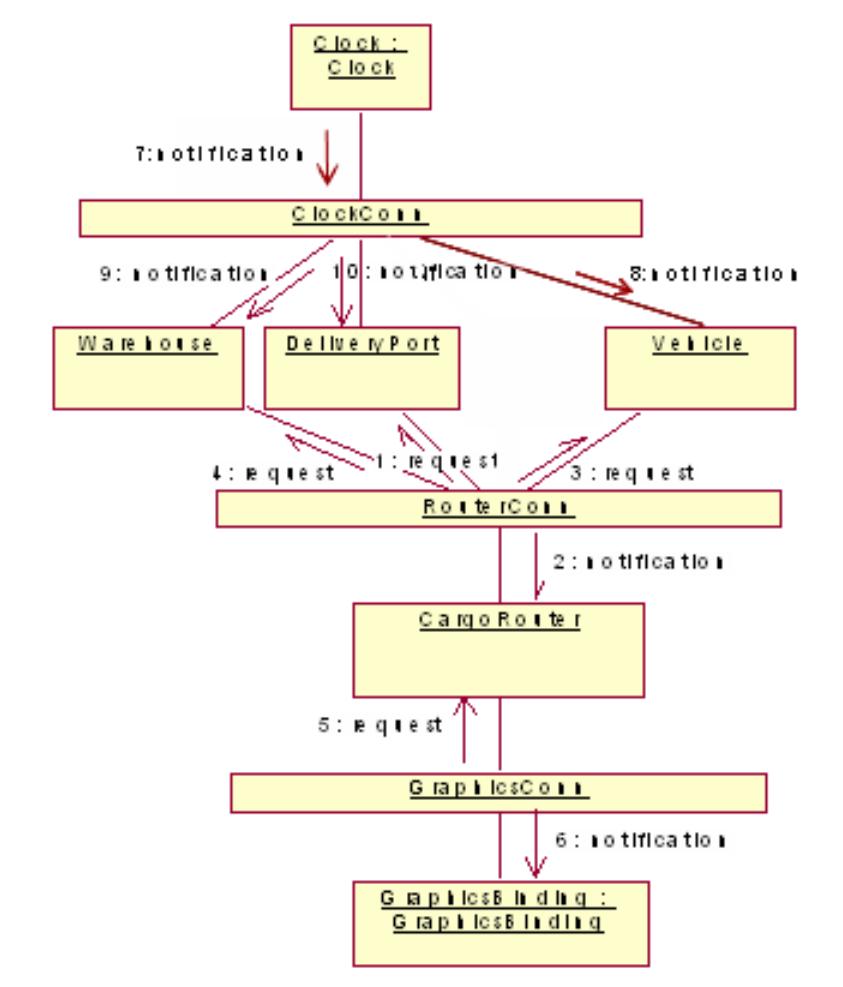
Temporal Aspect

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

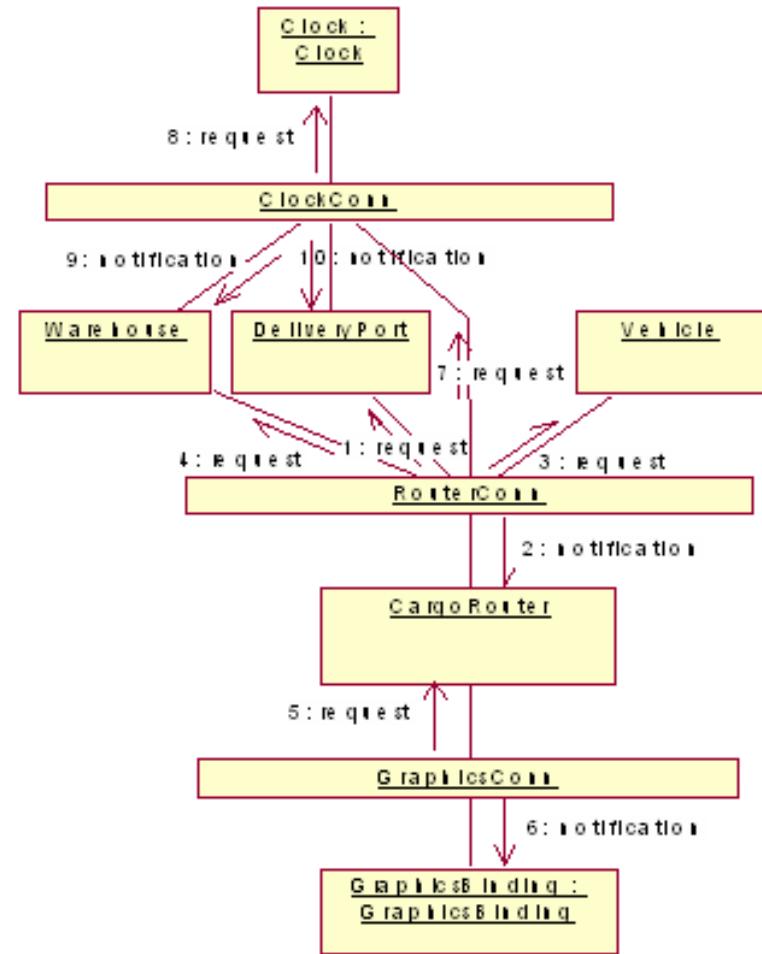
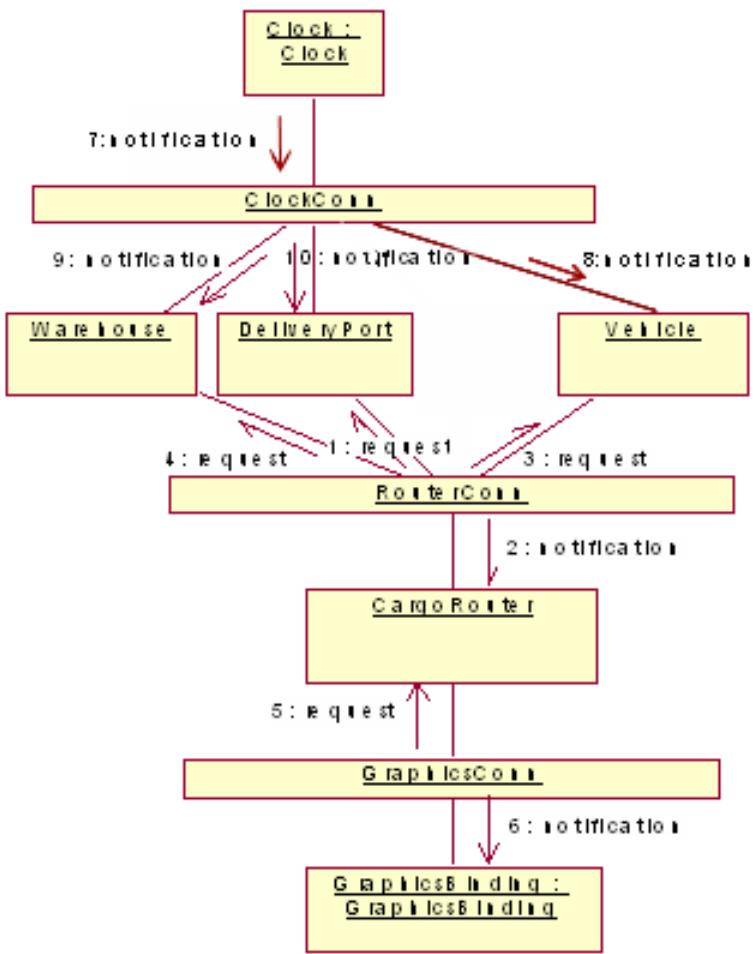
Prescriptive vs. Descriptive Architecture

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

As-Designed vs. As-Implemented Architecture

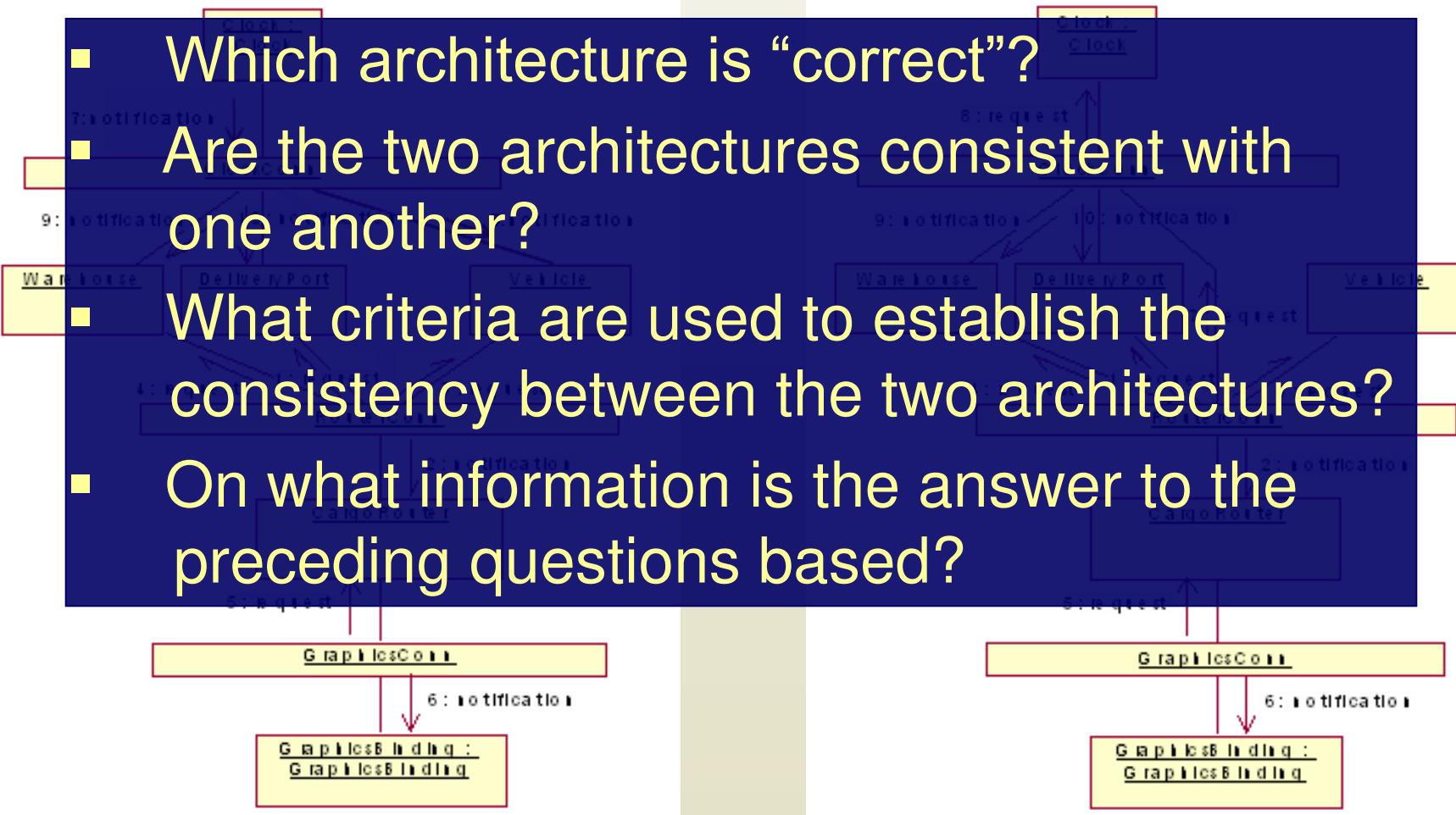


As-Designed vs. As-Implemented Architecture



As-Designed vs. As-Implemented Architecture

- Which architecture is “correct”?
- Are the two architectures consistent with one another?
- What criteria are used to establish the consistency between the two architectures?
- On what information is the answer to the preceding questions based?



Architectural Evolution

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

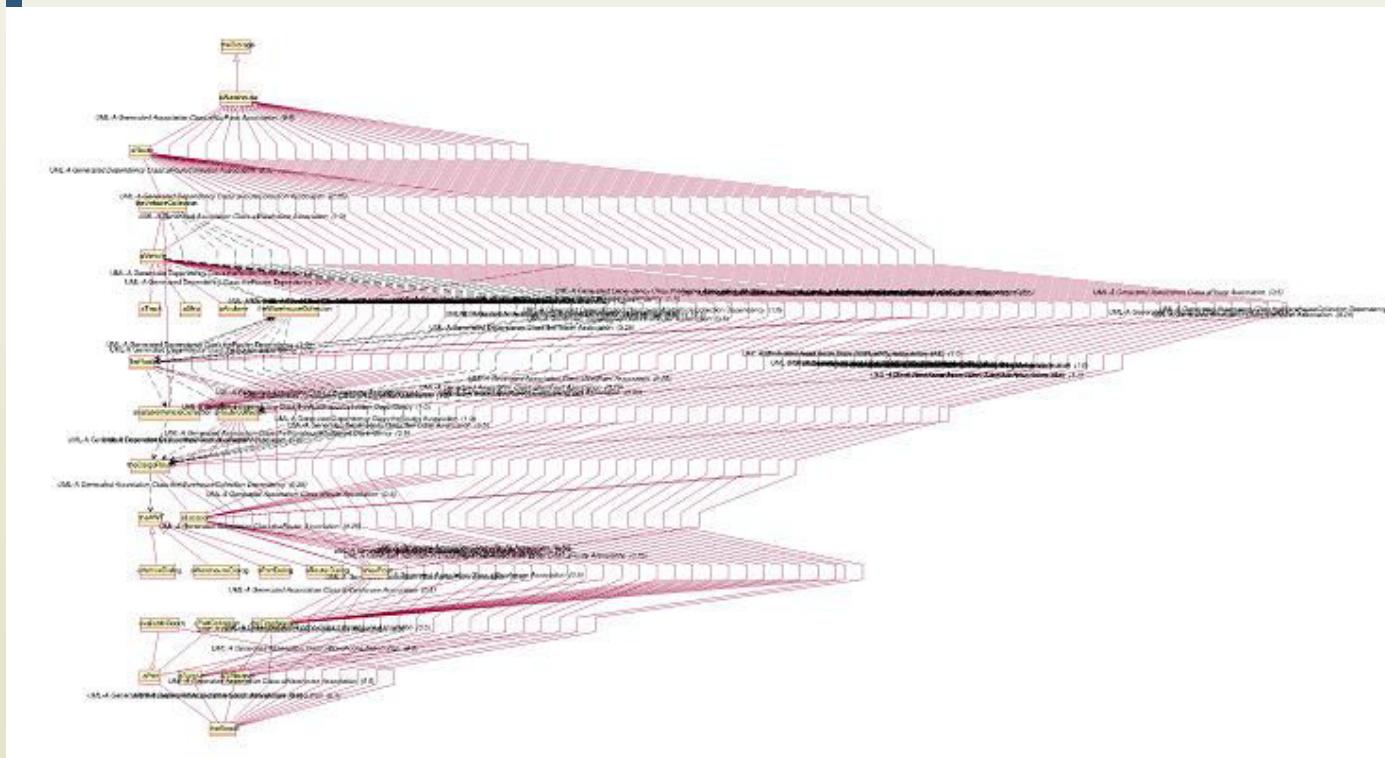
Architectural Degradation

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

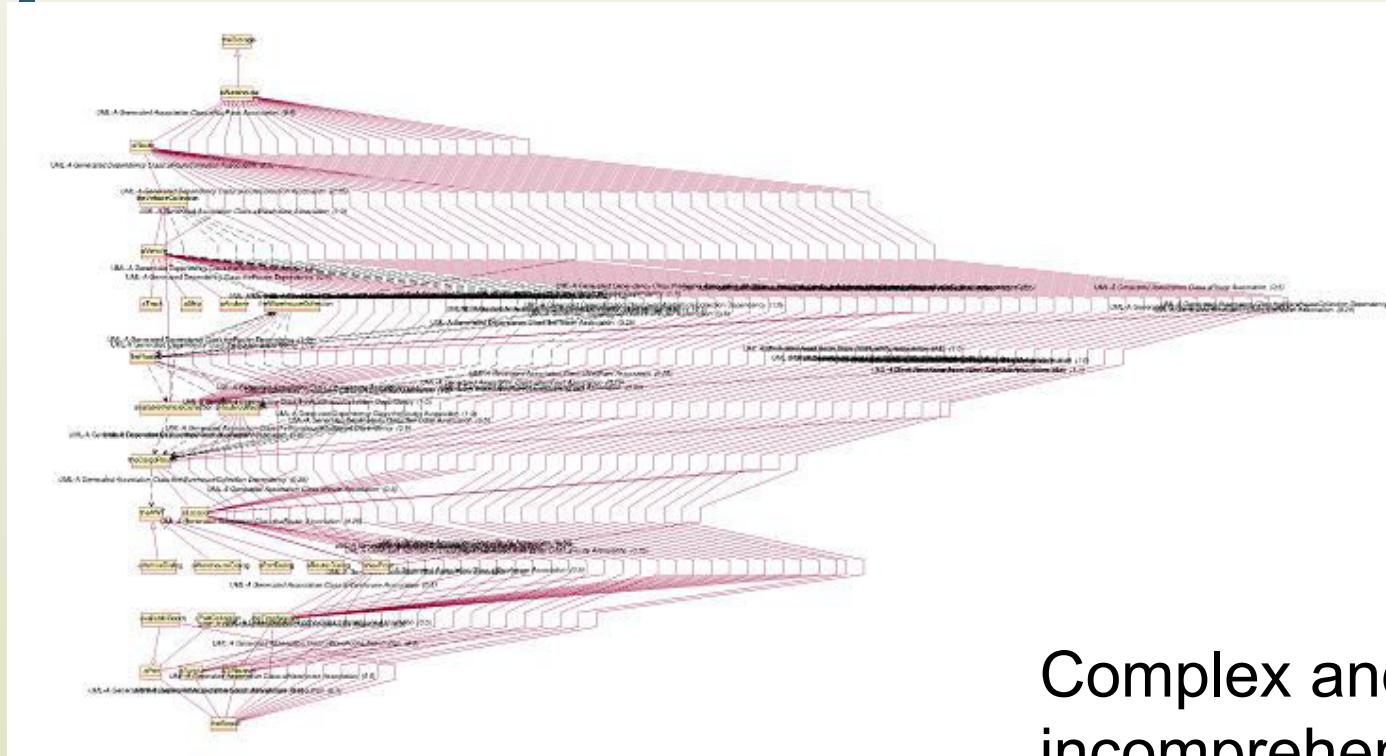
Architectural Recovery

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

Implementation-Level View of an Application



Implementation-Level View of an Application

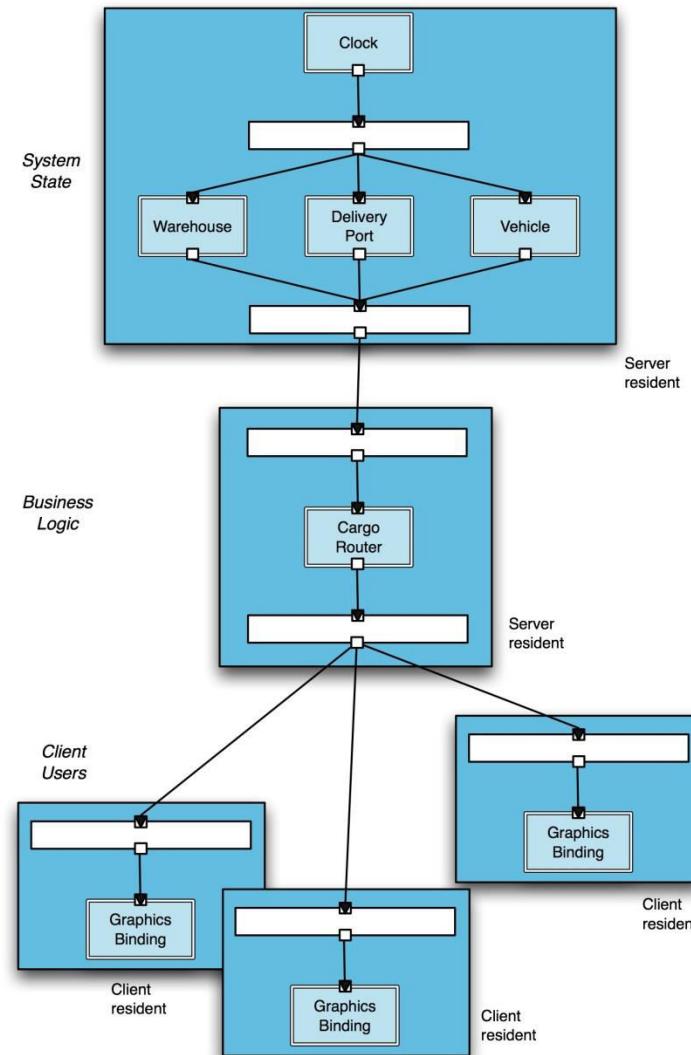


Complex and virtually
incomprehensible!

Deployment

- A software system cannot fulfill its purpose until it is *deployed*
 - ◆ Executable modules are physically placed on the hardware devices on which they are supposed to run
- The deployment view of an architecture can be critical in assessing whether the system will be able to satisfy its requirements
- Possible assessment dimensions
 - ◆ Available memory
 - ◆ Power consumption
 - ◆ Required network bandwidth

A System's Deployment Architectural Perspective



Software Architecture's Elements

- A software system's architecture typically is not (and should not be) a uniform monolith
- A software system's architecture should be a composition and interplay of different elements
 - ◆ Processing
 - ◆ Data, also referred as information or state
 - ◆ Interaction

Components

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

Connectors

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

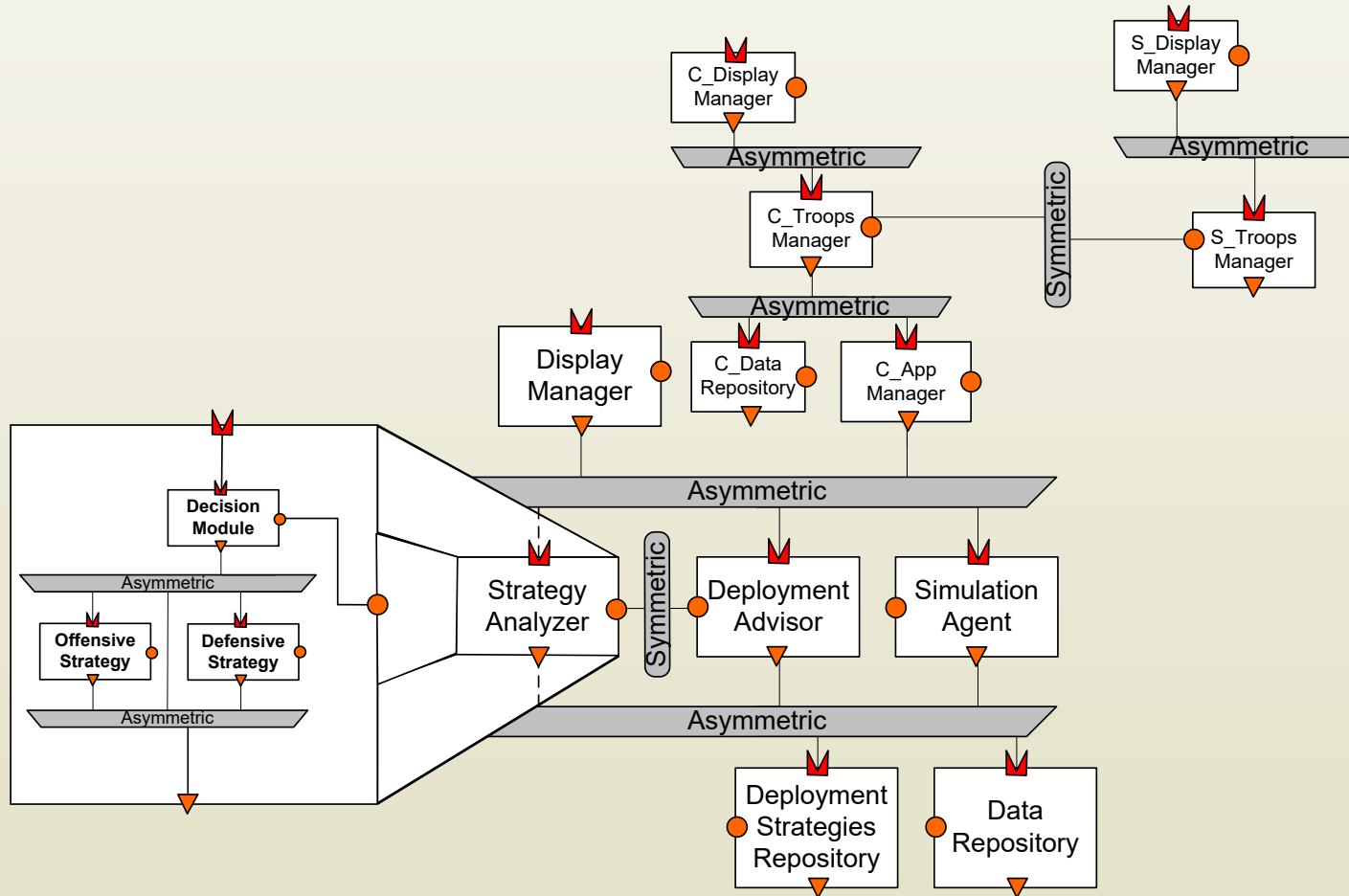
Examples of Connectors

- Procedure call connectors
- Shared memory connectors
- Message passing connectors
- Streaming connectors
- Distribution connectors
- Wrapper/adaptor connectors

Configurations

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

An Example Configuration



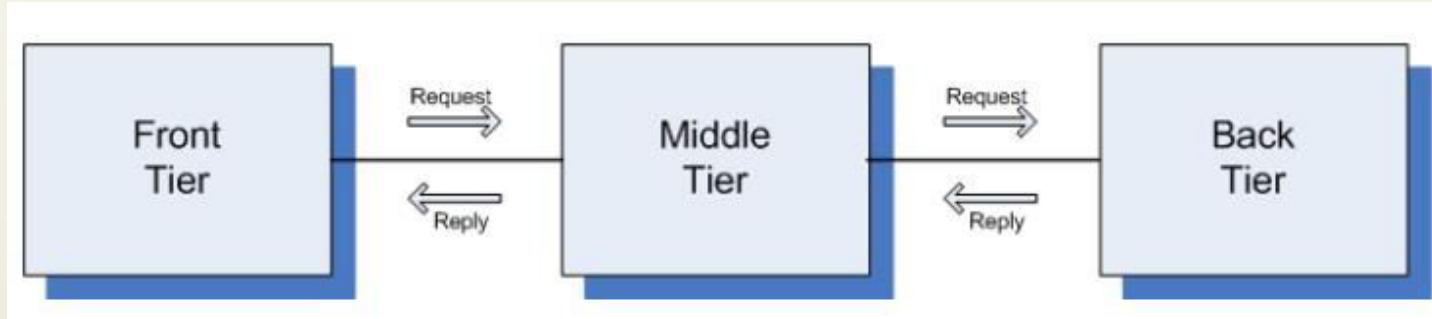
Architectural Styles

- Certain design choices regularly result in solutions with superior properties
 - ◆ Compared to other possible alternatives, solutions such as this are more elegant, effective, efficient, dependable, evolvable, scalable, and so on
- **Definition**
 - ◆ An *architectural style* is a named collection of architectural design decisions that
 - are applicable in a given development context
 - constrain architectural design decisions that are specific to a particular system within that context
 - elicit beneficial qualities in each resulting system

Architectural Patterns

- **Definition**
 - ◆ An *architectural pattern* is a set of architectural design decisions that are applicable to a recurring design problem, and parameterized to account for different software development contexts in which that problem appears
- A widely used pattern in modern distributed systems is the *three-tiered system* pattern
 - ◆ Science
 - ◆ Banking
 - ◆ E-commerce
 - ◆ Reservation systems

Three-Tiered Pattern



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

Architectural Models, Views, and Visualizations

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

Architectural Processes

- Architectural design
- Architecture modeling and visualization
- Architecture-driven system analysis
- Architecture-driven system implementation
- Architecture-driven system deployment, runtime redeployment, and mobility
- Architecture-based design for non-functional properties, including security and trust
- architectural adaptation

Stakeholders in a System's Architecture

- Architects
- Developers
- Testers
- Managers
- Customers
- Users
- Vendors

Designing Architectures

**Software Architecture
Lecture 4**

How Do You Design?

Where do architectures come from?

Creativity

- 1) Fun!
- 2) Fraught with peril
- 3) May be unnecessary
- 4) May yield the best

- 1) Efficient in familiar terrain
- 2) Not always successful
- 3) Predictable outcome (+ & -)
- 4) Quality of methods varies

Method

Objectives

- Creativity
 - ◆ Enhance your skillset
 - ◆ Provide new tools
- Method
 - ◆ Focus on highly effective techniques
- Develop judgment: when to develop novel solutions, and when to follow established method

Engineering Design Process

- Feasibility stage: identifying a set of feasible concepts for the design as a whole
- Preliminary design stage: selection and development of the best concept.
- Detailed design stage: development of engineering descriptions of the concept.
- Planning stage: evaluating and altering the concept to suit the requirements of production, distribution, consumption and product retirement.

Potential Problems

- If the designer is unable to produce a set of feasible concepts, progress stops.
- As problems and products increase in size and complexity, the probability that any one individual can successfully perform the first steps decreases.
- The standard approach does not directly address the situation where system design is at stake, i.e. when relationship between a set of products is at issue.
- → As complexity increases or the experience of the designer is not sufficient, alternative approaches to the design process must be adopted.

Alternative Design Strategies

- Standard
 - ◆ Linear model described above
- Cyclic
 - ◆ Process can revert to an earlier stage
- Parallel
 - ◆ Independent alternatives are explored in parallel
- Adaptive ("lay tracks as you go")
 - ◆ The next design strategy of the design activity is decided at the end of a given stage
- Incremental
 - ◆ Each stage of development is treated as a task of incrementally improving the existing design

Identifying a Viable Strategy

- Use fundamental design tools: abstraction and modularity.
 - ◆ *But how?*
- Inspiration, where inspiration is needed. Predictable techniques elsewhere.
 - ◆ *But where is creativity required?*
- Applying own experience or experience of others.

The Tools of “Software Engineering 101”

- Abstraction
 - ◆ Abstraction(1): look at details, and abstract “up” to concepts
 - ◆ Abstraction(2): choose concepts, then add detailed substructure, and move “down”
 - Example: design of a stack class
- Separation of concerns

A Few Definitions... from the *OED Online*

- Abstraction: “The act or process of separating in thought, of considering a thing independently of its associations; or a substance independently of its attributes; or an attribute or quality independently of the substance to which it belongs.”
- Reification: “The mental conversion of ... [an] abstract concept into a thing.”
- Deduction: “The process of drawing a conclusion from a principle already known or assumed; spec. in Logic, inference by reasoning from generals to particulars; opposed to INDUCTION.”
- Induction: “The process of inferring a general law or principle from the observation of particular instances (opposed to DEDUCTION, q.v.).”

Abstraction and the Simple Machines

- What concepts should be chosen at the outset of a design task?
 - ◆ One technique: Search for a “simple machine” that serves as an abstraction of a potential system that will perform the required task
 - ◆ For instance, what kind of simple machine makes a software system embedded in a fax machine?
 - At core, it is basically just a little state machine.
- Simple machines provide a plausible first conception of how an application might be built.
- Every application domain has its common simple machines.

Simple Machines

Domain	Simple Machines
Graphics	Pixel arrays Transformation matrices Widgets Abstract depiction graphs
Word processing	Structured documents Layouts
Process control	Finite state machines
Income Tax Software	Hypertext Spreadsheets Form templates
Web pages	Hypertext Composite documents
Scientific computing	Matrices Mathematical functions
Banking	Spreadsheets Databases Transactions

Choosing the Level and Terms of Discourse

- Any attempt to use abstraction as a tool must choose a level of discourse, and once that is chosen, must choose the terms of discourse.
- *Alternative 1*: initial level of discourse is one of the application as a whole (step-wise refinement).
- *Alternative 2*: work, initially, at a level lower than that of the whole application.
 - ◆ Once several such sub-problems are solved they can be composed together to form an overall solution
- *Alternative 3*: work, initially, at a level above that of the desired application.
 - ◆ E.g. handling simple application input with a general parser.

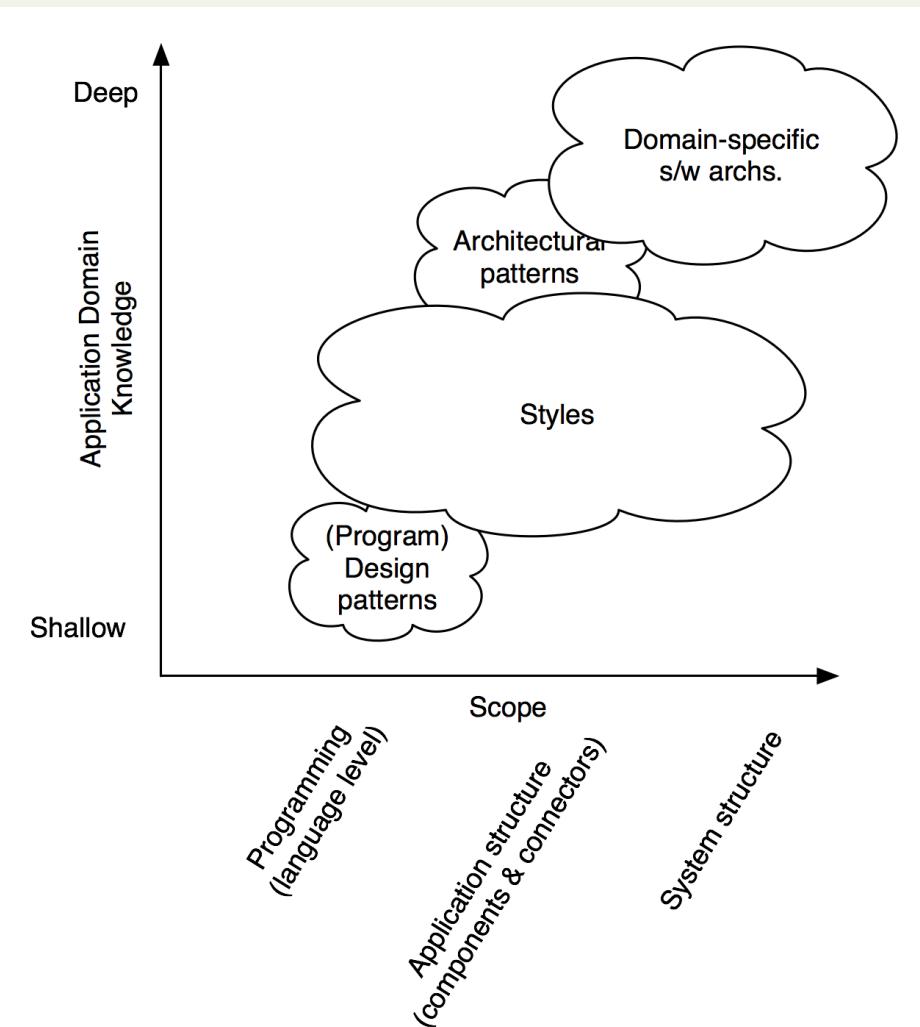
Separation of Concerns

- Separation of concerns is the subdivision of a problem into (hopefully) independent parts.
- The difficulties arise when the issues are either actually or apparently intertwined.
- Separations of concerns frequently involves many tradeoffs
- Total independence of concepts may not be possible.
- Key example from software architecture: separation of components (computation) from connectors (communication)

The Grand Tool: Refined Experience

- Experience must be reflected upon and refined.
- The lessons from prior work include not only the lessons of successes, but also the lessons arising from failure.
- Learn from success and failure of other engineers
 - ◆ Literature
 - ◆ Conferences
- Experience can provide that initial feasible set of “alternative arrangements for the design as a whole”.

Patterns, Styles, and DSSAs



Domain-Specific Software Architectures

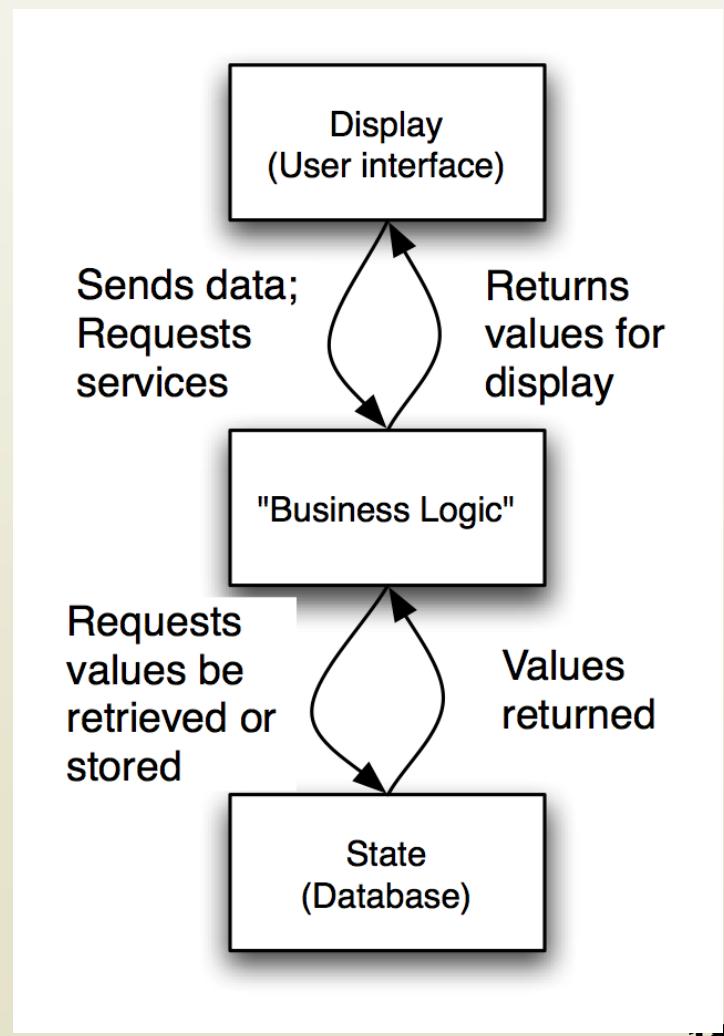
- A DSSA is an assemblage of software components
 - ◆ specialized for a particular type of task (domain),
 - ◆ generalized for effective use across that domain, and
 - ◆ composed in a standardized structure (topology) effective for building successful applications.
- Since DSSAs are specialized for a particular domain they are only of value if one exists for the domain wherein the engineer is tasked with building a new application.
- DSSAs are the pre-eminent means for maximal reuse of knowledge and prior development and hence for developing a new architectural design.

Architectural Patterns

- An architectural pattern is a set of architectural design decisions that are applicable to a recurring design problem, and parameterized to account for different software development contexts in which that problem appears.
- Architectural patterns are similar to DSSAs but applied “at a lower level” and within a much narrower scope.

State-Logic-Display: Three-Tiered Pattern

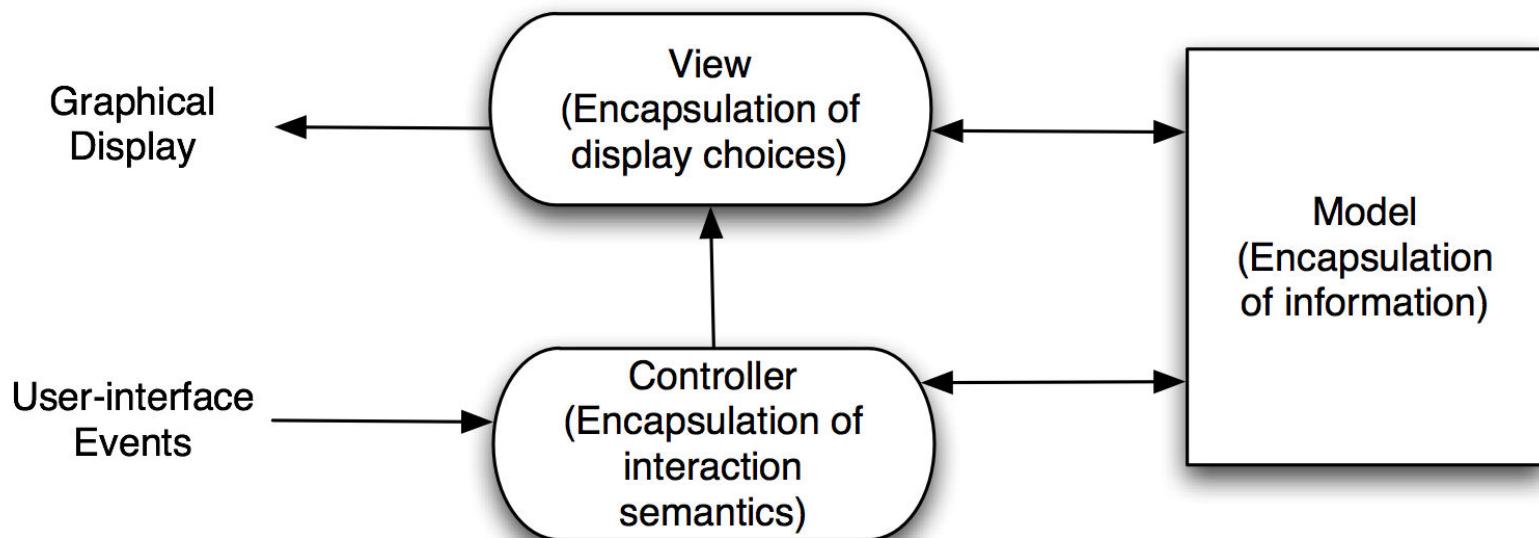
- Application Examples
 - Business applications
 - Multi-player games
 - Web-based applications



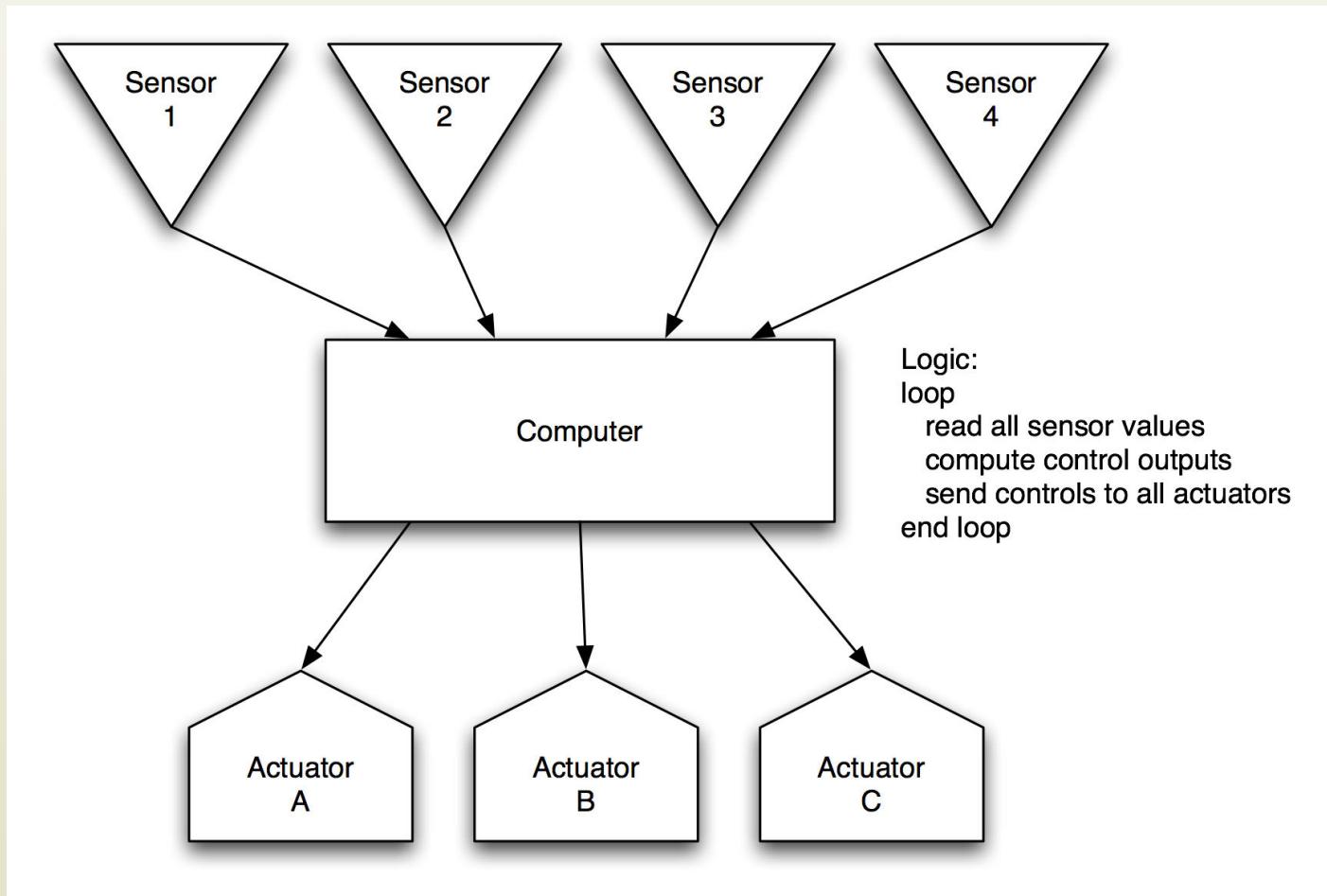
Model-View-Controller (MVC)

- Objective: Separation between information, presentation and user interaction.
- When a model object value changes, a notification is sent to the view and to the controller. So that the view can update itself and the controller can modify the view if its logic so requires.
- When handling input from the user the windowing system sends the user event to the controller; If a change is required, the controller updates the model object.

Model-View-Controller



Sense-Compute-Control

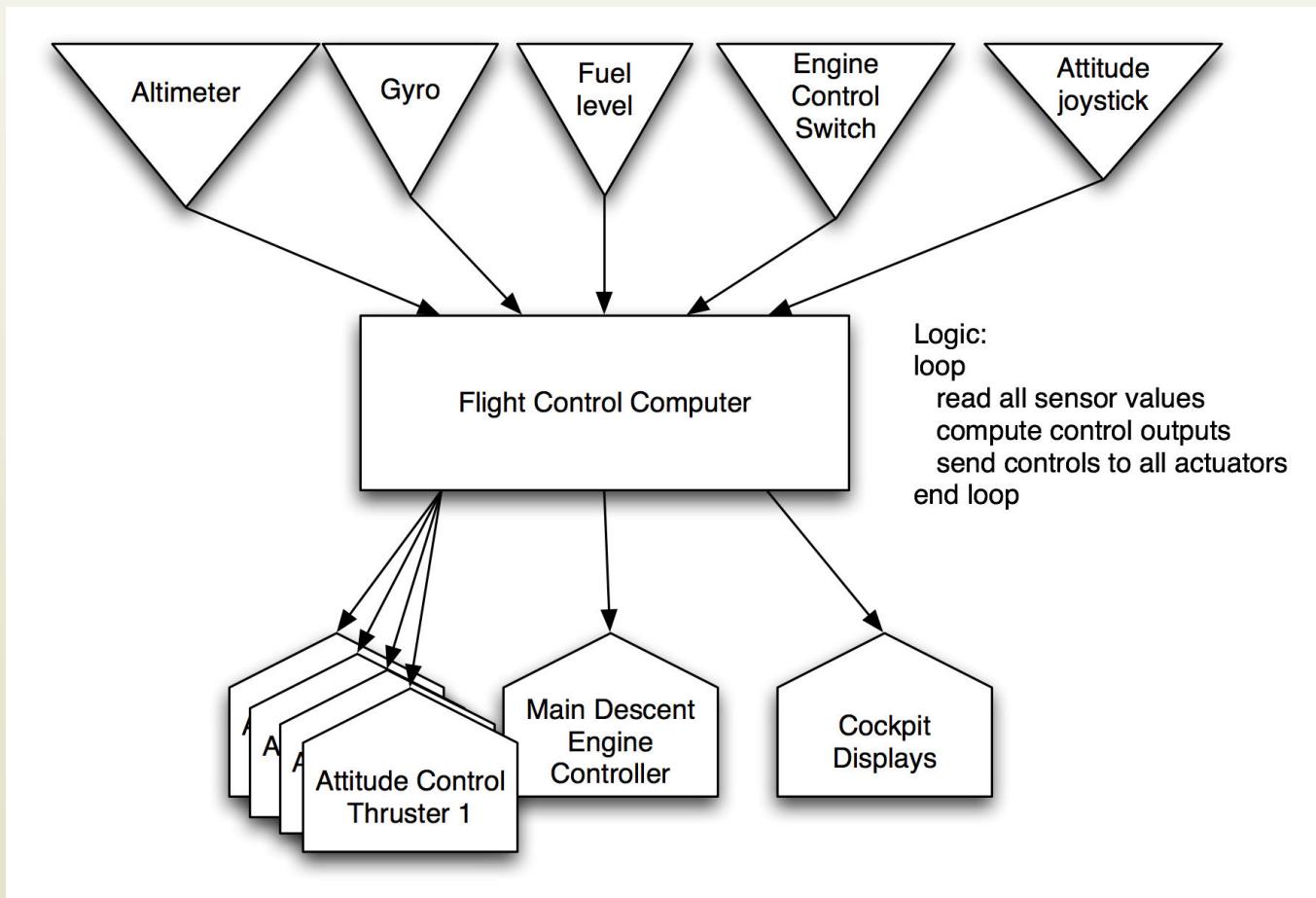


Objective: Structuring embedded control applications

The Lunar Lander: A Long-Running Example

- A simple computer game that first appeared in the 1960's
- Simple concept:
 - ◆ You (the pilot) control the descent rate of the Apollo-era Lunar Lander
 - Throttle setting controls descent engine
 - Limited fuel
 - Initial altitude and speed preset
 - If you land with a descent rate of < 5 fps: you win (whether there's fuel left or not)
 - ◆ "Advanced" version: joystick controls attitude & horizontal motion

Sense-Compute-Control LL



Architectural Styles

- An architectural style is a named collection of architectural design decisions that
 - are applicable in a given development context
 - constrain architectural design decisions that are specific to a particular system within that context
 - elicit beneficial qualities in each resulting system
- A primary way of characterizing lessons from experience in software system design
- Reflect less domain specificity than architectural patterns
- Useful in determining everything from subroutine structure to top-level application structure
- Many styles exist and we will discuss them in detail in the next lecture

Definitions of Architectural Style

- Definition. An architectural style is a named collection of architectural design decisions that
 - ◆ are applicable in a given development context
 - ◆ constrain architectural design decisions that are specific to a particular system within that context
 - ◆ elicit beneficial qualities in each resulting system.
- Recurring organizational patterns & idioms
 - ◆ Established, shared understanding of common design forms
 - ◆ Mark of mature engineering field.
 - Shaw & Garlan
- Abstraction of recurring composition & interaction characteristics in a set of architectures
 - Taylor

Basic Properties of Styles

- A vocabulary of design elements
 - ◆ Component and connector types; data elements
 - ◆ e.g., pipes, filters, objects, servers
- A set of configuration rules
 - ◆ Topological constraints that determine allowed compositions of elements
 - ◆ e.g., a component may be connected to at most two other components
- A semantic interpretation
 - ◆ Compositions of design elements have well-defined meanings
- Possible analyses of systems built in a style

Benefits of Using Styles

- Design reuse
 - ◆ Well-understood solutions applied to new problems
- Code reuse
 - ◆ Shared implementations of invariant aspects of a style
- Understandability of system organization
 - ◆ A phrase such as “client-server” conveys a lot of information
- Interoperability
 - ◆ Supported by style standardization
- Style-specific analyses
 - ◆ Enabled by the constrained design space
- Visualizations
 - ◆ Style-specific depictions matching engineers’ mental models

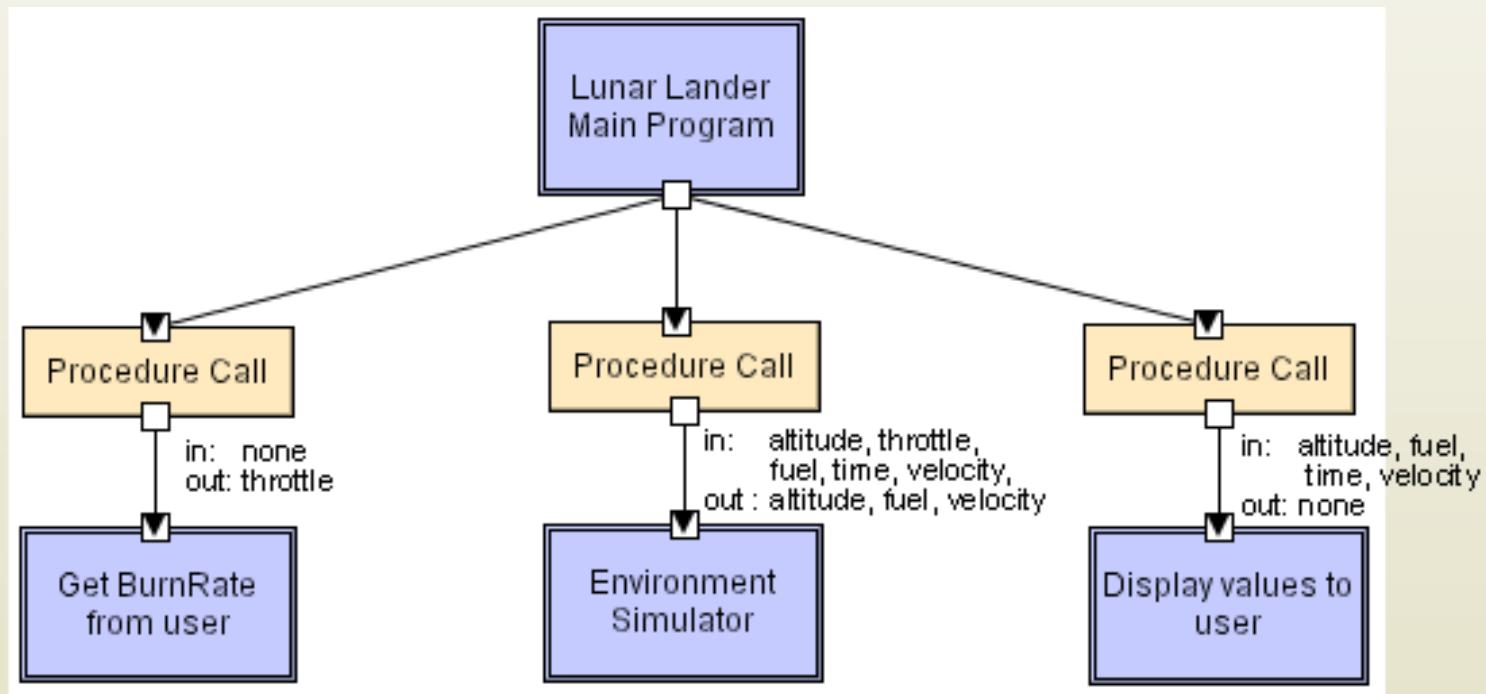
Style Analysis Dimensions

- What is the design vocabulary?
 - ◆ Component and connector types
- What are the allowable structural patterns?
- What is the underlying computational model?
- What are the essential invariants of the style?
- What are common examples of its use?
- What are the (dis)advantages of using the style?
- What are the style's specializations?

Some Common Styles

- Traditional, language-influenced styles
 - ◆ Main program and subroutines
 - ◆ Object-oriented
- Layered
 - ◆ Virtual machines
 - ◆ Client-server
- Data-flow styles
 - ◆ Batch sequential
 - ◆ Pipe and filter
- Shared memory
 - ◆ Blackboard
 - ◆ Rule based
- Interpreter
 - ◆ Interpreter
 - ◆ Mobile code
- Implicit invocation
 - ◆ Event-based
 - ◆ Publish-subscribe
- Peer-to-peer
- “Derived” styles
 - ◆ C2
 - ◆ CORBA

Main Program and Subroutines LL



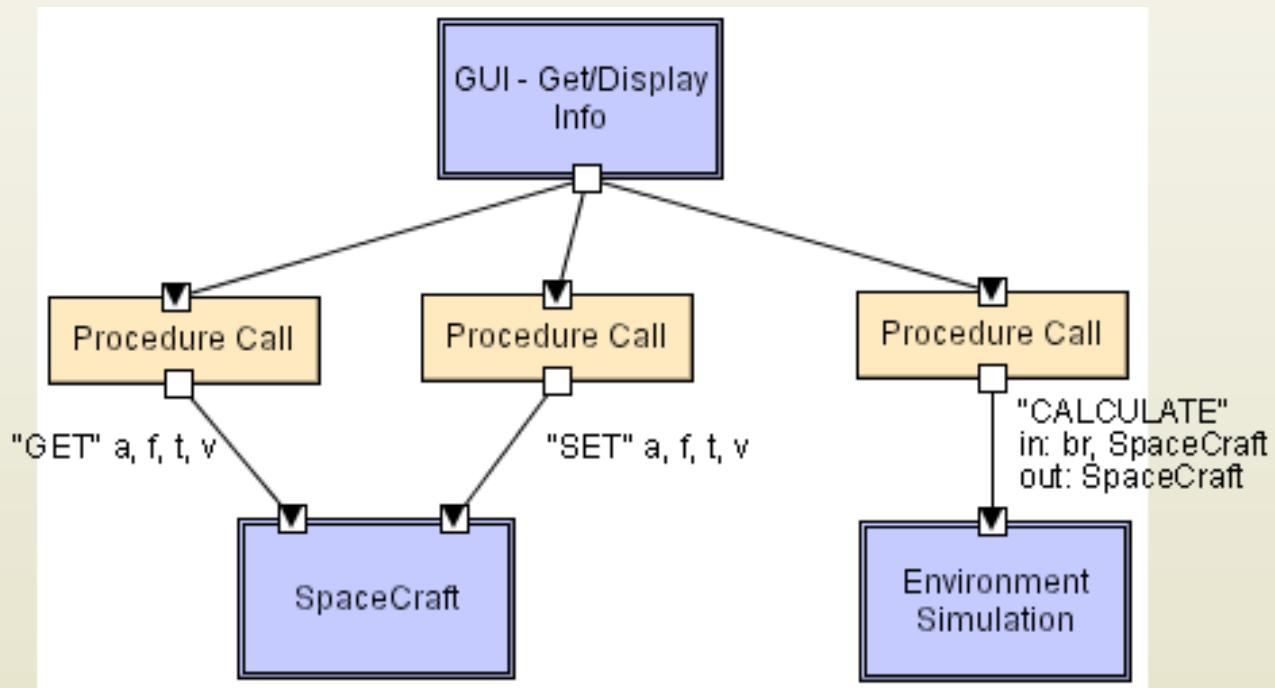
Architectural Styles

Software Architecture
Lecture 5

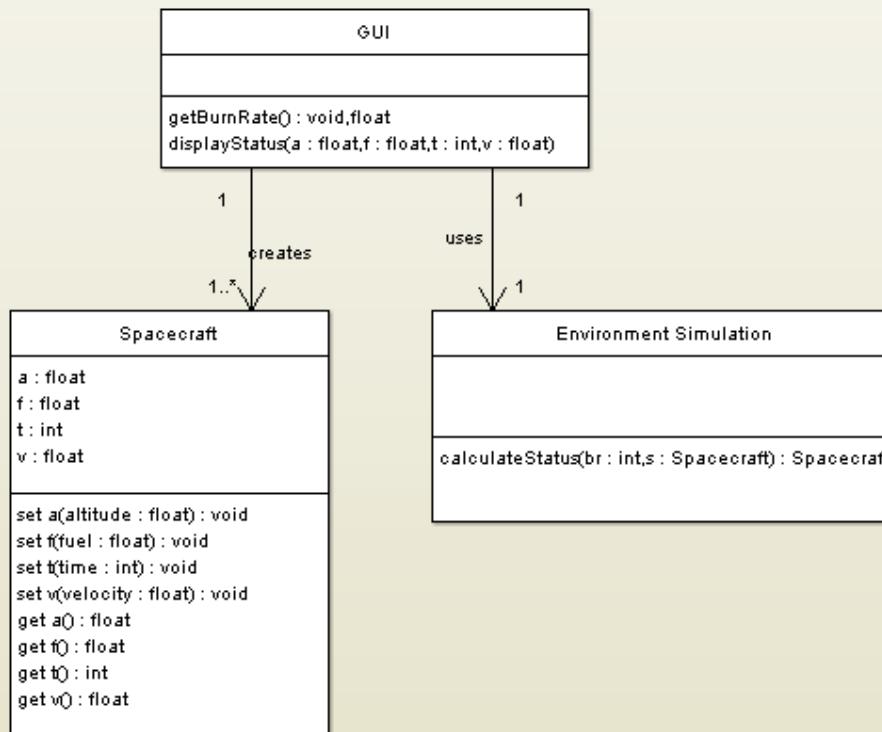
Object-Oriented Style

- Components are objects
 - ◆ Data and associated operations
- Connectors are messages and method invocations
- Style invariants
 - ◆ Objects are responsible for their internal representation integrity
 - ◆ Internal representation is hidden from other objects
- Advantages
 - ◆ “Infinite malleability” of object internals
 - ◆ System decomposition into sets of interacting agents
- Disadvantages
 - ◆ Objects must know identities of servers
 - ◆ Side effects in object method invocations

Object-Oriented LL



OO/LL in UML



Layered Style

- Hierarchical system organization
 - ◆ “Multi-level client-server”
 - ◆ Each layer exposes an interface (API) to be used by above layers
- Each layer acts as a
 - ◆ *Server*: service provider to layers “above”
 - ◆ *Client*: service consumer of layer(s) “below”
- Connectors are protocols of layer interaction
- Example: operating systems
- *Virtual machine* style results from fully opaque layers

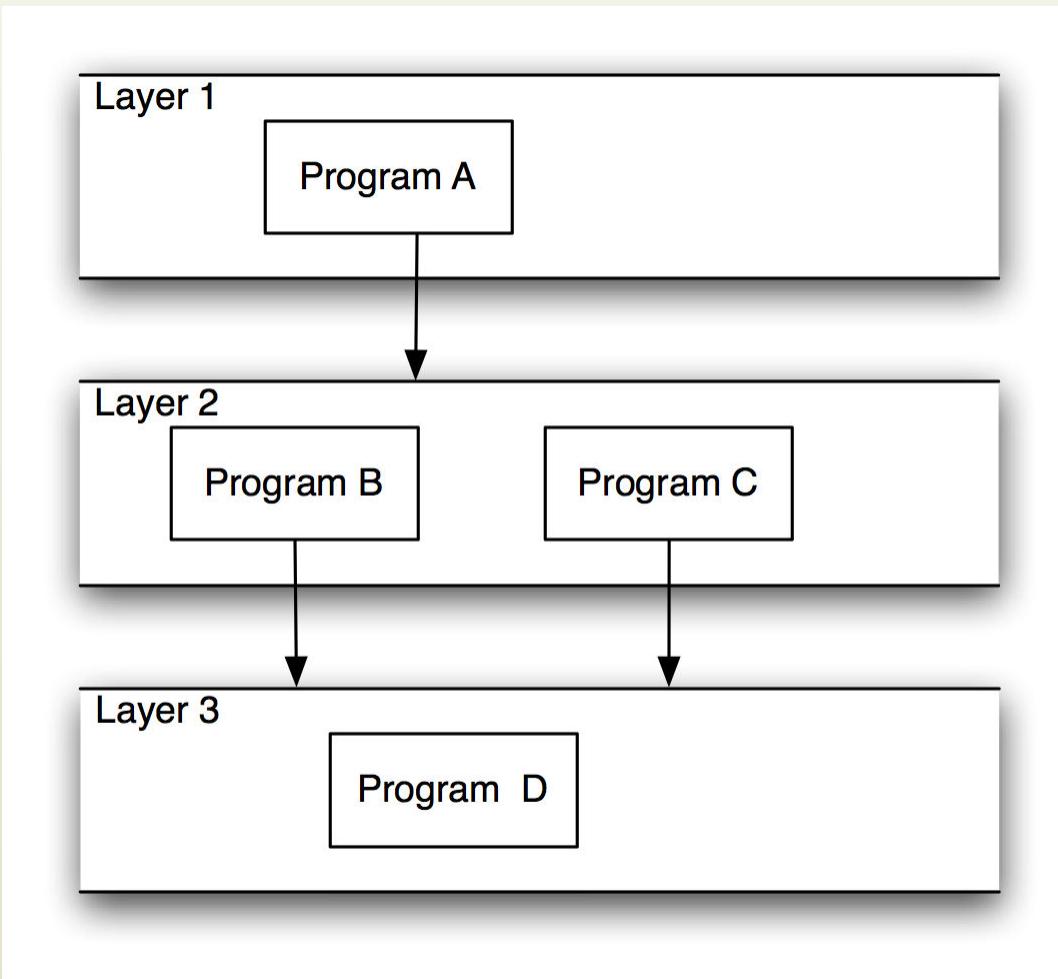
Layered Style (cont'd)

- Advantages
 - ◆ Increasing abstraction levels
 - ◆ Evolvability
 - ◆ Changes in a layer affect at most the adjacent two layers
 - Reuse
 - ◆ Different implementations of layer are allowed as long as interface is preserved
 - ◆ Standardized layer interfaces for libraries and frameworks

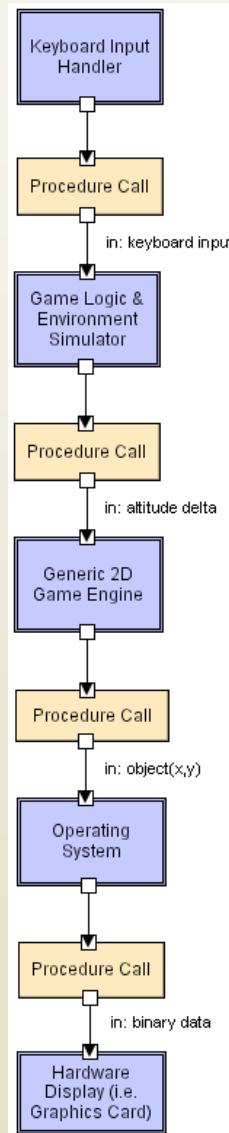
Layered Style (cont'd)

- Disadvantages
 - ◆ Not universally applicable
 - ◆ Performance
- Layers may have to be skipped
 - ◆ Determining the correct abstraction level

Layered Systems/Virtual Machines



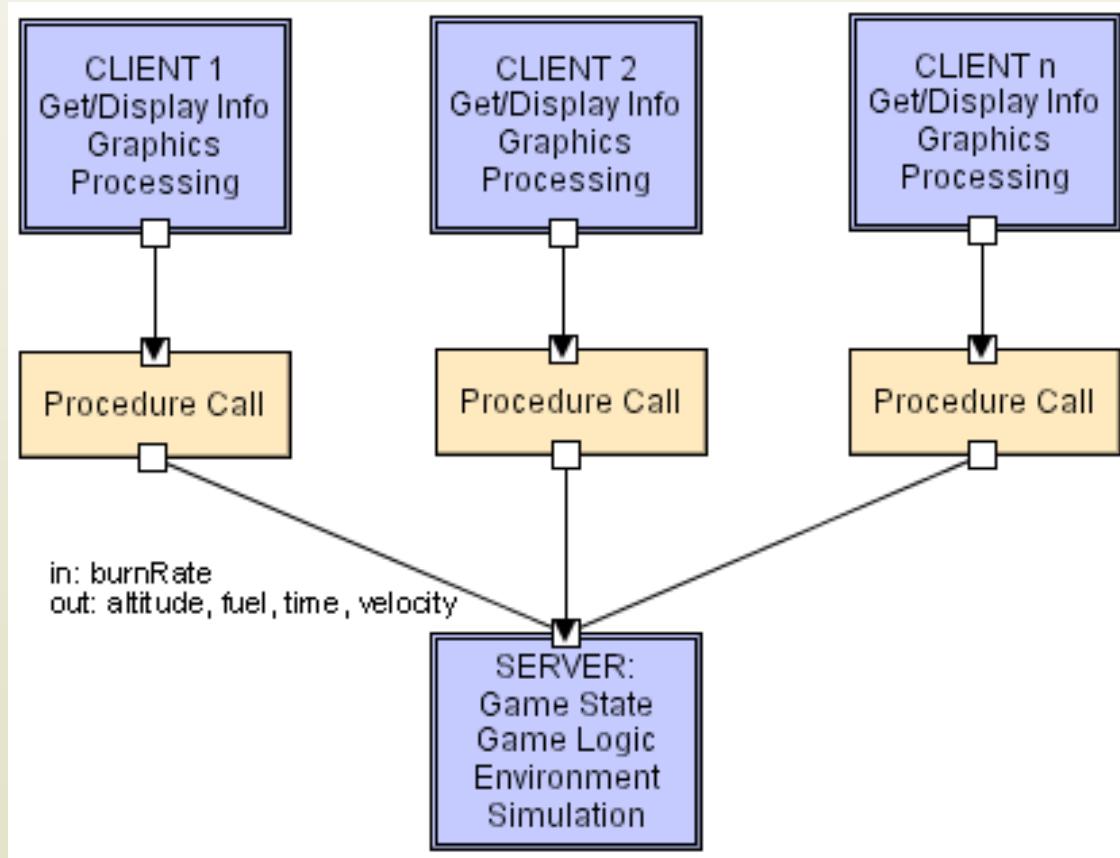
Layered LL



Client-Server Style

- Components are clients and servers
- Servers do not know number or identities of clients
- Clients know server's identity
- Connectors are RPC-based network interaction protocols

Client-Server LL

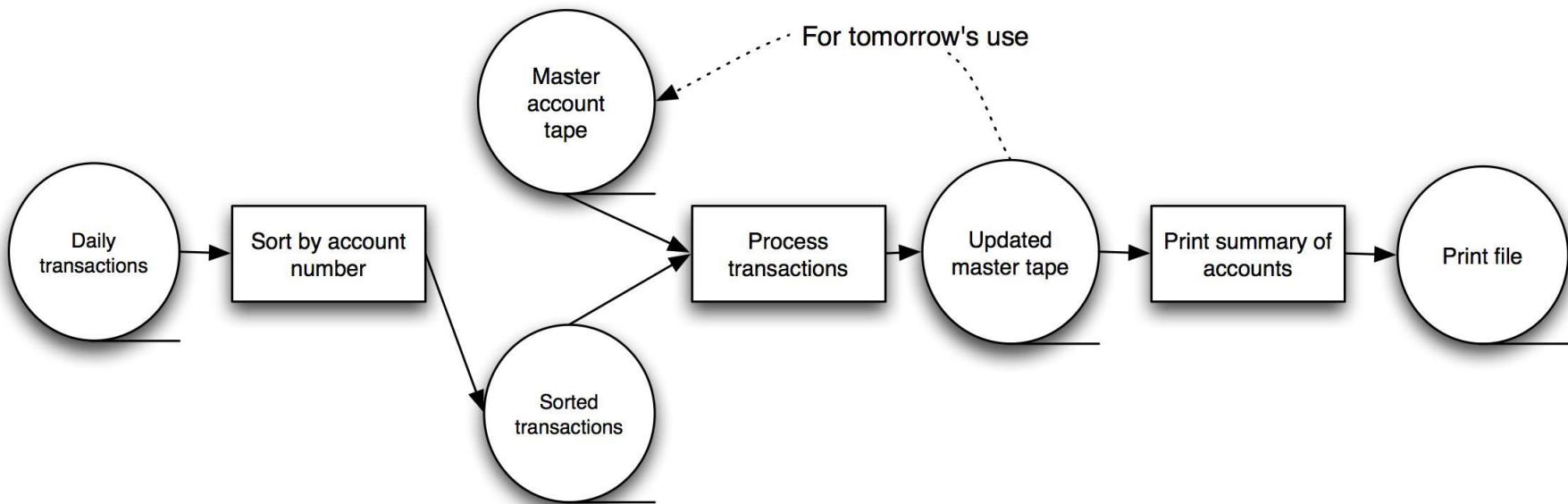


Data-Flow Styles

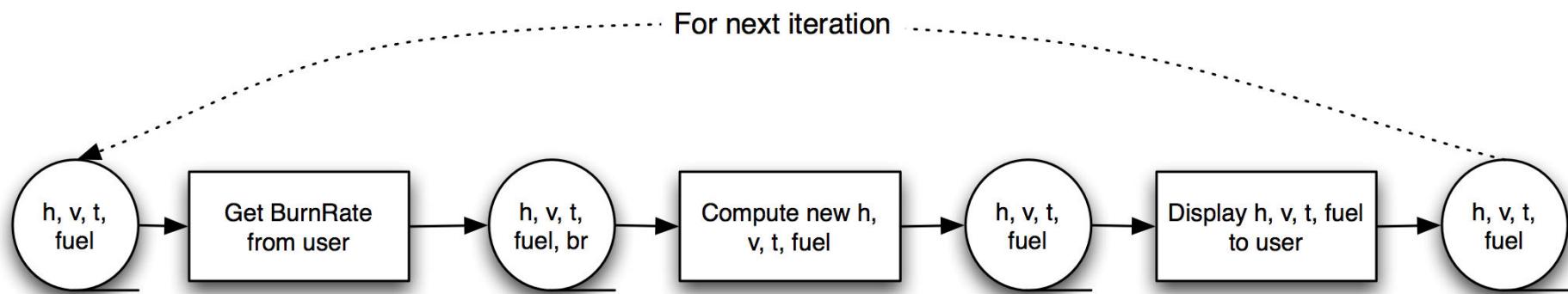
Batch Sequential

- ◆ Separate programs are executed in order; data is passed as an aggregate from one program to the next.
- ◆ Connectors: “The human hand” carrying tapes between the programs, a.k.a. “sneaker-net”
- ◆ Data Elements: Explicit, aggregate elements passed from one component to the next upon completion of the producing program’s execution.
- Typical uses: Transaction processing in financial systems. “The Granddaddy of Styles”

Batch-Sequential: A Financial Application



Batch-Sequential LL



Not a recipe for a successful lunar mission!

Pipe and Filter Style

- Components are filters
 - ◆ Transform input data streams into output data streams
 - ◆ Possibly incremental production of output
- Connectors are pipes
 - ◆ Conduits for data streams
- Style invariants
 - ◆ Filters are independent (no shared state)
 - ◆ Filter has no knowledge of up- or down-stream filters
- Examples
 - ◆ UNIX shell
 - ◆ Distributed systems
- Example: `ls invoices | grep -e August | sort`

signal processing

parallel programming

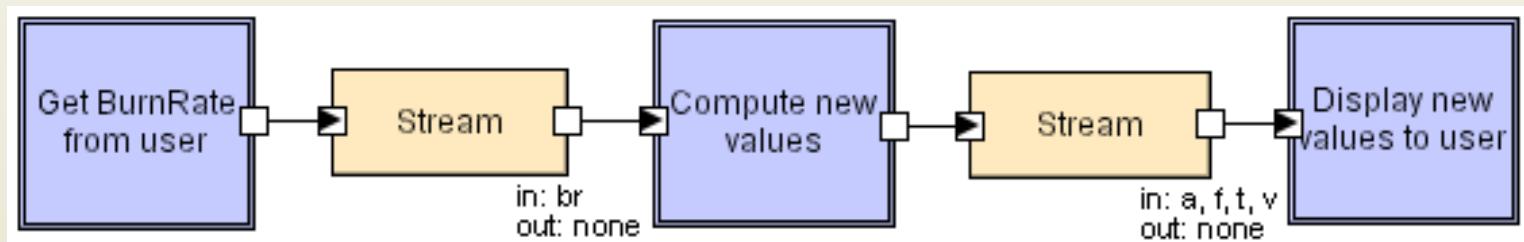
Pipe and Filter (cont'd)

- Variations
 - ◆ Pipelines — linear sequences of filters
 - ◆ Bounded pipes — limited amount of data on a pipe
 - ◆ Typed pipes — data strongly typed
- Advantages
 - ◆ System behavior is a succession of component behaviors
 - ◆ Filter addition, replacement, and reuse
 - Possible to hook any two filters together
 - ◆ Certain analyses
 - Throughput, latency, deadlock
 - ◆ Concurrent execution

Pipe and Filter (cont'd)

- Disadvantages
 - ◆ Batch organization of processing
 - ◆ Interactive applications
 - ◆ Lowest common denominator on data transmission

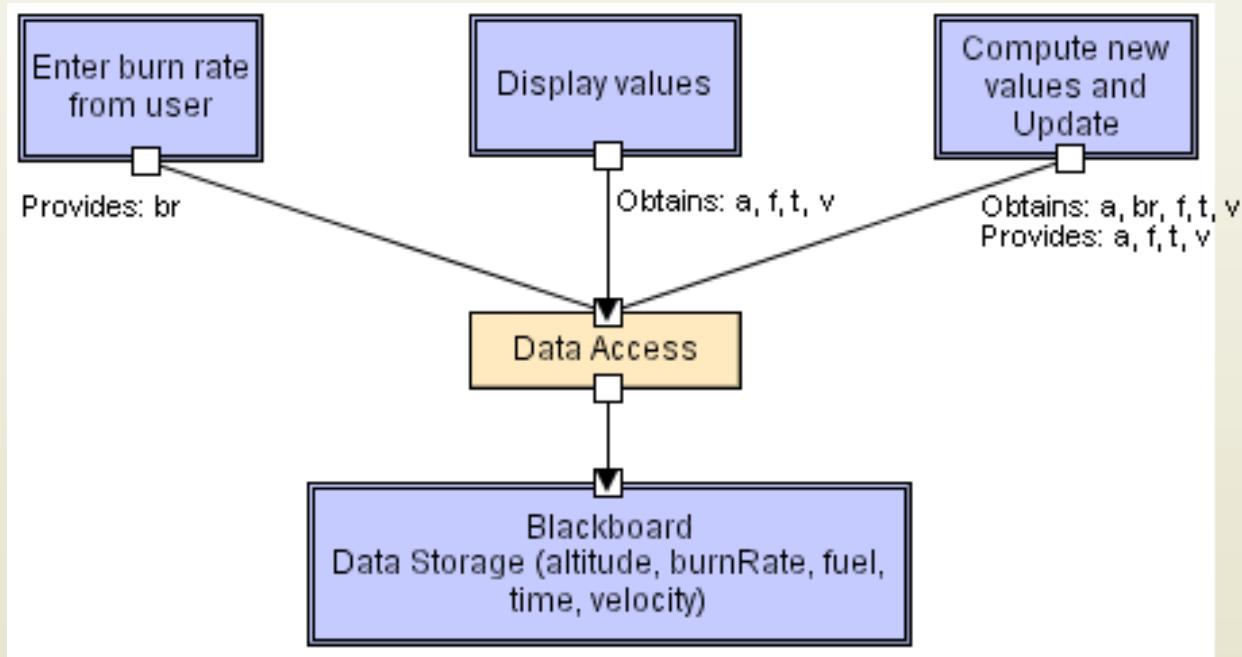
Pipe and Filter LL



Blackboard Style

- Two kinds of components
 - ◆ Central data structure — blackboard
 - ◆ Components operating on the blackboard
- System control is entirely driven by the blackboard state
- Examples
 - ◆ Typically used for AI systems
 - ◆ Integrated software environments (e.g., Interlisp)
 - ◆ Compiler architecture

Blackboard LL



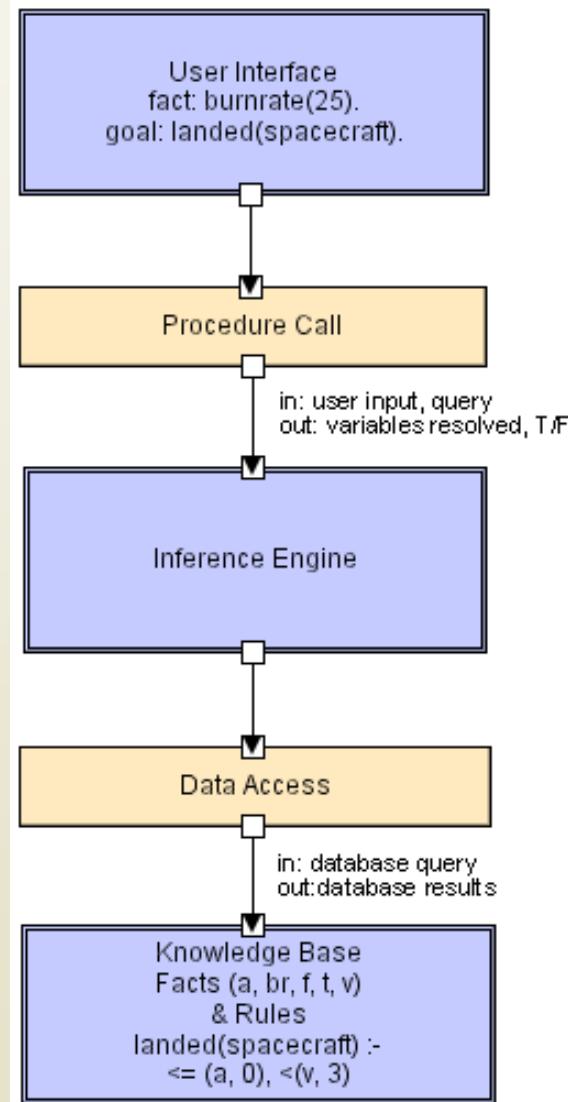
Rule-Based Style

Inference engine parses user input and determines whether it is a fact/rule or a query. If it is a fact/rule, it adds this entry to the knowledge base. Otherwise, it queries the knowledge base for applicable rules and attempts to resolve the query.

Rule-Based Style (cont'd)

- Components: User interface, inference engine, knowledge base
- Connectors: Components are tightly interconnected, with direct procedure calls and/or shared memory.
- Data Elements: Facts and queries
- Behavior of the application can be very easily modified through addition or deletion of rules from the knowledge base.
- Caution: When a large number of rules are involved understanding the interactions between multiple rules affected by the same facts can become *very* difficult.

Rule Based LL

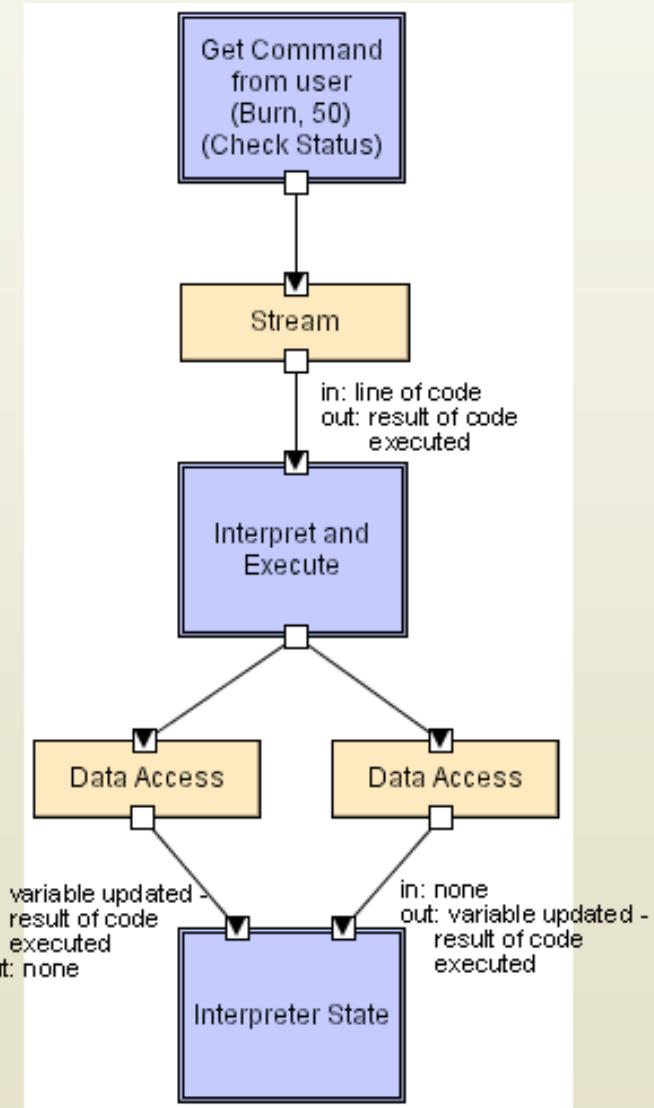


Interpreter Style

Interpreter parses and executes input commands, updating the state maintained by the interpreter

- Components: Command interpreter, program/interpreter state, user interface.
- Connectors: Typically very closely bound with direct procedure calls and shared state.
- Highly dynamic behavior possible, where the set of commands is dynamically modified. System architecture may remain constant while new capabilities are created based upon existing primitives.
- Superb for end-user programmability; supports dynamically changing set of capabilities
- Lisp and Scheme

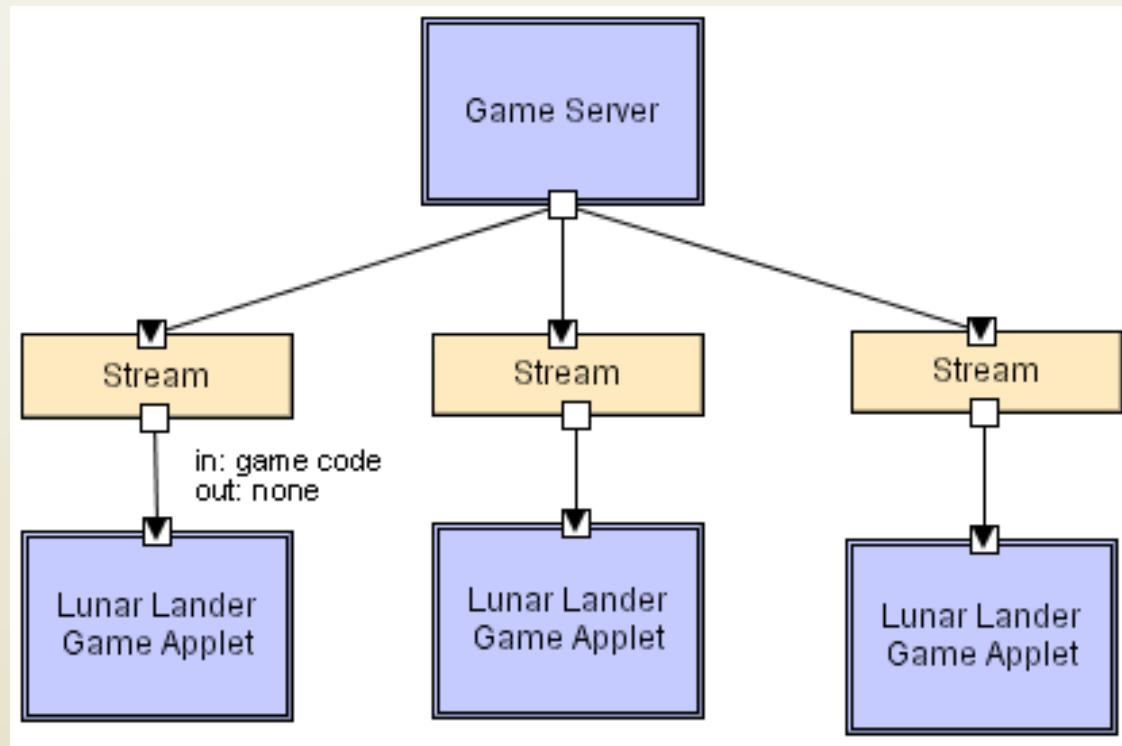
Interpreter LL



Mobile-Code Style

- Summary: a data element (some representation of a program) is dynamically transformed into a data processing component.
- Components: “Execution dock”, which handles receipt of code and state; code compiler/interpreter
- Connectors: Network protocols and elements for packaging code and data for transmission.
- Data Elements: Representations of code as data; program state; data
- Variants: Code-on-demand, remote evaluation, and mobile agent.

Mobile Code LL



Scripting languages (i.e. JavaScript, VBScript), ActiveX control, embedded Word/Excel macros.

Implicit Invocation Style

- Event announcement instead of method invocation
 - ◆ “Listeners” register interest in and associate methods with events
 - ◆ System invokes all registered methods implicitly
- Component interfaces are methods and events
- Two types of connectors
 - ◆ Invocation is either explicit or implicit in response to events
- Style invariants
 - ◆ “Announcers” are unaware of their events’ effects
 - ◆ No assumption about processing in response to events

Implicit Invocation (cont'd)

- Advantages
 - ◆ Component reuse
 - ◆ System evolution
 - Both at system construction-time & run-time
- Disadvantages
 - ◆ Counter-intuitive system structure
 - ◆ Components relinquish computation control to the system
 - ◆ No knowledge of what components will respond to event
 - ◆ No knowledge of order of responses

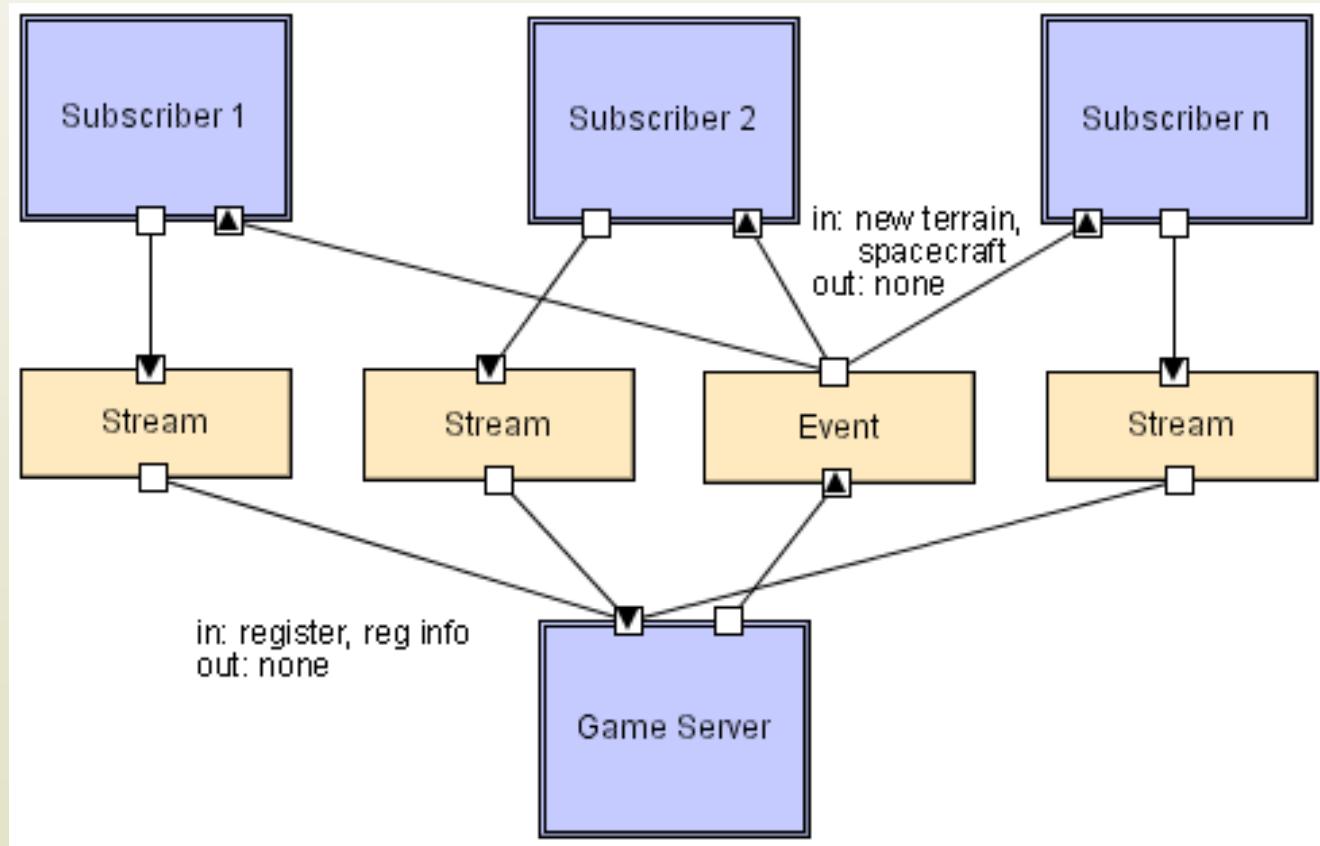
Publish-Subscribe

Subscribers register/deregister to receive specific messages or specific content. Publishers broadcast messages to subscribers either synchronously or asynchronously.

Publish-Subscribe (cont'd)

- Components: Publishers, subscribers, proxies for managing distribution
- Connectors: Typically a network protocol is required. Content-based subscription requires sophisticated connectors.
- Data Elements: Subscriptions, notifications, published information
- Topology: Subscribers connect to publishers either directly or may receive notifications via a network protocol from intermediaries
- Qualities yielded Highly efficient one-way dissemination of information with very low-coupling of components

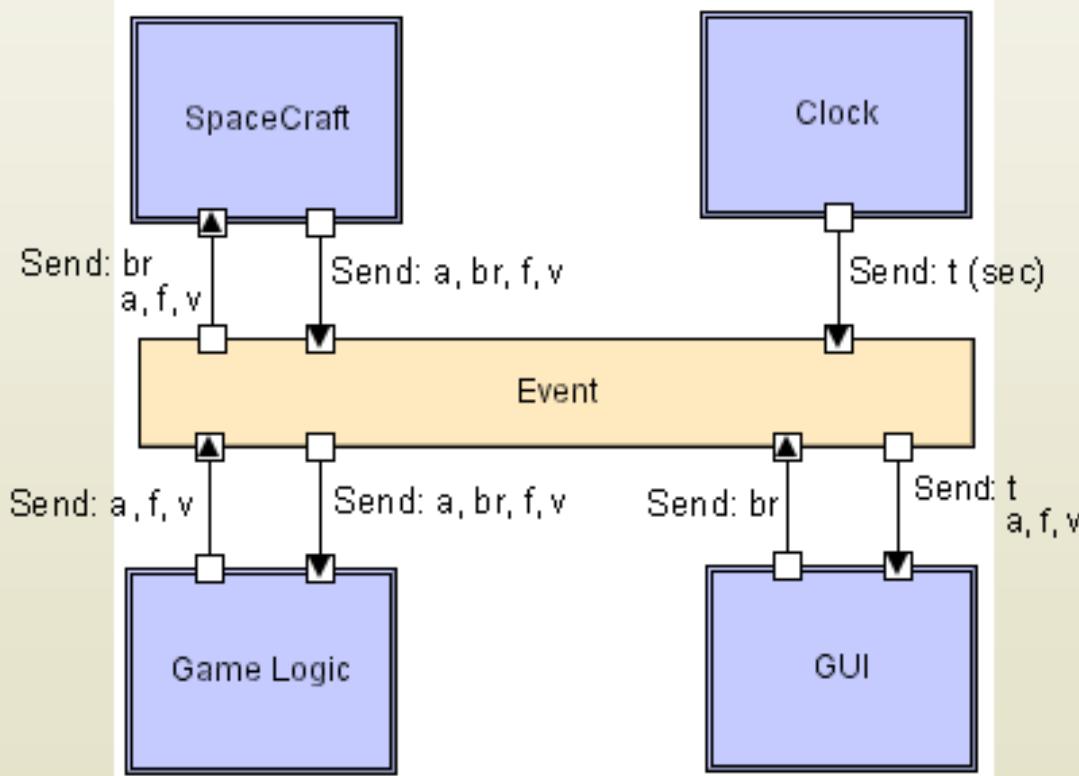
Pub-Sub LL



Event-Based Style

- Independent components asynchronously emit and receive events communicated over event buses
- Components: Independent, concurrent event generators and/or consumers
- Connectors: Event buses (at least one)
- Data Elements: Events – data sent as a first-class entity over the event bus
- Topology: Components communicate with the event buses, not directly to each other.
- Variants: Component communication with the event bus may either be push or pull based.
- Highly scalable, easy to evolve, effective for highly distributed applications.

Event-based LL



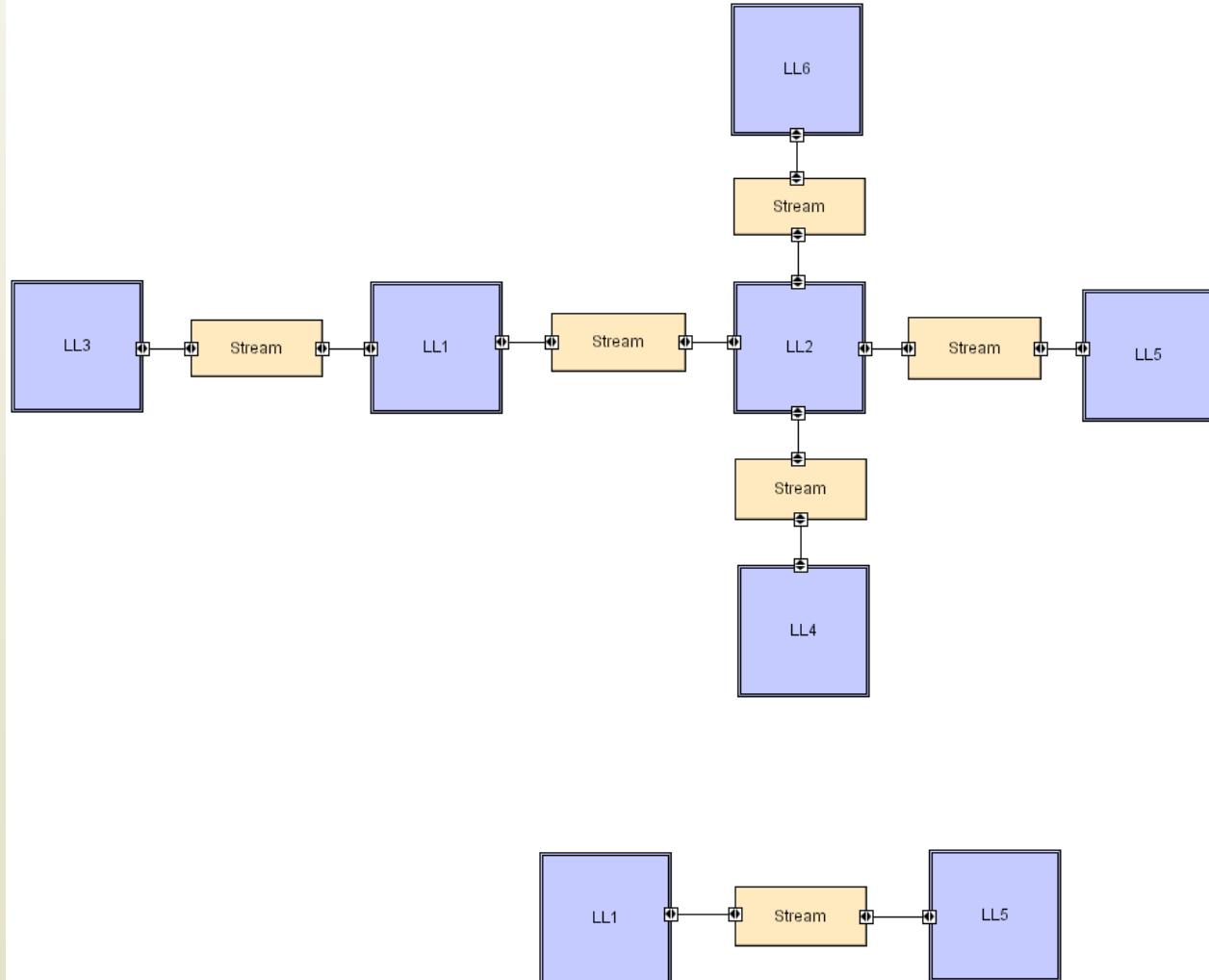
Peer-to-Peer Style

- 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

Peer-to-Peer Style (cont'd)

- 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



Styles and Greenfield Design

Software Architecture
Lecture 6

Heterogeneous Styles

- More complex styles created through composition of simpler styles
- REST (from the first lecture)
 - ◆ Complex history presented later in course
- C2
 - ◆ Implicit invocation + Layering + other constraints
- Distributed objects
 - ◆ OO + client-server network style
 - ◆ CORBA

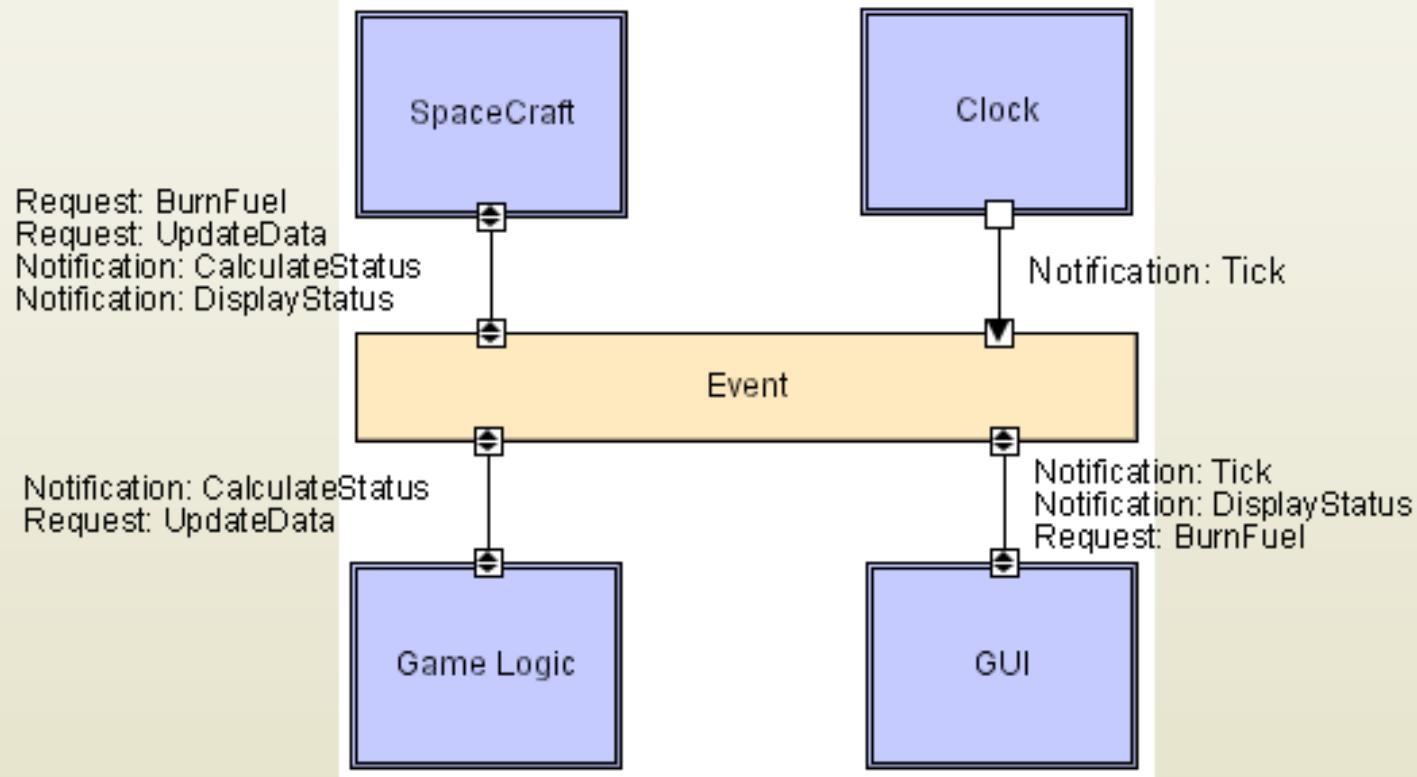
C2 Style

An indirect invocation style in which independent components communicate exclusively through message routing connectors. Strict rules on connections between components and connectors induce layering.

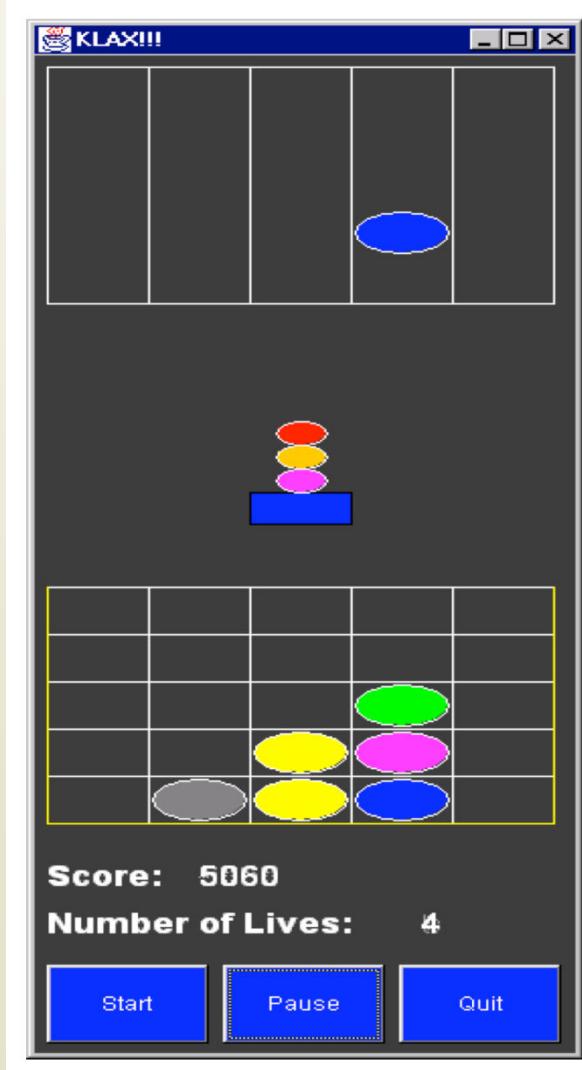
C2 Style (cont'd)

- Components: Independent, potentially concurrent message generators and/or consumers
- Connectors: Message routers that may filter, translate, and broadcast messages of two kinds: notifications and requests.
- Data Elements: Messages – data sent as first-class entities over the connectors. Notification messages announce changes of state. Request messages request performance of an action.
- Topology: Layers of components and connectors, with a defined “top” and “bottom”, wherein notifications flow downwards and requests upwards.

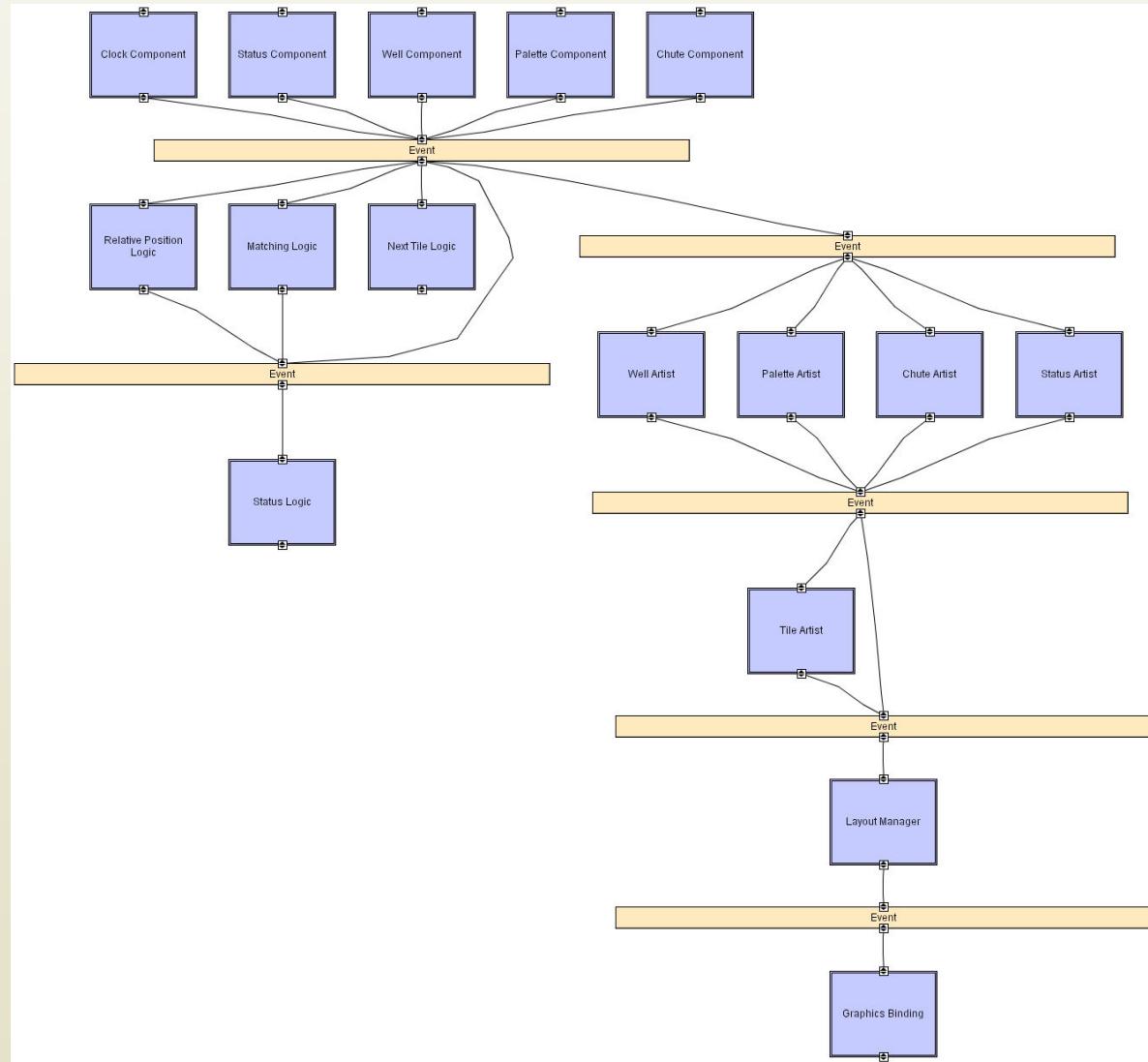
C2 LL



KLAX



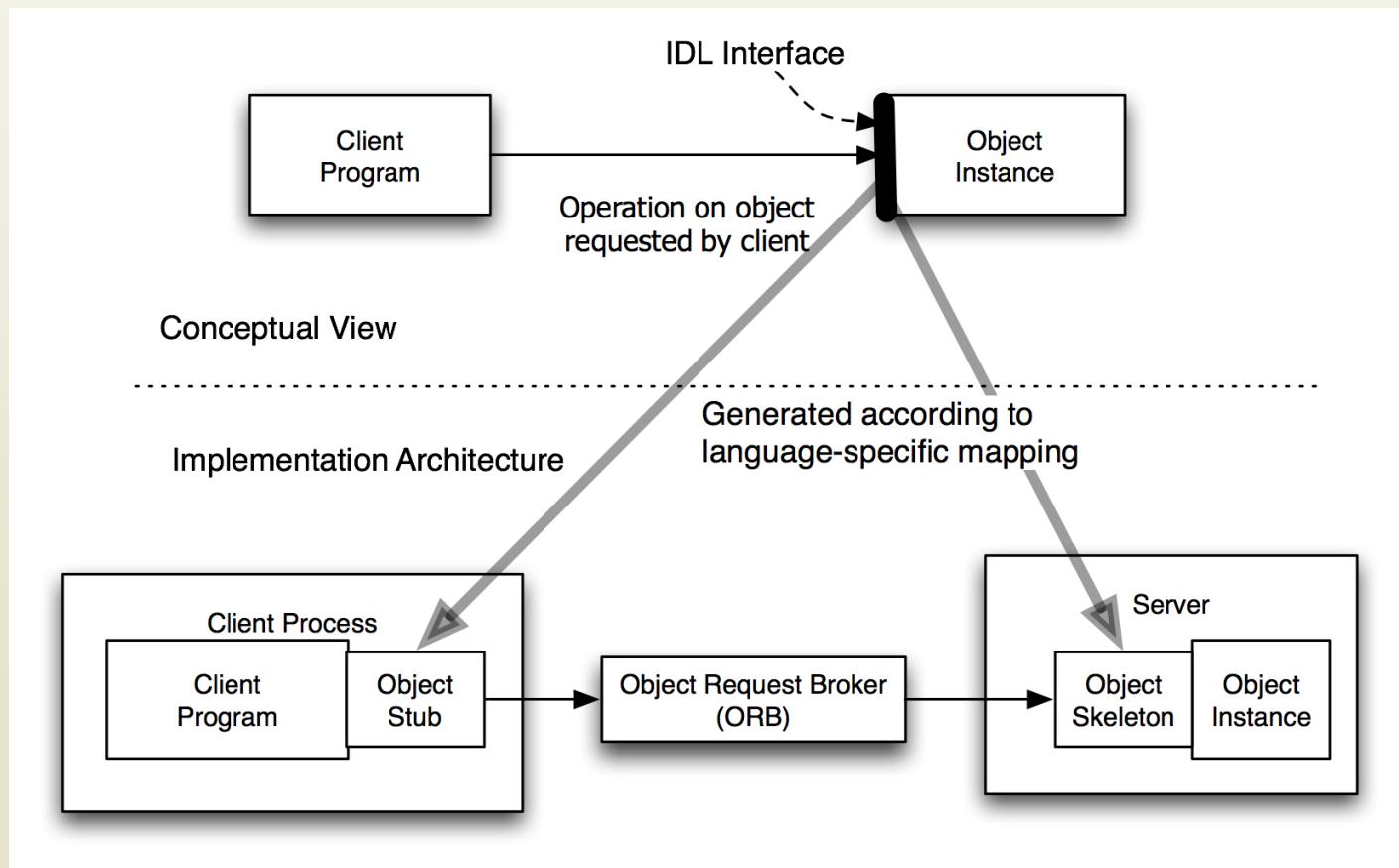
KLAX in C2



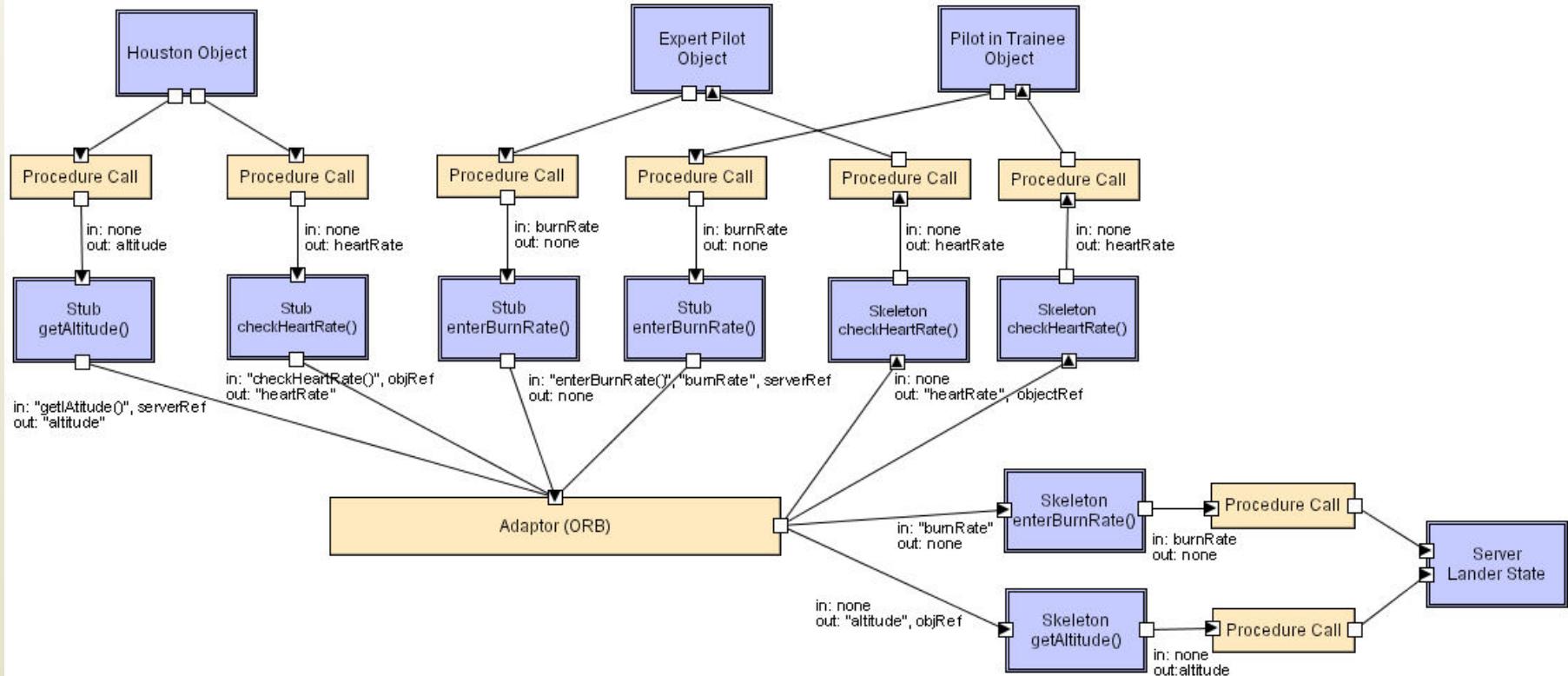
Distributed Objects: CORBA

- “Objects” (coarse- or fine-grained) run on heterogeneous hosts, written in heterogeneous languages. Objects provide services through well-defined interfaces. Objects invoke methods across host, process, and language boundaries via remote procedure calls (RPCs).
- Components: Objects (software components exposing services through well-defined provided interfaces)
- Connector: (Remote) Method invocation
- Data Elements: Arguments to methods, return values, and exceptions
- Topology: General graph of objects from callers to callees.
- Additional constraints imposed: Data passed in remote procedure calls must be serializable. Callers must deal with exceptions that can arise due to network or process faults.
- Location, platform, and language “transparency”. **CAUTION**

CORBA Concept and Implementation



CORBA LL



Observations

- Different styles result in
 - ◆ Different architectures
 - ◆ Architectures with greatly differing properties
- A style does not fully determine resulting architecture
 - ◆ A single style can result in different architectures
 - ◆ Considerable room for
 - Individual judgment
 - Variations among architects
- A style defines domain of discourse
 - ◆ About problem (domain)
 - ◆ About resulting system

Style Summary (1/4)

Style Category & Name	Summary	Use It When	Avoid It When
<i>Language-influenced styles</i>			
Main Program and Subroutines	Main program controls program execution, calling multiple subroutines.	Application is small and simple.	Complex data structures needed. Future modifications likely.
Object-oriented	Objects encapsulate state and accessing functions	Close mapping between external entities and internal objects is sensible. Many complex and interrelated data structures.	Application is distributed in a heterogeneous network. Strong independence between components necessary. High performance required.
<i>Layered</i>			
Virtual Machines	Virtual machine, or a layer, offers services to layers above it	Many applications can be based upon a single, common layer of services. Interface service specification resilient when implementation of a layer must change.	Many levels are required (causes inefficiency). Data structures must be accessed from multiple layers.
Client-server	Clients request service from a server	Centralization of computation and data at a single location (the server) promotes manageability and scalability; end-user processing limited to data entry and presentation.	Centrality presents a single-point-of-failure risk; Network bandwidth limited; Client machine capabilities rival or exceed the server's.

Style Summary, continued (2/4)

Data-flow styles

Batch sequential	Separate programs executed sequentially, with batched input	Problem easily formulated as a set of sequential, severable steps.	Interactivity or concurrency between components necessary or desirable.
Pipe-and-filter	Separate programs, a.k.a. filters, executed, potentially concurrently. Pipes route data streams between filters	[As with batch-sequential] Filters are useful in more than one application. Data structures easily serializable.	Random-access to data required. Interaction between components required. Exchange of complex data structures between components required.

Shared memory

Blackboard	Independent programs, access and communicate exclusively through a global repository known as blackboard	All calculation centers on a common, changing data structure; Order of processing dynamically determined and data-driven.	Programs deal with independent parts of the common data. Interface to common data susceptible to change. When interactions between the independent programs require complex regulation.
Rule-based	Use facts or rules entered into the knowledge base to resolve a query	Problem data and queries expressible as simple rules over which inference may be performed.	Number of rules is large. Interaction between rules present. High-performance required.

Style Summary, continued (3/4)

Interpreter

Interpreter	Interpreter parses and executes the input stream, updating the state maintained by the interpreter	Highly dynamic behavior required. High degree of end-user customizability.	High performance required.
Mobile Code	Code is mobile, that is, it is executed in a remote host	When it is more efficient to move processing to a data set than the data set to processing. When it is desirable to dynamically customize a local processing node through inclusion of external code	Security of mobile code cannot be assured, or sandboxed. When tight control of versions of deployed software is required.

Style Summary, continued (4/4)

Implicit Invocation

Publish-subscribe Publishers broadcast messages to subscribers Components are very loosely coupled. Subscription data is small and efficiently transported. When middleware to support high-volume data is unavailable.

Event-based Independent components asynchronously emit and receive events communicated over event buses Components are concurrent and independent. Components heterogeneous and network-distributed. Guarantees on real-time processing of events is required.

Peer-to-peer Peers hold state and behavior and can act as both clients and servers Peers are distributed in a network, can be heterogeneous, and mutually independent. Robust in face of independent failures. Highly scalable. Trustworthiness of independent peers cannot be assured or managed. Resource discovery inefficient without designated nodes.

More complex styles

C2 Layered network of concurrent components communicating by events When independence from substrate technologies required. Heterogeneous applications. When support for product-lines desired. When high-performance across many layers required. When multiple threads are inefficient.

Distributed Objects Objects instantiated on different hosts Objective is to preserve illusion of location-transparency When high overhead of supporting middleware is excessive. When network properties are unmaskable, in practical terms.

Design Recovery

- What happens if a system is already implemented but has no recorded architecture?
- The task of design recovery is
 - ◆ examining the existing code base
 - ◆ determining what the system's components, connectors, and overall topology are.
- A common approach to architectural recovery is clustering of the implementation-level entities into architectural elements.
 - ◆ Syntactic clustering
 - ◆ Semantic clustering

Syntactic Clustering

- Focuses exclusively on the static relationships among code-level entities
- Can be performed without executing the system
- Embodies inter-component (a.k.a. coupling) and intra-component (a.k.a. cohesion) connectivity
- May ignore or misinterpret many subtle relationships, because dynamic information is missing

Semantic Clustering

- Includes all aspects of a system's domain knowledge and information about the behavioral similarity of its entities.
- Requires interpreting the system entities' meaning, and possibly executing the system on a representative set of inputs.
- Difficult to automate
- May also be difficult to avail oneself of it

When There's No Experience to Go On....

- The first effort a designer should make in addressing a novel design challenge is to attempt to determine that it is genuinely a novel problem.
- Basic Strategy
 - ◆ Divergence – shake off inadequate prior approaches and discover or admit a variety of new ideas
 - ◆ Transformation – combination of analysis and selection
 - ◆ Convergence – selecting and further refining ideas
- Repeatedly cycling through the basic steps until a feasible solution emerges.

Analogy Searching

- Examine other fields and disciplines unrelated to the target problem for approaches and ideas that are analogous to the problem.
- Formulate a solution strategy based upon that analogy.
- A common “unrelated domain” that has yielded a variety of solutions is nature, especially the biological sciences.
 - ◆ E.g., Neural Networks

Brainstorming

- Technique of rapidly generating a wide set of ideas and thoughts pertaining to a design problem
 - ◆ without (initially) devoting effort to assessing the feasibility.
- Brainstorming can be done by an individual or, more commonly, by a group.
- Problem: A brainstorming session can generate a large number of ideas... all of which might be low-quality.
- The chief value of brainstorming is in identifying categories of possible designs, not any specific design solution suggested during a session.
- After brainstorm the design process may proceed to the Transformation and Convergence steps.

“Literature” Searching

- Examining published information to identify material that can be used to guide or inspire designers
- Many historically useful ways of searching “literature” are available
- Digital library collections make searching extraordinarily faster and more effective
 - ◆ IEEE Xplore
 - ◆ ACM Digital Library
 - ◆ Google Scholar
- The availability of free and open-source software adds special value to this technique.

Morphological Charts

- The essential idea:
 - ◆ identify all the primary functions to be performed by the desired system
 - ◆ for each function identify a means of performing that function
 - ◆ attempt to choose one means for each function such that the collection of means performs all the required functions in a compatible manner.
- The technique does not demand that the functions be shown to be independent when starting out.
- Sub-solutions to a given problem do not need to be compatible with all the sub-solutions to other functions in the beginning.

Removing Mental Blocks

- If you can't solve the problem, change the problem to one you can solve.
 - ◆ If the new problem is “close enough” to what is needed, then closure is reached.
 - ◆ If it is not close enough, the solution to the revised problem may suggest new venues for attacking the original.

Controlling the Design Strategy

- The potentially chaotic nature of exploring diverse approaches to the problem demands that some care be used in managing the activity
- Identify and review critical decisions
- Relate the costs of research and design to the penalty for taking wrong decisions
- Insulate uncertain decisions
- Continually re-evaluate system “requirements” in light of what the design exploration yields

Insights from Requirements

- In many cases new architectures can be created based upon experience with and improvement to pre-existing architectures.
- Requirements can use a vocabulary of known architectural choices and therefore reflect experience.
- The interaction between past design and new requirements means that many critical decisions for a new design can be identified or made as a requirement

Insights from Implementation

- Constraints on the implementation activity may help shape the design.
- Externally motivated constraints might dictate
 - ◆ Use of a middleware
 - ◆ Use of a particular programming language
 - ◆ Software reuse
- Design and implementation may proceed cooperatively and contemporaneously
 - ◆ Initial partial implementation activities may yield critical performance or feasibility information

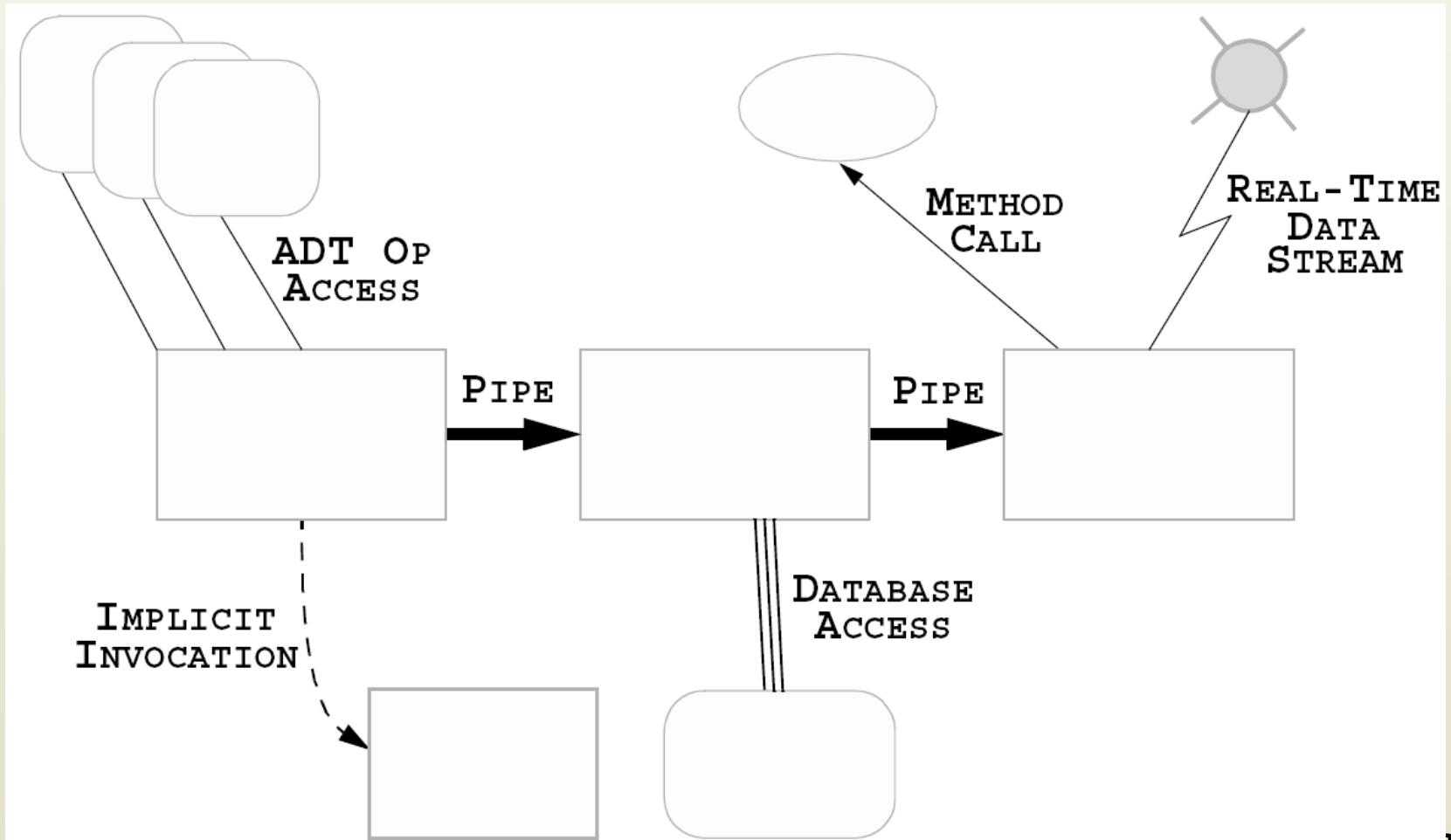
Software Connectors

Software Architecture
Lecture 7

What is a Software Connector?

- Architectural element that models
 - ◆ Interactions among components
 - ◆ Rules that govern those interactions
- Simple interactions
 - ◆ Procedure calls
 - ◆ Shared variable access
- Complex & semantically rich interactions
 - ◆ Client-server protocols
 - ◆ Database access protocols
 - ◆ Asynchronous event multicast
- Each connector provides
 - ◆ Interaction duct(s)
 - ◆ Transfer of control and/or data

Where are Connectors in Software Systems?



Implemented vs. Conceptual Connectors

- Connectors in software system implementations
 - ◆ Frequently no dedicated code
 - ◆ Frequently no identity
 - ◆ Typically do not correspond to compilation units
 - ◆ Distributed implementation
 - Across multiple modules
 - Across interaction mechanisms

Implemented vs. Conceptual Connectors (cont'd)

- Connectors in software architectures
 - ◆ First-class entities
 - ◆ Have identity
 - ◆ Describe all system interaction
 - ◆ Entitled to their own specifications & abstractions

Reasons for Treating Connectors Independently

- Connector \neq Component
 - ◆ Components provide application-specific functionality
 - ◆ Connectors provide application-independent interaction mechanisms
- Interaction abstraction and/or parameterization
- Specification of complex interactions
 - ◆ Binary vs. N-ary
 - ◆ Asymmetric vs. Symmetric
 - ◆ Interaction protocols

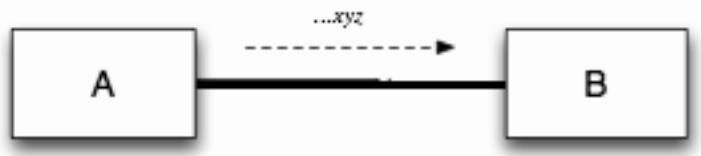
Treating Connectors Independently (cont'd)

- Localization of interaction definition
- Extra-component system (interaction) information
- Component independence
- Component interaction flexibility

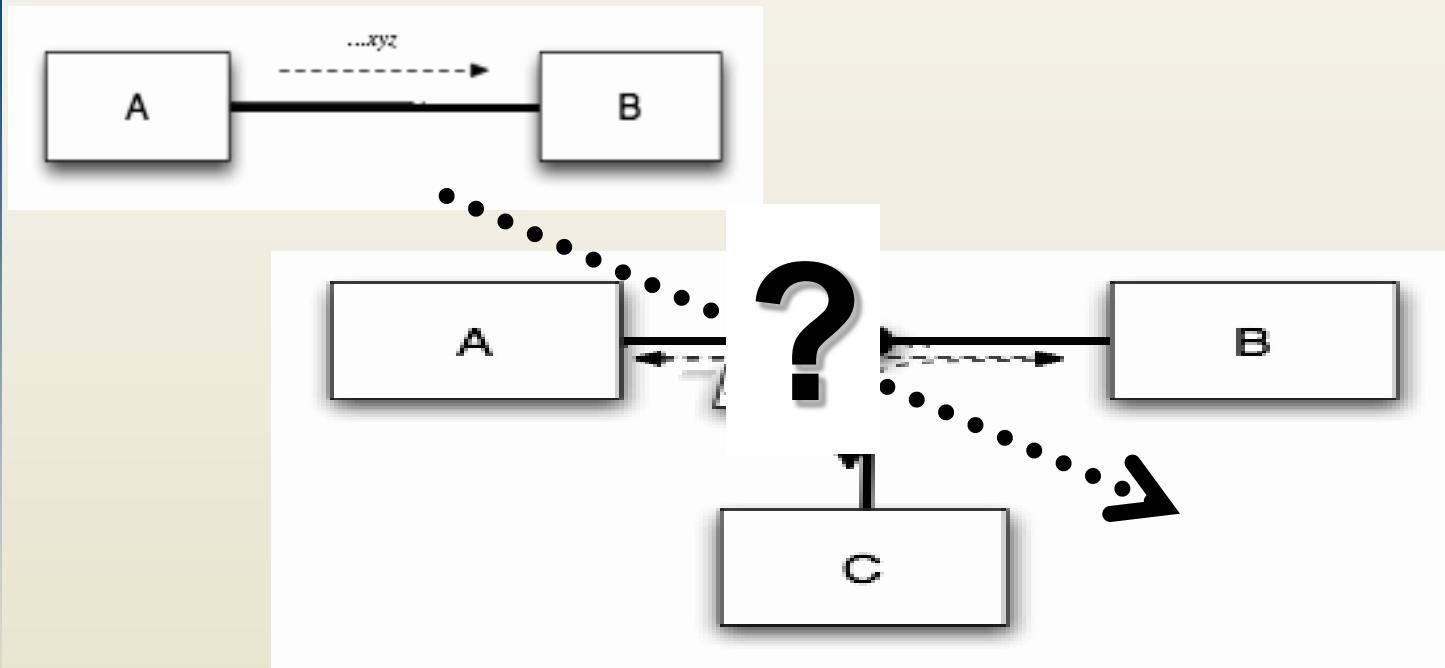
Benefits of First-Class Connectors

- Separate computation from interaction
- Minimize component interdependencies
- Support software evolution
 - ◆ At component-, connector-, & system-level
- Potential for supporting dynamism
- Facilitate heterogeneity
- Become points of distribution
- Aid system analysis & testing

An Example of Explicit Connectors



An Example of Explicit Connectors (cont'd)



Software Connector Roles

- Locus of interaction among set of components
- Protocol specification (sometimes implicit) that defines its properties
 - ◆ Types of interfaces it is able to mediate
 - ◆ Assurances about interaction properties
 - ◆ Rules about interaction ordering
 - ◆ Interaction commitments (e.g., performance)
- Roles
 - ◆ Communication
 - ◆ Coordination
 - ◆ Conversion
 - ◆ Facilitation

Connectors as Communicators

- Main role associated with connectors
- Supports
 - ◆ Different communication mechanisms
 - e.g. procedure call, RPC, shared data access, message passing
 - ◆ Constraints on communication structure/direction
 - e.g. pipes
 - ◆ Constraints on quality of service
 - e.g. persistence
- Separates communication from computation
- May influence non-functional system characteristics
 - ◆ e.g. performance, scalability, security

Connectors as Coordinators

- Determine computation control
- Control delivery of data
- Separates control from computation
- Orthogonal to communication, conversion, and facilitation
 - ◆ Elements of control are in communication, conversion and facilitation

Connectors as Converters

- Enable interaction of independently developed, mismatched components
- Mismatches based on interaction
 - ◆ Type
 - ◆ Number
 - ◆ Frequency
 - ◆ Order
- Examples of converters
 - ◆ Adaptors
 - ◆ Wrappers

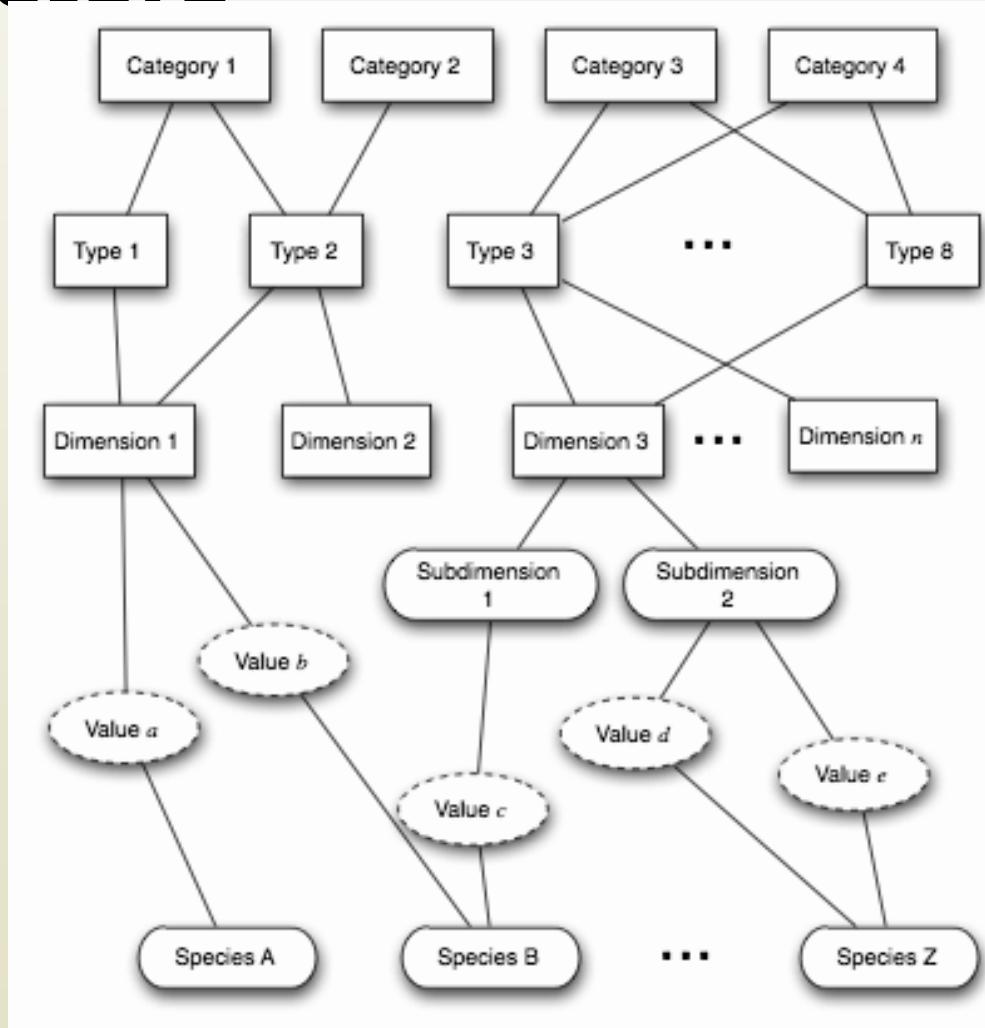
Connectors as Facilitators

- Enable interaction of components intended to interoperate
 - ◆ Mediate and streamline interaction
- Govern access to shared information
- Ensure proper performance profiles
 - ◆ e.g., load balancing
- Provide synchronization mechanisms
 - ◆ Critical sections
 - ◆ Monitors

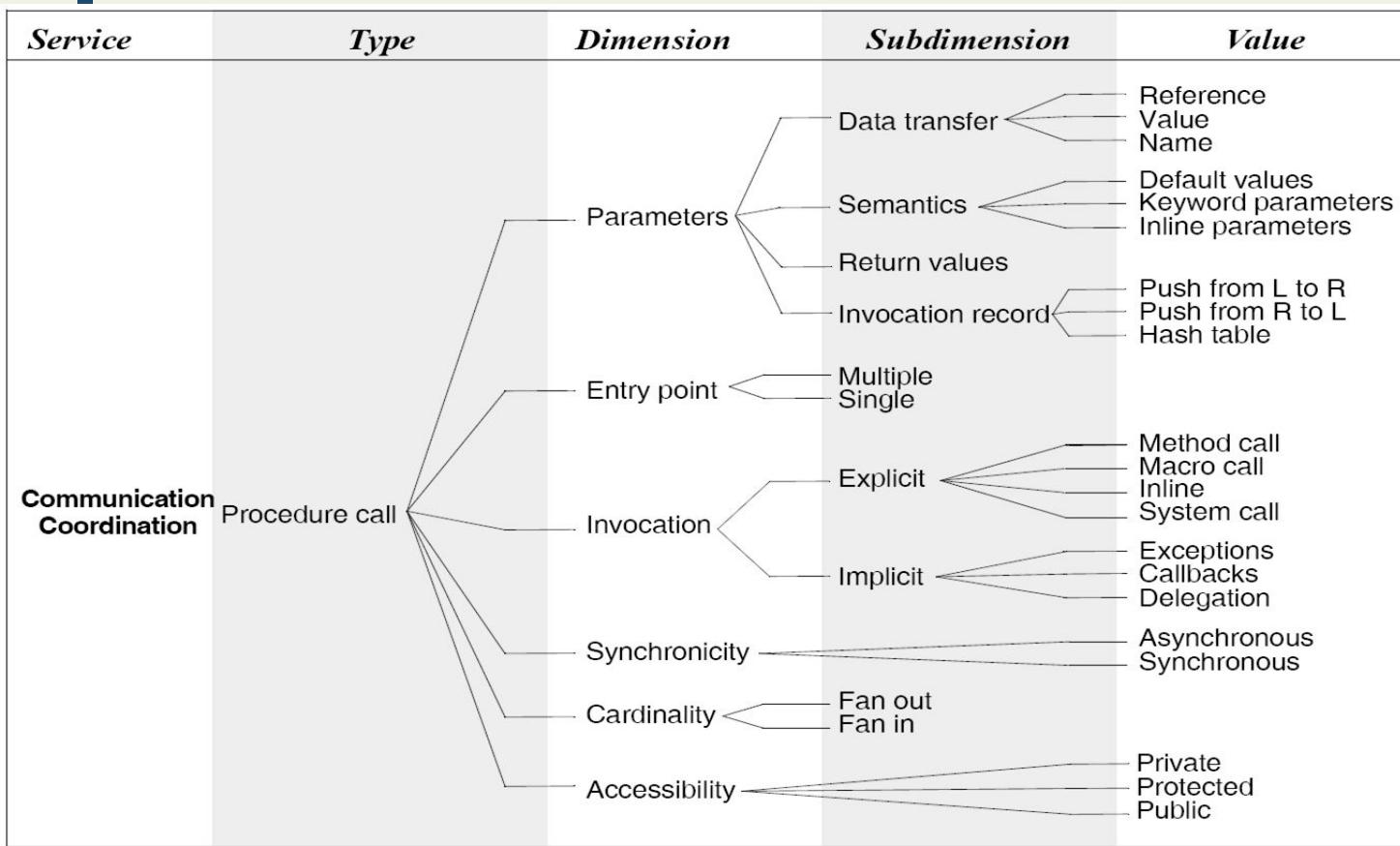
Connector Types

- Procedure call
- Data access
- Event
- Stream
- Linkage
- Distributor
- Arbitrator
- Adaptor

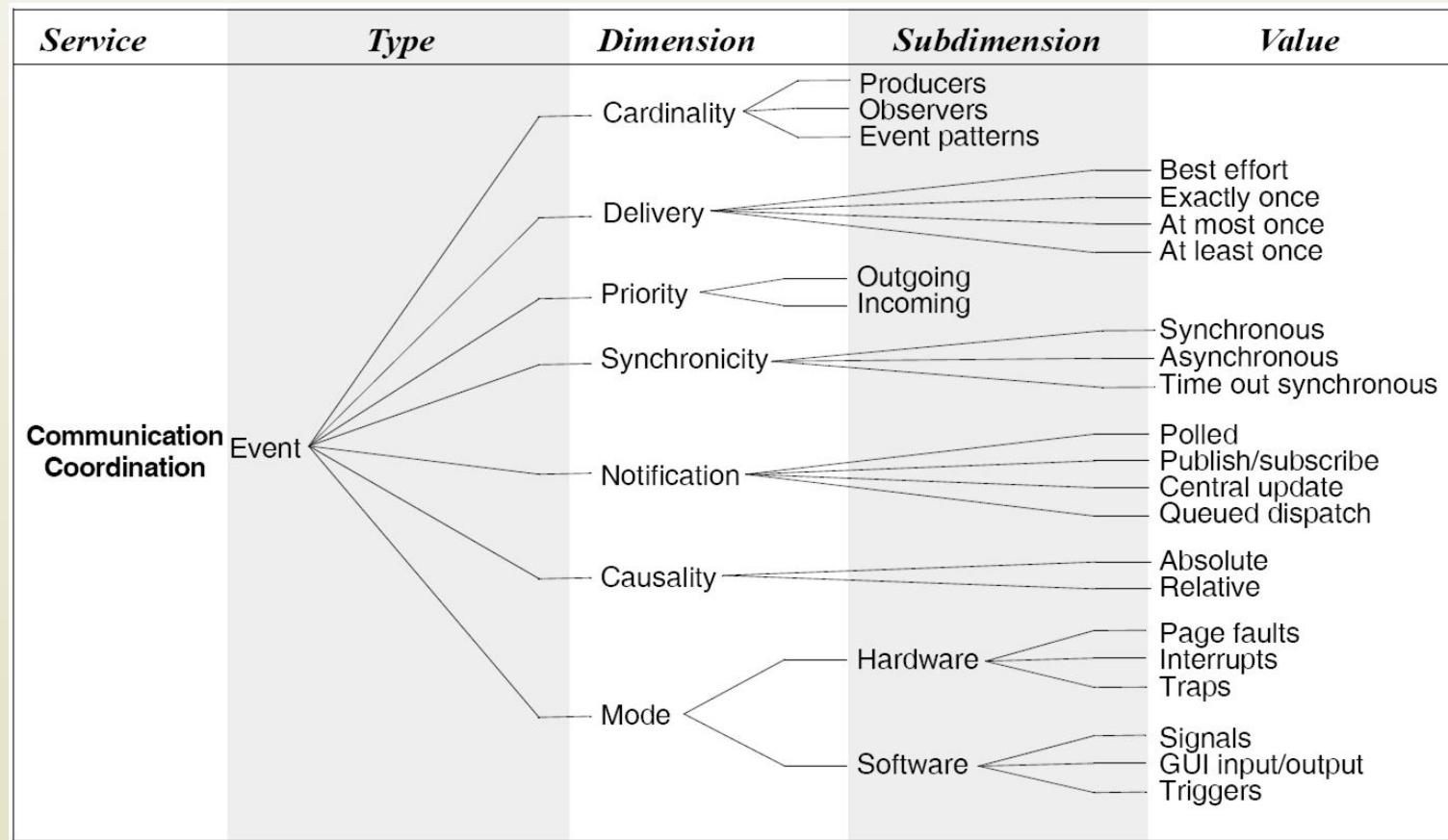
A Framework for Classifying Connectors



Procedure Call Connectors



Event Connectors



Data Access Connectors

<i>Service</i>	<i>Type</i>	<i>Dimension</i>	<i>Subdimension</i>	<i>Value</i>
Communication Conversion	Data Access	<pre> graph TD DA[Data Access] --> Loc[Locality] DA --> Acc[Access] DA --> Avail[Availability] DA --> Acces[Accessibility] DA --> Life[Lifecycle] DA --> Card[Cardinality] Loc --> TSS[Thread specific] Loc --> PSS[Process specific] Loc --> Global[Global] Acc --> Accs[Accessor] Acc --> Mut[Mutator] Avail --> Trans[Transient] Avail --> Pers[Persistent] Trans --> Reg[Register] Trans --> Cache[Cache] Trans --> DMA[DMA] Trans --> Heap[Heap] Trans --> Stack[Stack] Pers --> RA[Repository access] Pers --> FIO[File I/O] Pers --> DDE[Dynamic data exchange] Pers --> DBA[Database Access] Acces --> Priv[Private] Acces --> Prot[Protected] Acces --> Pub[Public] Life --> Init[Initialization] Life --> Term[Termination] Card --> Def[Defines] Card --> Use[Uses] </pre>	Locality Access Availability Accessibility Lifecycle Cardinality	Thread specific Process specific Global Accessor Mutator Register Cache DMA Heap Stack Repository access File I/O Dynamic data exchange Database Access Private Protected Public Initialization Termination Defines Uses

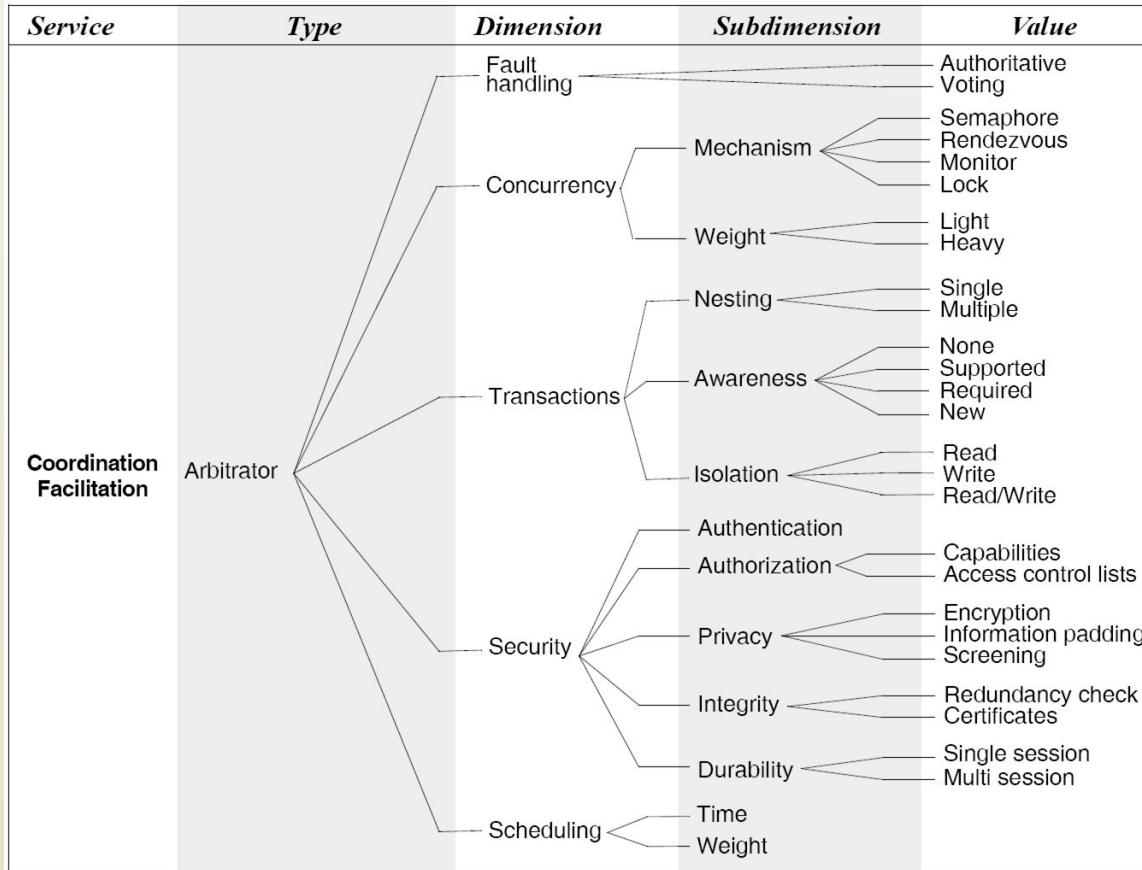
Linkage Connectors

<i>Service</i>	<i>Type</i>	<i>Dimension</i>	<i>Subdimension</i>	<i>Value</i>
Facilitation	Linkage	Reference Granularity Cardinality Binding	Implicit Explicit Unit Syntactic Semantic Defines Uses Provides Requires	Variable Procedure Function Constant Type Compile-time Run-time Pre-compile-time

Stream Connectors

<i>Service</i>	<i>Type</i>	<i>Dimension</i>	<i>Subdimension</i>	<i>Value</i>
Communication Stream	Delivery			Best effort Exactly once At most once At least once
	Bounds			Bounded Unbounded
	Buffering			Buffered Unbuffered
	Throughput			Atomic units Higher-order units
	State			Stateless Stateful
	Identity			Named Unnamed
	Locality			Local Remote
	Synchronicity			Synchronous Asynchronous Time out synchronous
	Format			Raw Structured
	Cardinality		Binary	Multi sender Multi receiver
			N-ary	Multi sender/receiver

Arbitrator Connectors



Adaptor Connectors

<i>Service</i>	<i>Type</i>	<i>Dimension</i>	<i>Subdimension</i>	<i>Value</i>
Conversion	Adaptor	Invocation conversion Packaging conversion Protocol conversion Presentation conversion	Address mapping Marshalling Translation Wrappers Packagers	

Distributor Connectors

<i>Service</i>	<i>Type</i>	<i>Dimension</i>	<i>Subdimension</i>	<i>Value</i>
Facilitation	Distributor	Naming	Structure based Attribute based	Hierarchical Flat
		Delivery	Semantics Mechanism	Best effort Exactly once At most once At least once Unicast Multicast Broadcast
		Routing	Membership Path	Bounded Ad-hoc Static Cached Dynamic

```

graph LR
    Facilitation[Facilitation] --> Naming[Naming]
    Facilitation --> Delivery[Delivery]
    Facilitation --> Routing[Routing]
    Naming --> StructureBased[Structure based]
    Naming --> AttributeBased[Attribute based]
    Delivery --> Semantics[Semantics]
    Delivery --> Mechanism[Mechanism]
    Routing --> Membership[Membership]
    Routing --> Path[Path]
    StructureBased --> Hierarchical[Hierarchical]
    StructureBased --> Flat[Flat]
    Semantics --> BestEffort[Best effort]
    Semantics --> ExactlyOnce[Exactly once]
    Semantics --> AtMostOnce[At most once]
    Semantics --> AtLeastOnce[At least once]
    Mechanism --> Unicast[Unicast]
    Mechanism --> Multicast[Multicast]
    Mechanism --> Broadcast[Broadcast]
    Membership --> Bounded[Bounded]
    Membership --> AdHoc[Ad-hoc]
    Path --> Static[Static]
    Path --> Cached[Cached]
    Path --> Dynamic[Dynamic]
  
```

Discussion

- Connectors allow modeling of arbitrarily complex interactions
- Connector flexibility aids system evolution
 - ◆ Component addition, removal, replacement, reconnection, migration
- Support for connector interchange is desired
 - ◆ Aids system evolution
 - ◆ May not affect system functionality

Discussion

- Libraries of OTS connector implementations allow developers to focus on application-specific issues
- Difficulties
 - ◆ Rigid connectors
 - ◆ Connector “dispersion” in implementations
- Key issue
 - ◆ Performance vs. flexibility

Choosing Connectors

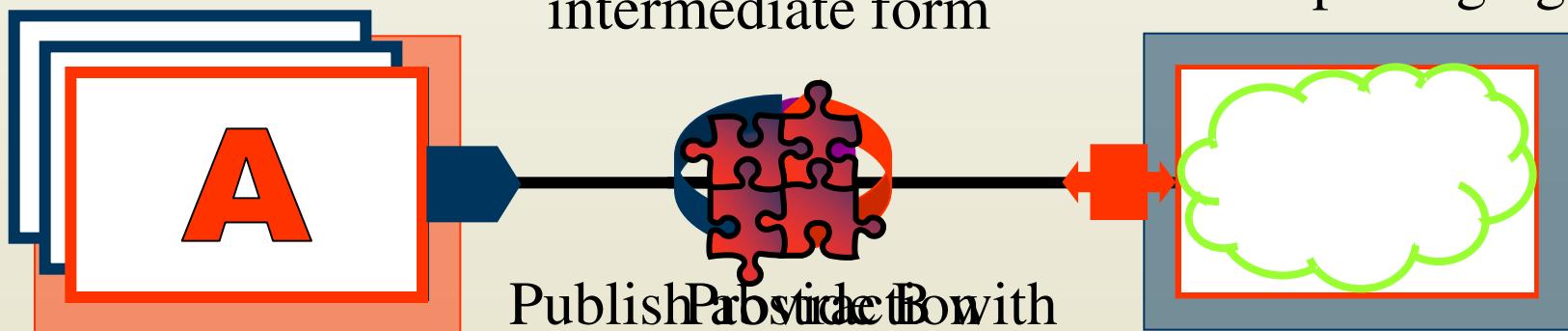
Software Architecture
Lecture 8

Role and Challenge of Software Connectors

How do we enable

Attach adapters to components A and B to ~~interact with B's essence~~
Introduce intermediate form

from its packaging



Maintain multiple versions of A and B
Change versions of A and B
What is the right answer?

How Does One Select a Connector?

- Determine a system's interconnection and interaction needs
 - ◆ Software interconnection models can help
- Determine roles to be fulfilled by the system's connectors
 - ◆ Communication, coordination, conversion, facilitation
- For each connector
 - ◆ Determine its appropriate type(s)
 - ◆ Determine its dimensions of interest
 - ◆ Select appropriate values for each dimension
- For multi-type, i.e., composite connectors
 - ◆ Determine the atomic connector compatibilities

Simple Example

- System components will execute in two processes on the same host
 - ◆ Mostly intra-process
 - ◆ Occasionally inter-process
- The interaction among the components is synchronous
- The components are primarily computation-intensive
 - ◆ There are some data storage needs, but those are secondary

Simple Example (cont'd)

- Select procedure call connectors for intra-process interaction
- Combine procedure call connectors with distributor connectors for inter-process interaction
 - RPC
- Select the values for the different connector dimensions
 - What are the appropriate values?
 - What values are imposed by your favorite programming language(s)?

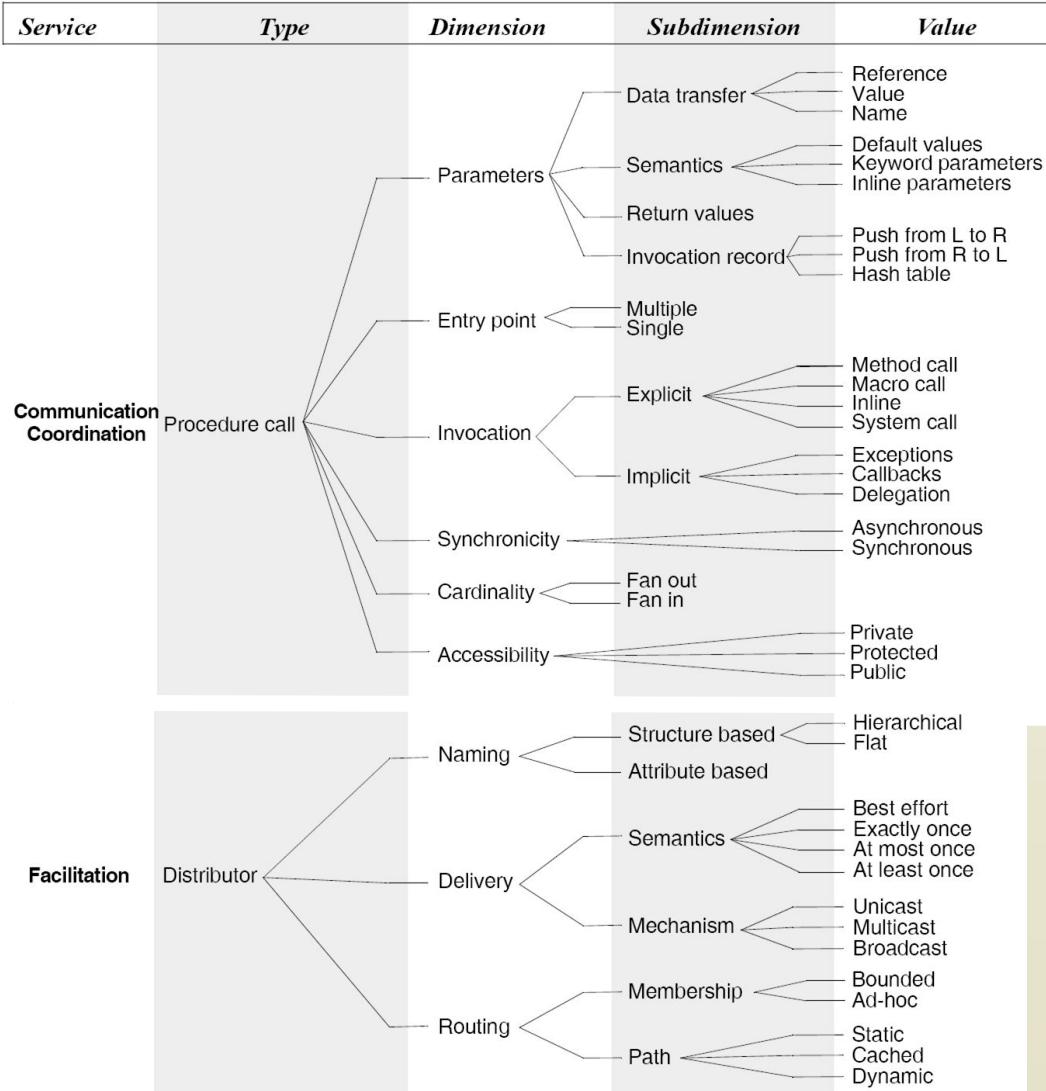
Procedure Call Connectors Revisited

<i>Service</i>	<i>Type</i>	<i>Dimension</i>	<i>Subdimension</i>	<i>Value</i>
Communication Coordination	Procedure call	<ul style="list-style-type: none"> Parameters Entry point Invocation Synchronicity Cardinality Accessibility 	<ul style="list-style-type: none"> Data transfer Semantics Return values Invocation record Multiple Single Explicit Implicit Fan out Fan in Reference Value Name Default values Keyword parameters Inline parameters Push from L to R Push from R to L Hash table Method call Macro call Inline System call Exceptions Callbacks Delegation Asynchronous Synchronous Private Protected Public 	

Distributor Connectors Revisited

<i>Service</i>	<i>Type</i>	<i>Dimension</i>	<i>Subdimension</i>	<i>Value</i>
Facilitation	Distributor	Naming	Structure based Attribute based	Hierarchical Flat
		Delivery	Semantics Mechanism	Best effort Exactly once At most once At least once Unicast Multicast Broadcast
		Routing	Membership Path	Bounded Ad-hoc Static Cached Dynamic

Two Connector Types in Tandem



Select the appropriate values for PC and RPC!

Software Interconnection Models

- Interconnection models (IM) as defined by Perry
 - ◆ Unit interconnection
 - ◆ Syntactic interconnection
 - ◆ Semantic interconnection
- All three are present in each system
- Are all equally appropriate at architectural level?

Unit Interconnection

- Defines relations between system's units
 - ◆ Units are components (modules or files)
 - ◆ Basic unit relationship is dependency
 - $Unit-IM = (\{units\}, \{"depends on"\})$
- Examples
 - ◆ Determining context of compilation
 - e.g., C preprocessor
 - $IM = (\{files\}, \{"include"\})$
 - ◆ Determining recompilation strategies
 - e.g., Make facility
 - $IM = (\{compile_units\}, \{"depends on", "has changed"\})$
 - ◆ System modeling
 - e.g., RCS, DVS, SVS, SCCS
 - $IM = (\{systems, files\}, \{"is composed of"\})$

Unit Interconnection Characteristics

- Coarse-grain interconnections
 - ◆ At level of entire components
- Interconnections are static
- Does not describe component interactions
 - ◆ Focus is exclusively on dependencies

Syntactic Interconnection

- Describes relations among syntactic elements of programming languages
 - ◆ Variable definition/use
 - ◆ Method definition/invocation
 - $IM = (\{methods, types, variables, locations\}, \{"is def at", "is set at", "is used at", "is del from", "is changed to", "is added to"\})$
- Examples
 - ◆ Automated software change management
 - e.g., Interlisp's masterscope
 - ◆ Static analysis
 - e.g., Detection of unreachable code by compilers
 - ◆ Smart recompilation
 - Changes inside unit → recompilation of only the changes
 - ◆ System modeling
 - Finer level of granularity than unit-IM

Syntactic Interconnection Characteristics

- Finer-grain interconnections
 - ◆ At level of individual syntactic objects
- Interconnections are static & dynamic
- Incomplete interconnection specification
 - ◆ Valid syntactic interconnections may not be allowed by semantics
 - ◆ Operation ordering, communication transactions
 - e.g., Pop on an empty stack
 - ◆ Violation of (intended) operation semantics
 - e.g., Trying to use calendar **add** operation to add integers

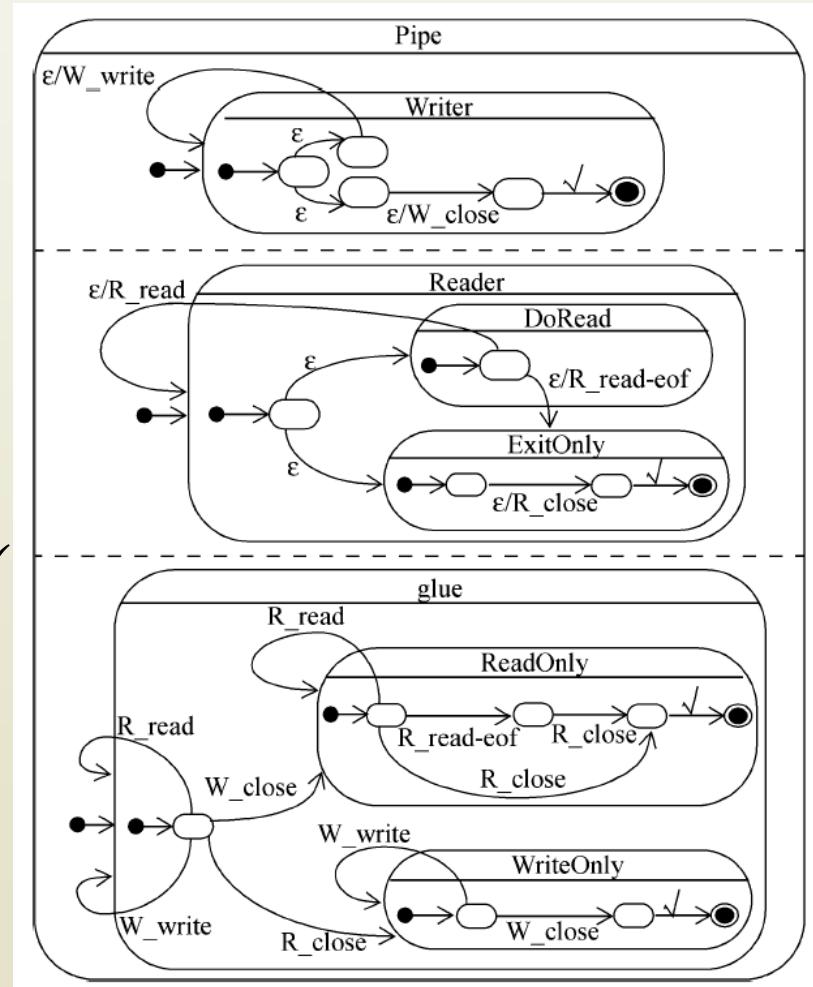
Semantic Interconnection

- Expresses how system components are meant to be used
 - ◆ Component designers' intentions
- Captures how system components are actually used
 - ◆ Component users' (i.e., system builders') intention
- Interconnection semantics can be formally specified
 - ◆ Pre- & post-conditions
 - ◆ Dynamic interaction protocols (e.g. CSP, FSM)
 - $IM = (\{methods, types, variables, \dots, predicates\}, \{“is set at”, “is used at”, “calls”, “called by”, \dots, “satisfies”\})$

Example of Semantic Interconnection

```

connector Pipe =
  role Writer = write → Writer ∏ close → ✓
  role Reader =
    let ExitOnly = close → ✓
    in let DoRead = (read → Reader
      ⊓ read-eof → ExitOnly)
    in DoRead ∏ ExitOnly
  glue = let ReadOnly = Reader.read → ReadOnly
    ⊓ Reader.read-eof
      → Reader.close → ✓
    ⊓ Reader.close → ✓
    in let WriteOnly = Writer.write → WriteOnly
      ⊓ Writer.close → ✓
    in Writer.write → glue
    ⊓ Reader.read → glue
    ⊓ Writer.close → ReadOnly
    ⊓ Reader.close → WriteOnly
  
```



Semantic Interconnection Characteristics

- Builds on syntactic interconnections
- Interconnections are static & dynamic
- Complete interconnection specification
 - ◆ Specifies both syntactic & semantic interconnection validity
- Necessary at level of architectures
 - ◆ Large components
 - ◆ Complex interactions
 - ◆ Heterogeneity
 - ◆ Component reuse
- What about ensuring other properties of interaction?
 - ◆ Robustness, reliability, security, availability, ...

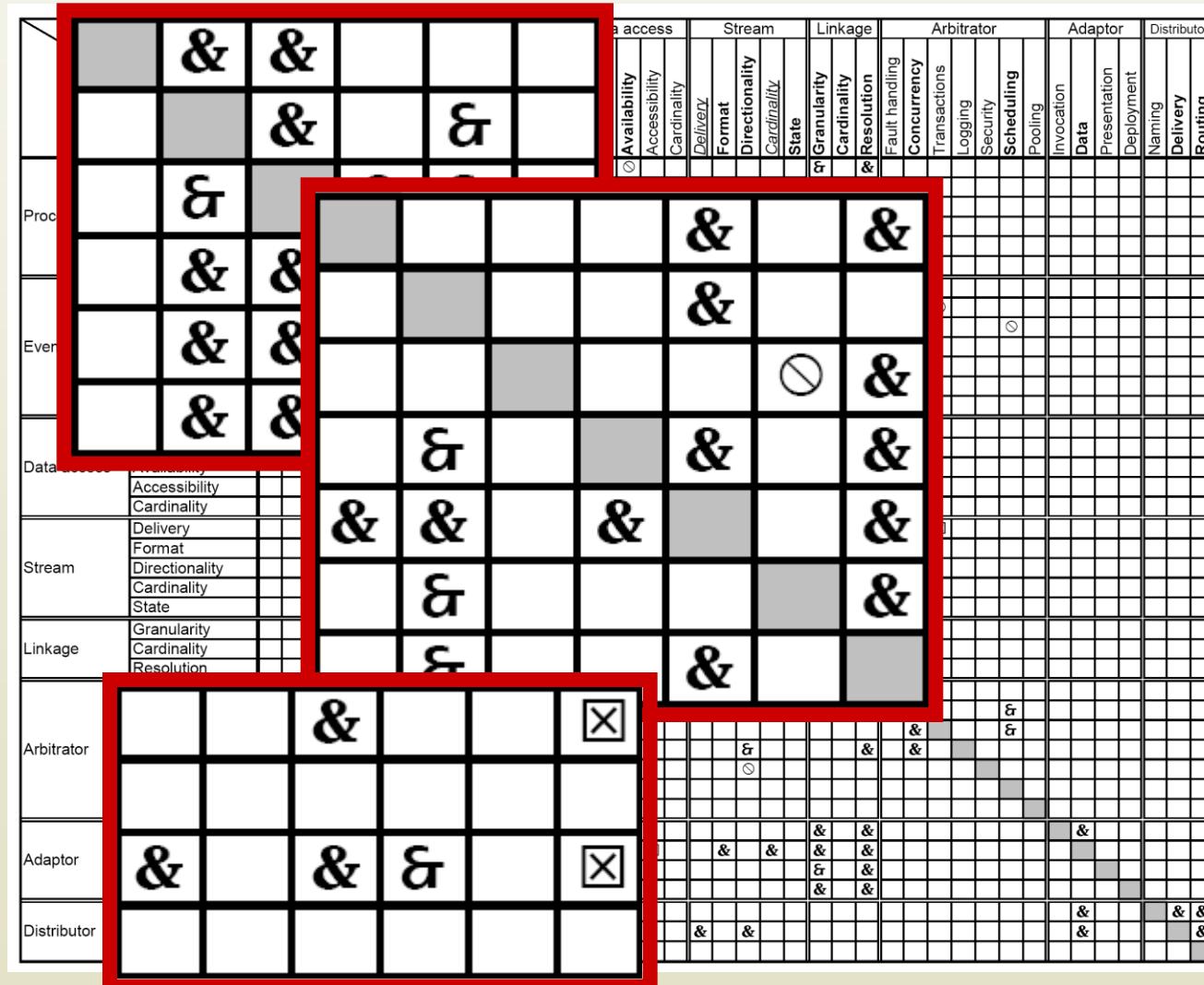
Composing Basic Connectors

- In many systems a connector of multiple types may be required to service (a subset of) the components
- All connectors cannot be composed
 - ◆ Some are naturally interoperable
 - ◆ Some are incompatible
 - ◆ All are likely to require trade-offs
- The composition can be considered at the level of connector type dimensions and subdimensions

Connector Dimension Inter-Relationships

- Requires – 
 - ◆ Choice of one dimension mandates the choice of another
- Prohibits – 
 - ◆ Two dimensions can never be composed into a single connector
- Restricts – 
 - ◆ Dimensions are not always required to be used together
 - ◆ Certain dimension combinations may be invalid
- Cautions – 
 - ◆ Combinations may result in unstable or unreliable connectors

Dimension Inter-Relationships



Well Known Composite Connectors

- Grid connectors (e.g., Globus)
 - ◆ Procedure call
 - ◆ Data access
 - ◆ Stream
 - ◆ Distributor
- Peer-to-peer connectors (e.g., BitTorrent)
 - ◆ Arbitrator
 - ◆ Data access
 - ◆ Stream
 - ◆ Distributor
- Client-server connectors
- Event-based connectors

Introduction to Modeling

**Software Architecture
Lecture 9**

Objectives

- Concepts
 - ◆ What is modeling?
 - ◆ How do we choose what to model?
 - ◆ What kinds of things do we model?
 - ◆ How can we characterize models?
 - ◆ How can we break up and organize models?
 - ◆ How can we evaluate models and modeling notations?
- Examples
 - ◆ Concrete examples of many notations used to model software architectures
 - Revisiting Lunar Lander as expressed in different modeling notations

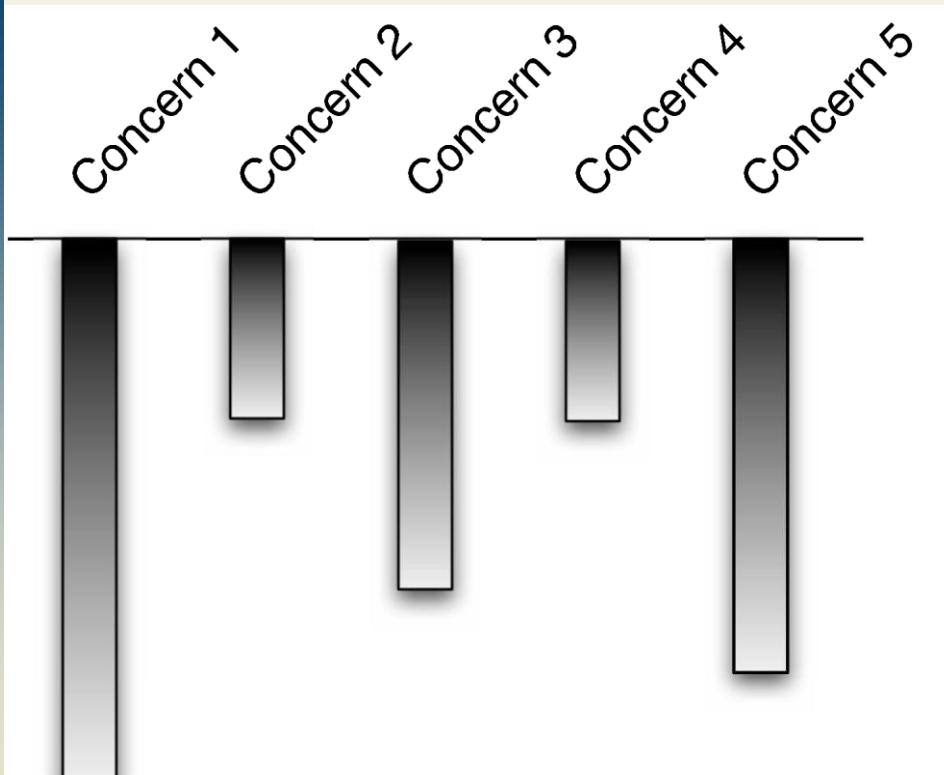
What is Architectural Modeling?

- Recall that we have characterized architecture as *the set of principal design decisions* made about a system
- We can define models and modeling in those terms
 - ◆ An architectural **model** is an artifact that captures some or all of the design decisions that comprise a system's architecture
 - ◆ Architectural **modeling** is the reification and documentation of those design decisions
- How we model is strongly influenced by the notations we choose:
 - ◆ An architectural modeling **notation** is a language or means of capturing design decisions.

How do We Choose What to Model?

- Architects and other stakeholders must make critical decisions:
 1. What architectural decisions and concepts should be modeled,
 2. At what level of detail, and
 3. With how much rigor or formality
- These are cost/benefit decisions
 - ◆ The benefits of creating and maintaining an architectural model must exceed the cost of doing so

Stakeholder-Driven Modeling

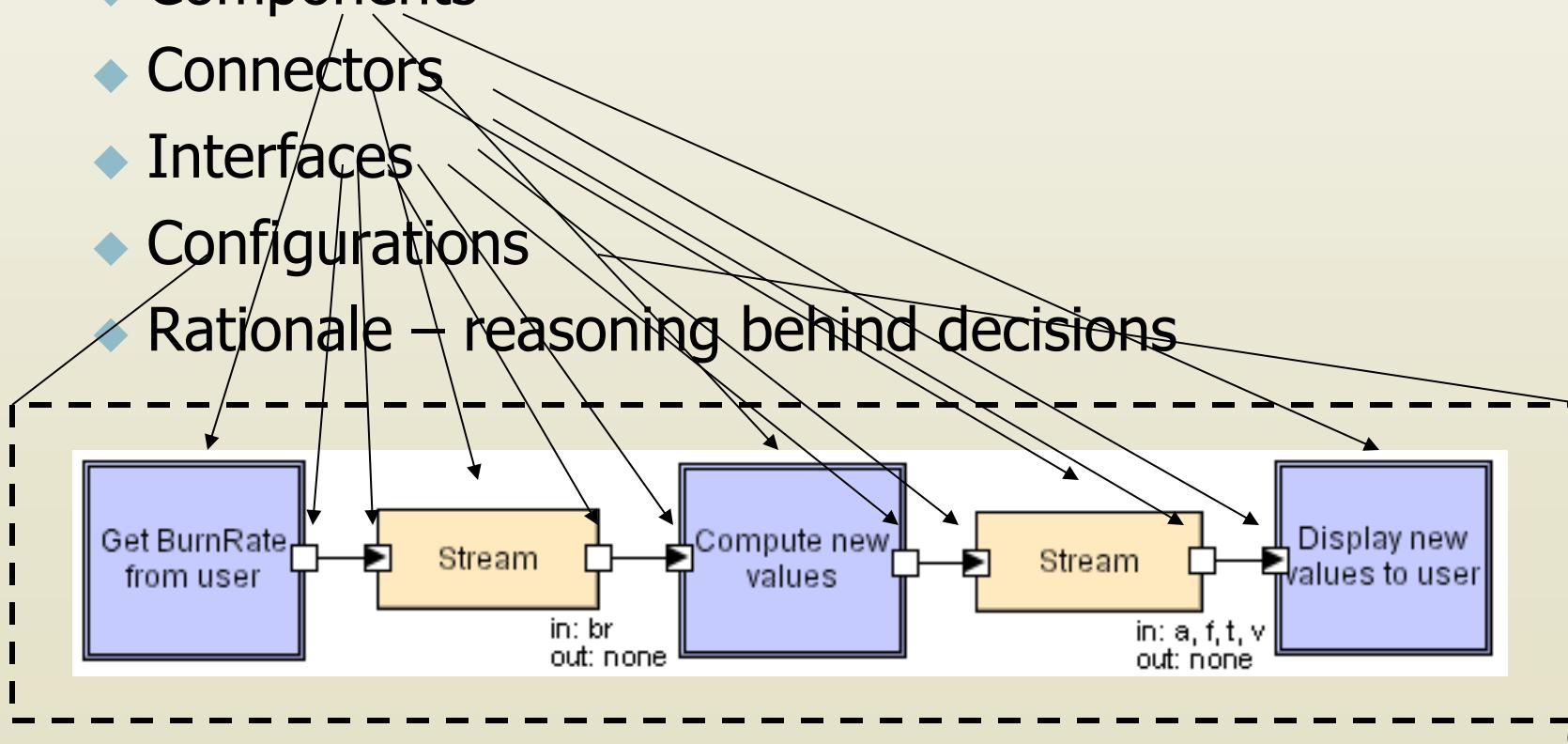


- Stakeholders identify aspects of the system they are concerned about
- Stakeholders decide the relative importance of these concerns
- Modeling depth should roughly mirror the relative importance of concerns

From Maier and Rechtin, “The Art of Systems Architecting” (2000) 5

What do We Model?

- Basic architectural elements
 - ◆ Components
 - ◆ Connectors
 - ◆ Interfaces
 - ◆ Configurations
 - ◆ Rationale – reasoning behind decisions



What do We Model? (cont'd)

- Elements of the architectural style
 - ◆ Inclusion of specific basic elements (e.g., components, connectors, interfaces)
 - ◆ Component, connector, and interface types
 - ◆ Constraints on interactions
 - ◆ Behavioral constraints
 - ◆ Concurrency constraints
 - ◆ ...

What do We Model? (cont'd)

- Static and Dynamic Aspects
 - ◆ Static aspects of a system *do not* change as a system runs
 - e.g., topologies, assignment of components/connectors to hosts, ...
 - ◆ Dynamic aspects *do* change as a system runs
 - e.g., State of individual components or connectors, state of a data flow through a system, ...
 - ◆ This line is often unclear
 - Consider a system whose topology is relatively stable but changes several times during system startup

What do We Model? (cont'd)

- Important distinction between:
 - ◆ Models of dynamic aspects of a system (models do not change)
 - ◆ Dynamic models (the models themselves change)

What do We Model? (cont'd)

- Functional and non-functional aspects of a system
 - ◆ Functional
 - “The system prints medical records”
 - ◆ Non-functional
 - “The system prints medical records *quickly* and *confidentially*.”
- Architectural models tend to be functional, but like rationale it is often important to capture non-functional decisions even if they cannot be automatically or deterministically interpreted or analyzed

Important Characteristics of Models

- Ambiguity
 - ◆ A model is **ambiguous** if it is open to more than one interpretation
- Accuracy and Precision
 - ◆ Different, but often conflated concepts
 - A model is **accurate** if it is correct, conforms to fact, or deviates from correctness within acceptable limits
 - A model is **precise** if it is sharply exact or delimited

Accuracy vs. Precision

Inaccurate and imprecise:
incoherent or contradictory assertions



(a)



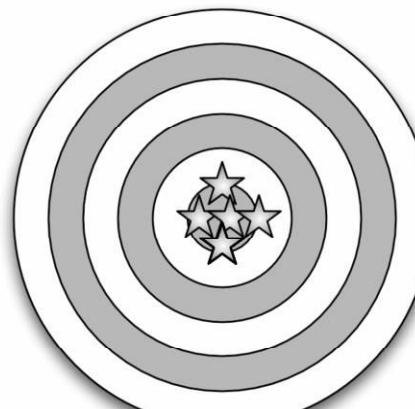
(b)

Inaccurate but precise:
detailed assertions that are wrong



(c)

Accurate but imprecise:
ambiguous or shallow assertions



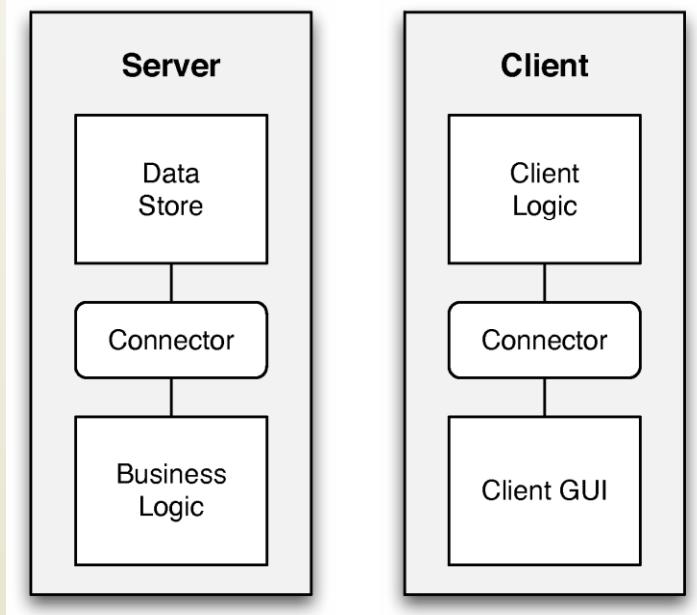
(d)

Accurate and precise:
detailed assertions that are correct

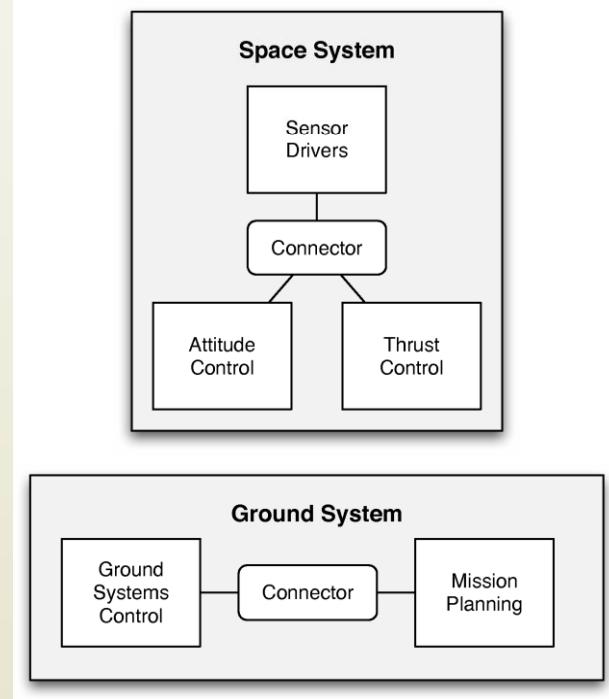
Views and Viewpoints

- Generally, it is not feasible to capture everything we want to model in a single model or document
 - ◆ The model would be too big, complex, and confusing
- So, we create several coordinated models, each capturing a subset of the design decisions
 - ◆ Generally, the subset is organized around a particular concern or other selection criteria
- We call the subset-model a 'view' and the concern (or criteria) a 'viewpoint'

Views and Viewpoints Example



Deployment view of a 3-tier application



Deployment view of a Lunar Lander system

Both instances of the deployment *viewpoint*

Commonly-Used Viewpoints

- Logical Viewpoints
 - ◆ Capture the logical (often software) entities in a system and how they are interconnected.
- Physical Viewpoints
 - ◆ Capture the physical (often hardware) entities in a system and how they are interconnected.
- Deployment Viewpoints
 - ◆ Capture how logical entities are mapped onto physical entities.

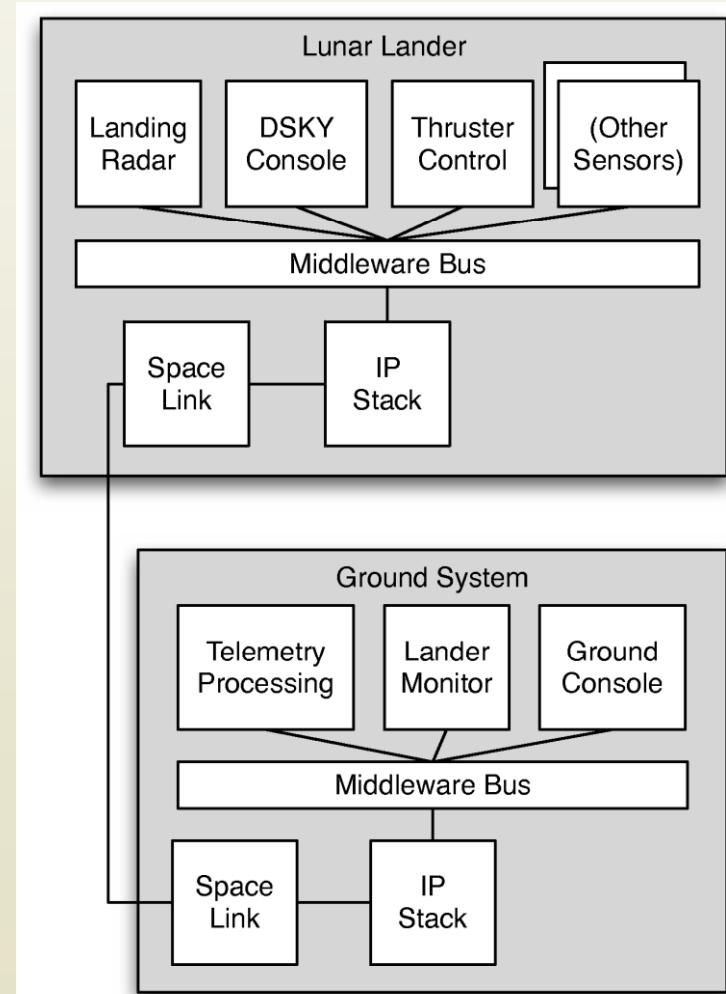
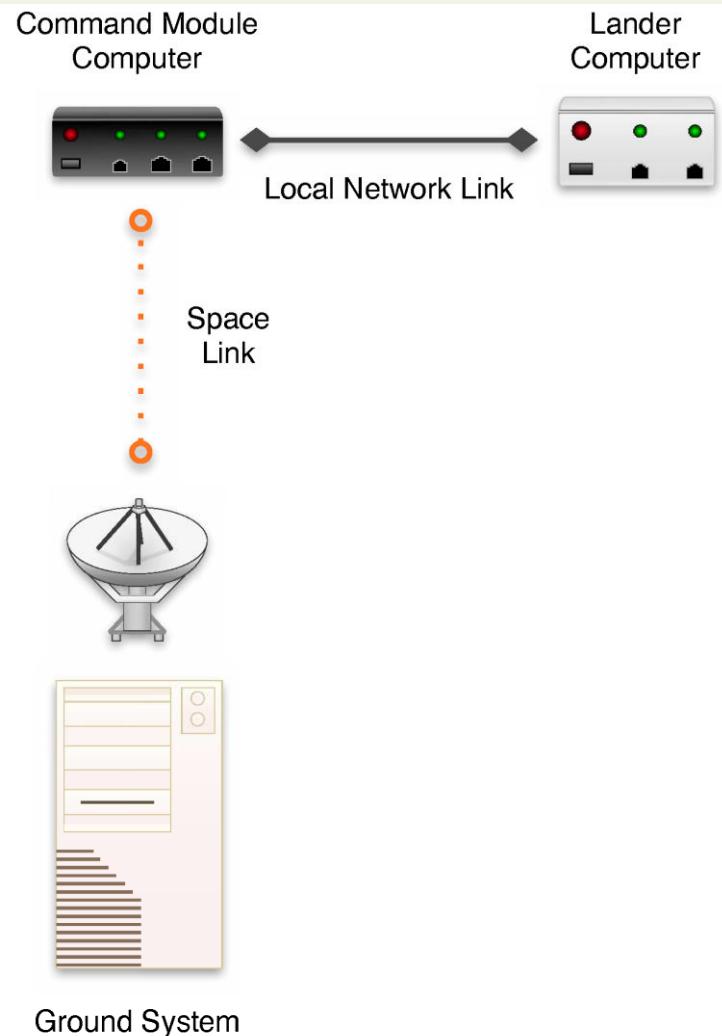
Commonly-Used Viewpoints (cont'd)

- Concurrency Viewpoints
 - ◆ Capture how concurrency and threading will be managed in a system.
- Behavioral Viewpoints
 - ◆ Capture the expected behavior of (parts of) a system.

Consistency Among Views

- Views can contain overlapping and related design decisions
 - ◆ There is the possibility that the views can thus become inconsistent with one another
- Views are **consistent** if the design decisions they contain are compatible
 - ◆ Views are **inconsistent** if two views assert design decisions that cannot simultaneously be true
- Inconsistency is usually but not always indicative of problems
 - ◆ Temporary inconsistencies are a natural part of exploratory design
 - ◆ Inconsistencies cannot always be fixed

Example of View Inconsistency



Common Types of Inconsistencies

- Direct inconsistencies
 - ◆ E.g., “The system runs on two hosts” and “the system runs on three hosts.”
- Refinement inconsistencies
 - ◆ High-level (more abstract) and low-level (more concrete) views of the same parts of a system conflict
- Static vs. dynamic aspect inconsistencies
 - ◆ Dynamic aspects (e.g., behavioral specifications) conflict with static aspects (e.g., topologies)

Common Types of Inconsistencies (cont'd)

- Dynamic vs. dynamic aspect inconsistencies
 - ◆ Different descriptions of dynamic aspects of a system conflict
- Functional vs. non-functional inconsistencies

Evaluating Modeling Approaches

- Scope and purpose
 - ◆ What does the technique help you model? What does it *not* help you model?
- Basic elements
 - ◆ What are the basic elements (the 'atoms') that are modeled? How are they modeled?
- Style
 - ◆ To what extent does the approach help you model elements of the underlying architectural style? Is the technique bound to one particular style or family of styles?

Evaluating Modeling Approaches (cont'd)

- Static and dynamic aspects
 - ◆ What static and dynamic aspects of an architecture does the approach help you model?
- Dynamic modeling
 - ◆ To what extent does the approach support models that change as the system executes?
- Non-functional aspects
 - ◆ To what extent does the approach support (explicit) modeling of non-functional aspects of architecture?

Evaluating Modeling Approaches (cont'd)

- Ambiguity
 - ◆ How does the approach help you to avoid (or embrace) ambiguity?
- Accuracy
 - ◆ How does the approach help you to assess the correctness of models?
- Precision
 - ◆ At what level of detail can various aspects of the architecture be modeled?

Evaluating Modeling Approaches (cont'd)

- Viewpoints
 - ◆ Which viewpoints are supported by the approach?
- Viewpoint Consistency
 - ◆ How does the approach help you assess or maintain consistency among different viewpoints?

Surveying Modeling Approaches

- Generic approaches
 - ◆ Natural language
 - ◆ PowerPoint-style modeling
 - ◆ UML, the Unified Modeling Language
- Early architecture description languages
 - ◆ Darwin
 - ◆ Rapide
 - ◆ Wright
- Domain- and style-specific languages
 - ◆ Koala
 - ◆ Weaves
 - ◆ AADL
- Extensible architecture description languages
 - ◆ Acme
 - ◆ ADML
 - ◆ xADL

Surveying Modeling Approaches (cont'd)

- Generic approaches
 - ◆ Natural language
 - ◆ PowerPoint-style modeling
 - ◆ UML, the Unified Modeling Language
- Early architecture description languages
 - ◆ Darwin
 - ◆ Rapide
 - ◆ Wright
- Domain- and style-specific languages
 - ◆ Koala
 - ◆ Weaves
 - ◆ AADL
- Extensible architecture description languages
 - ◆ Acme
 - ◆ ADML
 - ◆ xADL

Natural Language

- Spoken/written languages such as English
- Advantages
 - ◆ Highly expressive
 - ◆ Accessible to all stakeholders
 - ◆ Good for capturing non-rigorous or informal architectural elements like rationale and non-functional requirements
 - ◆ Plentiful tools available (word processors and other text editors)
- Disadvantages
 - ◆ Ambiguous, non-rigorous, non-formal
 - ◆ Often verbose
 - ◆ Cannot be effectively processed or analyzed by machines/software

Natural Language Example

*“The Lunar Lander application consists of three components: a **data store component**, a **calculation component**, and a **user interface component**.*

*The job of the **data store component** is to store and allow other components access to the height, velocity, and fuel of the lander, as well as the current simulator time.*

*The job of the **calculation component** is to, upon receipt of a burn-rate quantity, retrieve current values of height, velocity, and fuel from the data store component, update them with respect to the input burn-rate, and store the new values back. It also retrieves, increments, and stores back the simulator time. It is also responsible for notifying the calling component of whether the simulator has terminated, and with what state (landed safely, crashed, and so on).*

*The job of the **user interface component** is to display the current status of the lander using information from both the calculation and the data store components. While the simulator is running, it retrieves the new burn-rate value from the user, and invokes the calculation component.”*

Related Alternatives

- Ambiguity can be reduced and rigor can be increased through the use of techniques like 'statement templates,' e.g.:
 - ◆ The (name) interface on (name) component takes (list-of-elements) as input and produces (list-of-elements) as output (synchronously | asynchronously).
 - ◆ This can help to make rigorous data easier to read and interpret, but such information is generally better represented in a more compact format

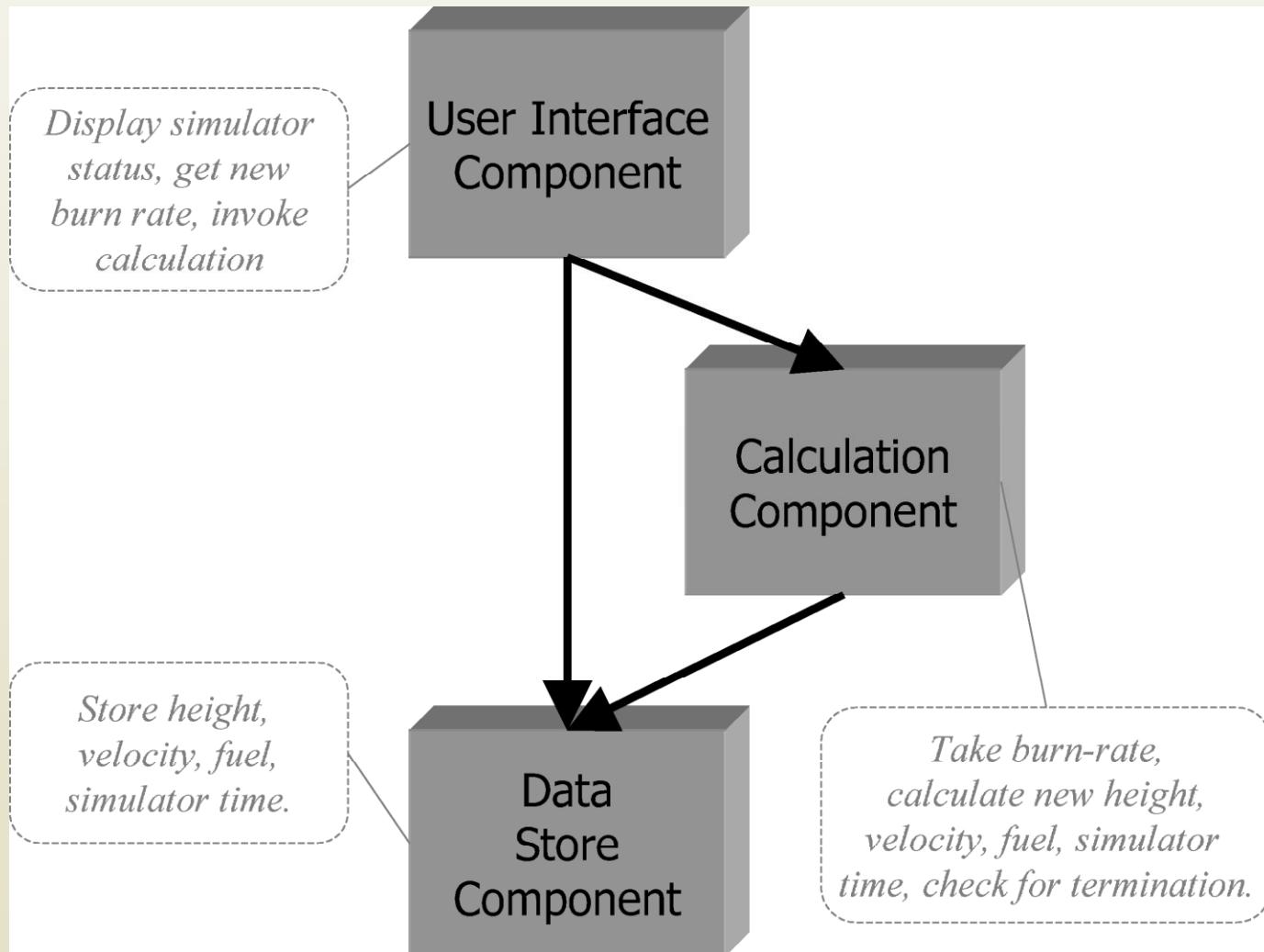
Natural Language Evaluation

- Scope and purpose
 - ◆ Capture design decisions in prose form
- Basic elements
 - ◆ Any concepts required
- Style
 - ◆ Can be described by using more general language
- Static & Dynamic Aspects
 - ◆ Any aspect can be modeled
- Dynamic Models
 - ◆ No direct tie to implemented/ running system
- Non-Functional Aspects
 - ◆ Expressive vocabulary available (but no way to verify)
- Ambiguity
 - ◆ Plain natural language tends to be ambiguous; statement templates and dictionaries help
- Accuracy
 - ◆ Manual reviews and inspection
- Precision
 - ◆ Can add text to describe any level of detail
- Viewpoints
 - ◆ Any viewpoint (but no specific support for any particular viewpoint)
- Viewpoint consistency
 - ◆ Manual reviews and inspection

Informal Graphical Modeling

- General diagrams produced in tools like PowerPoint and OmniGraffle
- Advantages
 - ◆ Can be aesthetically pleasing
 - ◆ Size limitations (e.g., one slide, one page) generally constrain complexity of diagrams
 - ◆ Extremely flexible due to large symbolic vocabulary
- Disadvantages
 - ◆ Ambiguous, non-rigorous, non-formal
 - But often treated otherwise
 - ◆ Cannot be effectively processed or analyzed by machines/software

Informal Graphical Model Example



Related Alternatives

- Some diagram editors (e.g., Microsoft Visio) can be extended with semantics through scripts and other additional programming
 - ◆ Generally ends up somewhere in between a custom notation-specific editor and a generic diagram editor
 - ◆ Limited by extensibility of the tool
- PowerPoint Design Editor (Goldman, Balzer) was an interesting project that attempted to integrate semantics into PowerPoint

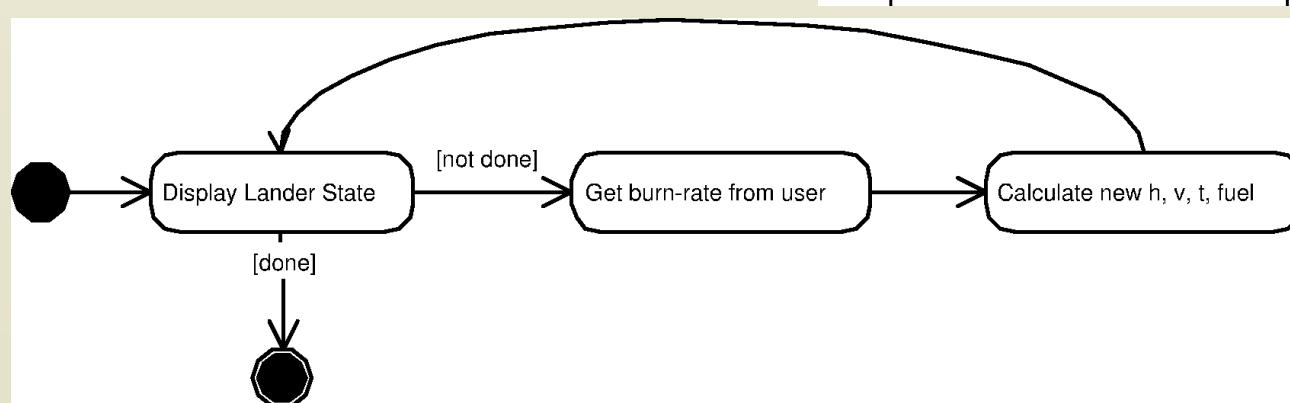
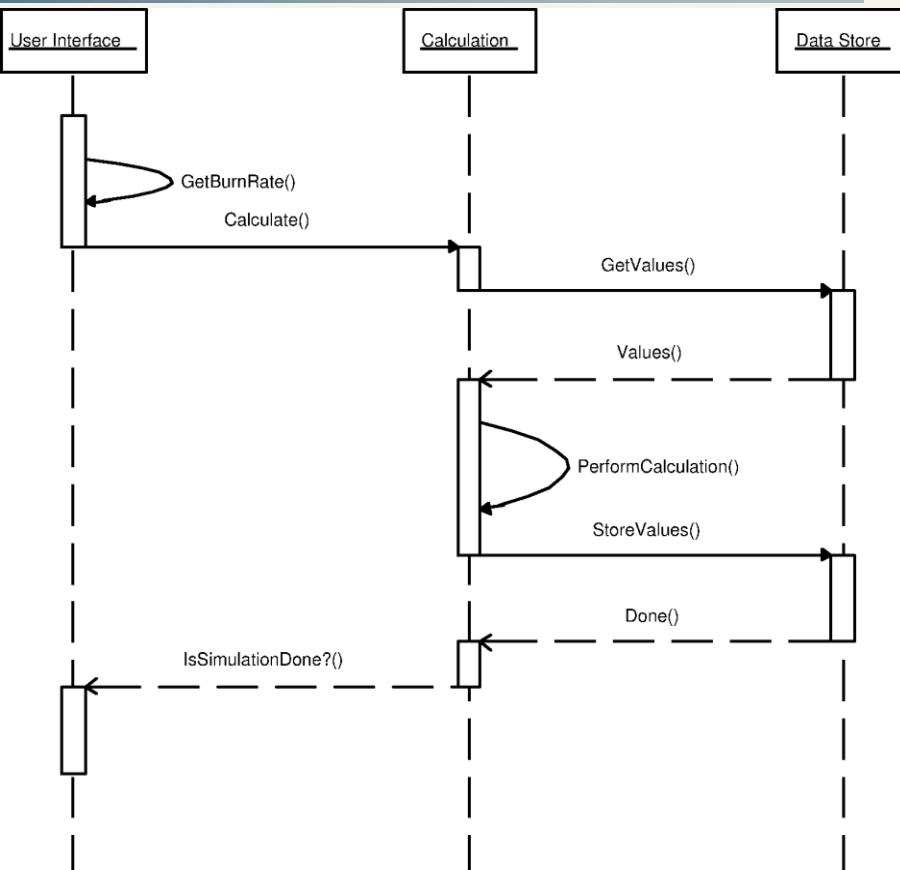
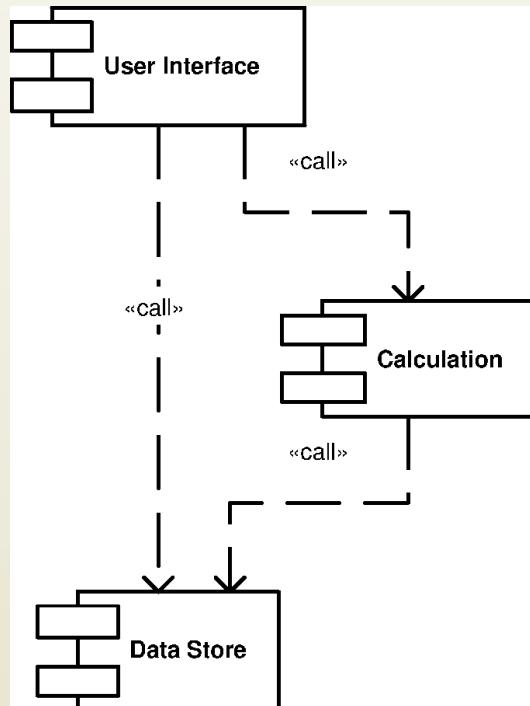
Informal Graphical Evaluation

- Scope and purpose
 - ◆ Arbitrary diagrams consisting of symbols and text
- Basic elements
 - ◆ Geometric shapes, splines, clip-art, text segments
- Style
 - ◆ In general, no support
- Static & Dynamic Aspects
 - ◆ Any aspect can be modeled, but no semantics behind models
- Dynamic Models
 - ◆ Rare, although APIs to manipulate graphics exist
- Non-Functional Aspects
 - ◆ With natural language annotations
- Ambiguity
 - ◆ Can be reduced through use of rigorous symbolic vocabulary/dictionaries
- Accuracy
 - ◆ Manual reviews and inspection
- Precision
 - ◆ Up to modeler; generally canvas is limited in size (e.g., one 'slide')
- Viewpoints
 - ◆ Any viewpoint (but no specific support for any particular viewpoint)
- Viewpoint consistency
 - ◆ Manual reviews and inspection

UML – the Unified Modeling Language

- 13 loosely-interconnected notations called diagrams that capture static and dynamic aspects of software-intensive systems
- Advantages
 - ◆ Support for a diverse array of viewpoints focused on many common software engineering concerns
 - ◆ Ubiquity improves comprehensibility
 - ◆ Extensive documentation and tool support from many vendors
- Disadvantages
 - ◆ Needs customization through profiles to reduce ambiguity
 - ◆ Difficult to assess consistency among views
 - ◆ Difficult to capture foreign concepts or views

UML Example



UML Evaluation

- Scope and purpose
 - ◆ Diverse array of design decisions in 13 viewpoints
- Basic elements
 - ◆ Multitude – states, classes, objects, composite nodes...
- Style
 - ◆ Through (OCL) constraints
- Static & Dynamic Aspects
 - ◆ Some static diagrams (class, package), some dynamic (state, activity)
- Dynamic Models
 - ◆ Rare; depends on the environment
- Non-Functional Aspects
 - ◆ No direct support; natural-language annotations
- Ambiguity
 - ◆ Many symbols are interpreted differently depending on context; profiles reduce ambiguity
- Accuracy
 - ◆ Well-formedness checks, automatic constraint checking, ersatz tool methods, manual
- Precision
 - ◆ Up to modeler; wide flexibility
- Viewpoints
 - ◆ Each diagram type represents a viewpoint; more can be added through overloading/profiles
- Viewpoint consistency
 - ◆ Constraint checking, ersatz tool methods, manual

Modeling and Notations

**Software Architecture
Lecture 10**

Continuing Our Survey

- Generic approaches
 - ◆ Natural language
 - ◆ PowerPoint-style modeling
 - ◆ UML, the Unified Modeling Language
- Early architecture description languages
 - ◆ Darwin
 - ◆ Rapide
 - ◆ Wright
- Domain- and style-specific languages
 - ◆ Koala
 - ◆ Weaves
 - ◆ AADL
- Extensible architecture description languages
 - ◆ Acme
 - ◆ ADML
 - ◆ xADL

Continuing Our Survey

- Generic approaches
 - ◆ Natural language
 - ◆ PowerPoint-style modeling
 - ◆ UML, the Unified Modeling Language
- Early architecture description languages
 - ◆ Darwin
 - ◆ Rapide
 - ◆ Wright
- Domain- and style-specific languages
 - ◆ Koala
 - ◆ Weaves
 - ◆ AADL
- Extensible architecture description languages
 - ◆ Acme
 - ◆ ADML
 - ◆ xADL

Early Architecture Description Languages

- Early ADLs proliferated in the 1990s and explored ways to model different aspects of software architecture
 - ◆ Many emerged from academia
 - ◆ Focus on structure: components, connectors, interfaces, configurations
 - ◆ Focus on formal analysis
 - ◆ None used actively in practice today, tool support has waned
 - Ideas influenced many later systems, though

Darwin

- General purpose language with graphical and textual visualizations focused on structural modeling of systems
- Advantages
 - ◆ Simple, straightforward mechanism for modeling structural dependencies
 - ◆ Interesting way to specify repeated elements through programmatic constructs
 - ◆ Can be modeled in pi-calculus for formal analysis
 - ◆ Can specify hierarchical (i.e., composite) structures
- Disadvantages
 - ◆ Limited usefulness beyond simple structural modeling
 - ◆ No notion of explicit connectors
 - Although components can act as connectors

Darwin Example

```

component DataStore{
    provide landerValues;
}

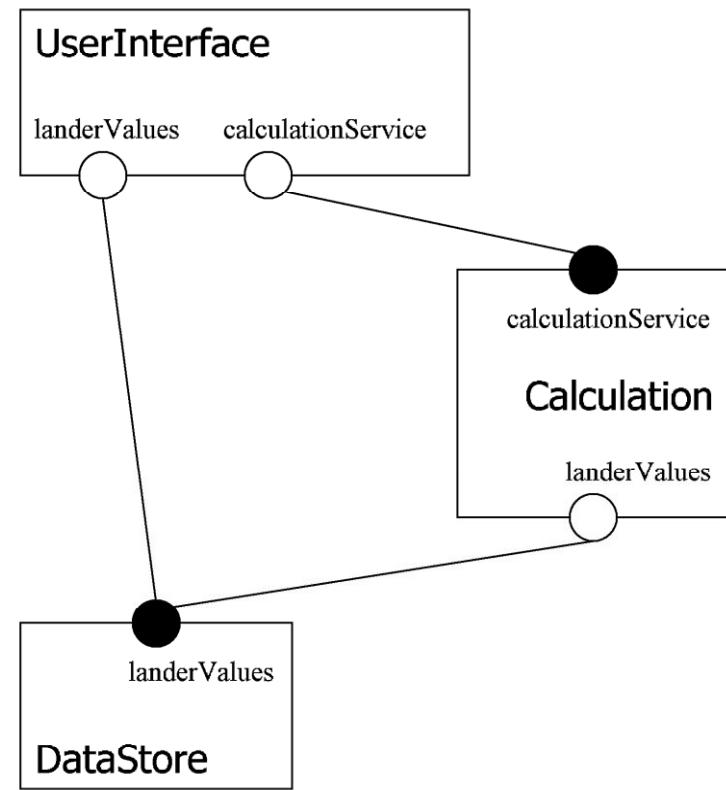
component Calculation{
    require landerValues;
    provide calculationService;
}

component UserInterface{
    require calculationService;
    require landerValues;
}

component LunarLander{
inst
    U: UserInterface;
    C: Calculation;
    D: DataStore;
bind
    C.landerValues -- D.landerValues;
    U.landerValues -- D.landerValues;
    U.calculationService -- C.calculationService;
}

```

LunarLander



Canonical Textual Visualization

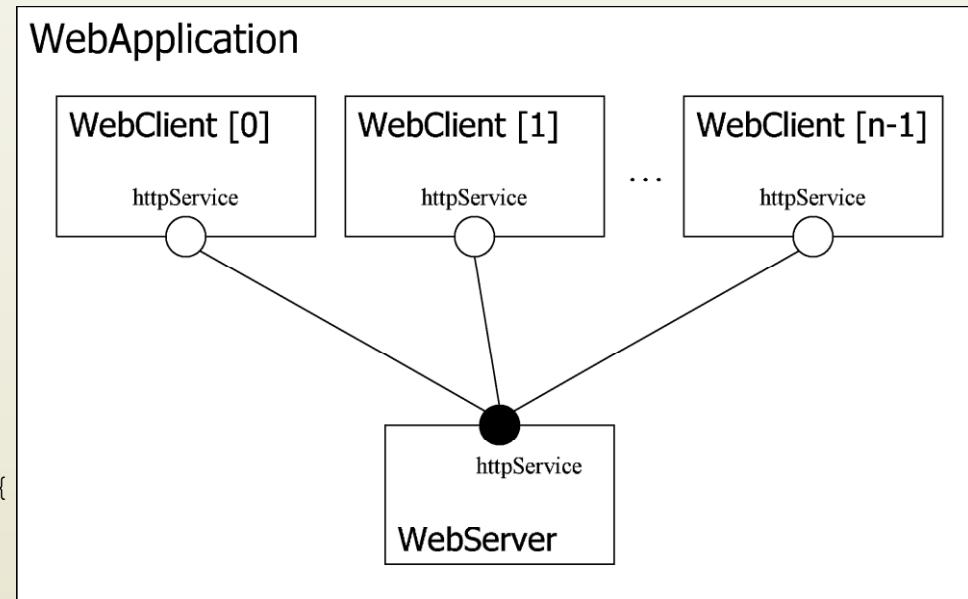
Graphical Visualization

Programmatic Darwin Constructs

```
component WebServer{
    provide httpService;
}

component WebClient{
    require httpService;
}

component WebApplication(int numClients) {
    inst S: WebServer;
    array C[numClients]: WebClient;
    forall k:0..numClients-1{
        inst C[k] @ k;
        bind C[k].httpService -- S.httpService;
    }
}
```



Darwin Evaluation

- Scope and purpose
 - ◆ Modeling software structure
- Basic elements
 - ◆ Components, interfaces, configurations, hierarchy
- Style
 - ◆ Limited support through programmatic constructs
- Static & Dynamic Aspects
 - ◆ Mostly static structure; some additional support for dynamic aspects through lazy and dynamic instantiation/binding
- Dynamic Models
 - ◆ N/A
- Non-Functional Aspects
 - ◆ N/A
- Ambiguity
 - ◆ Rigorous, but structural elements can be interpreted in many ways
- Accuracy
 - ◆ Pi-calculus analysis
- Precision
 - ◆ Modelers choose appropriate level of detail through hierarchy
- Viewpoints
 - ◆ Structural viewpoints
- Viewpoint consistency
 - ◆ N/A

Rapide

- Language and tool-set for exploring dynamic properties of systems of components that communicate through events
- Advantages
 - ◆ Unique and expressive language for describing asynchronously communicating components
 - ◆ Tool-set supports simulation of models and graphical visualization of event traces
- Disadvantages
 - ◆ No natural or explicit mapping to implemented systems
 - ◆ High learning curve
 - ◆ Important tool support is difficult to run on modern machines
 - Has morphed into the CEP project, however

Rapide Example

```
type DataStore is interface
    action in SetValues();
        out NotifyNewValues();
    behavior
    begin
        SetValues => NotifyNewValues();
    end DataStore;

type Calculation is interface
    action in SetBurnRate();
        out DoSetValues();
    behavior
        action CalcNewState();
    begin
        SetBurnRate => CalcNewState(); DoSetValues();
    end Calculation;

type Player is interface
    action out DoSetBurnRate();
        in NotifyNewValues();
    behavior
        TurnsRemaining : var integer := 1;
        action UpdateStatusDisplay();
        action Done();

```

Rapide Example (cont'd)

```

type DataStore is interface
    action in SetValues();
        out NotifyNewValues();
    behavior
    begin
        SetValues => NotifyNewValues();
    end DataStore;

type Calculation is interface
    action in SetBurnRate();
        out DoSetValues();
    behavior
        action CalcNewStat
    begin
        SetBurnRate => CalcNewStat();
    end Calculation;

type Player is interface
    action out DoSetBurnRate();
        in NotifyNewValues();
    behavior
        TurnsRemaining : v;
        action UpdateStatus();
        action Done();
    end Player;

```

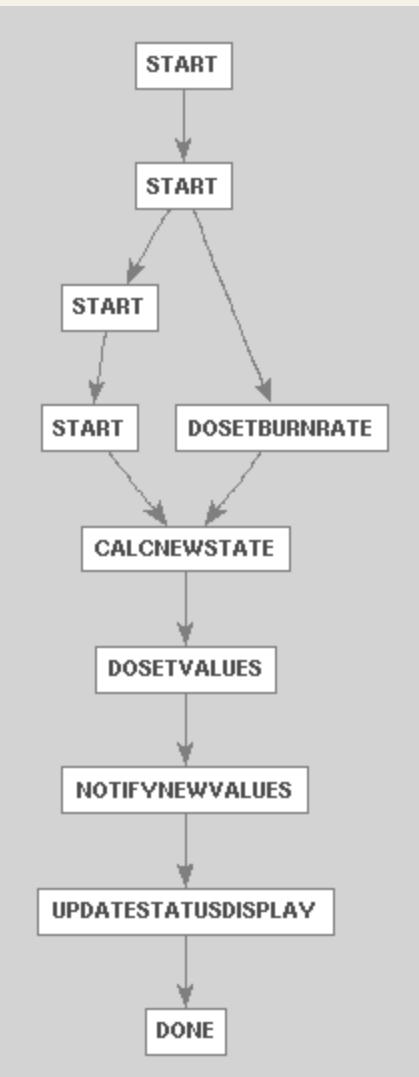
```

begin
    (start or UpdateStatusDisplay) where \
        ($TurnsRemaining > 0) => \
            if ( $TurnsRemaining > 0 ) then \
                TurnsRemaining := $TurnsRemaining - 1; \
                DoSetBurnRate(); \
            end if;;
    NotifyNewValues => UpdateStatusDisplay();;
    UpdateStatusDisplay where $TurnsRemaining == 0 \
        => Done();;
end UserInterface;

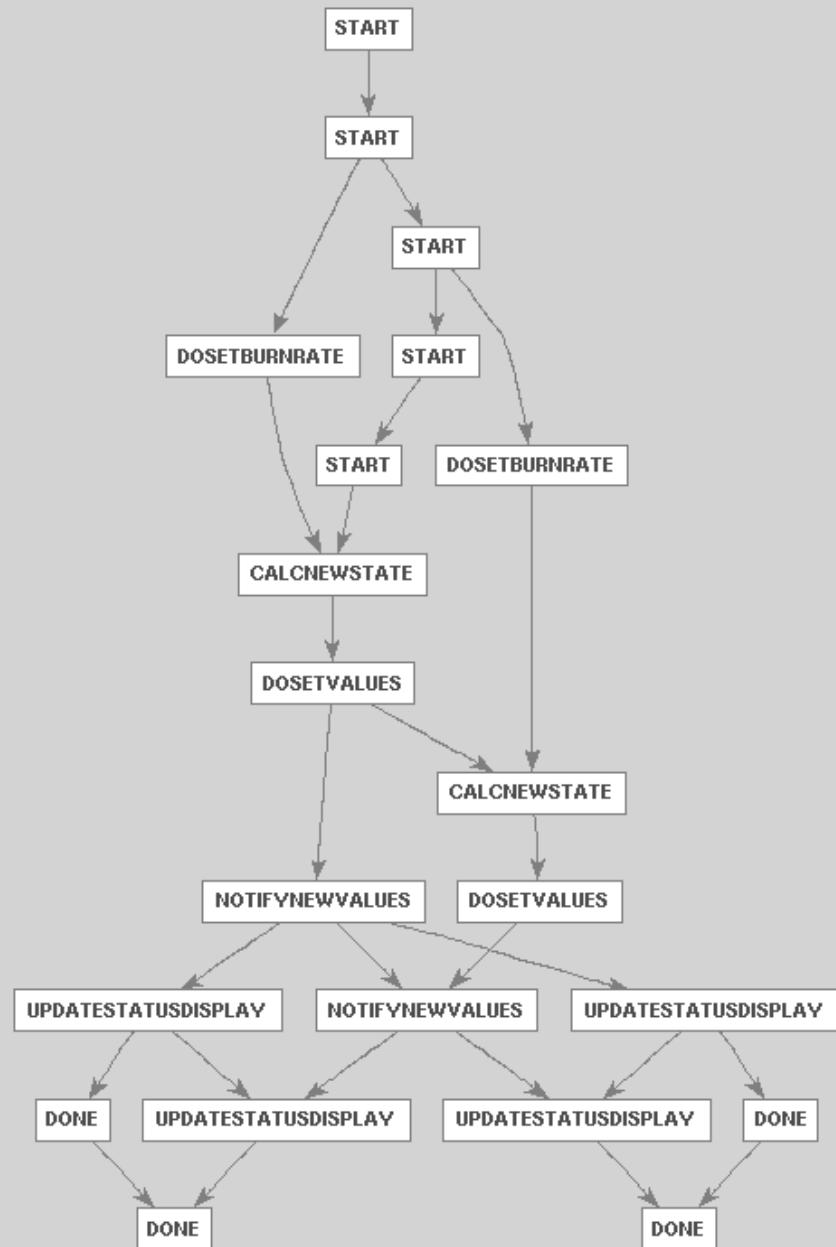
architecture lander() is
    P1, P2 : Player;
    C : Calculation;
    D : DataStore;
connect
    P1.DoSetBurnRate to C.SetBurnRate;
    P2.DoSetBurnRate to C.SetBurnRate;
    C.DoSetValues to D.SetValues;
    D.NotifyNewValues to P1.NotifyNewValues();
    D.NotifyNewValues to P2.NotifyNewValues();
end LunarLander;

```

Simulation Output



1-player
↔
2-player



Rapide Evaluation

- Scope and purpose
 - ◆ Interactions between components communicating with events
- Basic elements
 - ◆ Structures, components/interfaces, behaviors
- Style
 - ◆ N/A
- Static & Dynamic Aspects
 - ◆ Static structure and dynamic behavior co-modeled
- Dynamic Models
 - ◆ Some tools provide limited animation capabilities
- Non-Functional Aspects
 - ◆ N/A
- Ambiguity
 - ◆ Well-defined semantics limit ambiguity
- Accuracy
 - ◆ Compilers check syntax, simulators can be used to check semantics although simulation results are non-deterministic and non-exhaustive
- Precision
 - ◆ Detailed behavioral modeling possible
- Viewpoints
 - ◆ Single structural/behavioral viewpoint
- Viewpoint consistency
 - ◆ N/A

Wright

- An ADL that specifies structure and formal behavioral specifications for interfaces between components and connectors
- Advantages
 - ◆ Structural specification similar to Darwin or Rapide
 - ◆ Formal interface specifications can be translated automatically into CSP and analyzed with tools
 - Can detect subtle problems e.g., deadlock
- Disadvantages
 - ◆ High learning curve
 - ◆ No direct mapping to implemented systems
 - ◆ Addresses a small number of system properties relative to cost of use

Wright Example

Component DataStore

Port getValues (*behavior specification*)
Port storeValues (*behavior specification*)
Computation (*behavior specification*)

Component Calculation

Port getValues (*behavior specification*)
Port storeValues (*behavior specification*)
Port calculate (*behavior specification*)
Computation (*behavior specification*)

Component UserInterface

Port getValues (*behavior specification*)
Port calculate (*behavior specification*)
Computation (*behavior specification*)
call → return → Callee[]§

Connector Call

Caller.call → Callee.call → Glue
Role Caller.call → Callee.call → Glue
Role Callee.return → Caller.return → Glue
[]§

Glue =

Wright Example

Component DataStore

```
Port getValues (behavior specification)
Port storeValues (behavior specification)
Computation (behavior specification)
```

Component Calculation

```
Port getValues (behavior speci
Port storeValues (behavior spe
Port calculate (behavior speci
Computation (behavior specific
```

Component UserInterface

```
Port getValues (behavior speci
Port calculate (behavior speci
Computation (behavior specific
call → return → Caller
call → return → C
```

Connector Call → Callee → Glue
Role Caller = _____
Role Callee = _____
[]§

Glue =

Configuration LunarLander

Instances

```
DS : DataStore
C : Calculation
UI : UserInterface
CtoUIgetValues, CtoUIstoreValues, UItoC, UItoDS : Call
```

Attachments

```
C.getValues as CtoUIgetValues.Caller
DS.getValues as CtoUIgetValues.Callee
```

```
C.storeValues as CtoUIstoreValues.Caller
DS.storeValues as CtoUIstoreValues.Callee
```

```
UI.calculate as UItoC.Caller
C.calculate as UItoC.Callee
```

```
UI.getValues as UItoDS.Caller
DS.getValues as UItoDS.Callee
```

End LunarLander.

Wright Evaluation

- Scope and purpose
 - ◆ Structures, behaviors, and styles of systems composed of components & connectors
 - Basic elements
 - ◆ Components, connectors, interfaces, attachments, styles
 - Style
 - ◆ Supported through predicates over instance models
 - Static & Dynamic Aspects
 - ◆ Static structural models annotated with behavioral specifications
 - Dynamic Models
 - ◆ N/A
 - Non-Functional Aspects
 - ◆ N/A
-
- Ambiguity
 - ◆ Well-defined semantics limit ambiguity
 - Accuracy
 - ◆ Wright models can be translated into CSP for automated analysis
 - Precision
 - ◆ Detailed behavioral modeling possible
 - Viewpoints
 - ◆ Single structural/behavioral viewpoint plus styles
 - Viewpoint consistency
 - ◆ Style checking can be done automatically

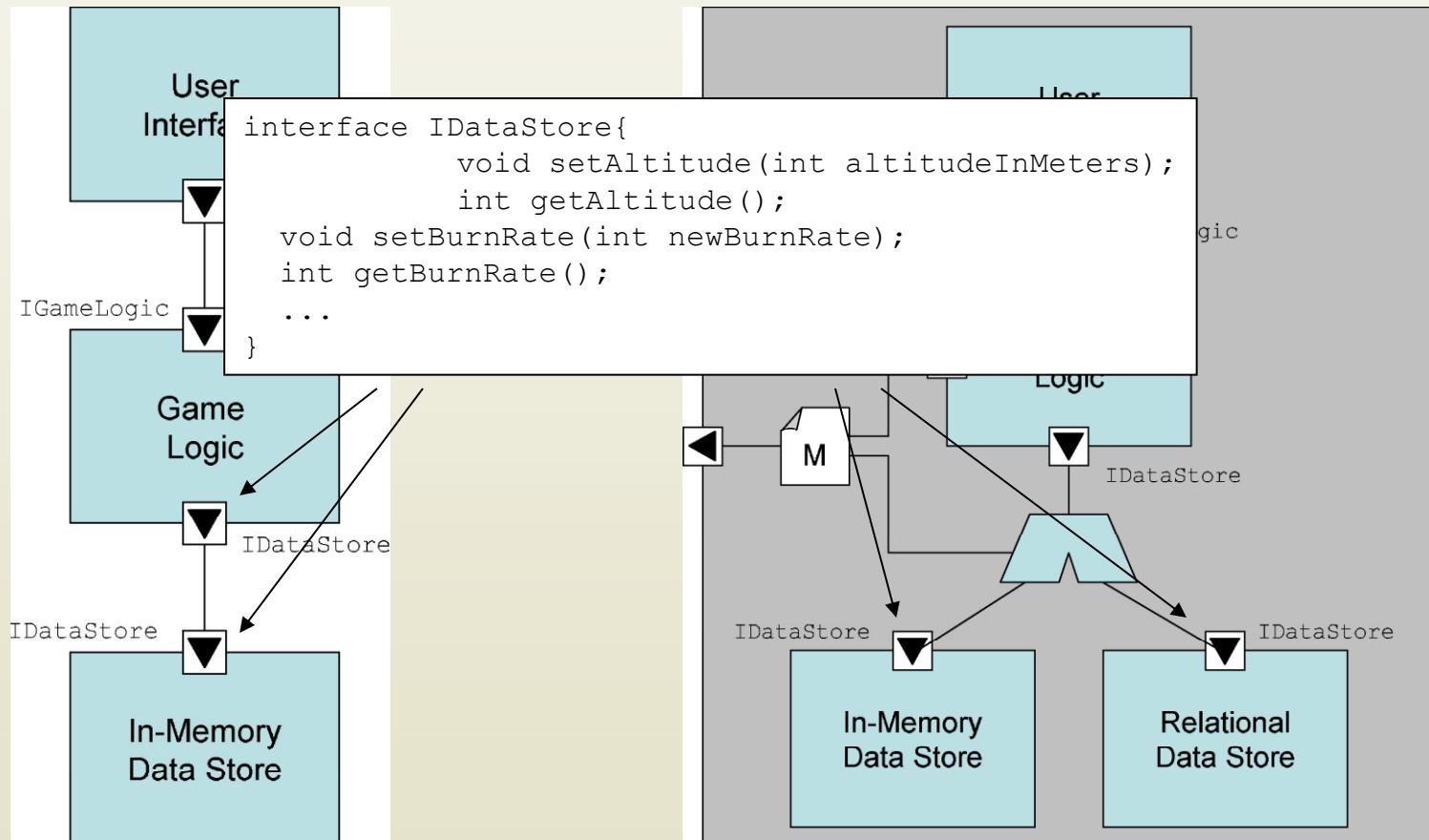
Domain- and Style-Specific ADLs

- Notations we have surveyed thus far have been generically applicable to many types of software systems
- If you restrict the target domain, you can provide more advanced features and/or reduce complexity
 - ◆ We'll talk a lot more about domain-specific software engineering later in the course

Koala

- Darwin-inspired notation for specifying product lines of embedded consumer-electronics devices
- Advantages
 - ◆ Advanced product-line features let you specify many systems in a single model
 - ◆ Direct mapping to implemented systems promotes design and code reuse
- Disadvantages
 - ◆ Limited to structural specification with additional focus on interfaces

Koala Example



Single system

Product line of two systems₂₀

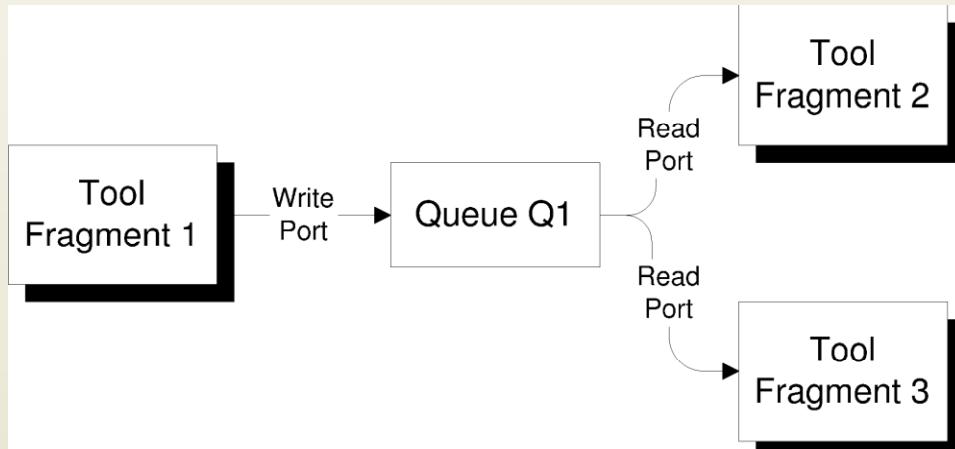
Koala Evaluation

- Scope and purpose
 - ◆ Structures and interfaces of product lines of component-based systems
 - Basic elements
 - ◆ Components, interfaces, elements for variation points: switches, diversity interfaces, etc.
 - Style
 - ◆ Product lines might be seen as very narrow styles
 - Static & Dynamic Aspects
 - ◆ Static structure only
 - Dynamic Models
 - ◆ N/A
 - Non-Functional Aspects
 - ◆ N/A
-
- Ambiguity
 - ◆ Close mappings to implementation limit ambiguity
 - Accuracy
 - ◆ Close mappings to implementations should reveal problems
 - Precision
 - ◆ Structural decisions are fully enumerated but other aspects left out
 - Viewpoints
 - ◆ Structural viewpoint with explicit points of variation
 - Viewpoint consistency
 - ◆ N/A

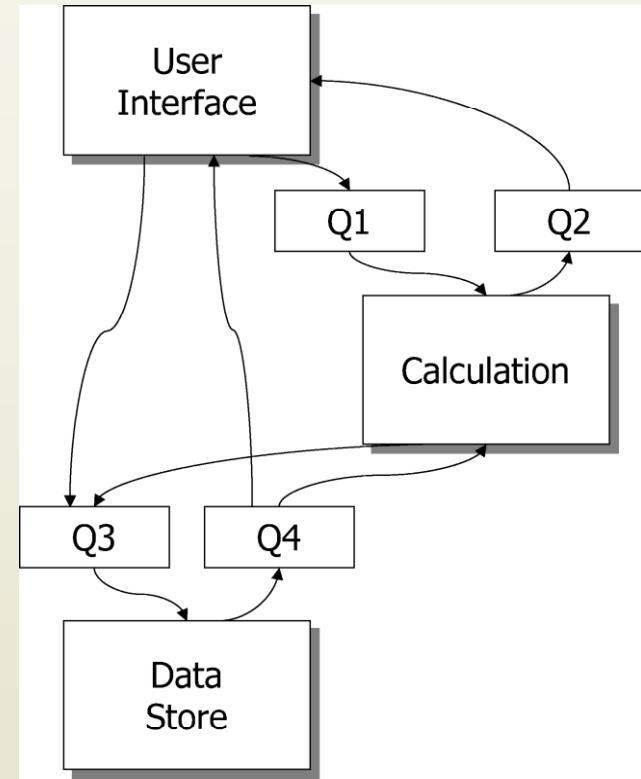
Weaves

- An architectural style and notation for modeling systems of small-grain tool fragments that communicate through data flows of objects
- Advantages
 - ◆ Extremely optimized notation
 - Even simpler than Darwin diagrams
 - ◆ Close mapping to implemented systems
- Disadvantages
 - ◆ Addresses structure and data flows only

Weaves Example



Generic Weaves
System



Lunar Lander in
Weaves

Augmenting Weaves

- Weaves diagrams do not capture the protocol or kinds of data that flow across component boundaries
- This could be rectified through, for example, additional natural language or more formal (e.g., CSP) protocol specifications

The connection from User Interface to Calculation (via Q1) carries objects that include a burn-rate and instruct the calculation component to calculate a new Lander state.

The connection from Calculation to User Interface (via Q2) indicates when the calculation is complete and also includes the termination state of the application.

The connections from User Interface and Calculation to Data Store (via Q3) carry objects that either update or query the state of the Lander.

The connections back to User Interface and Calculation from Data Store (via Q4) carry objects that contain the Lander state, and are sent out whenever the state of the Lander is updated.

Weaves Evaluation

- Scope and purpose
 - ◆ Structures of components and connectors in the Weaves style
- Basic elements
 - ◆ Components, queues, directed interconnections
- Style
 - ◆ Weaves style implicit
- Static & Dynamic Aspects
 - ◆ Static structure only
- Dynamic Models
 - ◆ N/A, although there is a 1-1 correspondence between model and implementation elements
- Non-Functional Aspects
 - ◆ N/A
- Ambiguity
 - ◆ Meanings of Weaves elements are well-defined although important elements (e.g., protocols) are subject to interpretation
- Accuracy
 - ◆ Syntactic (e.g., structural) errors easy to identify
- Precision
 - ◆ Structural decisions are fully enumerated but other aspects left out
- Viewpoints
 - ◆ Structural viewpoint
- Viewpoint consistency
 - ◆ N/A

AADL: The Architecture Analysis & Design Language

- Notation and tool-set for modeling hardware/software systems, particularly embedded and real-time systems
- Advantages
 - ◆ Allows detailed specification of both hardware and software aspects of a system
 - ◆ Automated analysis tools check interesting end-to-end properties of system
- Disadvantages
 - ◆ Verbose; large amount of detail required to capture even simple systems
 - ◆ Emerging tool support and UML profile support

AADL (Partial) Example

```
data lander_state_data
end lander_state_data;
bus lan_bus_type
end lan_bus_type;

bus implementation lan_bus_type.ethernet
properties
    Transmission_Time => 1 ms .. 5 ms;
    Allowed_Message_Size => 1 b .. 1 kb;
end lan_bus_type.ethernet;
system calculation_type
features
    network : requires bus access
        lan_bus.calculation_to_datastore;
    request_get : out event port;
    response_get : in event data port lander_state_data;
    request_store : out event port lander_state_data;
    response_store : in event port;
end calculation_type;

system implementation calculation_type.calculation
subcomponents
    the_calculation_processor :
        processor calculation_processor_type;
    the_calculation_process : process
        calculation_process_type.one_thread;
```

AADL (Partial) Example

```

data lander_state_dat connections
end lander_state_data bus access network -> the_calculation_processor.network;
bus lan_bus_type event data port response_get ->
end lan_bus_type; the_calculation_process.response_get;
bus implementation lan event port the_calculation_process.request_get ->
properties request_get;
event data port response_store ->
Transmission_Time = the_calculation_process.response_store;
Allowed_Message_Siz properties
end lan_bus_type.ether Actual_Processor_Binding => reference
system calculation_ty the_calculation_processor applies to
features the_calculation_process;
network : requires end calculation_type.calculation;
lan_bus.ou processor calculation_processor_type
request_get : ou features
response_get : in network : requires bus access
request_store : ou lan_bus.calculation_to_datastore;
response_store : in end calculation_processor_type;
end calculation_type;

system implementation process calculation_process_type
subcomponents
the_calculation_pro features
cess request_get : out event port;
process response_get : in event data port lander_state_data;
the_calculation_pro request_store : out event data port lander_state_data;
cess response_store : in event port;
calculated end calculation_process_type;

```

AADL (Partial) Example

```

data lander_state_data connections
end lander_state_data
bus lan_bus_type
end lan_bus_type;

bus implementation lan
properties
    Transmission_Time =
    Allowed_Message_Siz
end lan_bus_type.ethernet
system calculation_ty
features
    network : requires lan_bus.type;
    request_get : out event port;
    response_get : in event port;
    request_store : out event port;
    response_store : in event port;
end calculation_type;

system implementation
subcomponents
    the_calculation_process
        process
            the_calculation_process
                calculate
                    connections
                        bus acc
                            event d
                                thread calculation_thread_type
                                    features
                                        request_get : out event port;
                                        response_get : in event data port lander_state_data;
                                        request_store : out event data port lander_state_data;
                                        response_store : in event port;
                                    properties
                                        Dispatch_Protocol => periodic;
                                    end calculation_thread_type;
                                process implementation calculation_process_type.one_thread
                                    subcomponents
                                        calculation_thread : thread client_thread_type;
                                    connections
                                        event data port response_get ->
                                            calculation_thread.response_get;
                                        event port calculation_thread.request_get -> request_get;
                                        event port response_store ->
                                            calculation_thread.response_store;
                                        event data port request_store -> request_store;
                                    properties
                                        Dispatch_Protocol => Periodic;
                                        Period => 20 ms;
                                    end calculation_process_type.one_thread;
                                end calculation_process_type;
                            end calculation_thread_type;
                        end calculation_thread;
                    end calculate;
                end the_calculation_process;
            end process;
        end the_calculation_process;
    end process;
end system;

```

AADL Example Explained a Bit

- Note the level of detail at which the system is specified
 - ◆ A component (`calculation_type.calculation`) runs on...
 - ◆ a physical processor (`the_calculation_processor`), which runs...
 - ◆ a process (`calculation_process_type.one_thread`), which in turn contains...
 - ◆ a single thread of control (`calculation_thread`), all of which can make two kinds of request-response calls through...
 - ◆ ports (`request_get/response_get`,
`request_store/response_store`) over...
 - ◆ an Ethernet bus (`lan_bus_type.Ethernet`).
- All connected through composition, port-mapping, and so on
- This detail is what gives AADL its power and analyzability

AADL Evaluation

- Scope and purpose
 - ◆ Interconnected multi-level systems architectures
- Basic elements
 - ◆ Multitude – components, threads, hardware elements, configurations, mappings...
- Style
 - ◆ N/A
- Static & Dynamic Aspects
 - ◆ Primarily static structure but additional properties specify dynamic aspects
- Dynamic Models
 - ◆ N/A
- Non-Functional Aspects
 - ◆ N/A
- Ambiguity
 - ◆ Most elements have concrete counterparts with well-known semantics
- Accuracy
 - ◆ Structural as well as other interesting properties can be automatically analyzed
- Precision
 - ◆ Many complex interconnected levels of abstraction and concerns
- Viewpoints
 - ◆ Many viewpoints addressing different aspects of the system
- Viewpoint consistency
 - ◆ Mappings and refinement can generally be automatically checked or do not overlap

Extensible ADLs

- There is a tension between
 - ◆ The expressiveness of general-purpose ADLs and
 - ◆ The optimization and customization of more specialized ADLs
- How do we get the best of both worlds?
 - ◆ Use multiple notations in tandem
 - (Difficult to keep consistent, often means excessive redundancy)
 - ◆ Overload an existing notation or ADL (e.g., UML profiles)
 - Increases confusion, doesn't work well if the custom features don't map naturally onto existing features
 - ◆ Add additional features we want to an existing ADL
 - But existing ADLs provide little or no guidance for this
- Extensible ADLs attempt to provide such guidance

Acme

- Early general purpose ADL with support for extensibility through properties
- Advantages
 - ◆ Structural specification capabilities similar to Darwin
 - ◆ Simple property structure allows for arbitrary decoration of existing elements
 - ◆ Tool support with AcmeStudio
- Disadvantages
 - ◆ No way to add new views
 - ◆ Property specifications can become extremely complex and have entirely separate syntax/semantics of their own

Acme Example

```
//Global Types
Property Type returnsValueType = bool;
Connector Type CallType = {
    Roles { callerRole; calleeRole; };
    Property returnsValue : returnsValueType;
};

System LunarLander = {
    //Components
    Component DataStore = {
        Ports { getValues; storeValues; }
    };
    Component Calculation = {
        Ports { calculate; getValues; storeValues; }
    };
    Component UserInterface = {
        Ports { getValues; calculate; }
    };
};

// Connectors
Connector UserInterfaceToCalculation : CallType {
    Roles { callerRole; calleeRole; };
    Property returnsValue : returnsValueType = true;
}
Connector UserInterfaceToDataStore : CallType {
    Roles { callerRole; calleeRole; };
    Property returnsValue : returnsValueType = true;
}
```

Acme Example

```
//Global Types
Property Type returnsValueType = bool
Connector Type CallType = {
    Roles { callerRole; calleeRole; };
    Property returnsValue : returnsValueType;
};

System LunarLander = {
    //Components
    Component DataStore = {
        Ports { getValues; storeValues; }
    };
    Component Calculation = {
        Ports { calculate; getValues; storeValues; }
    };
    Component UserInterface = {
        Ports { getValues; calculate; }
    };

    // Connectors
    Connector UserInterfaceToCalculation : CallType {
        Roles { callerRole; calleeRole; };
        Property returnsValue : returnsValueType = false;
    }
    Connector UserInterfaceToDataStore : CallType {
        Roles { callerRole; calleeRole; };
        Property returnsValue : returnsValueType = true;
    }
}
```

```
Connector CalculationToDataStoreS : CallType {
    Roles { callerRole; calleeRole; };
    Property returnsValue : returnsValueType = false;
}
Connector CalculationToDataStoreG : CallType {
    Roles { callerRole; calleeRole; };
    Property returnsValue : returnsValueType = true;
}
Attachments {
    UserInterface.getValues to
        UserInterfaceToDataStore.callerRole;
    UserInterfaceToDataStore.calleeRole to
        DataStore.getValues;
    UserInterface.getValues to
        UserInterfaceToDataStore.callerRole;
    UserInterfaceToDataStore.calleeRole to
        DataStore.getValues;
    UserInterface.calculate to
        UserInterfaceToCalculation.callerRole;
    UserInterfaceToCalculation.calleeRole to
        Calculation.calculate;
    Calculation.storeValues to
        CalculationToDataStoreS.callerRole;
    CalculationToDataStoreS.calleeRole to
        DataStore.storeValues;
    Calculation.getValues to
        CalculationToDataStoreG.callerRole;
    CalculationToDataStoreG.calleeRole to
        DataStore.getValues;
}
```

Software Architecture: Foundations, Theory, and Practice

```
//Global Types
Property Type returnsValueType = bool
Connector Type CallType = {
    Roles { callerRole; calleeRole; };
    Property returnsValue : returnsValueType;
};
```

```
System LunarLander =
    //Components
    Component DataStore {
        Ports { getValues;
    };
    Component Calculation {
        Ports { calculate;
    };
    Component UserInterface {
        Ports { getValues;
    };
    // Connectors
    Connector UserInterfaceToCalculation {
        Roles { callerRole; calleeRole; }
        Property returnsValue : returnsValueType;
    }
    Connector UserInterfaceToDataStore {
        Roles { callerRole; calleeRole; }
        Property returnsValue : returnsValueType;
    }
```

```
Property Type StoreType = enum { file,
    relationalDatabase, objectDatabase };

Component DataStore = {
    Ports {
        getValues; storeValues;
    }
    Property storeType : StoreType =
        relationalDatabase;
    Property tableName : String = "LanderTable";
    Property numReplicas: int = 0;
};
```

```
Connector CalculationToDataStoreS : CallType {
    Roles { callerRole; calleeRole; };
    Property returnsValue : returnsValueType = false;
}
Connector CalculationToDataStoreG : CallType {
    Roles { callerRole; calleeRole; };
    Property returnsValue : returnsValueType = true;
}
```

Attachments

callerRole; calleeRole to
callerRole; calleeRole to
.callerRole; calleeRole to

```
Calculation.storeValues to
    CalculationToDataStoreS.callerRole;
CalculationToDataStoreS.calleeRole to
    DataStore.storeValues;
Calculation.getValues to
    CalculationToDataStoreG.callerRole;
CalculationToDataStoreG.calleeRole to
    DataStore.getValues;
};
```

Acme Evaluation

- Scope and purpose
 - ◆ Structures of components and connectors with extensible properties
 - Basic elements
 - ◆ Components, connectors, interfaces, hierarchy, properties
 - Style
 - ◆ Through type system
 - Static & Dynamic Aspects
 - ◆ Static structure is modeled natively, dynamic aspects in properties
 - Dynamic Models
 - ◆ AcmeLib allows programmatic model manipulation
 - Non-Functional Aspects
 - ◆ Through properties
-
- Ambiguity
 - ◆ Meanings of elements subject to some interpretation, properties may have arbitrary level of rigor/formality
 - Accuracy
 - ◆ Checkable syntactically, via type system, and properties by external tools
 - Precision
 - ◆ Properties can increase precision but cannot add new elements
 - Viewpoints
 - ◆ Structural viewpoint is native, properties might provide additional viewpoints
 - Viewpoint consistency
 - ◆ Via external tools that must be developed

ADML

- Effort to standardize the concepts in Acme and leverage XML as a syntactic base
- Advantages
 - ◆ XML parsers and tools readily available
 - ◆ Added some ability to reason about types of properties with meta-properties
- Disadvantages
 - ◆ Properties are still name-value pairs
 - ◆ Did not take advantage of XML extension mechanisms

ADML Example

- Similar to Acme, except in an XML format

```
<Component ID="datastore" name="Data Store">
  <ComponentDescription>
    <ComponentBody>
      <Port ID="getValues" name="getValues"/>
      <Port ID="storeValues" name="storeValues"/>
    </ComponentBody>
  </ComponentDescription>
</Component>
```

xADL

- Modular XML-based ADL intended to maximize extensibility both in notation and tools
- Advantages
 - ◆ Growing set of generically useful modules available already
 - ◆ Tool support in ArchStudio environment
 - ◆ Users can add their own modules via well-defined extensibility mechanisms
- Disadvantages
 - ◆ Extensibility mechanisms can be complex and increase learning curve
 - ◆ Heavy reliance on tools

xADL Example

```
<types:component xsi:type="types:Component"
    types:id="myComp">
  <types:description xsi:type="instance:Description">
    MyComponent
  </types:description>
  <types:interface xsi:type="types:Interface"
    types:id="iface1">
    <types:description xsi:type="instance:Description">
      Interface1
    </types:description>
    <types:direction xsi:type="instance:Direction">
      inout
    </types:direction>
  </types:interface>
</types:component>
```

xADL Example

```
<types:component xsi:type="types:Component"
    types:id="myComp">
    <types:description xsi:type="instance:Description">
        MyComponent
    </types:description>
    <types:interface xsi:type="types:Interface"
        types:id="iface1">
        <types:description xsi:type="instance:Description">
            Interface1
        </types:description>
        <types:direction xsi:type="i
            inout
        </types:direction>
    </types:interface>
</types:component>
```

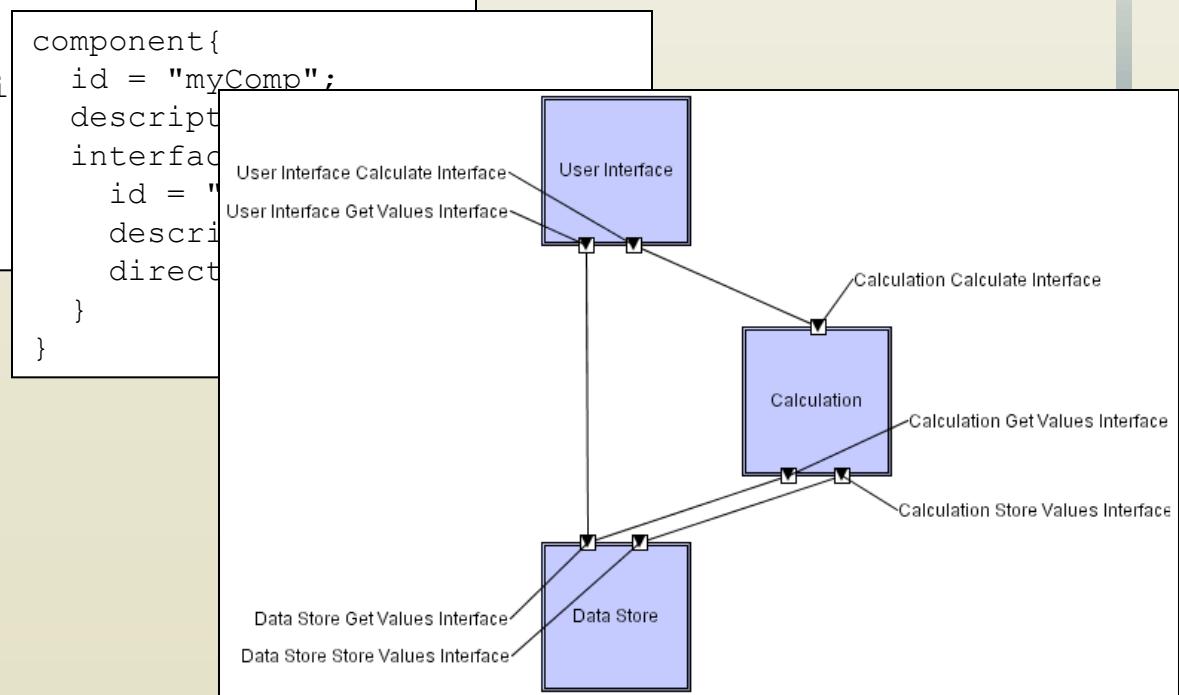
```
component{
    id = "myComp";
    description = "MyComponent";
    interface{
        id = "iface1";
        description = "Interface1";
        direction = "inout";
    }
}
```

xADL Example

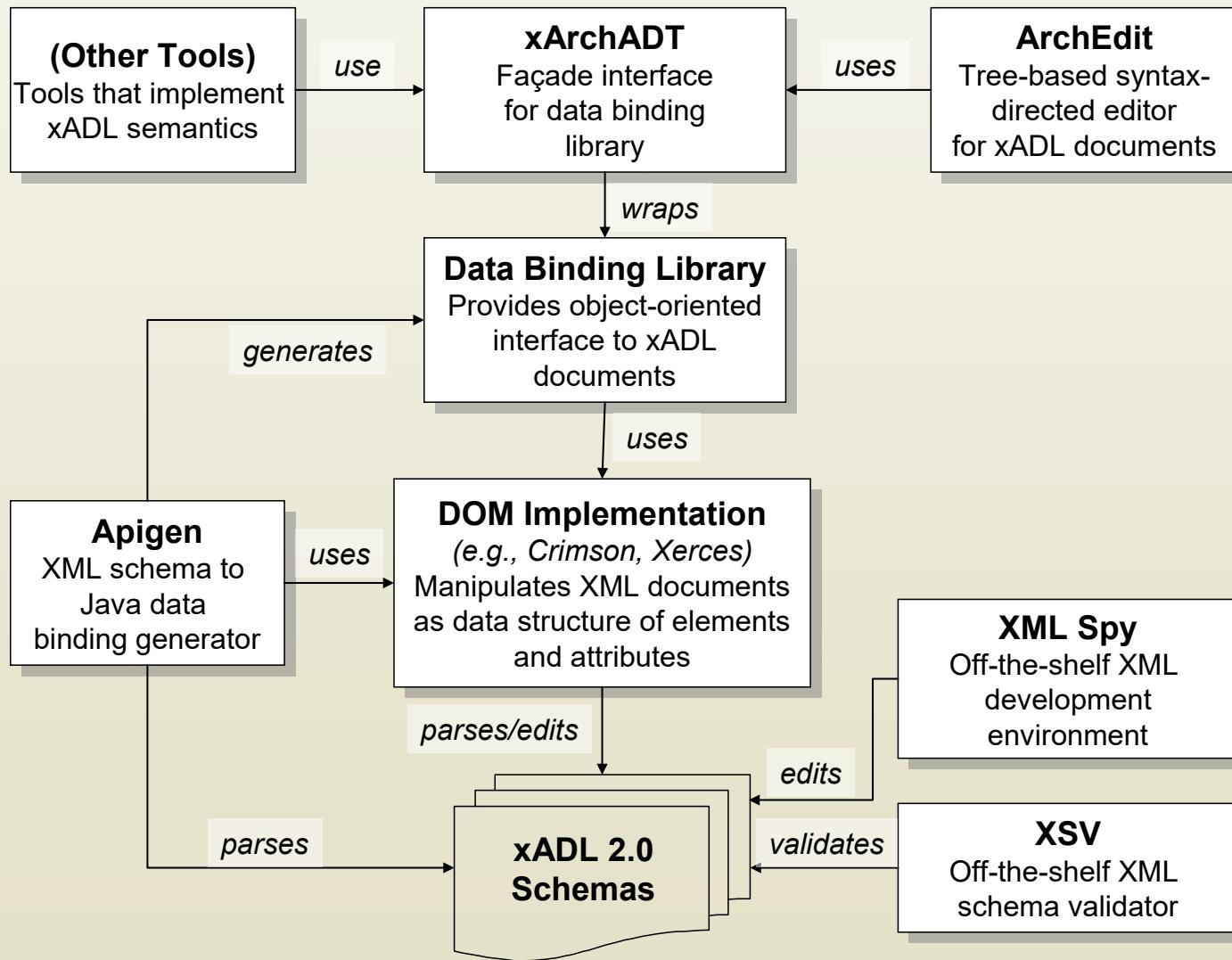
```

<types:component xsi:type="types:Component"
    types:id="myComp">
    <types:description xsi:type="instance:Description">
        MyComponent
    </types:description>
    <types:interface xsi:type="types:Interface"
        types:id="iface1">
        <types:description xsi:type="instance:Description">
            Interface1
        </types:description>
        <types:direction xsi:type="i
            inout
        </types:direction>
    </types:interface>
</types:component>

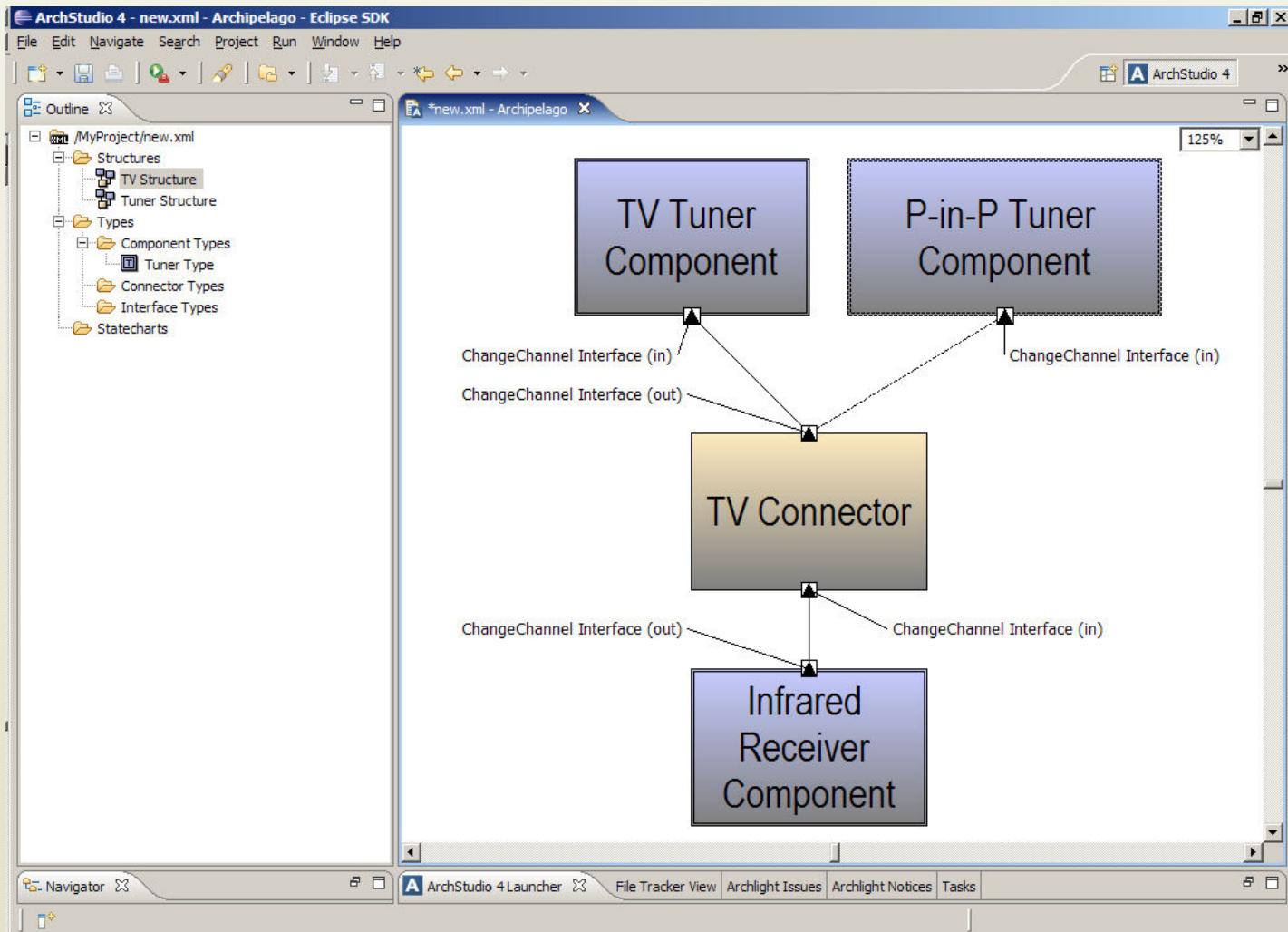
```



xADL Tools



ArchStudio Environment



xADL Schemas (Modules)

Schema	Features
Structure & Types	Defines basic structural modeling of prescriptive architectures: components, connectors, interfaces, links, general groups, as well as types for components, connectors, and interfaces.
Instances	Basic structural modeling of descriptive architectures: components, connectors, interfaces, links, general groups.
Abstract Implementation	Mappings from structural element types (component types, connector types) to implementations.
Java Implementation	Mappings from structural element types to Java implementations.
Options	Allows structural elements to be declared optional— included or excluded from an architecture depending on specified conditions.
Variants	Allows structural element types to be declared variant—taking on different concrete types depending on specified conditions.
Versions	Defines version graphs; allows structural element types to be versioned through association with versions in version graphs.

xADL Evaluation

- Scope and purpose
 - ◆ Modeling various architectural concerns with explicit focus on extensibility
- Basic elements
 - ◆ Components, connectors, interfaces, links, options, variants, versions, ..., plus extensions
- Style
 - ◆ Limited, through type system
- Static & Dynamic Aspects
 - ◆ Mostly static views with behavior and dynamic aspects provided through extensions
- Dynamic Models
 - ◆ Models can be manipulated programmatically
- Non-Functional Aspects
 - ◆ Through extensions
- Ambiguity
 - ◆ Base schemas are permissive; extensions add rigor or formality if needed
- Accuracy
 - ◆ Correctness checkers included in ArchStudio and users can add additional tools through well-defined mechanisms
- Precision
 - ◆ Base schemas are abstract, precision added in extensions
- Viewpoints
 - ◆ Several viewpoints provided natively, new viewpoints through extensions
- Viewpoint consistency
 - ◆ Checkable through external tools and additional consistency rules

Caveat 1

- The preceding overview optimized for breadth rather than depth
 - ◆ Semantics and capabilities of many of these notations quite deep and subtle
 - Some even have entire books written about them
 - ◆ You are encouraged to investigate individual notations more deeply

Caveat 2

- Some systems are difficult to model in traditional ways
 - ◆ 'Agile' systems that are not explicitly designed above the level of code modules
 - ◆ Extremely large, complex, or dynamic systems (e.g., the Web)
 - Can model limited or less complex aspects
 - Can model one instance of the system (e.g., one Web application)
 - Exploit regularity
 - Model the style
 - Model the protocols

Visualizing Software Architectures

Software Architecture
Lecture 11

Objectives

- Concepts
 - ◆ What is visualization?
 - ◆ Differences between modeling and visualization
 - ◆ What kinds of visualizations do we use?
 - ◆ Visualizations and views
 - ◆ How can we characterize and evaluate visualizations?
- Examples
 - ◆ Concrete examples of a diverse array of visualizations
- Constructing visualizations
 - ◆ Guidelines for constructing new visualizations
 - ◆ Pitfalls to avoid when constructing new visualizations
 - ◆ Coordinating visualizations

Objectives

- Concepts
 - ◆ What is visualization?
 - ◆ Differences between modeling and visualization
 - ◆ What kinds of visualizations do we use?
 - ◆ Visualizations and views
 - ◆ How can we characterize and evaluate visualizations?
- Examples
 - ◆ Concrete examples of a diverse array of visualizations
- Constructing visualizations
 - ◆ Guidelines for constructing new visualizations
 - ◆ Pitfalls to avoid when constructing new visualizations
 - ◆ Coordinating visualizations

What is Architectural Visualization?

- Recall that we have characterized architecture as *the set of principal design decisions* made about a system
- Recall also that models are artifacts that capture some or all of the design decisions that comprise an architecture
- An architectural **visualization** defines how architectural models are depicted, and how stakeholders interact with those depictions
 - ◆ Two key aspects here:
 - **Depiction** is a picture or other visual representation of design decisions
 - **Interaction** mechanisms allow stakeholders to interact with design decisions in terms of the depiction

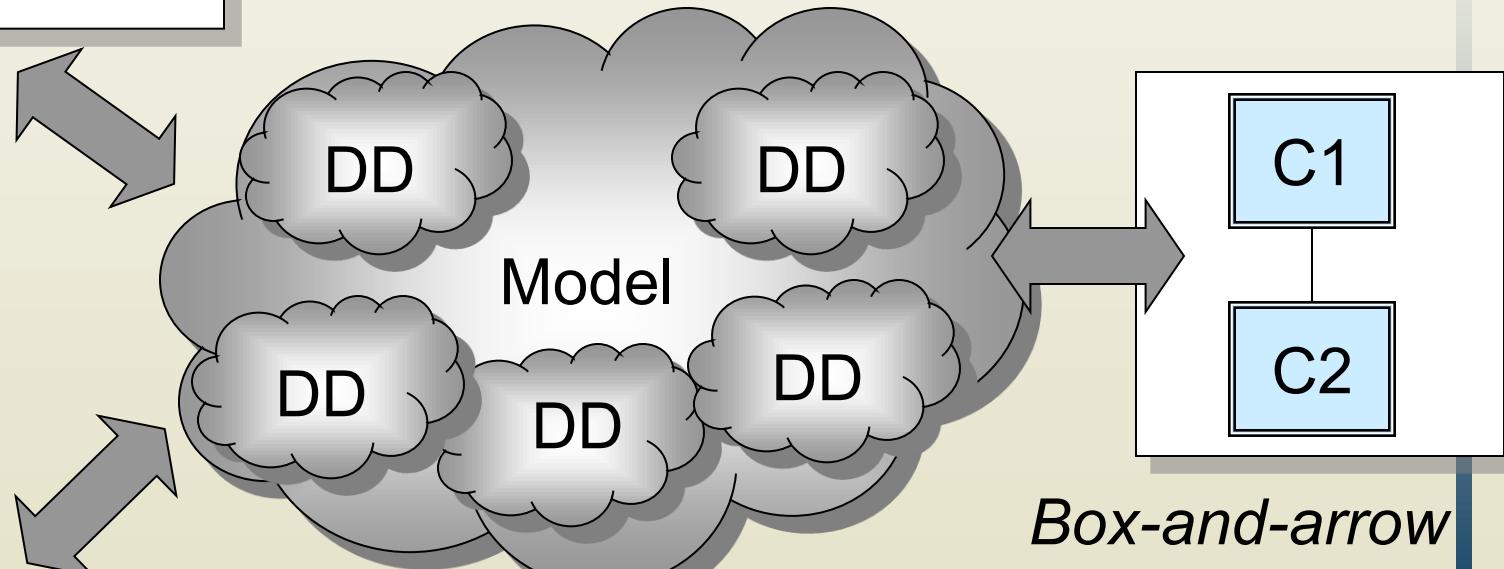
Models vs. Visualizations

- It is easy to confuse models and visualizations because they are very closely related
- In the previous lectures, we have not drawn out this distinction, but now we make it explicit
- A **model** is just abstract information – a set of design decisions
- **Visualizations** give those design decisions form: they let us **depict** those design decisions and **interact** with them in different ways
 - ◆ Because of the interaction aspect, visualizations are often active – they are both pictures AND tools

Models vs. Visualizations

```
<?xml version="1.0">
<model>
  <decision num="1" .../>
  <decision num="2" .../>
</model>
```

XML-based visualization



Our first decision is
that the system will
have two components,
C1 and C2...

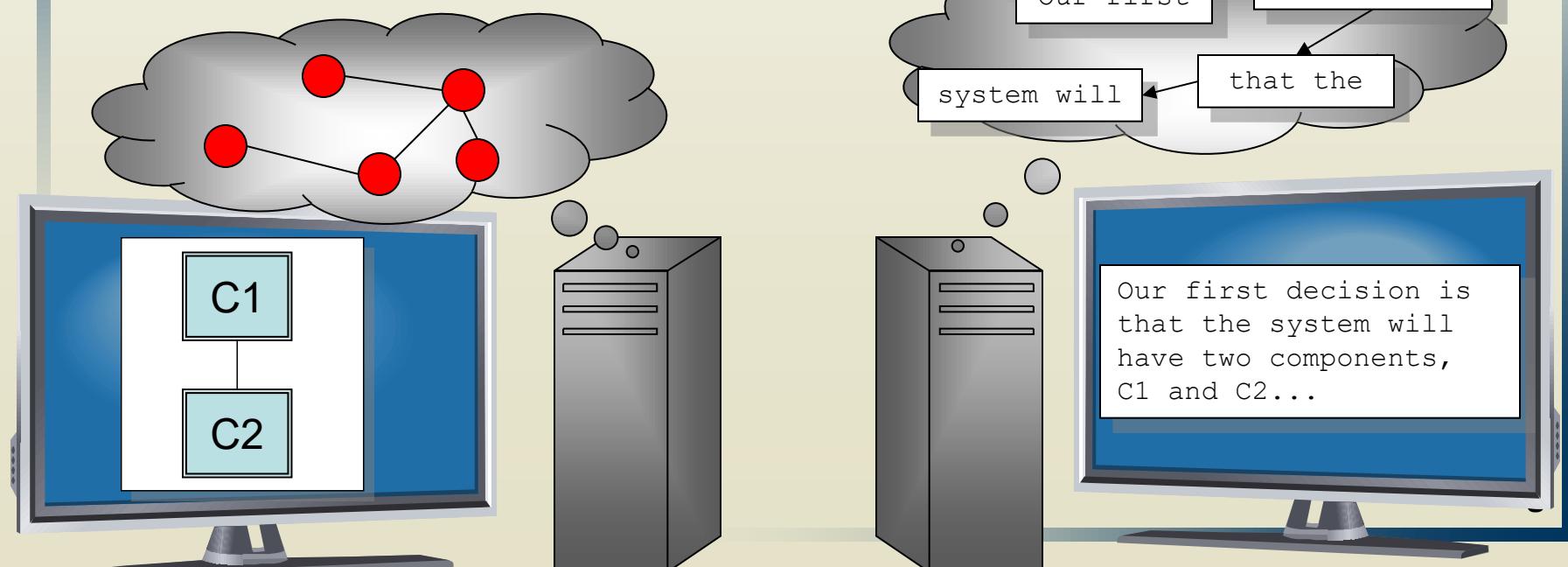
Natural language visualization

Canonical Visualizations

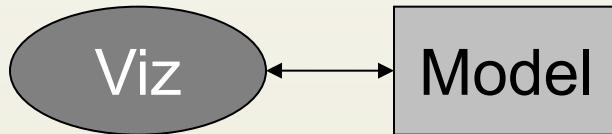
- Each modeling notation is associated with one or more canonical visualizations
 - ◆ This makes it easy to think of a notation and a visualization as the same thing, even though they are not
- Some notations are canonically textual
 - ◆ Natural language, XML-based ADLs
- ...or graphical
 - ◆ PowerPoint-style
- ...or a little of both
 - ◆ UML
- ...or have multiple canonical visualizations
 - ◆ Darwin

Another Way to Think About It

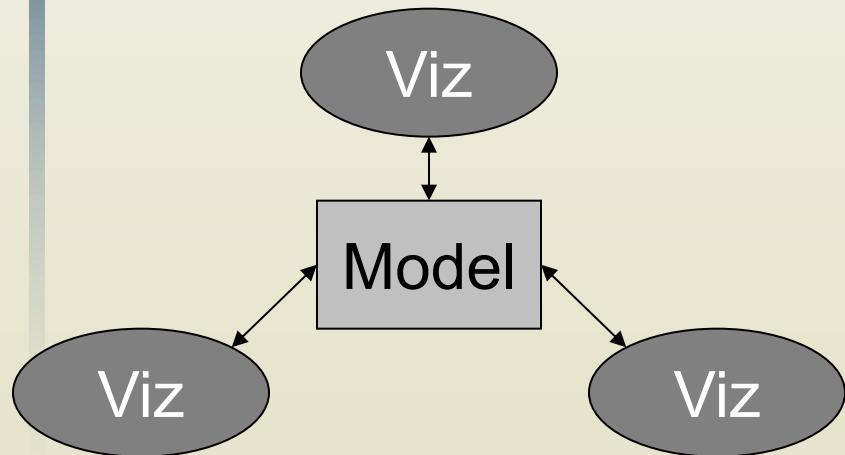
- We may ask “isn’t the canonical visualization the same as the notation since that is how the information is fundamentally organized?”
- Perhaps, but consider a piece of software that edits an architectural model



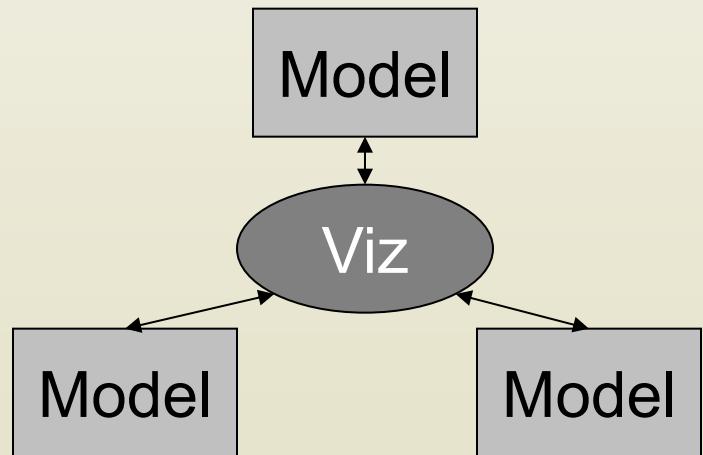
Different Relationships



*One (canonical) visualization
(common)*



*Many visualizations for one
model (common)*



*One visualization bringing together
many models (uncommon)*

Kinds of Visualizations: Textual Visualizations

- Depict architectures through ordinary text files
 - ◆ Generally conform to some syntactic format, like programs conform to a language
 - ◆ May be natural language, in which case the format is defined by the spelling and grammar rules of the language
 - ◆ Decorative options
 - Fonts, colors, bold/italics
 - Tables, bulleted lists/outlines

Textual Visualizations

```
<instance:xArch xsi:type="instance:XArch">
  <types:archStructure xsi:type="types:ArchStructure"
    types:id="ClientArch">
    <types:description xsi:type="instance:Description">
      Client Architecture
    </types:description>
    <types:component xsi:type="types:Component"
      types:id="WebBrowser">
      <types:description xsi:type="instance:Description">
        Web Browser
      </types:description>
      <types:interface xsi:type="types:Interface"
        types:id="WebBrowserInterface">
        <types:description xsi:type="instance:Description">
          Web Browser Interface
        </types:description>
        <types:direction xsi:type="instance:Direction">
          inout
        </types:direction>
      </types:interface>
    </types:component>
  </types:archStructure>
</instance:xArch>
```

XML visualization

Textual Visualizations (cont'd)

```
<instance:xArch xsi:type="instance:XArch">
  <types:archStructure xsi:type="types:ArchStructure"
    types:id="ClientArch">
    <types:description xsi:type="instance:Description">
      Client Architecture
    </types:description>
    <types:component xsi:type="types:Component"
      types:id="WebBrowser">
      <types:description xsi:type="instance:Description">
        Web Browser
      </types:description>
      <types:interface xsi:type="types:Interface"
        types:id="WebBrowserInterface">
        <types:description xsi:type="instance:Description">
          Web Browser Interface
        </types:description>
        <types:direction xsi:type="instance:Direction">
          inout
        </types:direction>
      </types:interface>
    </types:component>
  </types:archStructure>
</instance:xArch>
```

XML visualization

```
xArch{
  archStructure{
    id = "ClientArch"
    description = "Client Architecture"
    component{
      id = "WebBrowser"
      description = "Web Browser"
      interface{
        id = "WebBrowserInterface"
        description = "Web Browser Interface"
        direction = "inout"
      }
    }
  }
}
```

Compact visualization

Textual Visualizations: Interaction

- Generally through an ordinary text editor or word processor
- Some advanced mechanisms available
 - ◆ Syntax highlighting
 - ◆ Static checking
 - ◆ Autocomplete
 - ◆ Structural folding

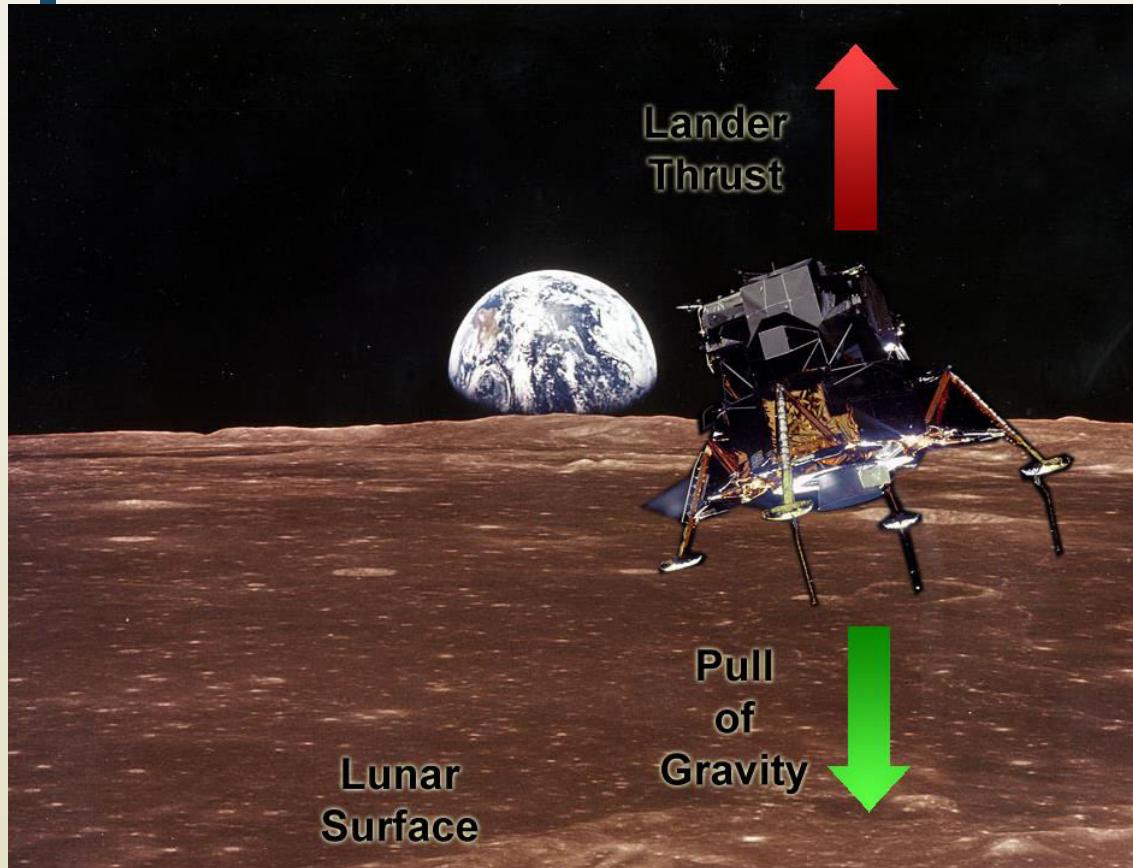
Textual Visualizations

- Advantages
 - ◆ Depict entire architecture in a single file
 - ◆ Good for linear or hierarchical structures
 - ◆ Hundreds of available editors
 - ◆ Substantial tool support if syntax is rigorous (e.g., defined in something like BNF)
- Disadvantages
 - ◆ Can be overwhelming
 - ◆ Bad for graphlike organizations of information
 - ◆ Difficult to reorganize information meaningfully
 - ◆ Learning curve for syntax/semantics

Kinds of Visualizations: Graphical Visualizations

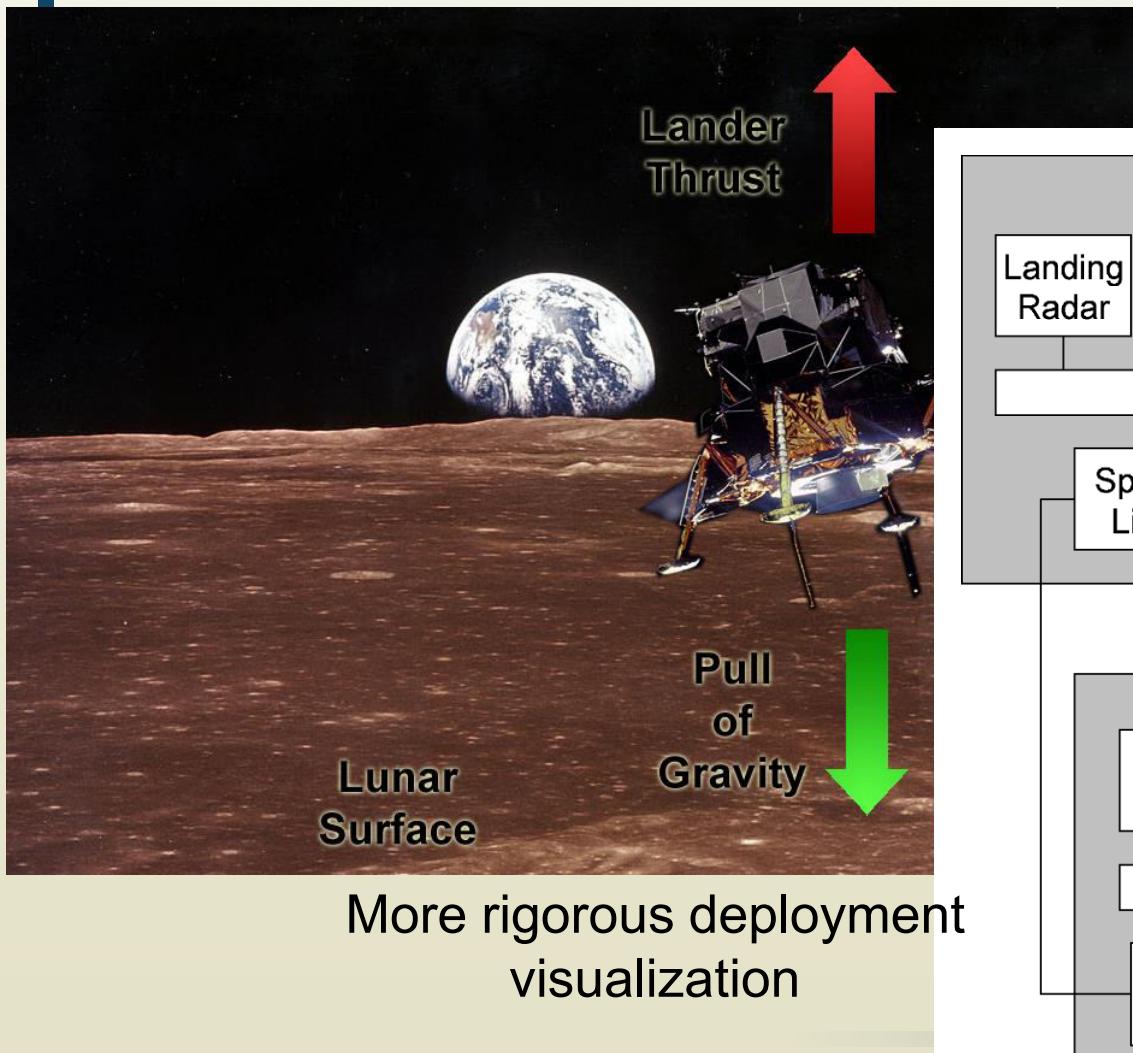
- Depict architectures (primarily) as graphical symbols
 - ◆ Boxes, shapes, pictures, clip-art
 - ◆ Lines, arrows, other connectors
 - ◆ Photographic images
 - ◆ Regions, shading
 - ◆ 2D or 3D
- Generally conform to a symbolic syntax
 - ◆ But may also be 'free-form' and stylistic

Graphical Visualizations

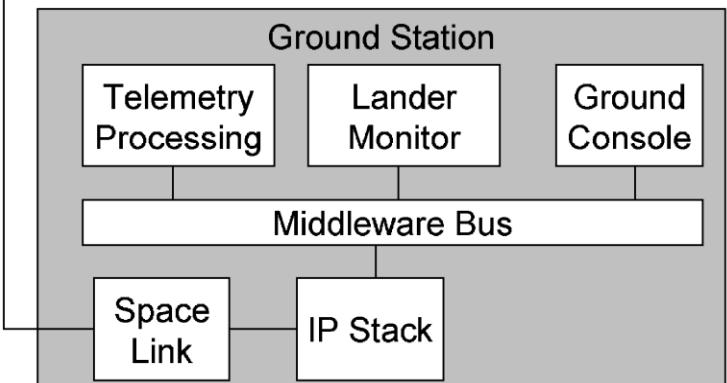
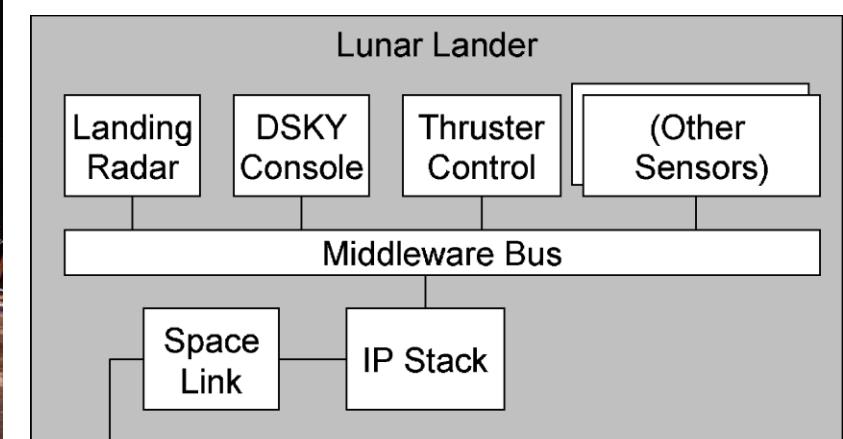


Abstract, stylized
visualization

Graphical Visualizations



Abstract, stylized visualization



Graphical Visualizations: Interaction

- Generally graphical editors with point-and-click interfaces
 - ◆ Employ metaphors like scrolling, zooming, ‘drill-down’
- Editors have varying levels of awareness for different target notations
 - ◆ For example, you can develop UML models in PowerPoint (or Photoshop), but the tools won’t help much
- More exotic editors and interaction mechanisms exist in research
 - ◆ 3D editors
 - ◆ “Sketching-based” editors

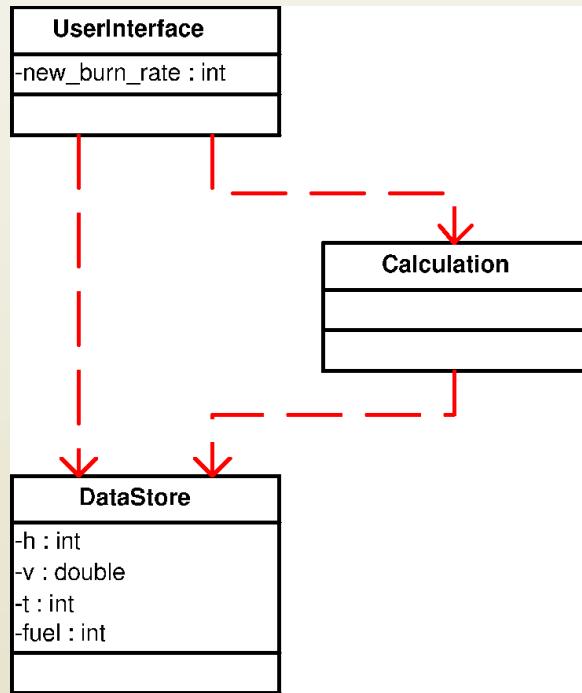
Graphical Visualizations

- Advantages
 - ◆ Symbols, colors, and visual decorations more easily parsed by humans than structured text
 - ◆ Handle non-hierarchical relationships well
 - ◆ Diverse spatial interaction metaphors (scrolling, zooming) allow intuitive navigation
- Disadvantages
 - ◆ Cost of building and maintaining tool support
 - Difficult to incorporate new semantics into existing tools
 - ◆ Do not scale as well as text to very large models

Hybrid Visualizations

- Many visualizations are text-only
- Few graphical notations are purely symbolic
 - ◆ Text labels, at a minimum
 - ◆ Annotations are generally textual as well
- Some notations incorporate substantial parts that are mostly graphical alongside substantial parts that are mostly or wholly textual

Hybrid Visualizations (cont'd)



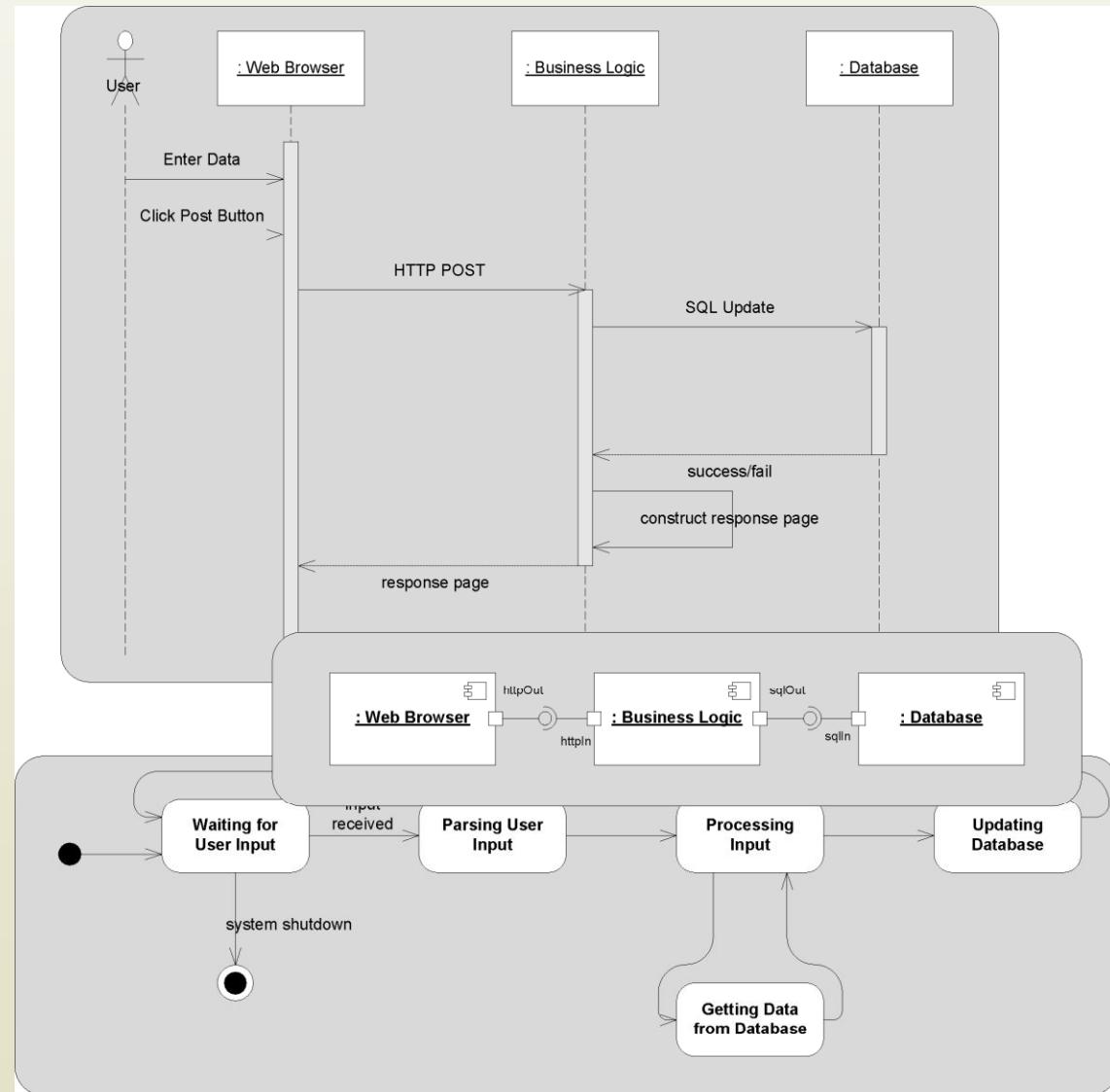
context UserInterface
inv: new_burn_rate ≥ 0

*Primarily graphical
UML class diagram*

*Architectural constraints
expressed in OCL*

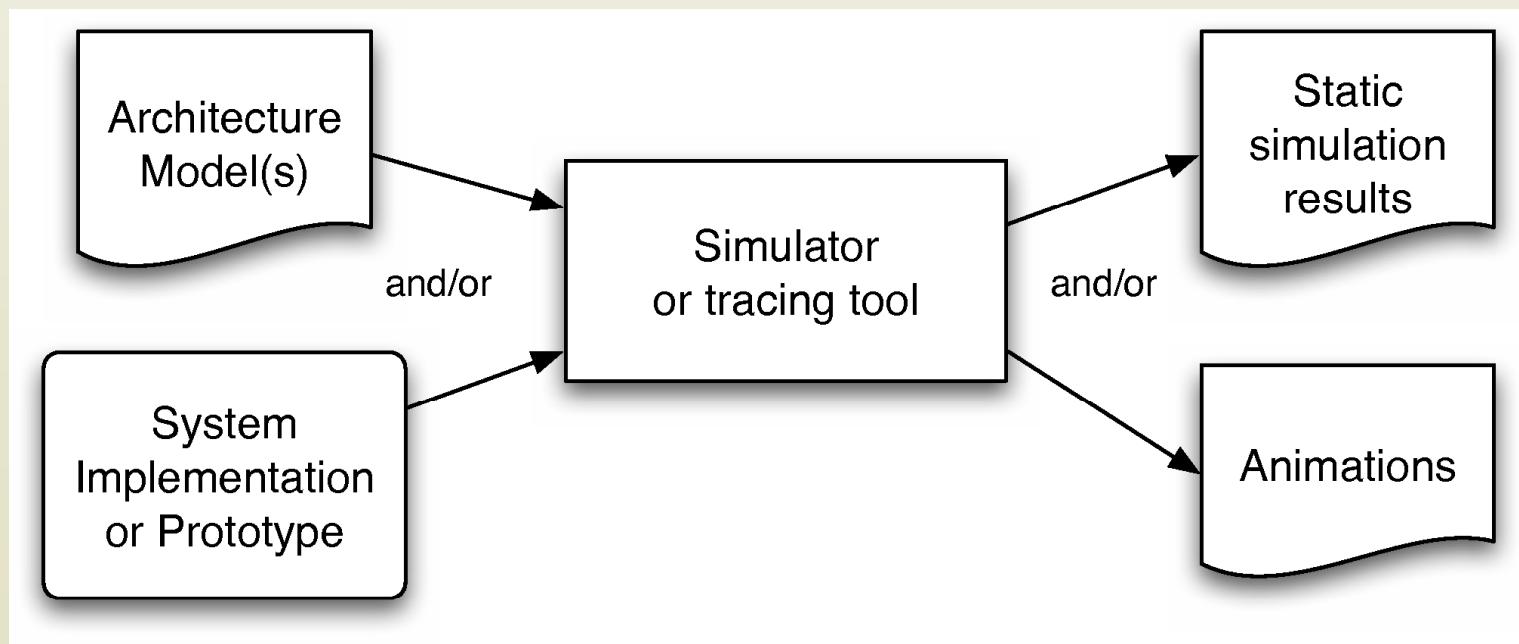
Views, Viewpoints, & Visualizations

- Recall that a view is a subset of the design decisions in an architecture
- And a viewpoint is the perspective from which a view is taken (i.e., the filter that selects the subset)
- Visualizations are associated with viewpoints



Effect Visualizations

- Not all visualizations used in architecture-centric development depict design decisions directly
- Some depict the results or effects of design decisions
 - ◆ We call these 'effect visualizations'
- May be textual, graphical, hybrid, etc.



Evaluating Visualizations

- Scope and Purpose
 - ◆ What is the visualization for? What can it visualize?
- Basic Type
 - ◆ Textual? Graphical? Hybrid? Effect?
- Depiction
 - ◆ What depiction mechanisms and metaphors are primarily employed by the visualization?
- Interaction
 - ◆ What interaction mechanisms and metaphors are primarily employed by the visualization?

Evaluating Visualizations (cont'd)

- Fidelity
 - ◆ How well/completely does the visualization reflect the information in the underlying model?
 - ◆ Consistency should be a minimum requirement, but details are often left out
- Consistency
 - ◆ How well does the visualization use similar representations for similar concepts?
- Comprehensibility
 - ◆ How easy is it for stakeholders to understand and use a visualization
 - Note: this is a function of both the visualization and the stakeholders

Evaluating Visualizations (cont'd)

- Dynamism
 - ◆ How well does the visualization support models that change over time (dynamic models)?
- View Coordination
 - ◆ How well the visualization is connected to and kept consistent with other visualizations
- Aesthetics
 - ◆ How pleasing is the visualization (look and feel) to its users?
 - A very subjective judgment
- Extensibility
 - ◆ How easy is it to add new capabilities to a visualization?

Objectives

- Concepts
 - ◆ What is visualization?
 - ◆ Differences between modeling and visualization
 - ◆ What kinds of visualizations do we use?
 - ◆ Visualizations and views
 - ◆ How can we characterize and evaluate visualizations?
- Examples
 - ◆ Concrete examples of a diverse array of visualizations
- Constructing visualizations
 - ◆ Guidelines for constructing new visualizations
 - ◆ Pitfalls to avoid when constructing new visualizations
 - ◆ Coordinating visualizations

Text Visualizations

- Text visualizations are generally provided through text editors
- Examples:
 - ◆ Simple: Windows Notepad, SimpleText, pico, joe
 - ◆ For experts: vi, emacs
 - ◆ With underlying language support: Eclipse, UltraEdit, many HTML editors
 - ◆ Free-form text documents: Microsoft Word, other word processors

Text Visualizations (cont'd)

- Advantages
 - ◆ Provide a uniform way of working with many different underlying notations
 - ◆ Wide range of editors available to suit any need
 - ◆ Many incorporate advanced 'content assist' capabilities
 - ◆ Many text editors can be extended to handle new languages or integrate new tools easily
- Disadvantages
 - ◆ Increasing complexity as models get bigger
 - ◆ Do not handle graph structures and complex interrelationships well

Advanced Interaction Mechanisms

Before Code Folding:

```
- public int getAltitude() {  
    ds = getDataStore();  
    a = ds.getProperty("altitude");  
    return a;  
}
```

After Code Folding:

```
+ public int getAltitude() { ... }
```

```
component GameLogic{  
    description = "my_description"  
    interface{  
        description = "my_description"  
        direction = "none / in / out / inout"  
    }  
    behavior{  
        my_behavior  
    }  
}
```

```
GameState st = application.getGameState();  
st.| getAltitude() : int  
    setAltitude(int a) : void  
    getFuel() : int  
    setFuel(int f) : void  
    ...
```

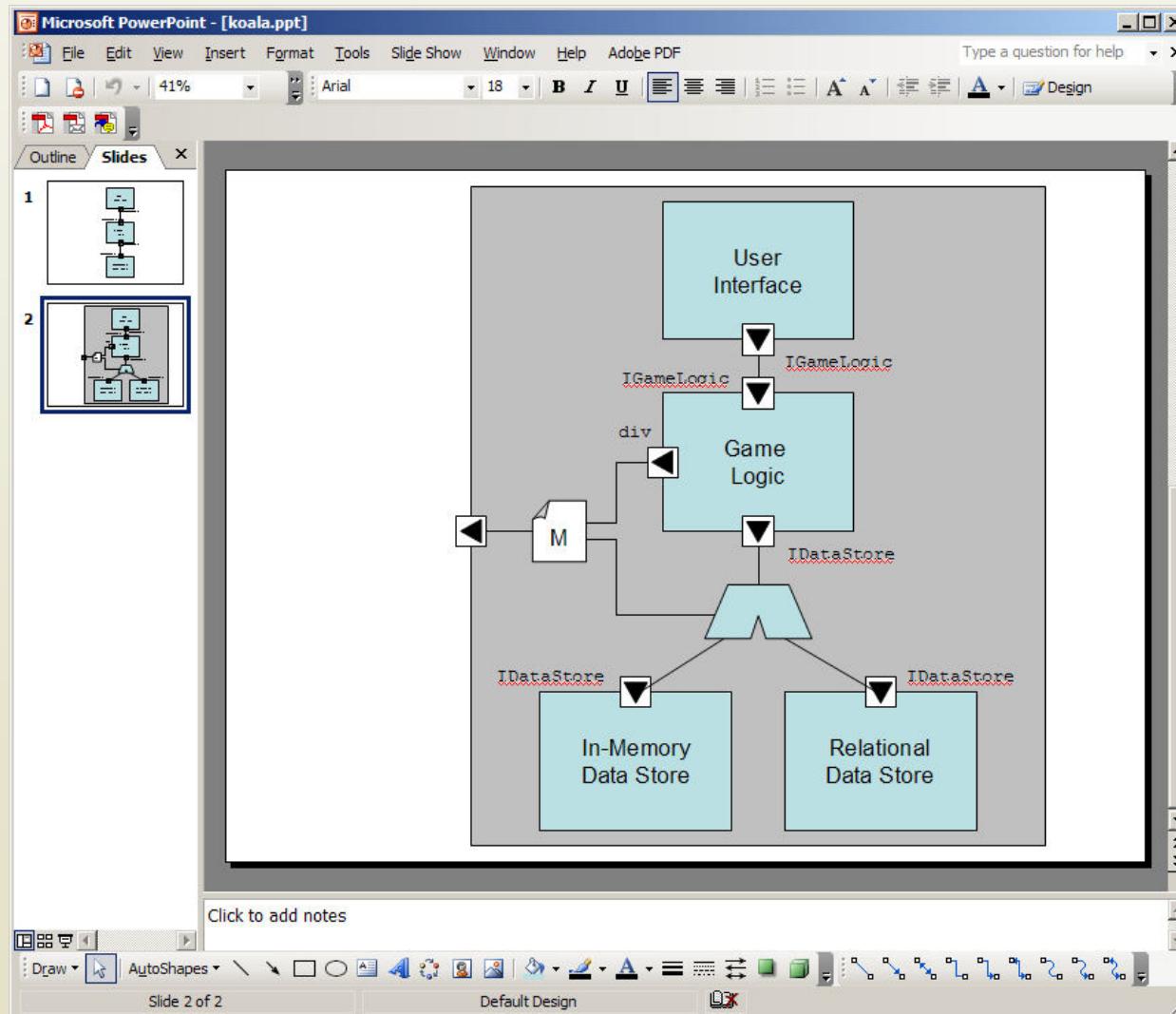
Text Visualizations: Evaluation

- Scope/Purpose
 - ◆ Visualizing design decisions or effects as (structured) text
- Basic Type
 - ◆ Textual
- Depiction
 - ◆ Ordered lines of characters possibly grouped into tokens
- Interaction
 - ◆ Basic: insert, delete, copy, paste
 - ◆ Advanced: coloring, code folding, etc.
- Fidelity
 - ◆ Generally canonical
- Consistency
 - ◆ Generally good; depends on underlying notation
- Comprehensibility
 - ◆ Drops with increasing complexity
- Dynamism
 - ◆ Rare, but depends on editor
- View coordination
 - ◆ Depends on editor
- Aesthetics
 - ◆ Varies; can be overwhelming or elegant and structured
- Extensibility
 - ◆ Many extensible editors

General Graphical Visualizations

- E.g., PowerPoint, OmniGraffle, etc.
- Provide point-and-click manipulation of graphical symbols, interconnections, and text blocks
- Advantages
 - ◆ Friendly UI can create nice-looking depictions
 - ◆ Nothing hidden; no information difference between model and depiction
- Disadvantages
 - ◆ No underlying semantics; difficult to add them
 - Visio is a partial exception
 - This means that interaction mechanisms can offer minimal support
 - ◆ Difficult to connect to other visualizations

General Graphical Example



General Graphical: Evaluation

- Scope/Purpose
 - ◆ Visualizing design decisions as symbolic pictures
- Basic Type
 - ◆ Graphical
- Depiction
 - ◆ (Possibly) interconnected symbols on a finite canvas
- Interaction
 - ◆ Point and click, drag-and-drop direct interactions with symbols, augmented by menus and dialogs
- Fidelity
 - ◆ Generally canonical
- Consistency
 - ◆ Manual
- Comprehensibility
 - ◆ Depends on skill of the modeler and use of consistent symbols/patterns
- Dynamism
 - ◆ Some animation capabilities
- View coordination
 - ◆ Difficult at best
- Aesthetics
 - ◆ Modeler's responsibility
- Extensibility
 - ◆ Adding new symbols is easy, adding semantics is harder

Next Time

- Continuing with survey of more specific visualizations
- Discussion of how to construct good (and bad) visualizations

Visualizing Software Architectures, Part 2

Software Architecture
Lecture 12

Objectives

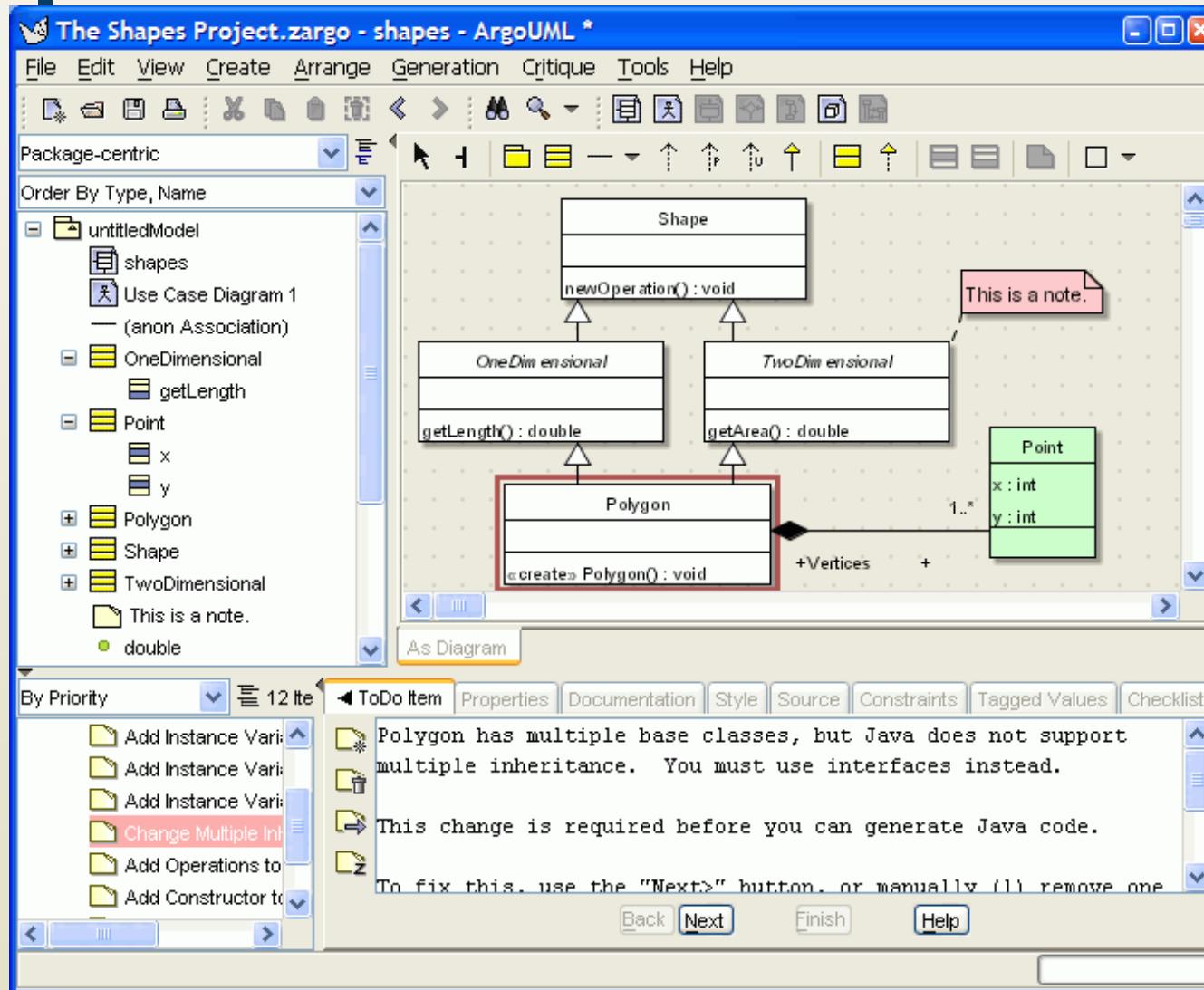
Concepts

- ◆ What is visualization?
- ◆ Differences between modeling and visualization
- ◆ What kinds of visualizations do we use?
- ◆ Visualizations and views
- ◆ How can we characterize and evaluate visualizations?
- Examples
 - ◆ Concrete examples of a diverse array of visualizations
- Constructing visualizations
 - ◆ Guidelines for constructing new visualizations
 - ◆ Pitfalls to avoid when constructing new visualizations
 - ◆ Coordinating visualizations

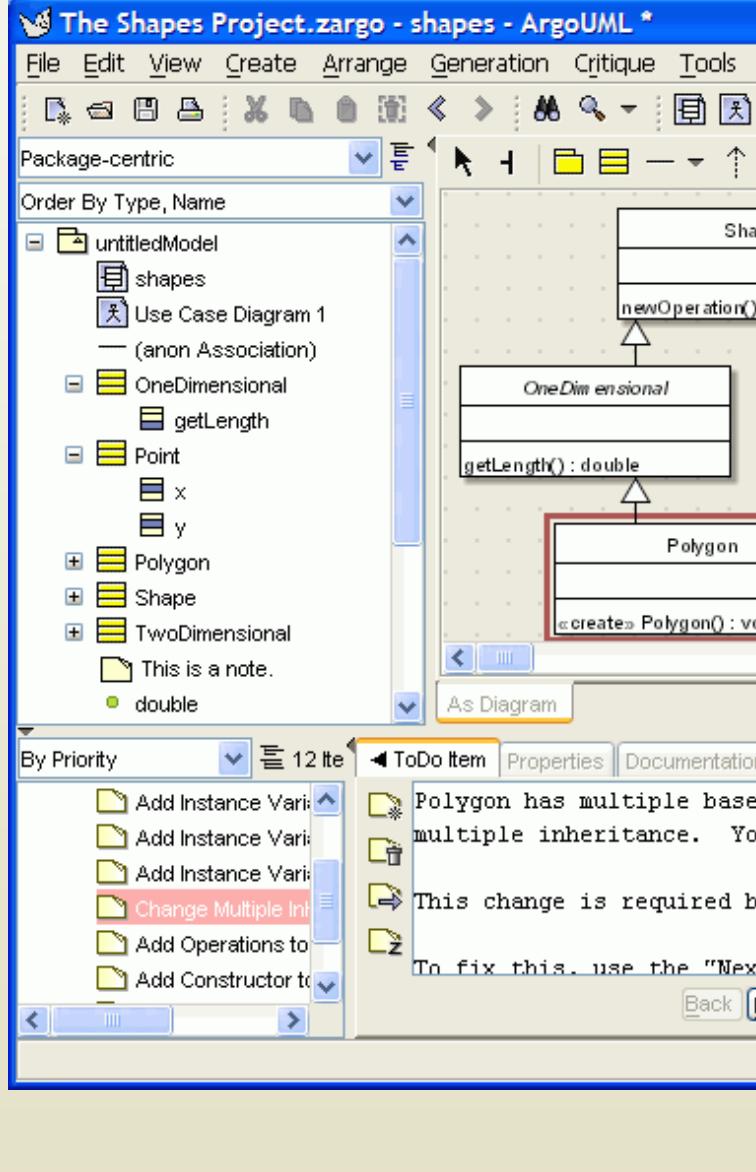
UML Visualizations

- Canonical graphical depictions + tool-specific interactions
- XMI: Textual depiction in XML + text-editor interactions
- Advantages
 - ◆ Canonical graphical depiction common across tools
 - ◆ Graphical visualizations have similar UI metaphors to PowerPoint-style editors, but with UML semantics
 - ◆ XMI projection provides textual alternative
- Disadvantages
 - ◆ No standard for interaction as there is for depiction
 - ◆ In some tools hard to tell where UML model ends and auxiliary models begin
 - ◆ Most UML visualizations are restricted to (slight variants) of the canonical UML depiction

UML Visualization



Software Architecture: Foundations, Theory, and Practice



```
<UML:Class xmi.id = '723'  
           name = 'Data Store'  
           visibility = 'public'  
           isSpecification = 'false'  
           isRoot = 'false'  
           isLeaf = 'false'  
           isAbstract = 'false'  
           isActive = 'false'>  
  
<UML:Association xmi.id = '725'  
                  name = ''  
                  isSpecification = 'false'  
                  isRoot = 'false'  
                  isLeaf = 'false'  
                  isAbstract = 'false'>  
  
<UML:Association.connection>  
  <UML:AssociationEnd xmi.id = '726'  
    visibility = 'public'  
    isSpecification = 'false'  
    isNavigable = 'true'  
    ordering = 'unordered'  
    aggregation = 'none'  
    targetScope = 'instance'  
    changeability = 'changeable'>  
  
<UML:AssociationEnd.multiplicity>  
  <UML:Multiplicity xmi.id = '727'>  
    <UML:Multiplicity.range>  
      <UML:MultiplicityRange xmi.id = '728'>  
        lower = '1'  
        upper = '1'>  
    ...
```

UML Visualizations: Evaluation

- Scope/Purpose
 - ◆ Visualization of UML models
- Basic Type
 - ◆ Graphical (diagrams), textual (XMI)
- Depiction
 - ◆ Diagrams in UML symbolic vocabulary/XML-formatted text
- Interaction
 - ◆ Depends on the editor; generally point-and-click for diagrams; text editor for XMI
- Fidelity
 - ◆ Diagrams are canonical, XMI elides layout info
- Consistency
 - ◆ Generally good across diagrams; small exceptions
- Comprehensibility
 - ◆ Ubiquity assists interpretations
- Dynamism
 - ◆ Rare
- View coordination
 - ◆ Some editors better than others
- Aesthetics
 - ◆ Simple symbols reduce complexity; uniform diagrams
- Extensibility
 - ◆ Profile support OK; major language extensions hard

Rapidé

- Rapidé models are generally written with a canonical textual visualization
 - ◆ Some graphical builders available as well
- Focus: Interesting *effect visualization* of simulation results
- Advantages
 - ◆ Provides an intuitive way to visualize the causal relationships between events
 - ◆ Automatically generated from Rapide specifications
- Disadvantages
 - ◆ Complex applications generate complex graphs
 - ◆ Difficult to identify why particular causal relationships exist
 - Simulation is not interactive

Rapidé Examples

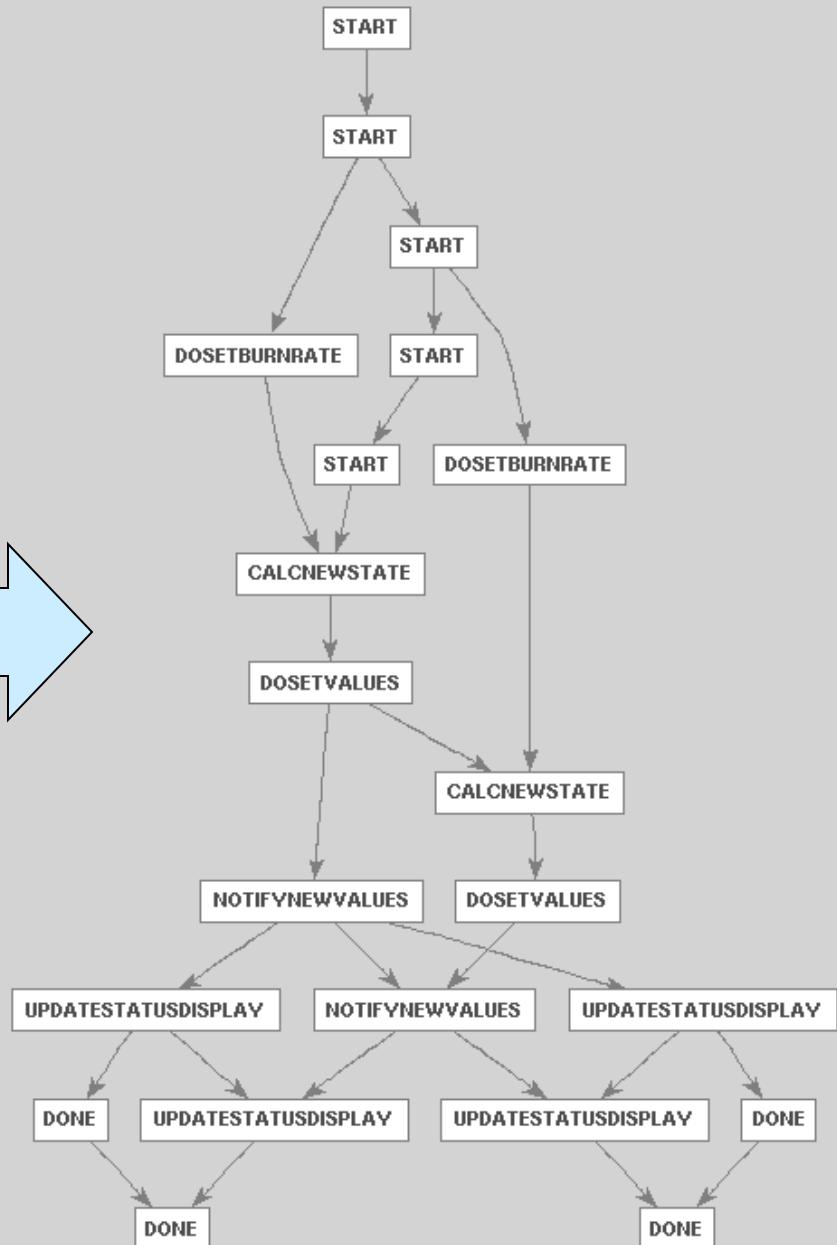
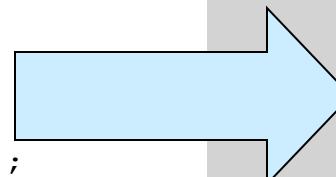
```

type DataStore is interface
    action in SetValues();
        out NotifyNewValues();
    behavior
    begin
        SetValues => NotifyNewValues();
    end DataStore;

type Calculation is interface
    action in SetBurnRate();
        out DoSetValues();
    behavior
        action CalcNewState();
    begin
        SetBurnRate => CalcNewState(); I
    end Calculation;

type Player is interface
    action out DoSetBurnRate();
        in NotifyNewValues();
    behavior
        TurnsRemaining : var integer := 0;
        action UpdateStatusDisplay();
        action Done();
    end Player;

```



Rapidé Effect Visualization: Evaluation

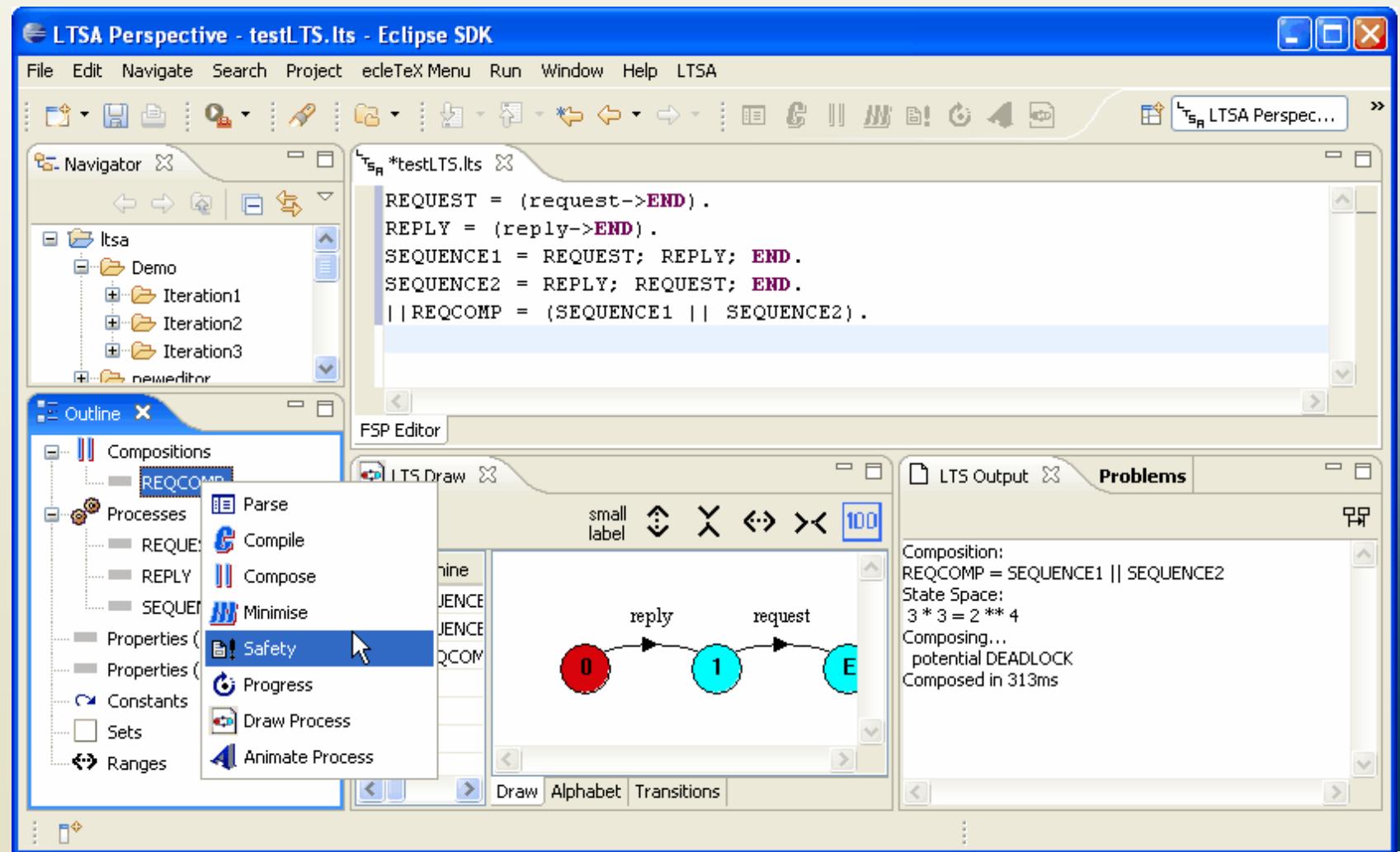
- Scope/Purpose
 - ◆ Graphical event traces
- Basic Type
 - ◆ Graphical
- Depiction
 - ◆ Directed acyclic graph of events
- Interaction
 - ◆ No substantial interaction with generated event traces
- Fidelity
 - ◆ Each trace is an instance; different simulation runs may produce different traces in a non-deterministic system
- Consistency
 - ◆ Tiny symbol vocabulary ensures consistency

- Comprehensibility
 - ◆ Easy to see causal relationships but difficult to understand why they're there
- Dynamism
 - ◆ No support
- View coordination
 - ◆ Event traces are generated automatically from architectural models
- Aesthetics
 - ◆ Simple unadorned directed acyclic graph of nodes and edges
- Extensibility
 - ◆ Tool set is effectively a 'black box'

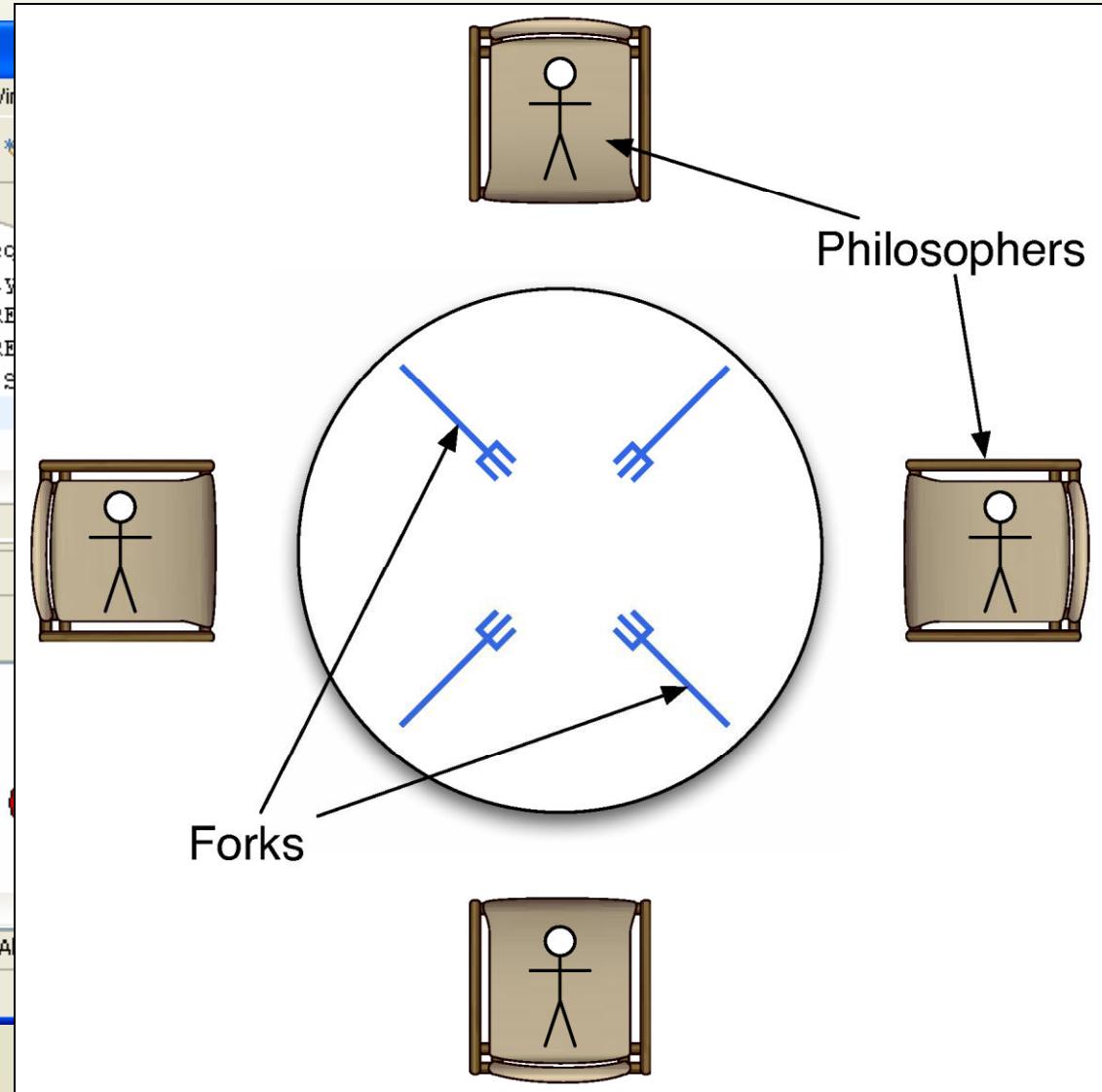
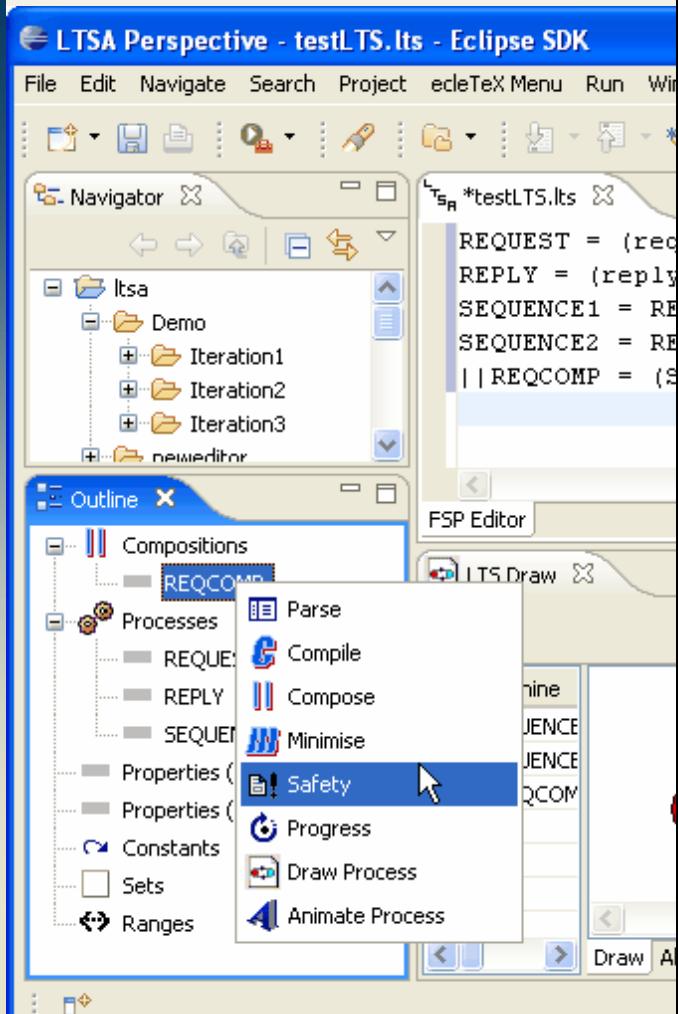
Labeled Transition State Analyzer (LTSA)

- A tool for analyzing and simultaneously visualizing concurrent systems' behavior using a modeling language called FSP
- Advantages
 - ◆ Provides multiple concurrent visualizations of concurrent behavior
 - ◆ Integrates both model and effect visualizations, textual and graphical depictions
 - ◆ Can develop domain-specific visualizations to understand abstract models
- Disadvantages
 - ◆ Behavior specification language has somewhat steep learning curve
 - ◆ Developing domain-specific graphical visualizations can be expensive

LTSA Examples



LTSA Examples



LTSA: Evaluation

- Scope/Purpose
 - ◆ Multiple coordinated visualizations of concurrent systems' behavior
- Basic Type
 - ◆ Textual, Graphical, Effect
- Depiction
 - ◆ Text & state machines for models, various effect viz.
- Interaction
 - ◆ FSP can be edited textually or graphically
- Fidelity
 - ◆ Graphical visualizations may elide some information
- Consistency
 - ◆ Limited vocabulary helps ensure consistency
- Comprehensibility
 - ◆ FSP has some learning curve; domain-specific effect visualizations are innovative
- Dynamism
 - ◆ Animation on state-transition diagrams and domain-specific visualizations
- View coordination
 - ◆ Views are coordinated automatically
- Aesthetics
 - ◆ State transition diagrams are traditional; domain-specific visualizations can enhance aesthetics
- Extensibility
 - ◆ New domain-specific effect visualizations as plug-ins

xADL Visualizations

- Coordinated set of textual, graphical, and effect visualizations for an extensible ADL
- Advantages
 - ◆ Provides an example of how to construct a wide variety of (often) coordinated or interrelated visualizations
 - ◆ Lets users move fluidly from one visualization to another
 - ◆ Guidance available for extending visualizations or adding new ones
- Disadvantages
 - ◆ Some learning curve to extend graphical editors
 - ◆ Adding or extending visualizations has to be done carefully so they play well with existing ones

xADL Visualization Examples

```
<types:component xsi:type="types:Component"
    types:id="myComp">
  <types:description xsi:type="instance:Description">
    MyComponent
  </types:description>
  <types:interface xsi:type="types:Interface"
    types:id="iface1">
    <types:description xsi:type="instance:Description">
      Interface1
    </types:description>
    <types:direction xsi:type="instance:Direction">
      inout
    </types:direction>
  </types:interface>
</types:component>
```

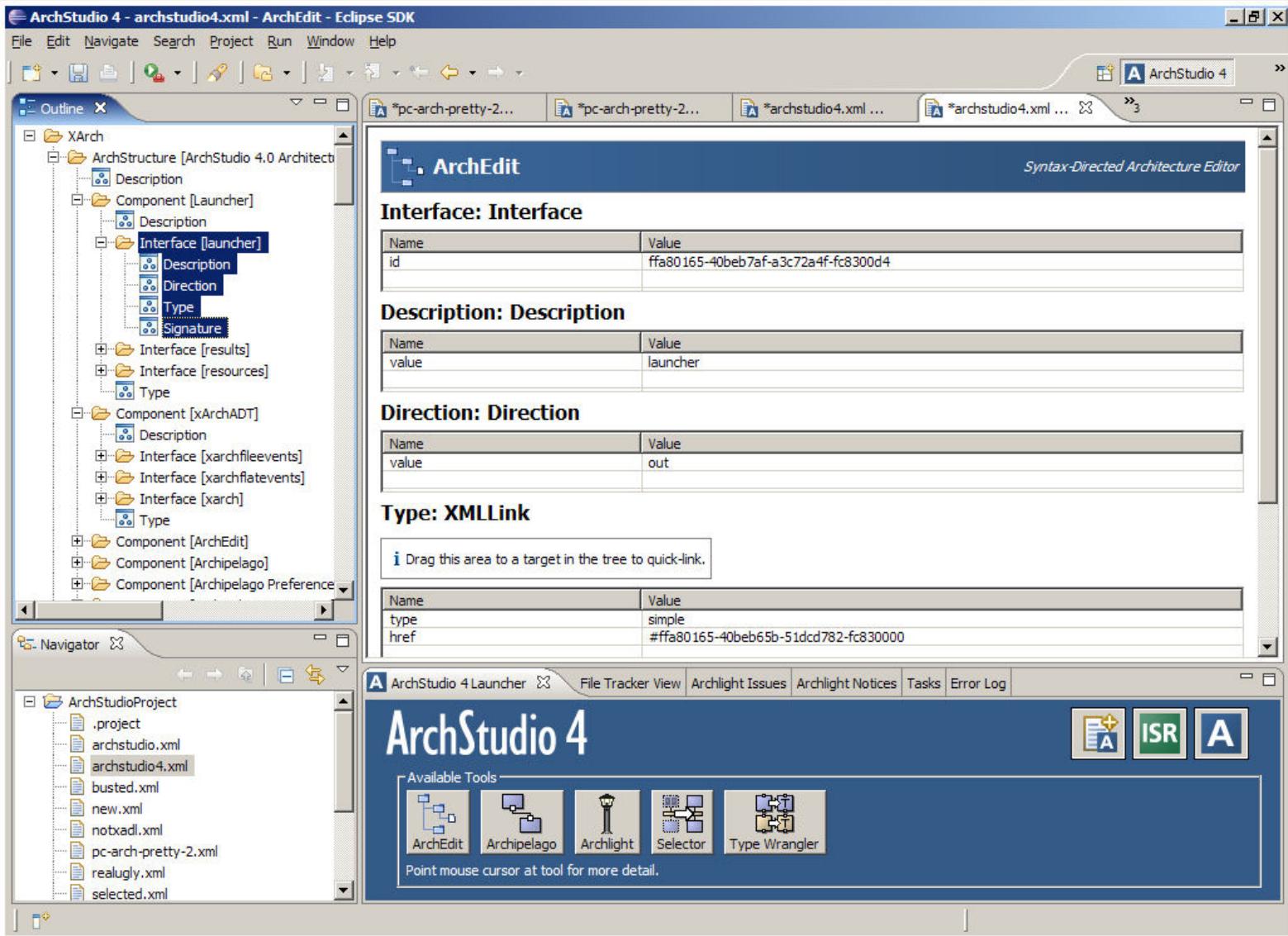
xADL Visualization Examples

```
<types:component xsi:type="types:Component"
    types:id="myComp">
<types:description xsi:type="instance:Description">
    MyComponent
</types:description>
<types:interface xsi:type="typ
    types:id="ifa
        <types:description xsi:type=
            Interface1
        </types:description>
        <types:direction xsi:type="i
            inout
        </types:direction>
    </types:interface>
</types:component>
```

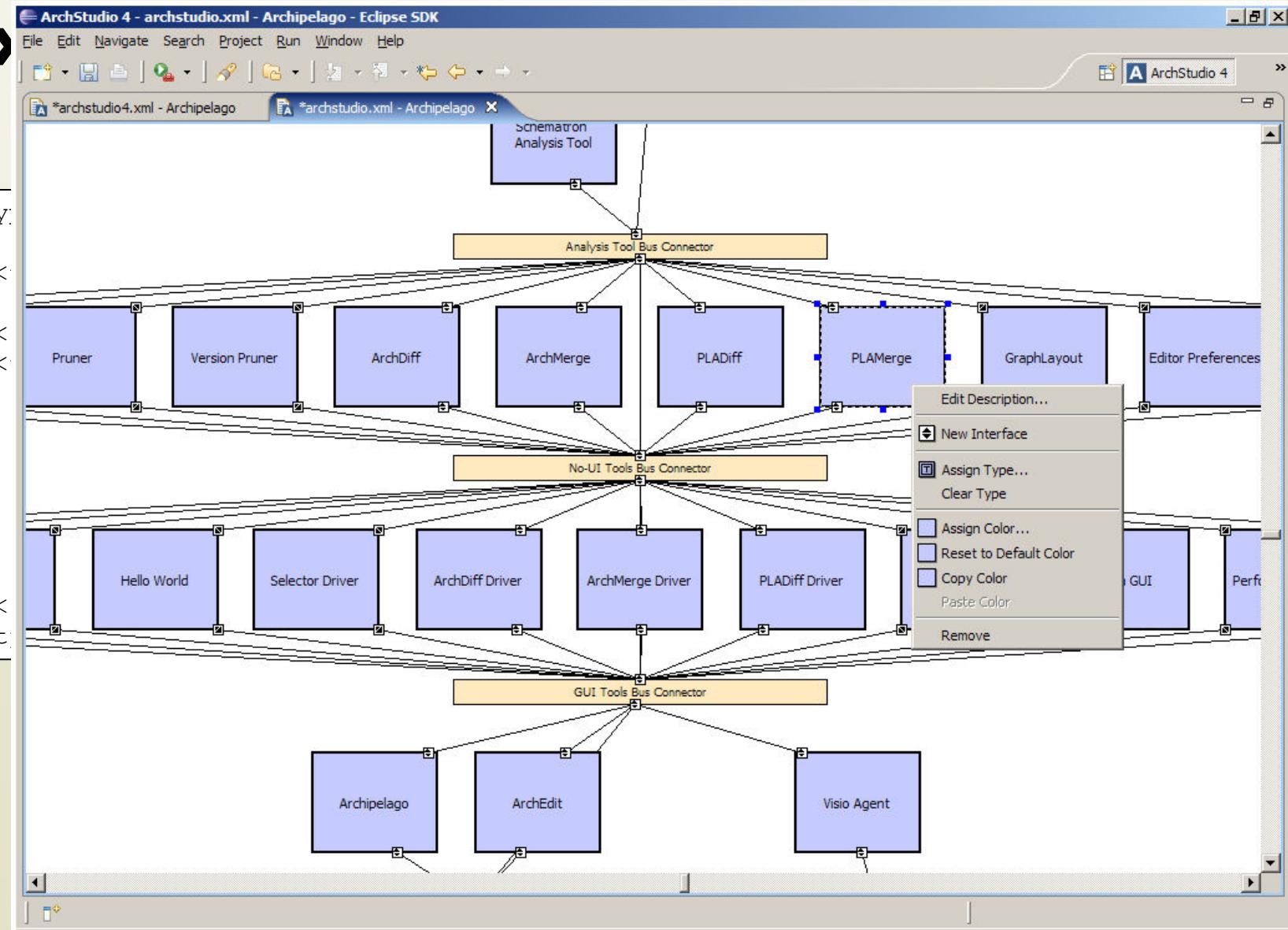
```
component{
    id = "myComp";
    description = "MyComponent";
    interface{
        id = "iface1";
        description = "Interface1";
        direction = "inout";
    }
}
```

Software Architecture: Foundations, Theory, and Practice

```
<types>
<typ My
</typ>
<typ
<t
</t
</
<t
</
</t
</type>
```



Software Architecture: Foundations, Theory, and Practice



Software Architecture: Foundations, Theory, and Practice

Message Trace Analysis Tool Alpha 1.0

Bricks

Name	Top	Bottom
ChuteArtist	<input type="checkbox"/>	<input type="checkbox"/>
ChuteComponent	<input type="checkbox"/>	<input type="checkbox"/>
ClockComponent	<input type="checkbox"/>	<input type="checkbox"/>
GraphicsBinding	<input type="checkbox"/>	<input type="checkbox"/>
LayoutManager	<input type="checkbox"/>	<input type="checkbox"/>
MatchingLogic	<input type="checkbox"/>	<input type="checkbox"/>
NextTileLogic	<input type="checkbox"/>	<input type="checkbox"/>
PaletteArtist	<input type="checkbox"/>	<input type="checkbox"/>
PaletteComponent	<input type="checkbox"/>	<input type="checkbox"/>
RelativePositionLogic	<input type="checkbox"/>	<input type="checkbox"/>
StatusArtist	<input type="checkbox"/>	<input type="checkbox"/>
StatusComponent	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>
StatusLogic	<input type="checkbox"/>	<input type="checkbox"/>
TitleArtist	<input type="checkbox"/>	<input type="checkbox"/>
WellArtist	<input type="checkbox"/>	<input type="checkbox"/>
WellComponent	<input type="checkbox"/>	<input type="checkbox"/>

Message Causality

ID	Relative Time	Source Identifier	Source Interface	Destination Identifier	Destination Interface	Name
4077	101947	Bus2	IFACE_TOP	Bus1	IFACE_BOTTOM	DecrementNumberOfLives

Get Message Information

ID	Relative Time	Source Identifier	Source Interface	Destination Identifier	Destination Interface	Name
4105	102017	StatusComponent	IFACE_BOTTOM	Bus1	IFACE_TOP	NumberOfLives

Brick Info

Class: c2.fw.NamedPropertyMessage
Id: 4104
Time: 101987

SourceBrickId: Bus1
SourceInterfaceId: IFACE_TOP
DestBrickId: StatusComponent
DestInterfaceId: IFACE_BOTTOM

DecrementNumberOfLives
C2_TYPE = REQUEST

toString() results:
(4104:101987 from BrickInterfaceIdPair(brickId="Bus1", in

Message List

ID	Relative Time	Source Identifier	Source Interface	Destination Identifier	Destination Interface	Name
3992	101697	Bus1	IFACE_TOP	StatusComponent	IFACE_BOTTOM	AdvanceWellTiles
4010	101767	Bus1	IFACE_TOP	StatusComponent	IFACE_BOTTOM	AdvanceWellTiles
4020	101787	Bus1	IFACE_TOP	StatusComponent	IFACE_BOTTOM	AdvanceChuteTiles
4072	101937	Bus1	IFACE_TOP	StatusComponent	IFACE_BOTTOM	SuspendClock
4092	101977	Bus1	IFACE_TOP	StatusComponent	IFACE_BOTTOM	AdvanceWellTiles
4104	101987	Bus1	IFACE_TOP	StatusComponent	IFACE_BOTTOM	DecrementNumberOfLives
4105	102017	StatusComponent	IFACE_BOTTOM	Bus1	IFACE_TOP	NumberOfLives
4110	102037	Bus1	IFACE_TOP	StatusComponent	IFACE_BOTTOM	GetNumberOfLives
4114	102047	StatusComponent	IFACE_BOTTOM	Bus1	IFACE_TOP	NumberOfLives
4158	212135	Bus1	IFACE_TOP	StatusComponent	IFACE_BOTTOM	ResumeClock
4176	212196	Bus1	IFACE_TOP	StatusComponent	IFACE_BOTTOM	AdvanceWellTiles
4193	212316	Bus1	IFACE_TOP	StatusComponent	IFACE_BOTTOM	AdvanceWellTiles
4211	212506	Bus1	IFACE_TOP	StatusComponent	IFACE_BOTTOM	AdvanceWellTiles
4216	212516	Bus1	IFACE_TOP	StatusComponent	IFACE_BOTTOM	PlaceTile

9

xADL Visualizations: Evaluation

- Scope/Purpose
 - ◆ Multiple coordinated visualizations of xADL models
- Basic Type
 - ◆ Textual, Graphical, Effect
- Depiction
 - ◆ XML, abbreviated XML, symbol graphs, hybrid effect (MTAT)
- Interaction
 - ◆ Visualizations emulate various editing paradigms
- Fidelity
 - ◆ Textual & ArchEdit complete; graphical leave detail out
- Consistency
 - ◆ Effort to follow conventions
- Comprehensibility
 - ◆ Varies; some easier than others
- Dynamism
 - ◆ Animation on state-transition diagrams and domain-specific visualizations
- View coordination
 - ◆ Many views coordinated 'live,' MTAT leverages some animation
- Aesthetics
 - ◆ Varies; Archipelago promotes aesthetic improvements by allowing fine customization
- Extensibility
 - ◆ Many extensibility mechanisms at different levels

Objectives

- Concepts
 - ◆ What is visualization?
 - ◆ Differences between modeling and visualization
 - ◆ What kinds of visualizations do we use?
 - ◆ Visualizations and views
 - ◆ How can we characterize and evaluate visualizations?
- Examples
 - ◆ Concrete examples of a diverse array of visualizations
- Constructing visualizations
 - ◆ Guidelines for constructing new visualizations
 - ◆ Pitfalls to avoid when constructing new visualizations
 - ◆ Coordinating visualizations

Constructing New Visualizations

- Developing a new visualization can be expensive both in initial development and maintenance
- Must answer many questions in advance
 - ◆ Can I achieve my goals by extending an existing visualization?
 - ◆ Can I translate into another notation and use a visualization already available there?
 - ◆ How will my visualization augment the existing set of visualizations for this notation?
 - ◆ How will my visualization coordinate with other visualizations?
 - ◆ (Plus all the evaluation categories we've been exploring)

New Visualizations: Guidelines

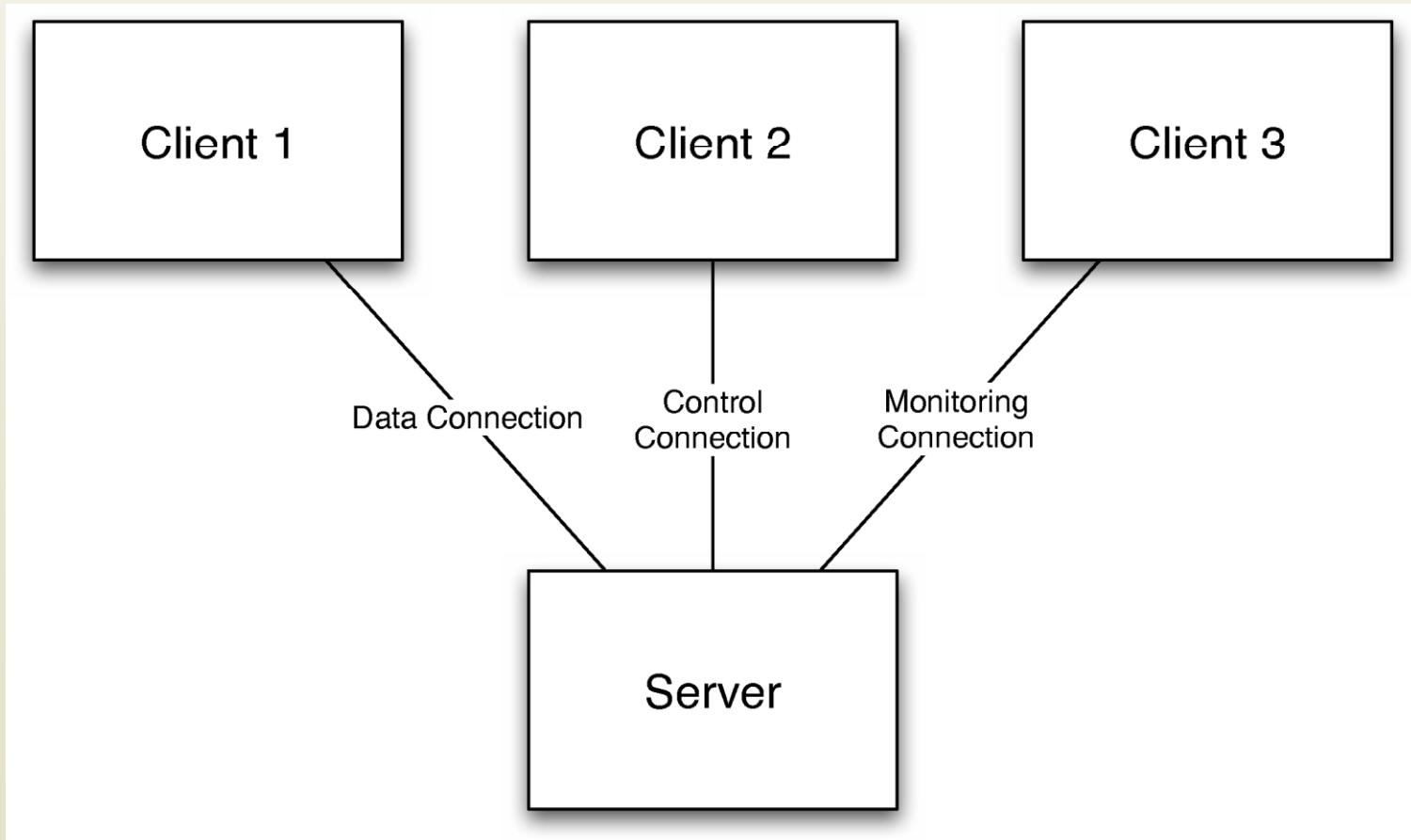
- Borrow elements from similar visualizations
 - ◆ Leverages existing stakeholder knowledge
 - ◆ Improves comprehensibility
- Be consistent among visualizations
 - ◆ Don't conflict with existing visualizations without a good reason (e.g., developing a domain-specific visualization where the concepts and metaphors are completely different)
- Give meaning to each visual aspect of elements
 - ◆ Parsimony is more important than aesthetics
 - ◆ Corollary: avoid having non-explicit meaning encoded in visualizations

New Visualizations: Guidelines (cont'd)

- Document the meaning of visualizations
 - ◆ Visualizations are rarely self-explanatory
 - ◆ Focus on mapping between model and visualization
- Balance traditional and innovative interfaces
 - ◆ Stakeholders bring a lot of interaction experience to the table
 - ◆ But just because a mechanism is popular doesn't mean it's ideal

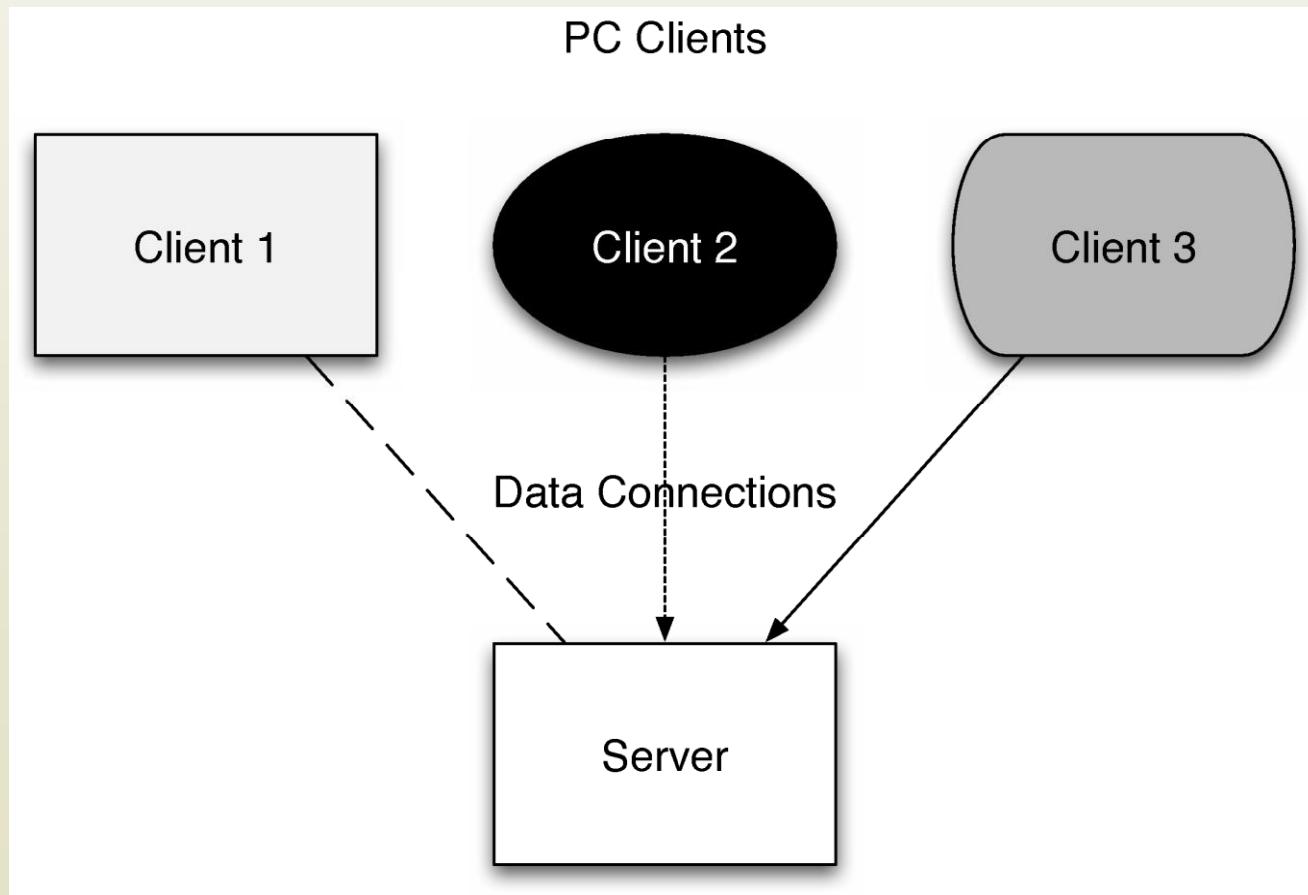
New Visualizations: Anti-Guidelines

- Same Symbol, Different Meaning



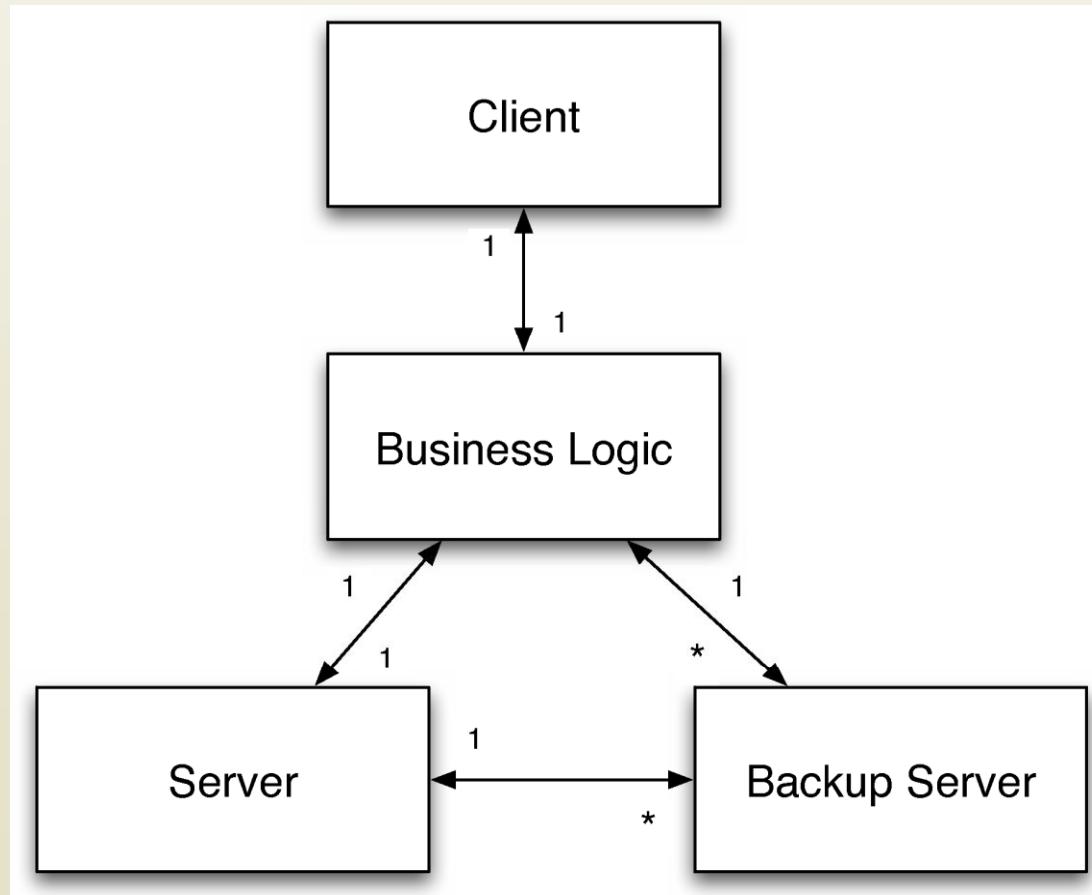
New Visualizations: Anti-Guidelines (cont'd)

- Differences without meaning



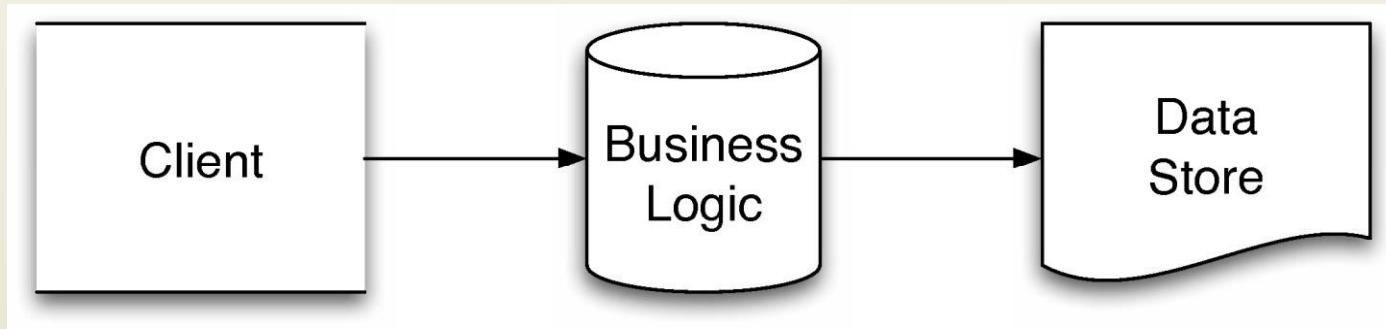
New Visualizations: Anti-Guidelines (cont'd)

- Decorations without meaning



New Visualizations: Anti-Guidelines (cont'd)

- Borrowed symbol, different meaning

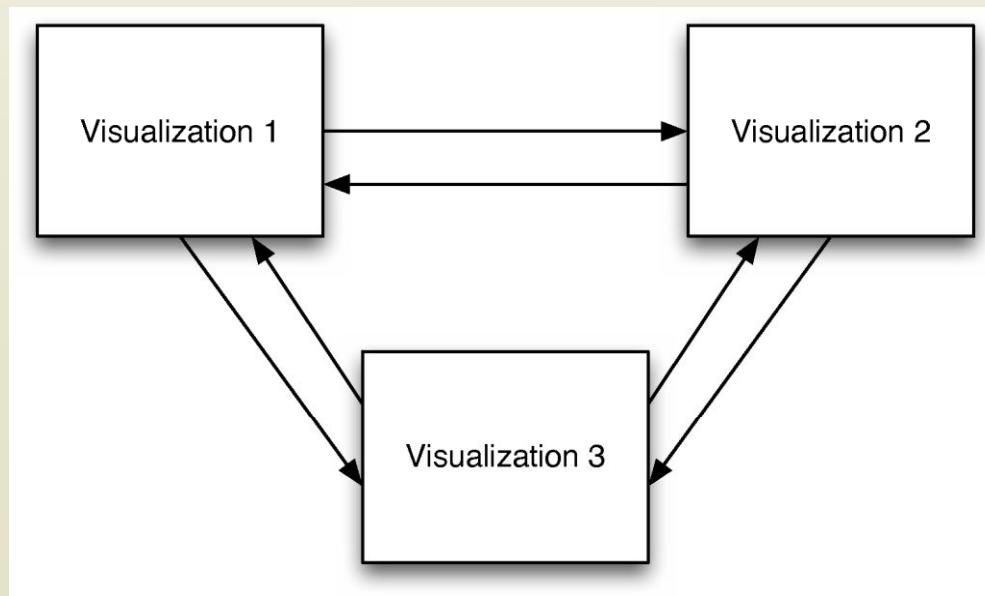


Coordinating Multiple Visualizations

- How do we keep multiple simultaneous visualizations of the same (part of the) architectural model consistent with each other and the model?
 - ◆ This is NOT the same as maintaining architectural consistency
 - ◆ If something is wrong with the model, this error would be reflected in the visualizations
- Can be made much easier by making simplifying assumptions, e.g.:
 - ◆ Only one visualization may operate at a time
 - ◆ Only one tool can operate on the model at a time
- But what if we can't simplify like this?

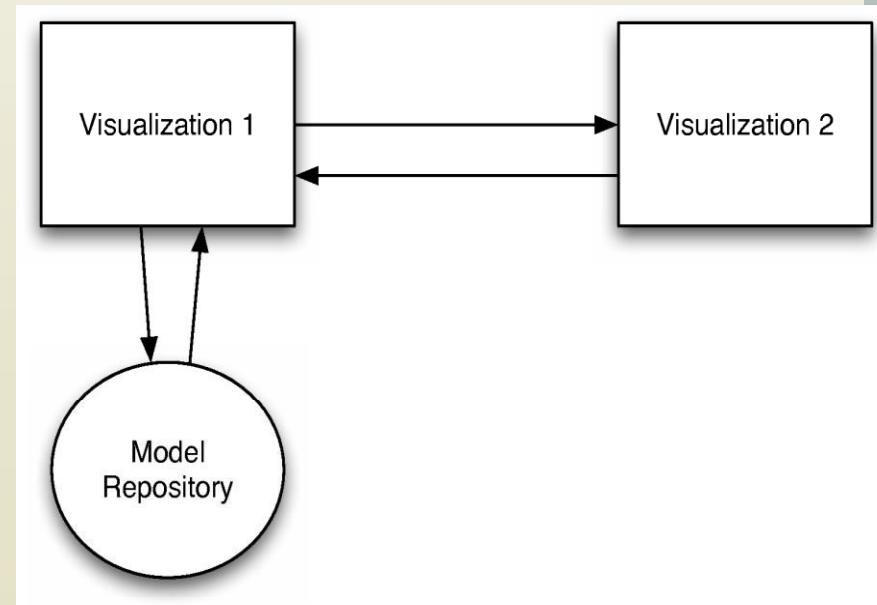
Strategy: Peer-to-Peer Coordination

- Each visualization communicates with each other visualization for updates
 - ◆ Has scaling problems
 - ◆ Works best for visualizations known *a priori*



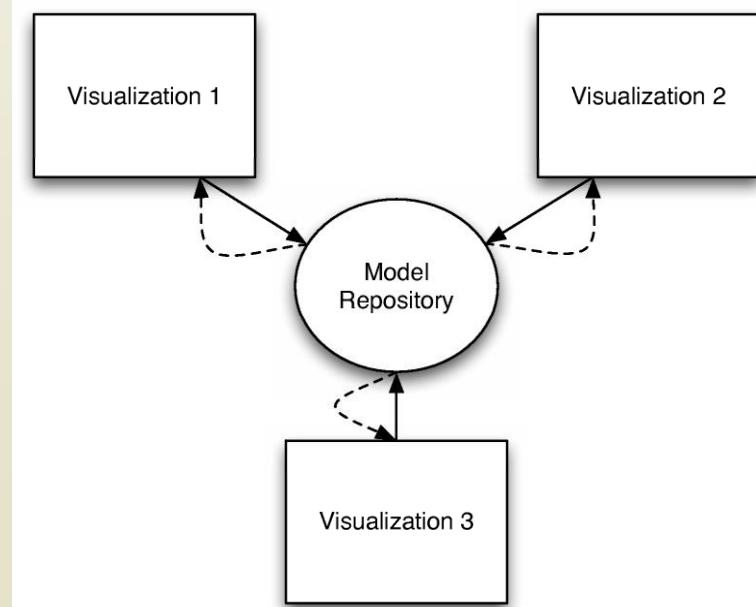
Strategy: Master-Slave

- One visualization is the master and others coordinate through it
- Works best when visualizations are subordinate
 - ◆ E.g., a “thumbnail” or “overview” next to a main, zoomed-in visualization



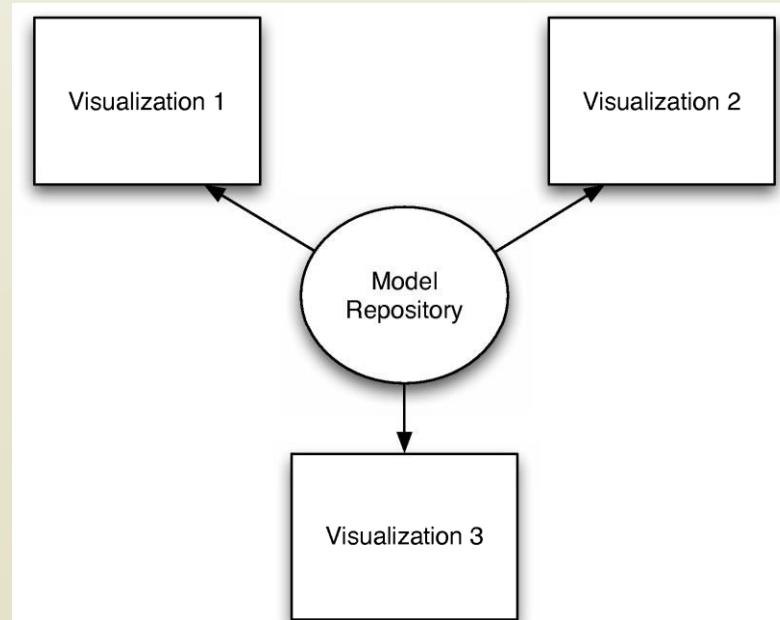
Strategy: Pull-based

- Visualizations repeatedly poll a model repository for changes
- Potential consistency/staleness problems
- May be necessary if model repository is entirely passive
- May save computing power



Strategy: Push-based

- Visualizations actively notified and update themselves whenever model changes for any reason
- Best for multiple simultaneous visualizations
- Hard to debug, must avoid infinite loops and subtle concurrency conditions



Caveats

- Like the modeling lectures, this optimized for breadth rather than depth
 - ◆ You are encouraged to explore these in depth, as well as visualizations you encounter in your own experiences
- Although we can attempt to conceptually separate modeling notations and visualizations, they are never truly distinct
 - ◆ Each influences the other in direct and indirect ways

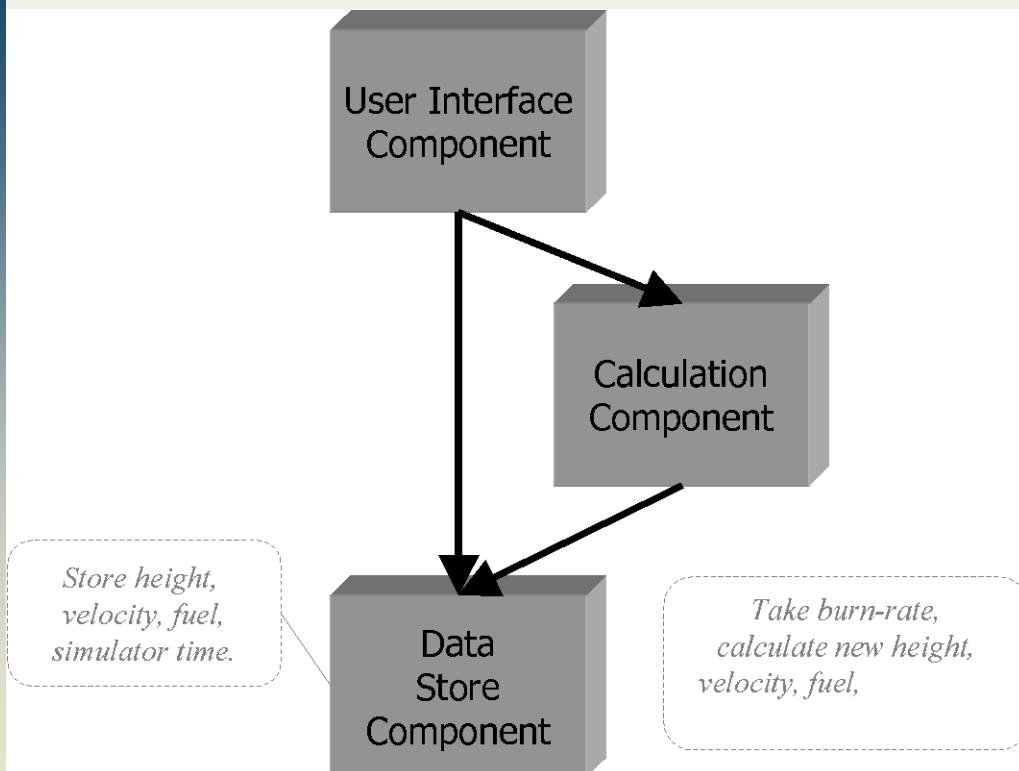
Analysis of Software Architectures

**Software Architecture
Lecture 13**

What Is Architectural Analysis?

- Architectural analysis is the activity of discovering important system properties using the system's architectural models.
 - ◆ Early, useful answers about relevant architectural aspects
 - ◆ Available prior to system's construction
- Important to know
 1. which questions to ask
 2. why to ask them
 3. how to ask them
 4. how to ensure that they can be answered

Informal Architectural Models and Analysis



- Helps architects get clarification from system customers
- Helps managers ensure project scope
- Not as useful to developers

Formal Architectural Models and Analysis

```
Component UserInterface
  Port getValues
  Port calculate
  Computation
Connector Call
  Role Caller =
  Role Callee =
  Glue =
Configuration LunarLander
  Instances
    DS : DataStore
    C : Calculation
    UI : UserInterface
    CtoUIgetValues, CtoUIstoreValues, UItoC, UItoDS : Call
Attachments
  C.getValues as CtoUIgetValues.Caller
  DS.getValues as CtoUIgetValues.Callee
  C.storeValues as CtoUIstoreValues.Caller
  DS.storeValues as CtoUIstoreValues.Callee
  UI.calculate as UItoC.Caller
  C.calculate as UItoC.Callee
  UI.getValues as UItoDS.Caller
  DS.getValues as UItoDS.Callee
End LunarLander.
```

- Helps architects determine component composability
- Helps developers with implementation-level decisions
- Helps with locating and selecting appropriateOTS components
- Helps with automated code generation
- Not as useful for discussions with non-technical stakeholders

Concerns Relevant to Architectural Analysis

- Goals of analysis
- Scope of analysis
- Primary architectural concern being analyzed
- Level of formality of architectural models
- Type of analysis
- Level of automation
- System stakeholders interested in analysis

Architectural Analysis Goals

- The four “C”s
 - ◆ Completeness
 - ◆ Consistency
 - ◆ Compatibility
 - ◆ Correctness

Architectural Analysis Goals – Completeness

- Completeness is both an external and an internal goal
- It is *external*/with respect to system requirements
 - ◆ Challenged by the complexity of large systems' requirements and architectures
 - ◆ Challenged by the many notations used to capture complex requirements as well as architectures
- It is *internal*/with respect to the architectural intent and modeling notation
 - ◆ Have all elements been fully modeled in the notation?
 - ◆ Have all design decisions been properly captured?

Architectural Analysis Goals – Consistency

- Consistency is an internal property of an architectural model
- Ensures that different model elements do not contradict one another
- Dimensions of architectural consistency
 - ◆ Name
 - ◆ Interface
 - ◆ Behavior
 - ◆ Interaction
 - ◆ Refinement

Name Consistency

- Component and connector names
- Component service names
- May be non-trivial to establish at the architectural level
 - ◆ Multiple system elements/services with identical names
 - ◆ Loose coupling via publish-subscribe or asynchronous event broadcast
 - ◆ Dynamically adaptable architectures

Interface Consistency

- Encompasses name consistency
- Also involves parameter lists in component services
- A rich spectrum of choices at the architectural level
- Example: matching provided and required interfaces

```
ReqInt:    getSubQ(Natural first, Natural last, Boolean remove)  
           returns FIFOQueue;
```

```
ProvInt1:  getSubQ(Index first, Index last)  
           returns FIFOQueue;
```

```
ProvInt2:  getSubQ(Natural first, Natural last, Boolean remove)  
           returns Queue;
```

Behavioral Consistency

- Names and interfaces of interacting components may match, but behaviors need not
- Example: subtraction

```
subtract(Integer x, Integer y) returns Integer;
```

- Can we be sure what the *subtract* operation does?
- Example: QueueClient and QueueServer components

QueueClient

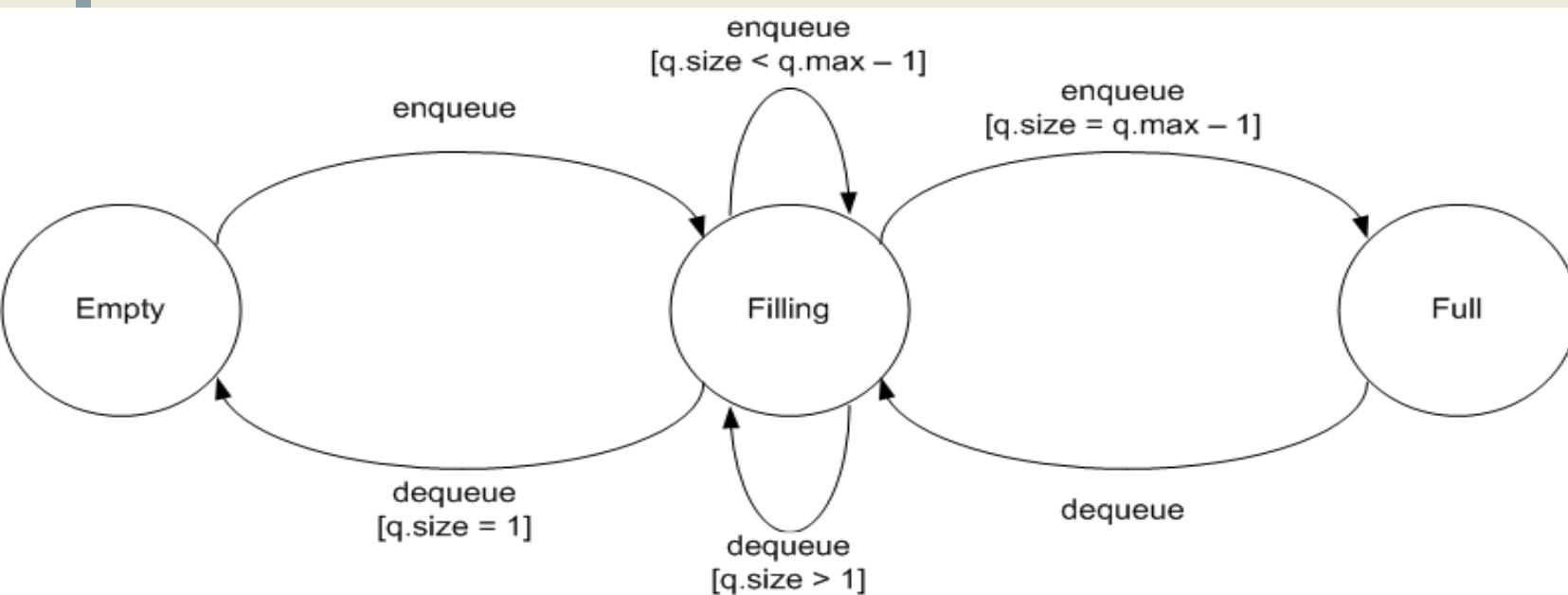
```
precondition q.size > 0;  
postcondition ~q.size = q.size;
```

QueueServer

```
precondition q.size > 1;  
postcondition ~q.size = q.size - 1;
```

Interaction Consistency

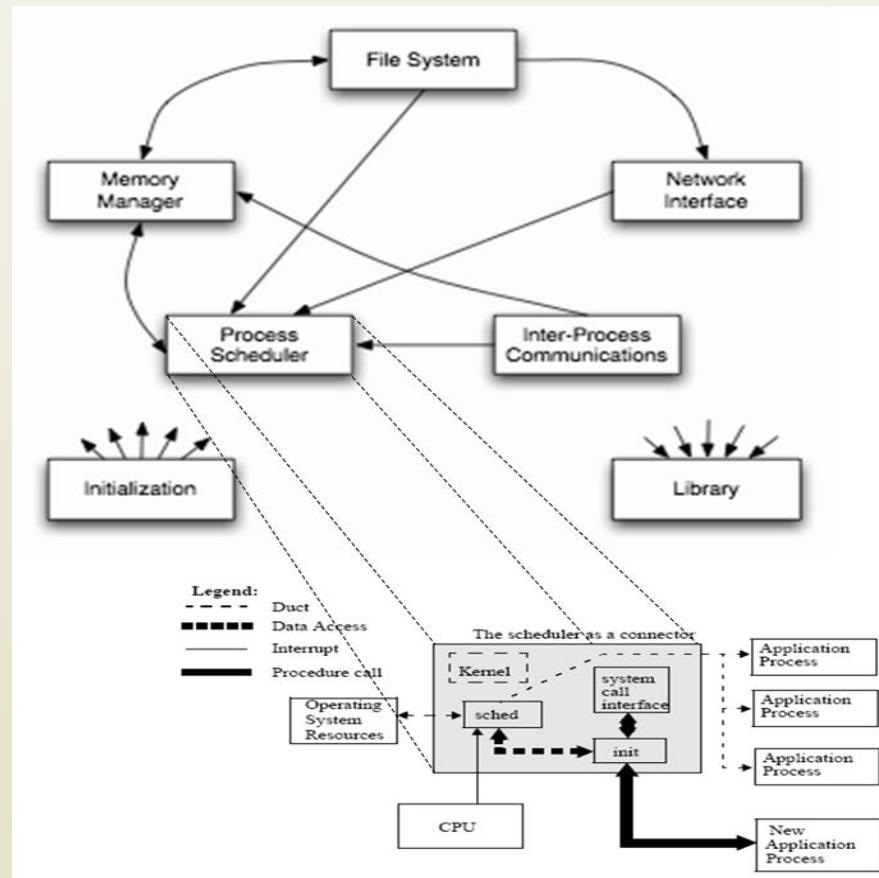
- Names, interfaces, and behaviors of interacting components may match, yet they may still be unable to interact properly
- Example: QueueClient and QueueServer components



Refinement Consistency

- Architectural models are refined during the design process
- A relationship must be maintained between higher and lower level models
 - ◆ All elements are preserved in the lower level model
 - ◆ All design decisions are preserved in the lower-level model
 - ◆ No new design decisions violate existing design decisions

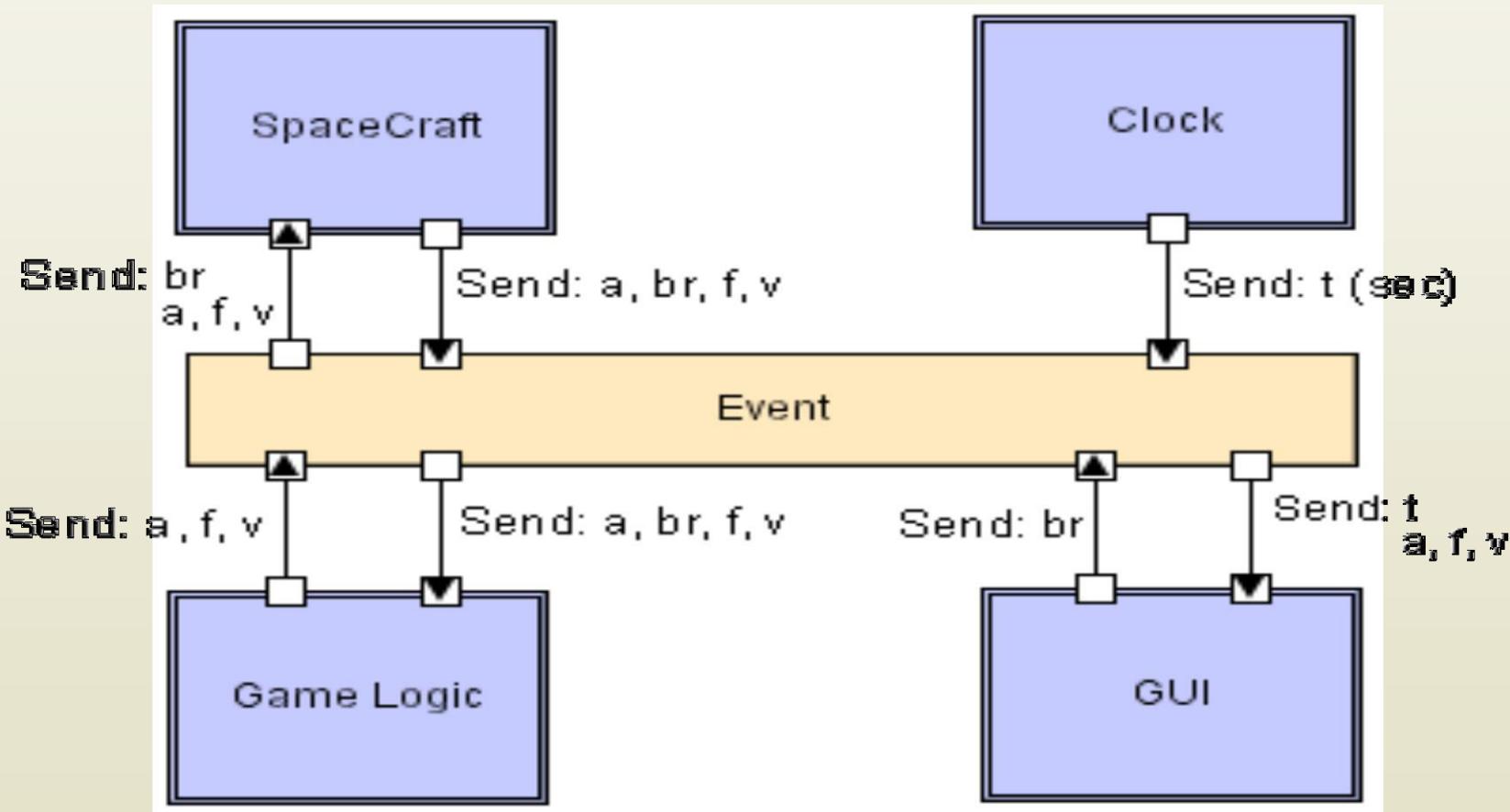
Refinement Consistency Example



Architectural Analysis Goals – Compatibility

- Compatibility is an external property of an architectural model
- Ensures that the architectural model adheres to guidelines and constraints of
 - ◆ a style
 - ◆ a reference architecture
 - ◆ an architectural standard

Architectural Compatibility Example – Lunar Lander



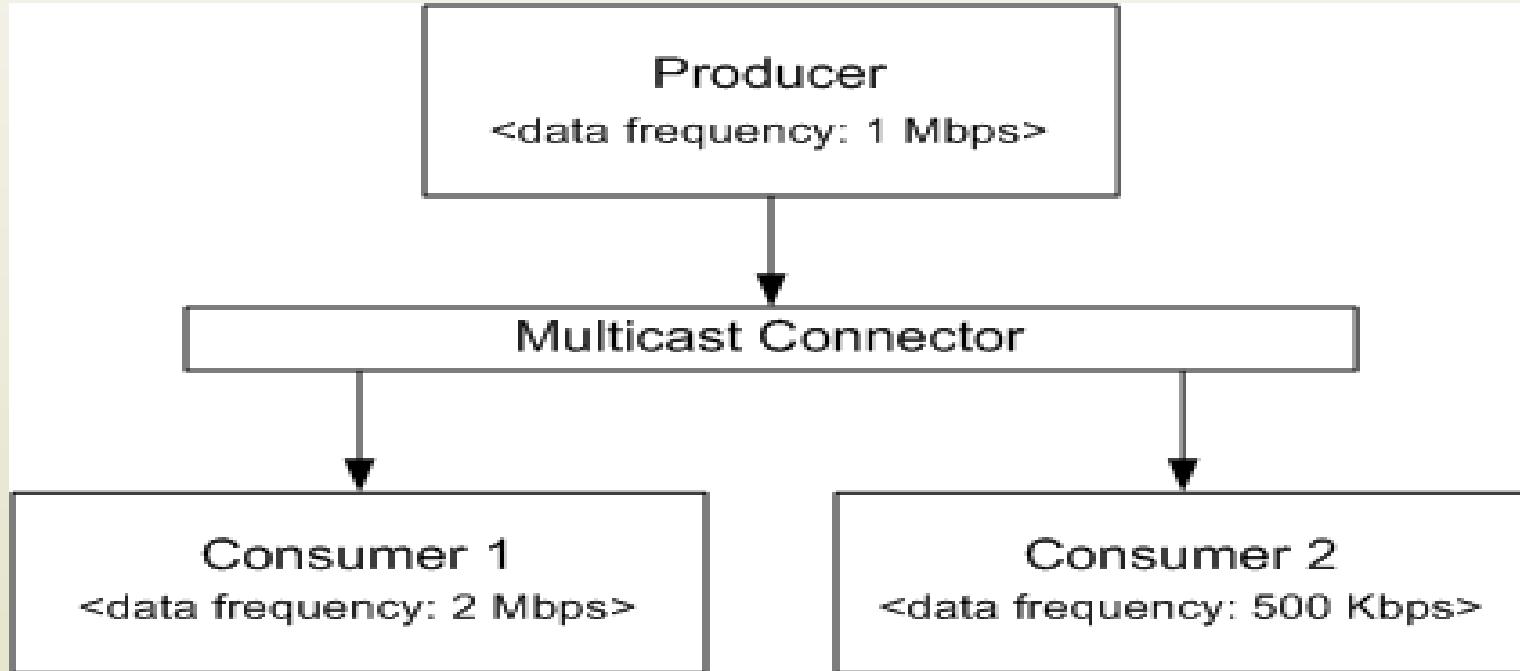
Architectural Analysis Goals – Correctness

- Correctness is an external property of an architectural model
- Ensures that
 1. the architectural model fully realizes a system specification
 2. the system's implementation fully realizes the architecture
- Inclusion of OTS elements impacts correctness
 - ◆ System may include structural elements, functionality, and non-functional properties that are not part of the architecture
 - ◆ The notion of *fulfillment* is key to ensuring architectural correctness

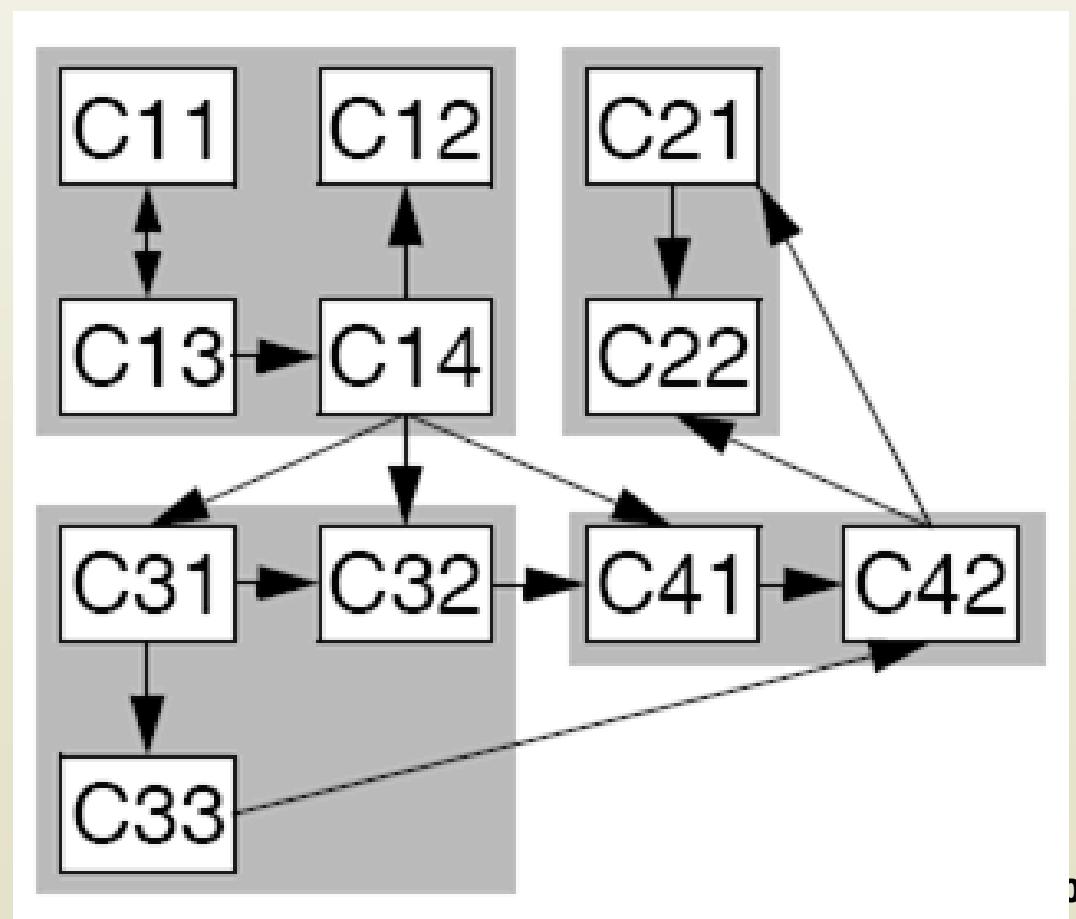
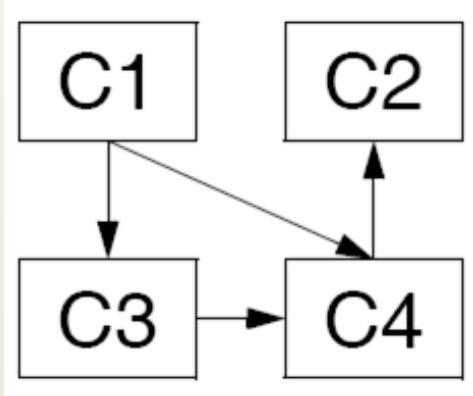
Scope of Architectural Analysis

- Component- and connector-level
- Subsystem- and system-level
 - ◆ Beware of the “honey-baked ham” syndrome
- Data exchanged in a system or subsystem
 - ◆ Data structure
 - ◆ Data flow
 - ◆ Properties of data exchange
- Architectures at different abstraction levels
- Comparison of two or more architectures
 - ◆ Processing
 - ◆ Data
 - ◆ Interaction
 - ◆ Configuration
 - ◆ Non-functional properties

Data Exchange Example



Architectures at Different Abstraction Levels



Architectural Concern Being Analyzed

- Structural characteristics
- Behavioral characteristics
- Interaction characteristics
- Non-functional characteristics

Level of Formality

- Informal models
- Semi-formal models
- Formal models

Type of Analysis

- Static analysis
- Dynamic analysis
- Scenario-driven analysis
 - ◆ Can be both static and dynamic

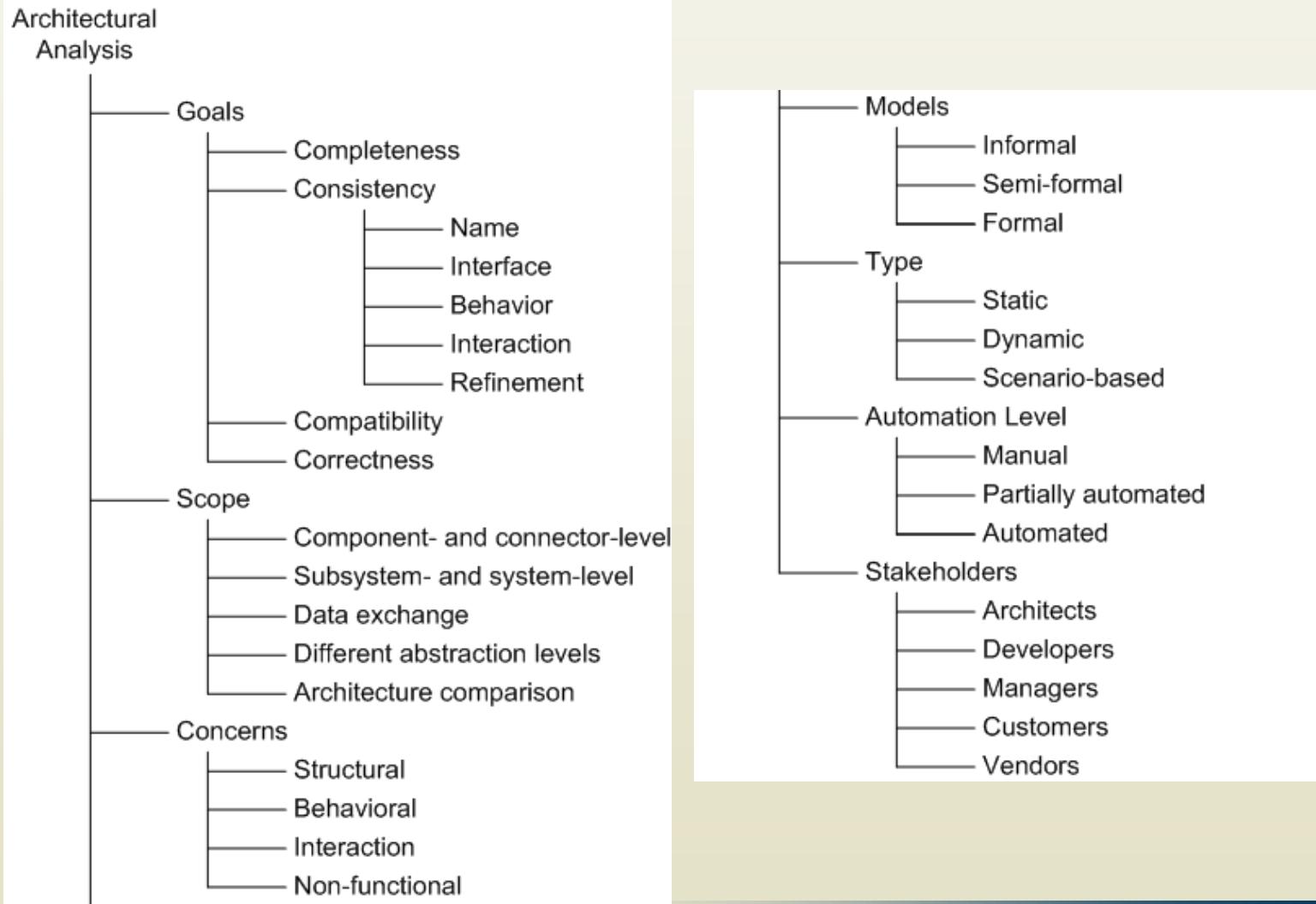
Level of Automation

- Manual
- Partially automated
- Fully automated

Analysis Stakeholders

- Architects
- Developers
- Managers
- Customers
- Vendors

Architectural Analysis in a Nutshell

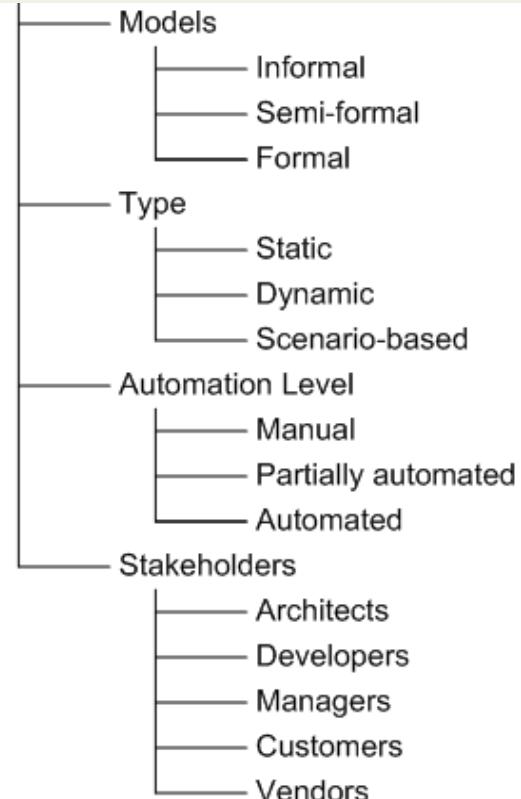
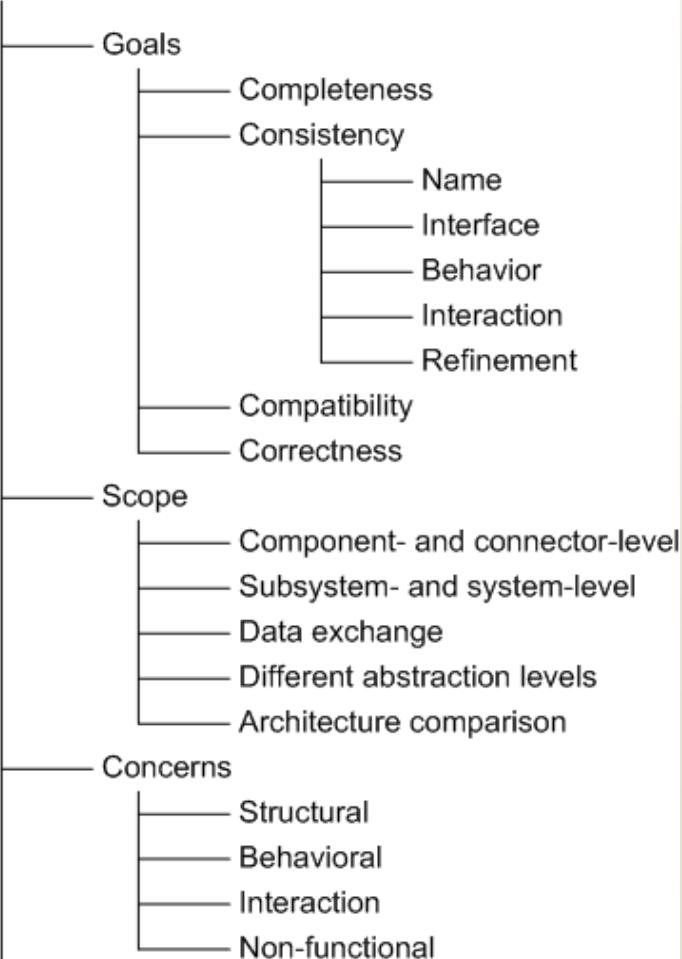


Analysis Techniques

Software Architecture
Lecture 14

Architectural Analysis in a Nutshell

Architectural Analysis



Analysis Technique Categories

- Inspection- and review-based
- Model-based
- Simulation-based

Architectural Inspections and Reviews

- Architectural models studied by human stakeholders for specific properties
- The stakeholders define analysis objective
- Manual techniques
 - ◆ Can be expensive
- Useful in the case of informal architectural descriptions
- Useful in establishing “soft” system properties
 - ◆ E.g., scalability or adaptability
- Able to consider multiple stakeholders’ objectives and multiple architectural properties

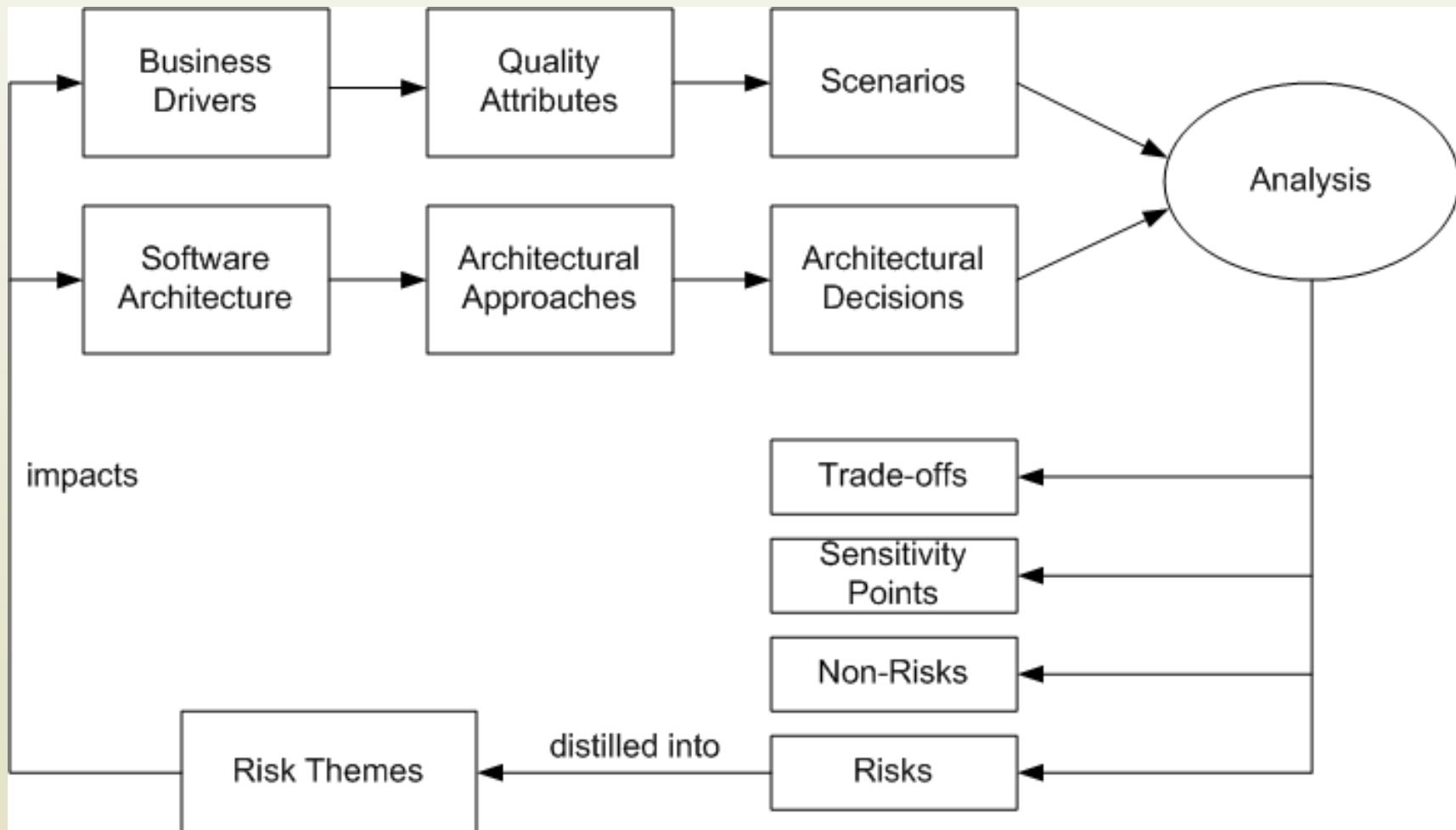
Inspections and Reviews in a Nutshell

- *Analysis Goals* – any
- *Analysis Scope* – any
- *Analysis Concern* – any, but particularly suited for non-functional properties
- *Architectural Models* – any, but must be geared to stakeholder needs and analysis objectives
- *Analysis Types* – mostly static and scenario-based
- *Automation Level* – manual, human intensive
- *Stakeholders* – any, except perhaps component vendors

Example – ATAM

- Stands for architectural trade-off analysis method
- Human-centric process for identifying risks early on in software design
- Focuses specifically on four quality attributes (NFPs)
 - ◆ Modifiability
 - ◆ Security
 - ◆ Performance
 - ◆ Reliability
- Reveals how well an architecture satisfies quality goals and how those goals trade-off

ATAM Process



ATAM Business Drivers

- The system's critical functionality
- Any technical, managerial, economic, or political constraints
- The project's business goals and context
- The major stakeholders
- The principal quality attribute (NFP) goals

ATAM Scenarios

- Use-case scenarios
 - ◆ Describe how the system is envisioned by the stakeholders to be used
- Growth scenarios
 - ◆ Describe planned and envisioned modifications to the architecture
- Exploratory scenarios
 - ◆ Try to establish the limits of architecture's adaptability with respect to
 - system's functionality
 - operational profiles
 - underlying execution platforms
 - ◆ Scenarios are prioritized based on importance to stakeholders

ATAM Architecture

- Technical constraints
 - ◆ Required hardware platforms, OS, middleware, programming languages, and OTS functionality
- Any other systems with which the system must interact
- *Architectural approaches* that have been used to meet the quality requirements
 - ◆ Sets of architectural design decisions employed to solve a problem
 - ◆ Typically architectural patterns and styles

ATAM Analysis

- Key step in ATAM
- Objective is to establish relationship between architectural approaches and quality attributes
- For each architectural approach a set of analysis questions are formulated
 - ◆ Targeted at the approach and quality attributes in question
- System architects and ATAM evaluation team work together to answer these questions and identify
 - ◆ Risks → these are distilled into risk *themes*
 - ◆ Non-Risks
 - ◆ Sensitivity points
 - ◆ Trade-off points
- Based on answers, further analysis may be performed

ATAM in a Nutshell

Goals	Completeness Consistency Compatibility Correctness
Scope	Subsystem- and system-level Data exchange
Concern	Non-functional
Models	Informal Semi-formal
Type	Scenario-driven
Automation Level	Manual
Stakeholders	Architects Developers Managers Customers

Model-Based Architectural Analysis

- Analysis techniques that manipulate architectural description to discover architectural properties
- Tool-driven, hence potentially less costly
- Typically useful for establishing “hard” architectural properties only
 - ◆ Unable to capture design intent and rationale
- Usually focus on a single architectural aspect
 - ◆ E.g., syntactic correctness, deadlock freedom, adherence to a style
- Scalability may be an issue
- Techniques typically used in tandem to provide more complete answers

Model-Based Analysis in a Nutshell

- *Analysis Goals* – consistency, compatibility, internal correctness
- *Analysis Scope* – any
- *Analysis Concern* – structural, behavioral, interaction, and possibly non-functional properties
- *Architectural Models* – semi-formal and formal
- *Analysis Types* – static
- *Automation Level* – partially and fully automated
- *Stakeholders* – mostly architects and developers

Model-Based Analysis in ADLs

- Wright – uses CSP to analyze for deadlocks
- Aesop – ensures style-specific constraints
- MetaH and UniCon – support schedulability analysis via NFPs such as component criticality and priority
- ADL parsers and compilers – ensure syntactic and semantic correctness
 - ◆ E.g., Rapide's generation of executable architectural simulations
- Architectural constraint enforcement
 - ◆ E.g., Armani or UML's OCL
- Architectural refinement
 - ◆ E.g., SADL and Rapide

ADLs' Analysis Foci in a Nutshell

Goals	Consistency Compatibility Completeness (internal)
Scope	Component- and connector-level Subsystem- and system-level Data exchange Different abstraction levels Architecture comparison
Concern	Structural Behavioral Interaction Non-functional
Models	Semi-formal Formal
Type	Static
Automation Level	Partially automated Automated
Stakeholders	Architects Developers Managers Customers

Architectural Reliability Analysis

- *Reliability* is the probability that the system will perform its intended functionality under specified design limits, without failure
- A *failure* is the occurrence of an incorrect output as a result of an input value that is received, with respect to the specification
- An *error* is a mental mistake made by the designer or programmer
- A *fault* or a *defect* is the manifestation of that error in the system
 - ◆ An abnormal condition that may cause a reduction in, or loss of, the capability of a component to perform a required function
 - ◆ A requirements, design, or implementation flaw or deviation from a desired or intended state

Reliability Metrics

- Time to failure
- Time to repair
- Time between failures

Assessing Reliability at Architectural Level

- Challenged by unknowns
 - ◆ Operational profile
 - ◆ Failure and recovery history
- Challenged by uncertainties
 - ◆ Multiple development scenarios
 - ◆ Varying granularity of architectural models
 - ◆ Different information sources about system usage
- Architectural reliability values must be qualified by assumptions made to deal with the above uncertainties
- Reliability modeling techniques are needed that deal effectively with uncertainties
 - ◆ E.g., Hidden Markov Models (HMMs)

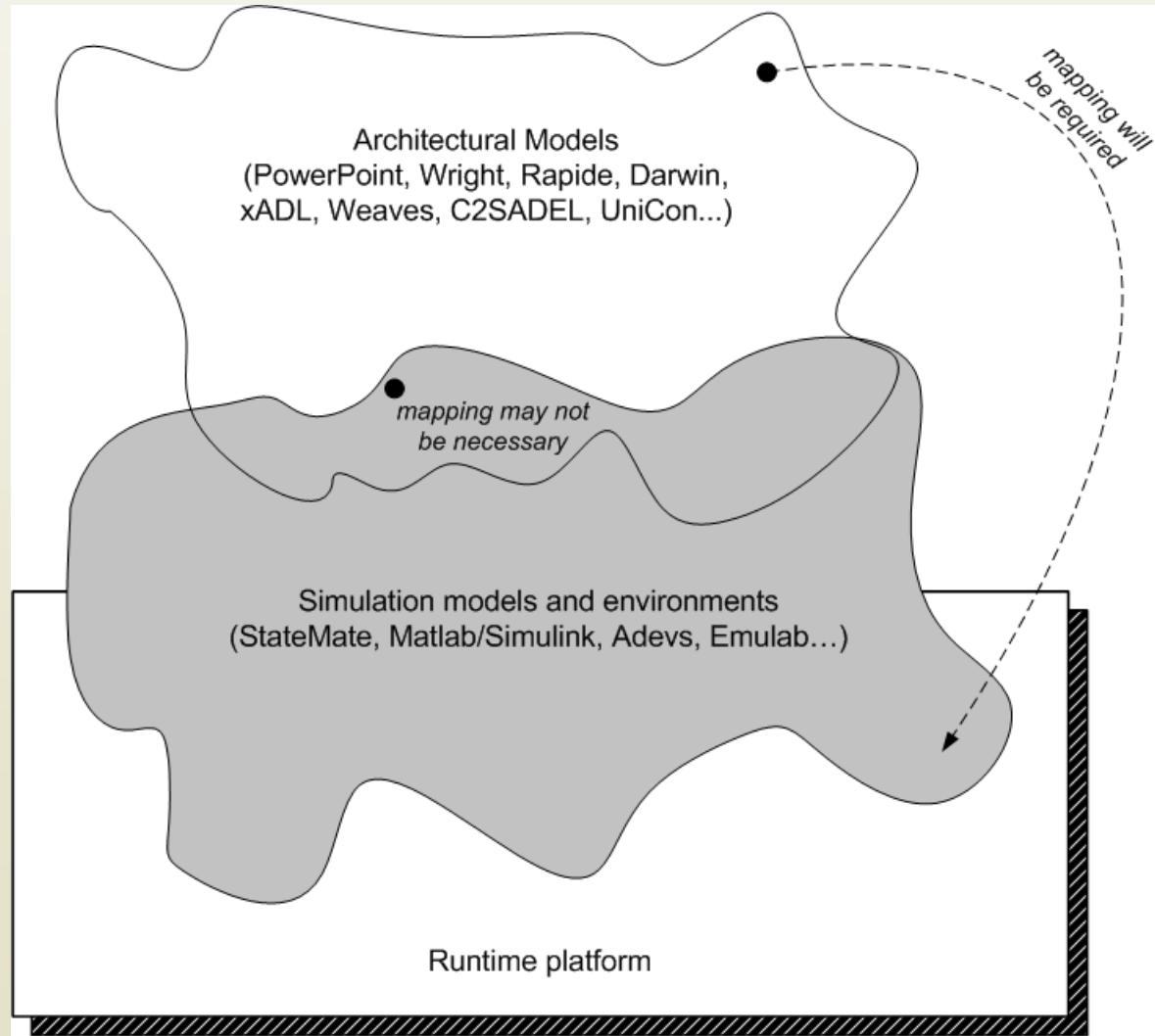
Architectural Reliability Analysis in a Nutshell

Goals	Consistency Compatibility Correctness
Scope	Component- and connector-level Subsystem- and system-level
Concern	Non-functional
Models	Formal
Type	Static Scenario-based
Automation Level	Partially automated
Stakeholders	Architects Managers Customers Vendors

Simulation-Based Analysis

- Requires producing an executable system model
- Simulation need not exhibit identical behavior to system implementation
 - ◆ Many low-level system parameters may be unavailable
- It needs to be precise and not necessarily accurate
- Some architectural models may not be amenable to simulation
 - ◆ Typically require translation to a simulatable language

Architectural and Simulation Models



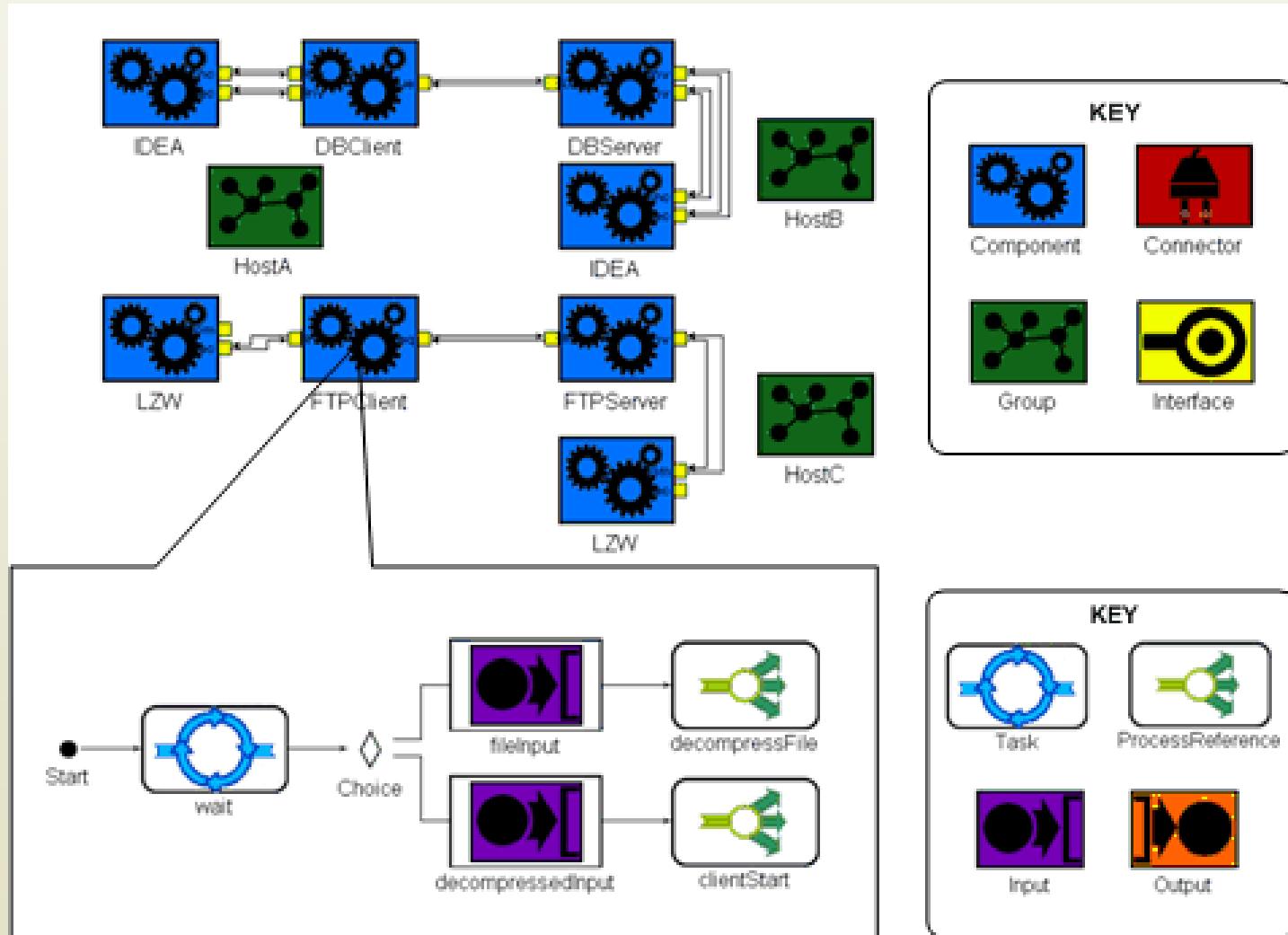
Simulation-Based Analysis in a Nutshell

- *Analysis Goals* – any
- *Analysis Scope* – any
- *Analysis Concern* – behavioral, interaction, and non-functional properties
- *Architectural Models* – formal
- *Analysis Types* – dynamic and scenario-based
- *Automation Level* – fully automated; model mapping may be manual
- *Stakeholders* – any

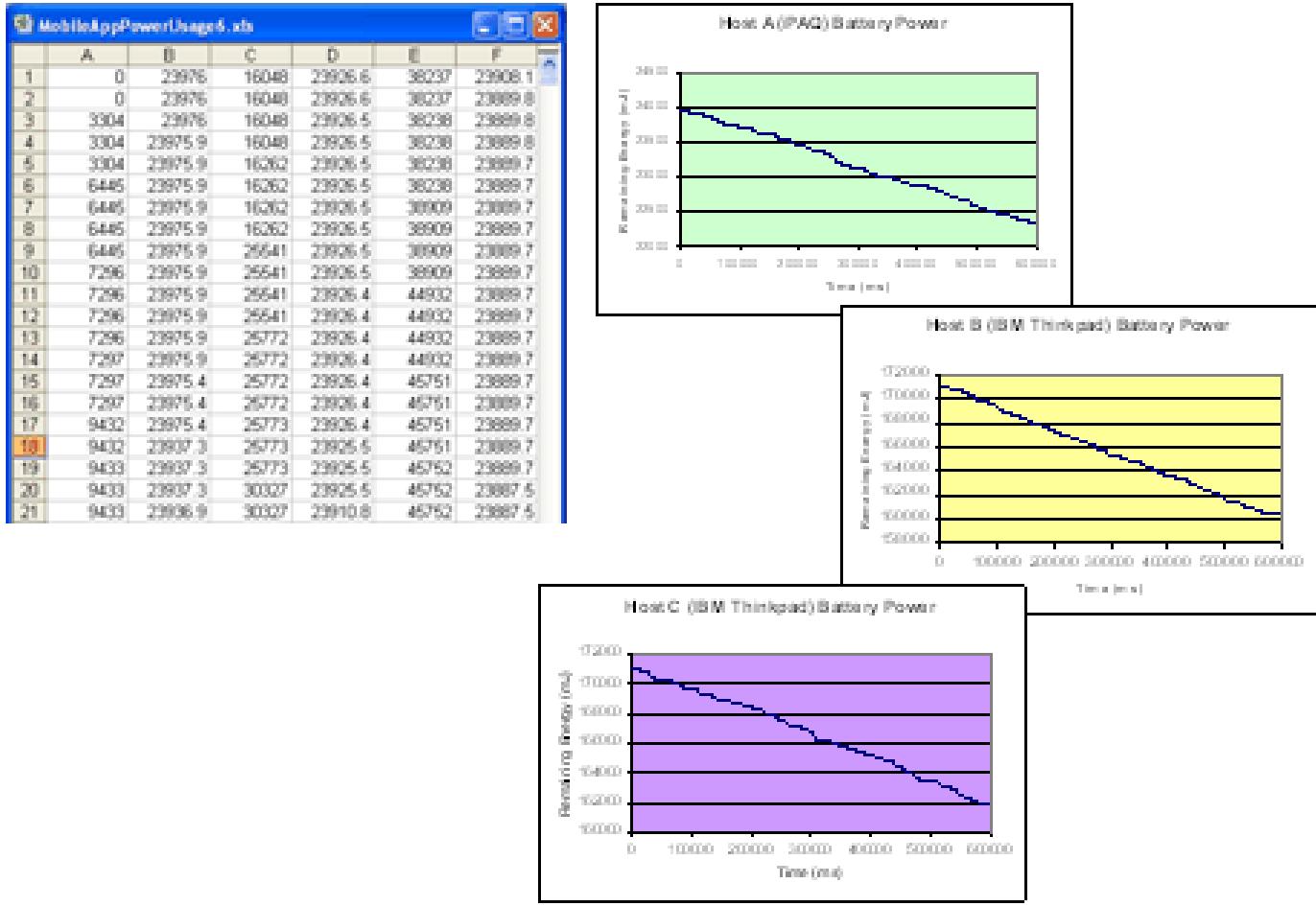
Example – XTEAM

- eXtensible Tool-chain for Evaluation of Architectural Models
- Targeted at mobile and resource-constrained systems
- Combines two underlying ADLs
 - ◆ xADL and FSP
- Maps architectural description to adevs
 - ◆ An OTS event simulation engine
- Implements different analyses via ADL extensions and a model interpreter
 - ◆ Latency, memory utilization, reliability, energy consumption

Example XTEAM Model



Example XTEAM Analysis



XTEAM in a Nutshell

Goals	Consistency Compatibility Correctness
Scope	Component- and connector-level Subsystem- and system-level Data exchange
Concern	Structural Behavioral Interaction Non-functional
Models	Formal
Type	Dynamic Scenario-based
Automation Level	Automated
Stakeholders	Architects Developers Managers Customers Vendors

Closing Remarks

- Architectural analysis is neither easy nor cheap
- The benefits typically far outweigh the drawbacks
- Early information about the system's key characteristics is indispensable
- Multiple analysis techniques often should be used in concert
- “How much analysis?”
 - ◆ This is the key facet of an architect’s job
 - ◆ Too many will expend resources unnecessarily
 - ◆ Too few will carry the risk of propagating defects into the final system
 - ◆ Wrong analyses will have both drawbacks

Implementing Architectures

**Software Architecture
Lecture 15**

Objectives

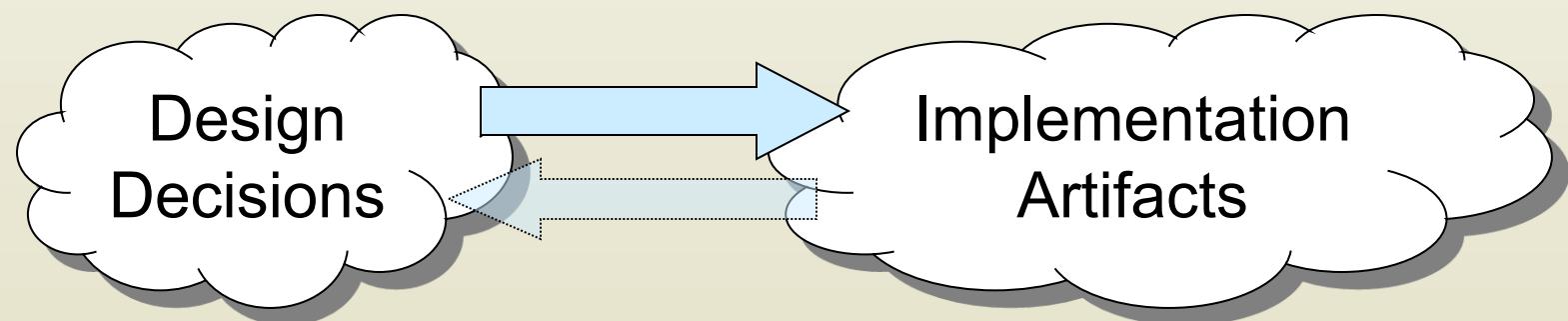
- Concepts
 - ◆ Implementation as a mapping problem
 - ◆ Architecture implementation frameworks
 - ◆ Evaluating frameworks
 - ◆ Relationships between middleware, frameworks, component models
 - ◆ Building new frameworks
 - ◆ Concurrency and generative technologies
 - ◆ Ensuring architecture-to-implementation consistency
- Examples
 - ◆ Different frameworks for pipe-and-filter
 - ◆ Different frameworks for the C2 style
- Application
 - ◆ Implementing Lunar Lander in different frameworks

Objectives

- Concepts
 - ◆ Implementation as a mapping problem
 - ◆ Architecture implementation frameworks
 - ◆ Evaluating frameworks
 - ◆ Relationships between middleware, frameworks, component models
 - ◆ Building new frameworks
 - ◆ Concurrency and generative technologies
 - ◆ Ensuring architecture-to-implementation consistency
- Examples
 - ◆ Different frameworks for pipe-and-filter
 - ◆ Different frameworks for the C2 style
- Application
 - ◆ Implementing Lunar Lander in different frameworks

The Mapping Problem

- Implementation is the one phase of software engineering that is not optional
- Architecture-based development provides a unique twist on the classic problem
 - ◆ It becomes, in large measure, a *mapping* activity



- Maintaining mapping means ensuring that our architectural intent is reflected in our constructed systems

Common Element Mapping

- Components and Connectors
 - ◆ Partitions of application computation and communication functionality
 - ◆ Modules, packages, libraries, classes, explicit components/connectors in middleware
- Interfaces
 - ◆ Programming-language level interfaces (e.g., APIs/function or method signatures) are common
 - ◆ State machines or protocols are harder to map

Common Element Mapping (cont'd)

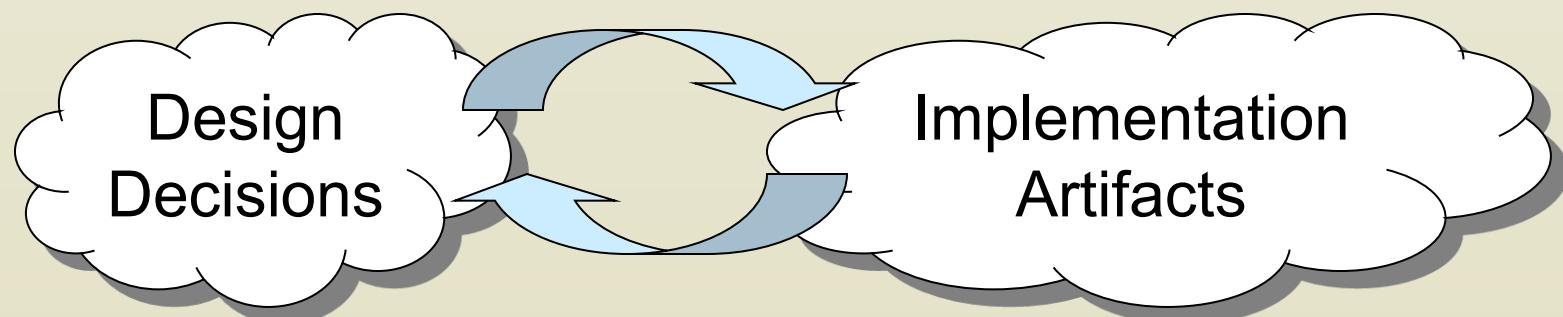
- Configurations
 - ◆ Interconnections, references, or dependencies between functional partitions
 - ◆ May be implicit in the implementation
 - ◆ May be externally specified through a MIL and enabled through middleware
 - ◆ May involve use of reflection
- Design rationale
 - ◆ Often does not appear directly in implementation
 - ◆ Retained in comments and other documentation

Common Element Mapping (cont'd)

- Dynamic Properties (e.g., behavior):
 - ◆ Usually translate to algorithms of some sort
 - ◆ Mapping strategy depends on how the behaviors are specified and what translations are available
 - ◆ Some behavioral specifications are more useful for generating analyses or testing plans
- Non-Functional Properties
 - ◆ Extremely difficult to do since non-functional properties are abstract and implementations are concrete
 - ◆ Achieved through a combination of human-centric strategies like inspections, reviews, focus groups, user studies, beta testing, and so on

One-Way vs. Round Trip Mapping

- Architectures inevitably change after implementation begins
 - ◆ For maintenance purposes
 - ◆ Because of time pressures
 - ◆ Because of new information
- Implementations can be a source of new information
 - ◆ We learn more about the feasibility of our designs when we implement
 - ◆ We also learn how to optimize them



One-Way vs. Round Trip Mapping (cont'd)

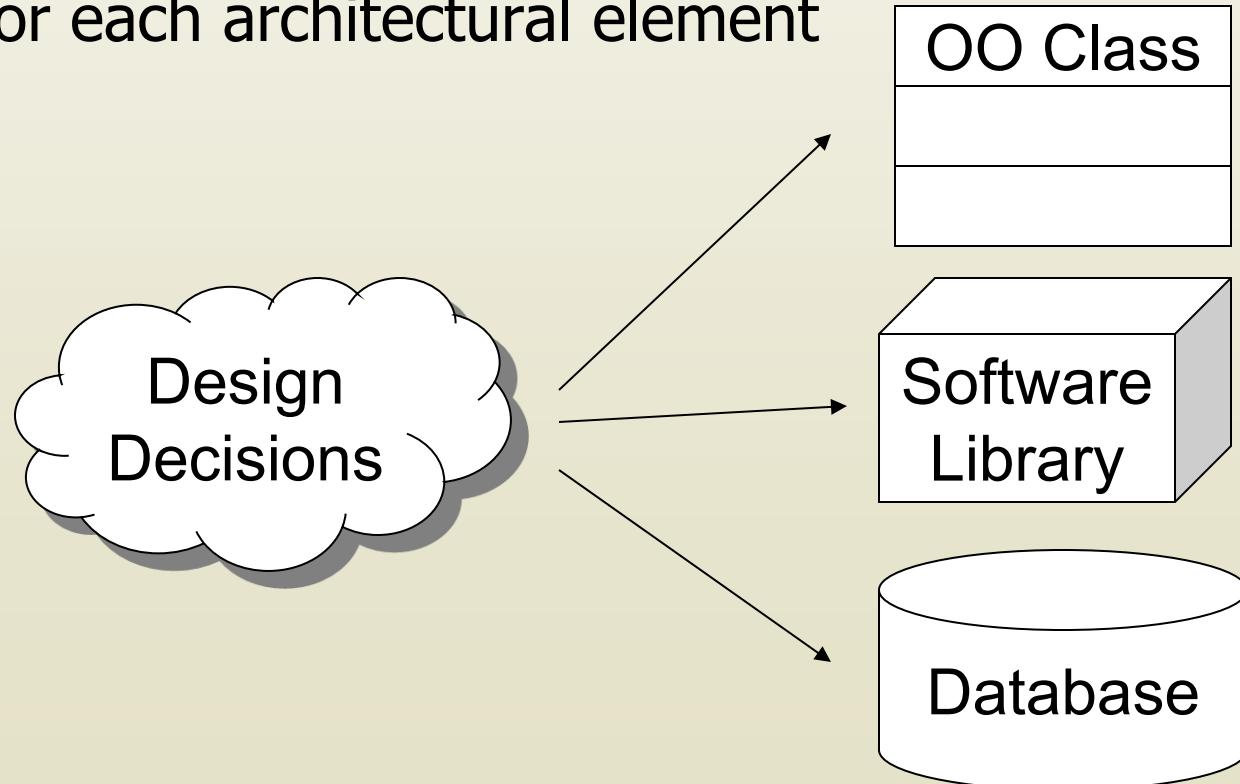
- Keeping the two in sync is a difficult technical and managerial problem
 - ◆ Places where strong mappings are not present are often the first to diverge
- One-way mappings are easier
 - ◆ Must be able to understand impact on implementation for an architectural design decision or change
- Two way mappings require more insight
 - ◆ Must understand how a change in the implementation impacts architecture-level design decisions

One-Way vs. Round Trip Mapping (cont'd)

- One strategy: limit changes
 - ◆ If all system changes must be done to the architecture first, only one-way mappings are needed
 - ◆ Works very well if many generative technologies in use
 - ◆ Often hard to control in practice; introduces process delays and limits implementer freedom
- Alternative: allow changes in either architecture or implementation
 - ◆ Requires round-trip mappings and maintenance strategies
 - ◆ Can be assisted (to a point) with automated tools

Architecture Implementation Frameworks

- Ideal approach: develop architecture based on a known style, select technologies that provide implementation support for each architectural element

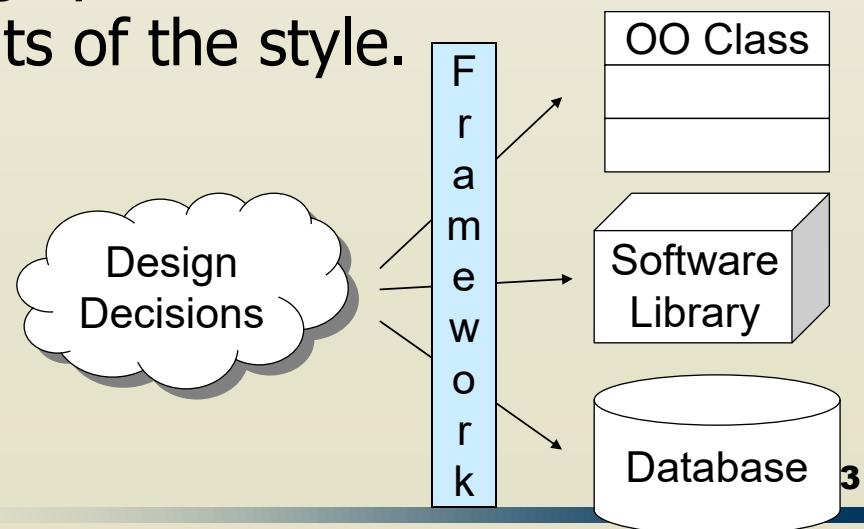


Architecture Implementation Frameworks

- This is rarely easy or trivial
 - ◆ Few programming languages have explicit support for architecture-level constructs
 - ◆ Support infrastructure (libraries, operating systems, etc.) also has its own sets of concepts, metaphors, and rules
- To mitigate these mismatches, we leverage an *architecture implementation framework*

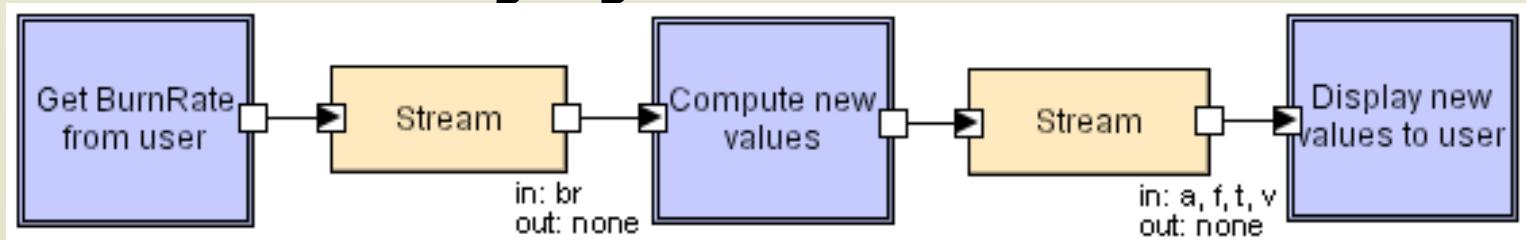
Architecture Implementation Frameworks

- **Definition:** An *architecture implementation framework* is a piece of software that acts as a bridge between a particular architectural style and a set of implementation technologies. It provides key elements of the architectural style *in code*, in a way that assists developers in implementing systems that conform to the prescriptions and constraints of the style.



Canonical Example

- The standard I/O ('stdio') framework in UNIX and other operating systems
 - ◆ Perhaps the most prevalent framework in use today
 - ◆ Style supported: pipe-and-filter
 - ◆ Implementation technologies supported: concurrent process-oriented operating system, (generally) non-concurrent language like C



More on Frameworks

- Frameworks are meant to assist developers in following a style
 - ◆ But generally do not *constrain* developers from violating a style if they really want to
- Developing applications in a target style does not *require* a framework
 - ◆ But if you follow good software engineering practices, you'll probably end up developing one anyway
- Frameworks are generally considered as underlying infrastructure or substrates from an architectural perspective
 - ◆ You won't usually see the framework show up in an architectural model, e.g., as a component

Same Style, Different Frameworks

- For a given style, there is no one perfect architecture framework
 - ◆ Different target implementation technologies induce different frameworks
 - stdio vs. iostream vs. java.io
- Even in the same (style/target technology) groupings, different frameworks exist due to different qualitative properties of frameworks
 - ◆ java.io vs. java.nio
 - ◆ Various C2-style frameworks in Java

Evaluating Frameworks

- Can draw out some of the qualitative properties just mentioned
- Platform support
 - ◆ Target language, operating system, other technologies
- Fidelity
 - ◆ How much style-specific support is provided by the framework?
 - Many frameworks are more general than one target style or focus on a subset of the style rules
 - ◆ How much enforcement is provided?

Evaluating Frameworks (cont'd)

- Matching Assumptions
 - ◆ Styles impose constraints on the target architecture/application
 - ◆ Frameworks can induce constraints as well
 - E.g., startup order, communication patterns ...
 - ◆ To what extent does the framework make too many (or too few) assumptions?
- Efficiency
 - ◆ Frameworks pervade target applications and can potentially get involved in any interaction
 - ◆ To what extent does the framework limit its slowdown and provide help to improve efficiency if possible (consider buffering in stdio)?

Evaluating Frameworks (cont'd)

- Other quality considerations
 - ◆ Nearly every other software quality can affect framework evaluation and selection
 - Size
 - Cost
 - Ease of use
 - Reliability
 - Robustness
 - Availability of source code
 - Portability
 - Long-term maintainability and support

Middleware and Component Models

- This may all sound similar to various kinds of middleware/component frameworks
 - ◆ CORBA, COM/DCOM, JavaBeans, .NET, Java Message Service (JMS), etc.
- They are closely related
 - ◆ Both provide developers with services not available in the underlying OS/language
 - ◆ CORBA provides well-defined interfaces, portability, remote procedure call...
 - ◆ JavaBeans provides a standardized packaging framework (the bean) with new kinds of introspection and binding

Middleware and Component Models (cont'd)

- Indeed, architecture implementation frameworks *are* forms of middleware
 - ◆ There's a subtle difference in how they emerge and develop
 - ◆ Middleware generally evolves based on a set of *services* that the developers want to have available
 - E.g., CORBA: Support for language heterogeneity, network transparency, portability
 - ◆ Frameworks generally evolve based on a particular *architectural style* that developers want to use
- Why is this important?

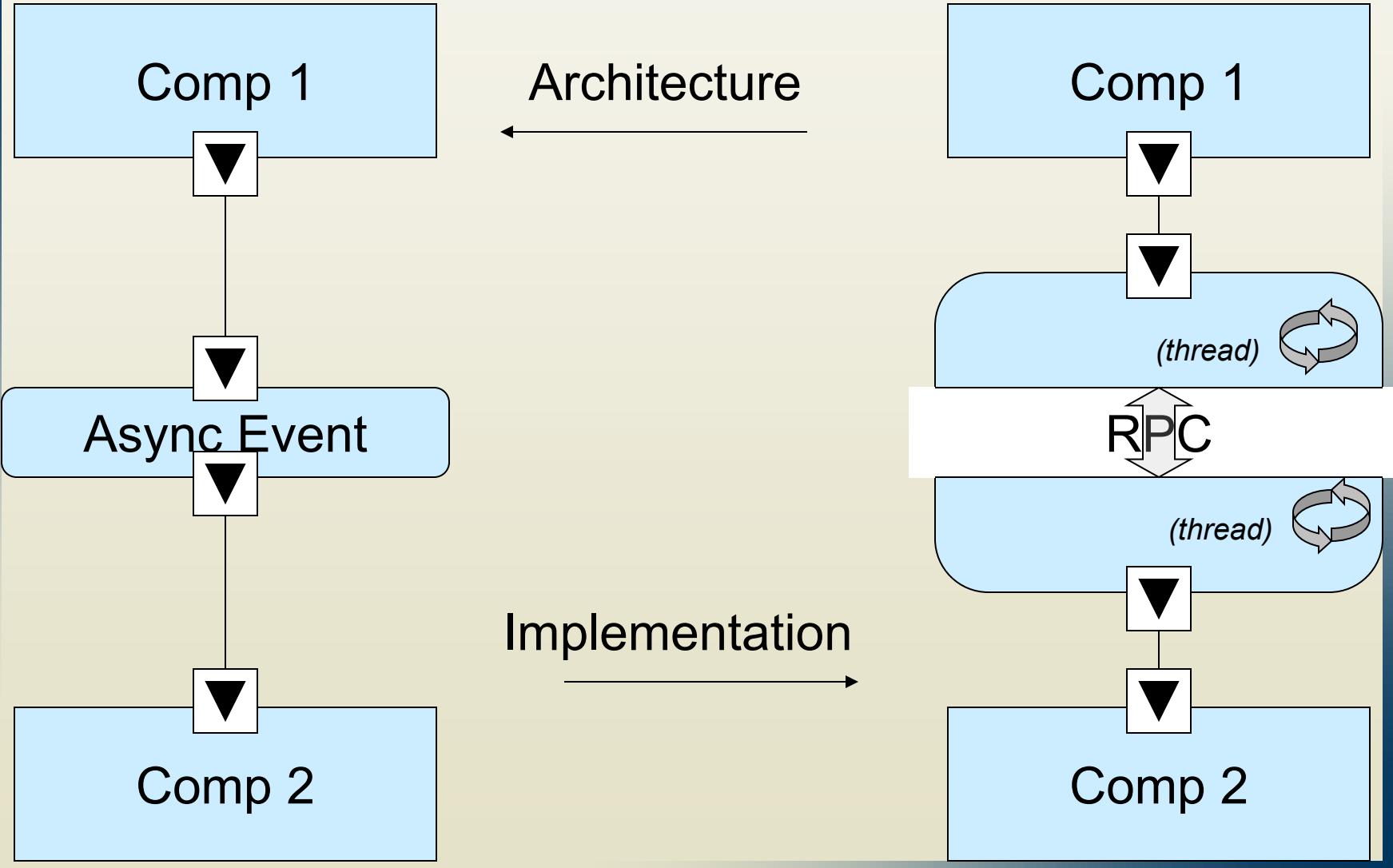
Middleware and Component Models (cont'd)

- By focusing on *services*, middleware developers often make other decisions that substantially impact architecture
- E.g., in supporting network transparency and language heterogeneity, CORBA uses RPC
 - ◆ But is RPC necessary for these services or is it just an enabling technique?
- In a very real way, middleware induces an architectural style
 - ◆ CORBA induces the 'distributed objects' style
 - ◆ JMS induces a distributed implicit invocation style
- Understanding these implications is essential for not having major problems when the tail wags the dog!

Resolving Mismatches

- A style is chosen first, but the middleware selected for implementation does not support (or contradicts) that style
 - A middleware is chosen first (or independently) and has undue influence on the architectural style used
 - Strategies
 - ◆ Change or adapt the style
 - ◆ Change the middleware selected
 - ◆ Develop glue code
 - ◆ Leverage parts of the middleware and ignore others
 - ◆ Hide the middleware in components/connectors
- 
- Use the middleware as the basis for a framework

Hiding Middleware in Connectors



Building a New Framework

- Occasionally, you need a new framework
 - ◆ The architectural style in use is novel
 - ◆ The architectural style is not novel but it is being implemented on a platform for which no framework exists
 - ◆ The architectural style is not novel and frameworks exist for the target platform, but the existing frameworks are inadequate
- Good framework development is extremely difficult
 - ◆ Frameworks pervade nearly every aspect of your system
 - ◆ Making changes to frameworks often means changing the entire system
 - ◆ A task for experienced developers/architects

New Framework Guidelines

- Understand the target style first
 - ◆ Enumerate all the rules and constraints in concrete terms
 - ◆ Provide example design patterns and corner cases
- Limit the framework to the rules and constraints of the style
 - ◆ Do not let a particular target application's needs creep into the framework
 - ◆ "Rule of three" for applications

New Framework Guidelines (cont'd)

- Choose the framework scope
 - ◆ A framework does not necessarily have to implement all possible stylistic advantages (e.g., dynamism or distribution)
- Avoid over-engineering
 - ◆ Don't add capabilities simply because they are clever or "cool", especially if known target applications won't use them
 - ◆ These often add complexity and reduce performance

New Framework Guidelines (cont'd)

- Limit overhead for application developers
 - ◆ Every framework induces some overhead (classes must inherit from framework base classes, communication mechanisms limited)
 - ◆ Try to put as little overhead as possible on framework users
- Develop strategies and patterns for legacy systems and components
 - ◆ Almost every large application will need to include elements that were not built to work with a target framework
 - ◆ Develop strategies for incorporating and wrapping these

Concurrency

- Concurrency is one of the most difficult concerns to address in implementation
 - ◆ Introduction of subtle bugs: deadlock, race conditions...
 - ◆ Another topic on which there are entire books written
- Concurrency is often an architecture-level concern
 - ◆ Decisions can be made at the architectural level
 - ◆ Done carefully, much concurrency management can be embedded into the architecture framework
- Consider our earlier example, or how pipe-and-filter architectures are made concurrent without direct user involvement

Generative Technologies

- With a sufficiently detailed architectural model, various implementation artifacts can be generated
 - ◆ Entire system implementations
 - Requires extremely detailed models including behavioral specifications
 - More feasible in domain-specific contexts
 - ◆ Skeletons or interfaces
 - With detailed structure and interface specifications
 - ◆ Compositions (e.g., glue code)
 - With sufficient data about bindings between two elements

Maintaining Consistency

- Strategies for maintaining one-way or round-trip mappings
 - ◆ Create and maintain traceability links from architectural implementation elements
 - Explicit links in a database, in architectural models, in code comments can all help with consistency checking
 - ◆ Make the architectural model part of the implementation
 - When the model changes, the implementation adapts automatically
 - May involve “internal generation”
 - ◆ Generate some or all of the implementation from the architecture

Implementation Techniques

**Software Architecture
Lecture 16**

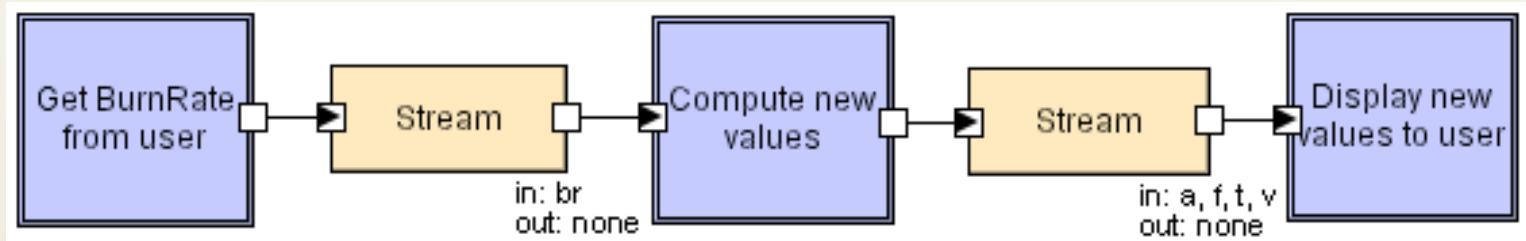
Objectives

- Concepts
 - ◆ Implementation as a mapping problem
 - ◆ Architecture implementation frameworks
 - ◆ Evaluating frameworks
 - ◆ Relationships between middleware, frameworks, component models
 - ◆ Building new frameworks
 - ◆ Concurrency and generative technologies
 - ◆ Ensuring architecture-to-implementation consistency
- Examples
 - ◆ Different frameworks for pipe-and-filter
 - ◆ Different frameworks for the C2 style
- Application
 - ◆ Implementing Lunar Lander in different frameworks

Objectives

- Concepts
 - ◆ Implementation as a mapping problem
 - ◆ Architecture implementation frameworks
 - ◆ Evaluating frameworks
 - ◆ Relationships between middleware, frameworks, component models
 - ◆ Building new frameworks
 - ◆ Concurrency and generative technologies
 - ◆ Ensuring architecture-to-implementation consistency
- Examples
 - ◆ Different frameworks for pipe-and-filter
 - ◆ Different frameworks for the C2 style
- Application
 - ◆ Implementing Lunar Lander in different frameworks

Recall Pipe-and-Filter



- Components ('filters') organized linearly, communicate through character-stream 'pipes,' which are the connectors
- Filters may run concurrently on partial data
- In general, all input comes in through the left and all output exits from the right

Framework #1: stdio

- Standard I/O framework used in C programming language
- Each process is a filter
 - ◆ Reads input from standard input (aka 'stdin')
 - ◆ Writes output to standard output (aka 'stdout')
 - Also a third, unbuffered output stream called standard error ('stderr') not considered here
 - ◆ Low and high level operations
 - `getchar(...)`, `putchar(...)` move one character at a time
 - `printf(...)` and `scanf(...)` move and format entire strings
 - ◆ Different implementations may vary in details (buffering strategy, etc.)

Evaluating stdio

- Platform support
 - ◆ Available with most, if not all, implementations of C programming language
 - ◆ Operates somewhat differently on OSes with no concurrency (e.g., MS-DOS)
- Fidelity
 - ◆ Good support for developing P&F applications, but no restriction that apps have to use this style
- Matching assumptions
 - ◆ Filters are processes and pipes are implicit. In-process P&F applications might require modifications
- Efficiency
 - ◆ Whether filters make maximal use of concurrency is partially up to filter implementations and partially up to the OS

Framework #2: java.io

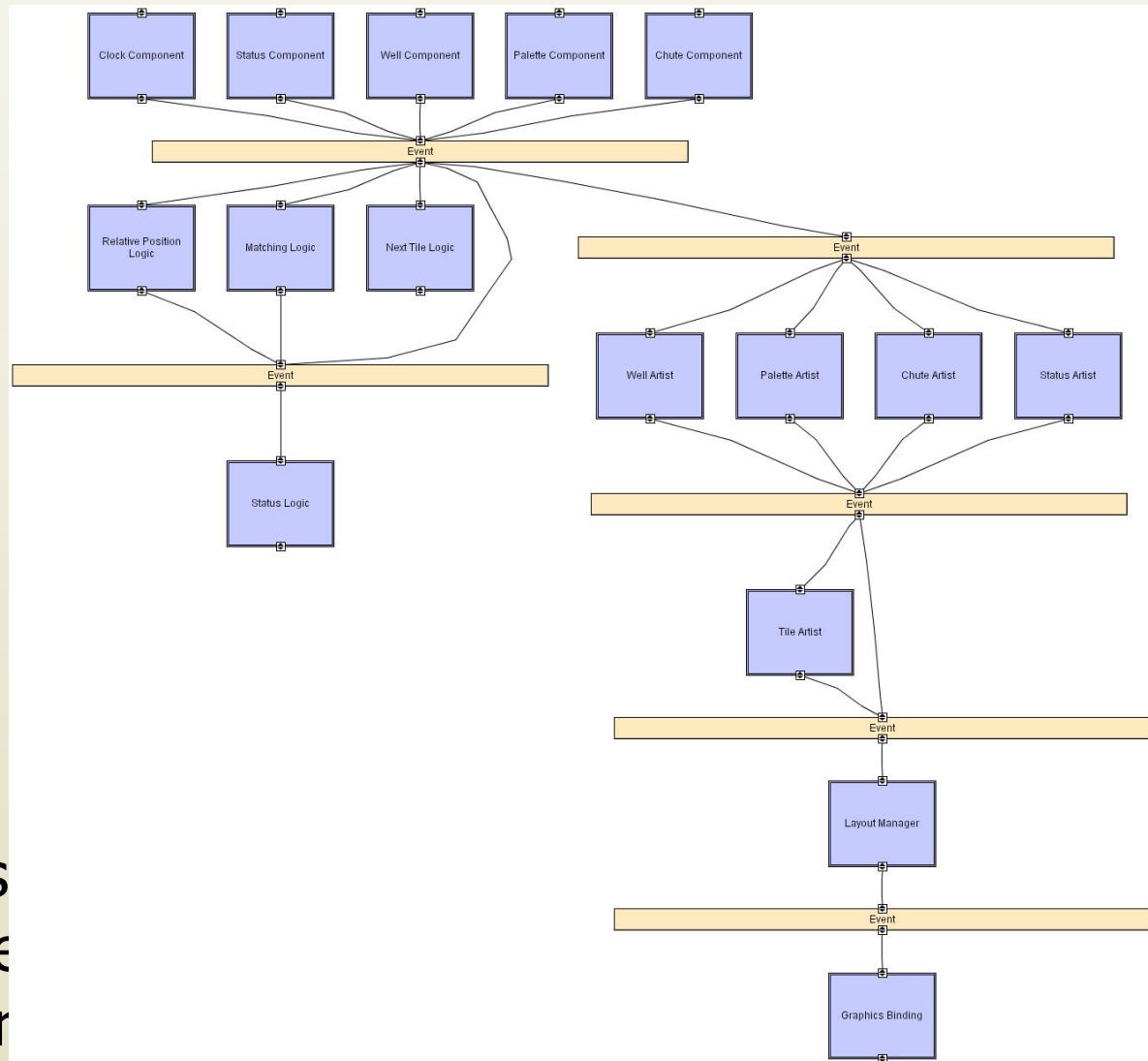
- Standard I/O framework used in Java language
- Object-oriented
- Can be used for in-process or inter-process P&F applications
 - ◆ All stream classes derive from `InputStream` or `OutputStream`
 - ◆ Distinguished objects (`System.in` and `System.out`) for writing to process' standard streams
 - ◆ Additional capabilities (formatting, buffering) provided by creating composite streams (e.g., a `Formatting-Buffered-InputStream`)

Evaluating java.io

- Platform support
 - ◆ Available with all Java implementations on many platforms
 - ◆ Platform-specific differences abstracted away
- Fidelity
 - ◆ Good support for developing P&F applications, but no restriction that apps have to use this style
- Matching assumptions
 - ◆ Easy to construct intra- and inter-process P&F applications
 - ◆ Concurrency can be an issue; many calls are blocking
- Efficiency
 - ◆ Users have fine-grained control over, e.g., buffering
 - ◆ Very high efficiency mechanisms (memory mapped I/O, channels) not available (but are in java.nio)

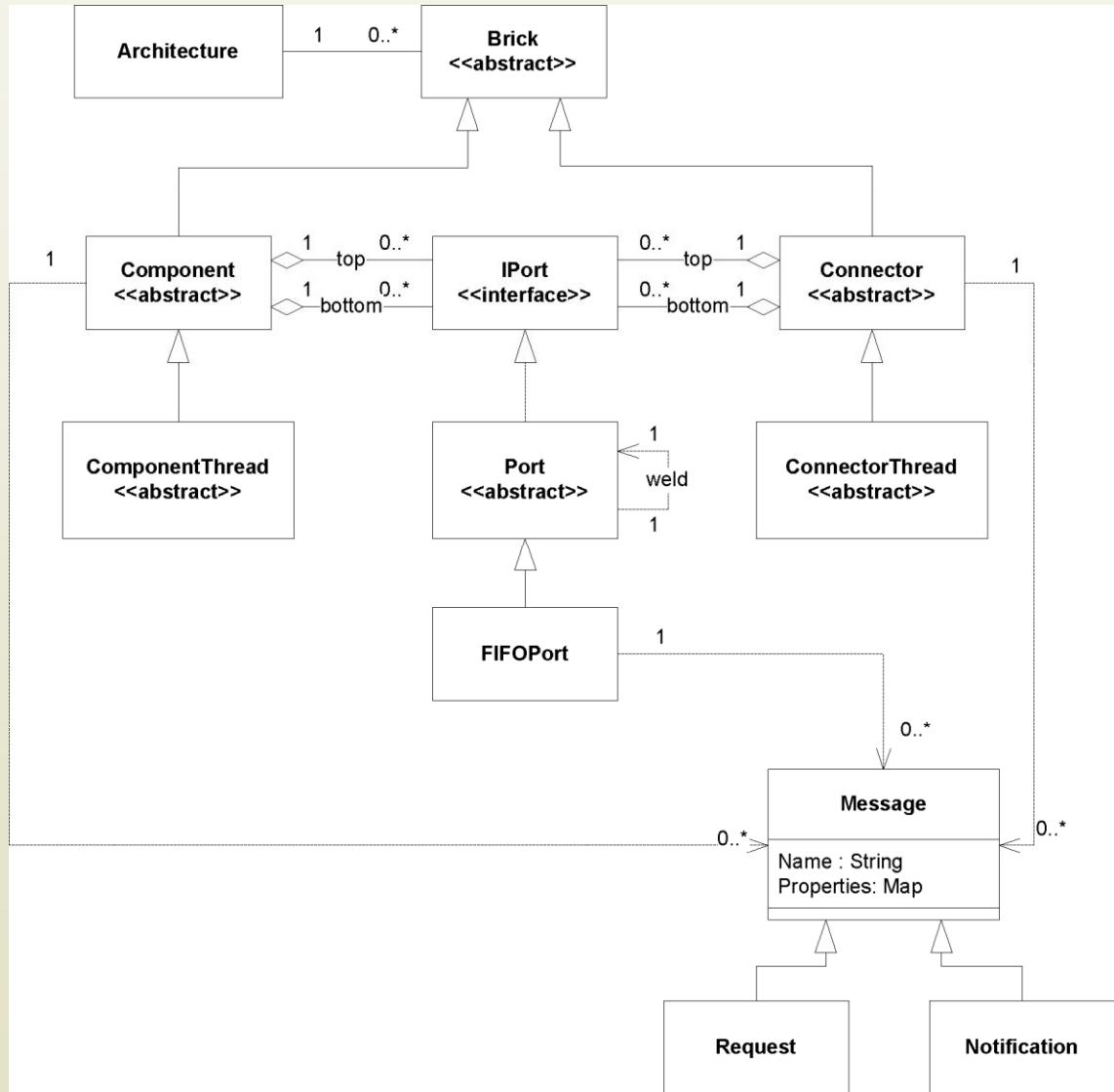
Recall the C2 Style

- Layered style with event-based communication over two-way broadcast buses
- Strict rules on concurrency, dependencies, and so on
- Many frameworks different languages alternative Java fr



Framework #1: Lightweight C2 Framework

- 16 classes, 3000 lines of code
- Components & connectors extend abstract base classes
- Concurrency, queuing handled at individual comp/conn level
- Messages are request or notification objects

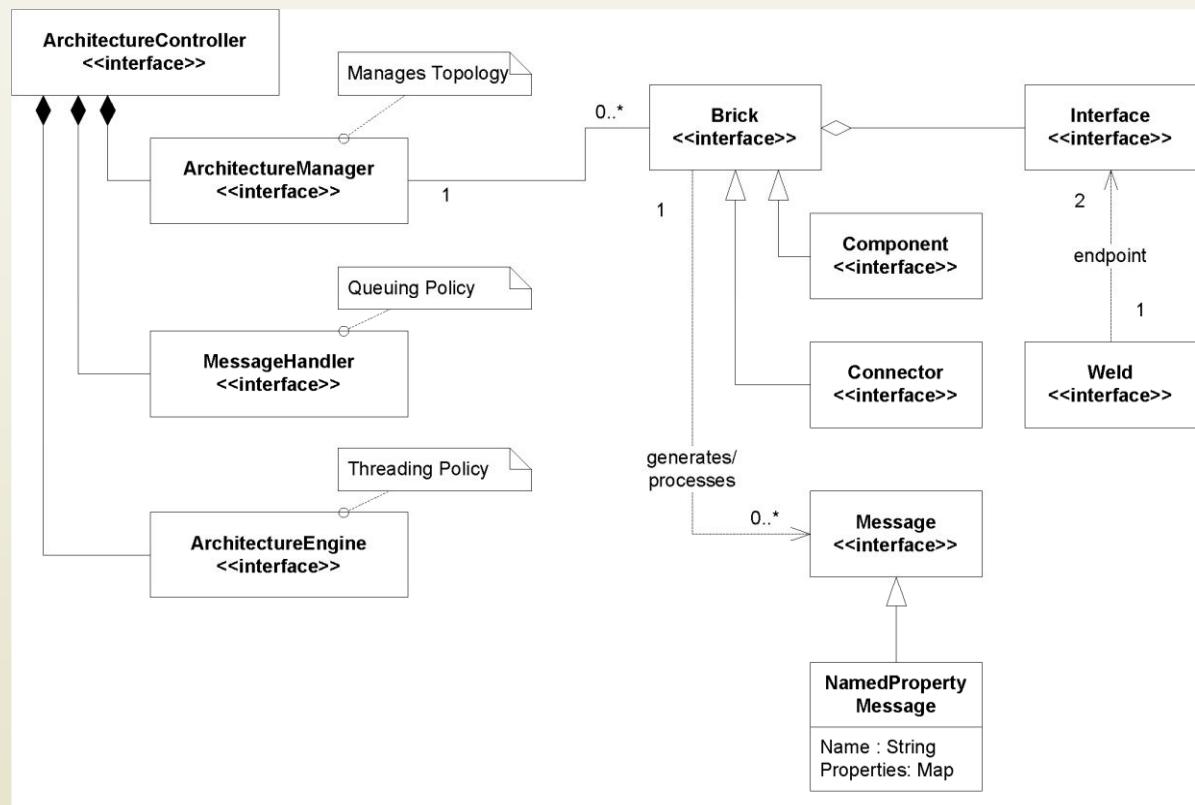


Evaluating Lightweight C2 Framework

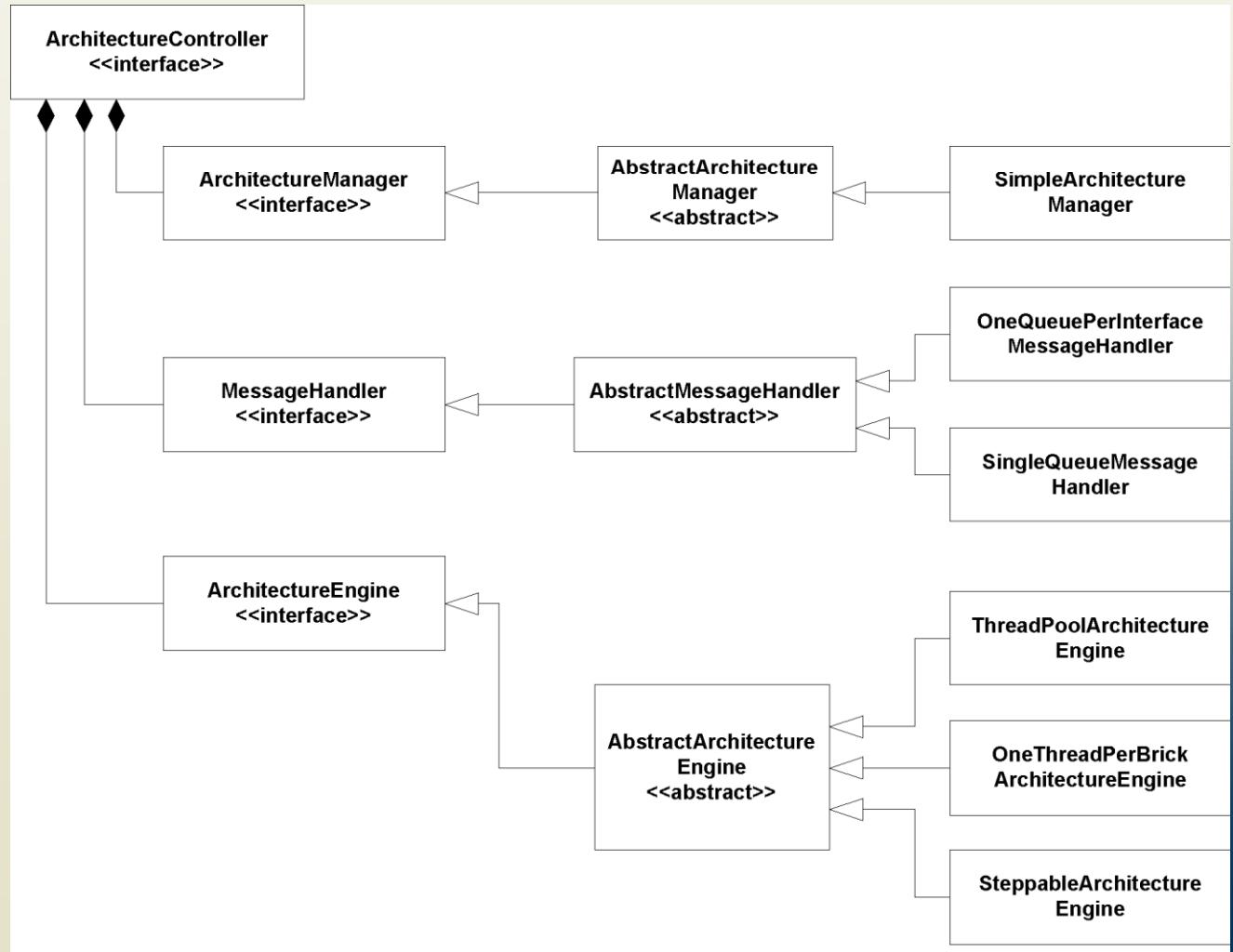
- Platform support
 - ◆ Available with all Java implementations on many platforms
- Fidelity
 - ◆ Assists developers with many aspects of C2 but does not enforce these constraints
 - ◆ Leaves threading and queuing policies up to individual elements
- Matching assumptions
 - ◆ Comp/conn main classes must inherit from distinguished base classes
 - ◆ All messages must be in dictionary form
- Efficiency
 - ◆ Lightweight framework; efficiency may depend on threading and queuing policy implemented by individual elements

Framework #2: Flexible C2 Framework

- 73 classes, 8500 lines of code
- Uses interfaces rather than base classes
- Threading policy for application is pluggable
- Message queuing policy is also pluggable



Framework #2: Flexible C2 Framework



Evaluating Flexible C2 Framework

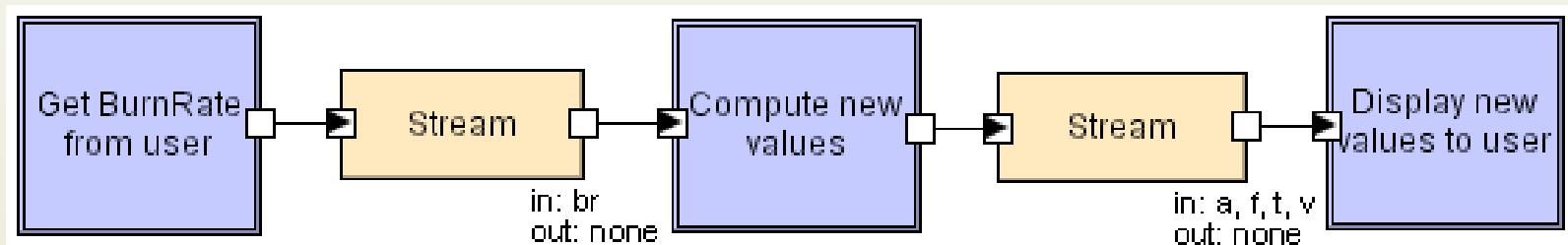
- Platform support
 - ◆ Available with all Java implementations on many platforms
- Fidelity
 - ◆ Assists developers with many aspects of C2 but does not enforce these constraints
 - ◆ Provides several alternative application-wide threading and queuing policies
- Matching assumptions
 - ◆ Comp/conn main classes must implement distinguished interfaces
 - ◆ Messages can be any serializable object
- Efficiency
 - ◆ User can easily swap out and tune threading and queuing policies without disturbing remainder of application code

Objectives

- Concepts
 - ◆ Implementation as a mapping problem
 - ◆ Architecture implementation frameworks
 - ◆ Evaluating frameworks
 - ◆ Relationships between middleware, frameworks, component models
 - ◆ Building new frameworks
 - ◆ Concurrency and generative technologies
 - ◆ Ensuring architecture-to-implementation consistency
- Examples
 - ◆ Different frameworks for pipe-and-filter
 - ◆ Different frameworks for the C2 style
- Application
 - ◆ Implementing Lunar Lander in different frameworks

Implementing Pipe and Filter

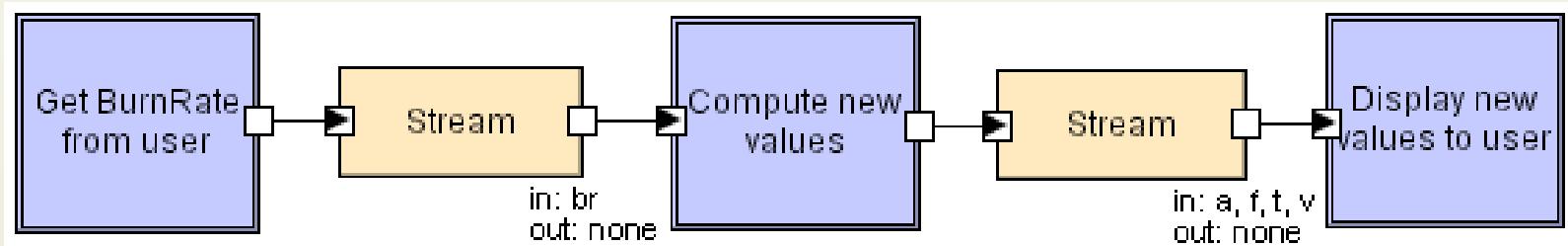
Lunar Lander



- Framework: `java.io`
- Implementing as a multi-process application
 - ◆ Each component (filter) will be a separate OS process
 - ◆ Operating system will provide the pipe connectors
- Going to use just the standard input and output streams
 - ◆ Ignoring standard error
- Ignoring good error handling practices and corner cases for simplicity

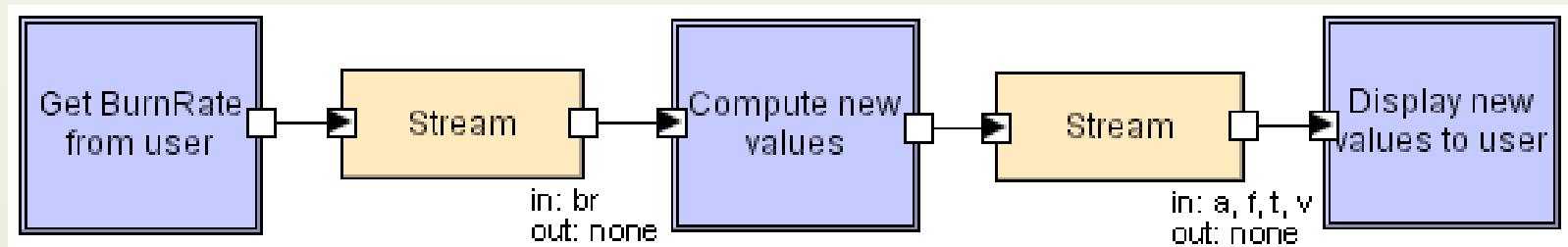
Implementing Pipe and Filter

Lunar Lander



- A note on I/O:
 - ◆ Some messages sent from components are intended for output to the console (to be read by the user)
 - These messages must be passed all the way through the pipeline and output at the end
 - We will preface these with a '#'
 - ◆ Some messages are control messages meant to communicate state to a component down the pipeline
 - These messages are intercepted by a component and processed
 - We will preface these with a '%'

Implementing Pipe and Filter Lunar Lander



- First: GetBurnRate component
 - Loops; on each loop:
 - Prompt user for new burn rate
 - Read burn rate from the user on standard input
 - Send burn rate to next component
 - Quit if burn rate read < 0

GetBurnRate Filter

```
//Import the java.io framework
import java.io.*;

public class GetBurnRate{
    public static void main(String[] args){

        //Send welcome message
        System.out.println("#Welcome to Lunar Lander");

        try{
            //Begin reading from System input
            BufferedReader inputReader =
                new BufferedReader(new InputStreamReader(System.in));

            //Set initial burn rate to 0
            int burnRate = 0;
            do{
                //Prompt user
                System.out.println("#Enter burn rate or <0 to quit:");

                . . .
            }
        }
    }
}
```

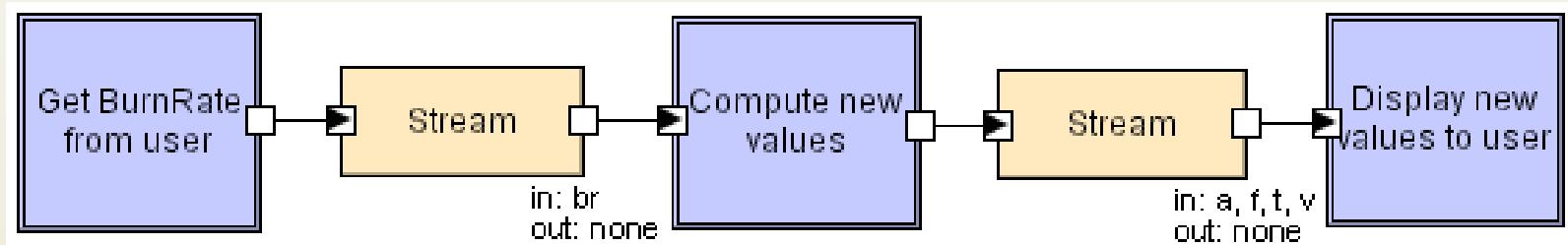
GetBurnRate Filter

```
//Import the java.io framework
import java.io.*;

public class GetBurnRateFilter {
    . . .
    public void filter(ServletInputStream inputReader, ServletOutputStream outputWriter) throws IOException {
        . . .
        //Read user response
        try{
            String burnRateString = inputReader.readLine();
            burnRate = Integer.parseInt(burnRateString);

            //Send user-supplied burn rate to next filter
            System.out.println("%" + burnRate);
        }
        catch (NumberFormatException nfe){
            System.out.println("#Invalid burn rate.");
        }
        //Send user-supplied burn rate to next filter
        int burnRate;
        do{
            burnRate = inputReader.readInt();
        }while(burnRate >= 0);
        inputReader.close();
    }
    catch (IOException ioe){
        ioe.printStackTrace();
    }
    . . .
}
}
```

Implementing Pipe and Filter Lunar Lander



- Second: CalcNewValues Component
 - ◆ Read burn rate from standard input
 - ◆ Calculate new game state including game-over
 - ◆ Send new game state to next component
 - New game state is not sent in a formatted string; that's the display component's job

CalcBurnRate Filter

```
import java.io.*;  
  
public class CalcNewValues{  
  
    public static void main(String[] args) {  
        //Initialize values  
        final int GRAVITY = 2;  
        int altitude = 1000;  
        int fuel = 500;  
        int velocity = 70;  
        int time = 0;  
  
        try{  
            BufferedReader inputReader = new  
                BufferedReader(new InputStreamReader(System.in));  
  
            //Print initial values  
            System.out.println("%a" + altitude);  
            System.out.println("%f" + fuel);  
            System.out.println("%v" + velocity);  
            System.out.println("%t" + time);  
  
            . . .  
        } catch (IOException e) {  
            e.printStackTrace();  
        }  
    }  
}
```

CalcBurnRate Filter

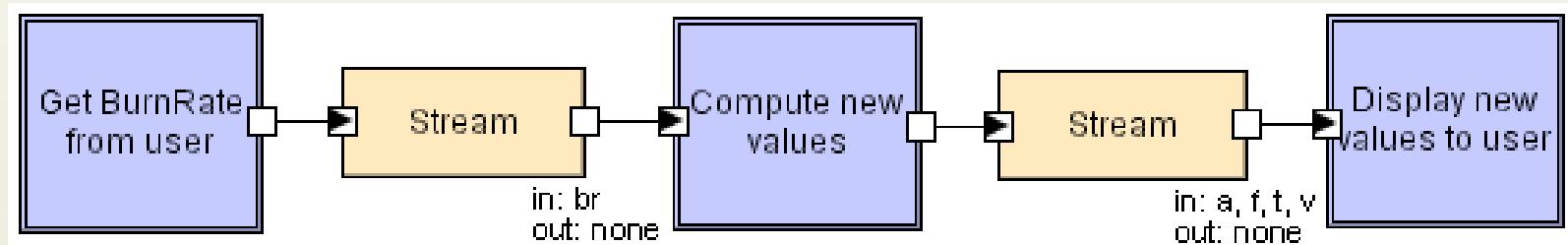
```
import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;
import java.util.Scanner;
import java.util.StringTokenizer;

public class CalcBurnRate {
    public static void main(String[] args) throws IOException {
        BufferedReader inputReader = new BufferedReader(new InputStreamReader(System.in));
        String inputLine = null;
        do{
            inputLine = inputReader.readLine();
            if((inputLine != null) && (inputLine.length() > 0)){
                if(inputLine.startsWith("#")){
                    //This is a status line of text, and
                    //should be passed down the pipeline
                    System.out.println(inputLine);
                }
                else if(inputLine.startsWith("%")){
                    //This is an input burn rate
                    try{
                        int burnRate = Integer.parseInt(inputLine.substring(1));
                        if(burnRate <= 0){
                            System.out.println("#The game is over.");
                        }
                        else if(burnRate > fuel){
                            System.out.println("#Sorry, you don't" +
                                "have that much fuel.");
                        }
                    }
                    catch(NumberFormatException e){
                        System.out.println("Input must be a number");
                    }
                }
            }
        }while(true);
    }
}
```

```
else{
    //Calculate new application state
    time = time + 1;
    altitude = altitude - velocity;
    velocity = ((velocity + GRAVITY) * 10 -
                burnRate * 2) / 10;
    fuel = fuel - burnRate;
    if(altitude <= 0) {
        altitude = 0;
        if(velocity <= 5) {
            System.out.println("#You have landed safely.");
        }
        else{
            System.out.println("#You have crashed.");
        }
    }
    //Print new values
    System.out.println("%a" + altitude);
    System.out.println("%f" + fuel);
    System.out.println("%v" + velocity);
    System.out.println("%t" + time);
}
catch(NumberFormatException nfe) {
}
.
.
.
```

```
else{
    //Calculate new application state
    time = time + 1;
    altitude = altitude - velocity;
    velocity = ((velocity + GRAVITY) * 10 -
        burnRate * 2) / 10;
    fuel = fuel - burnRate;
    if(                                )
        a                                }
    i        }while((inputLine != null) && (altitude > 0));
        inputReader.close();
    }
e        catch(IOException ioe){
        ioe.printStackTrace();
    }
}
}
//Print new values
System.out.println("%a" + altitude);
System.out.println("%f" + fuel);
System.out.println("%v" + velocity);
System.out.println("%t" + time);
}
catch(NumberFormatException nfe) {
}
```

Implementing Pipe and Filter Lunar Lander



- Third: DisplayValues component
 - ◆ Read value updates from standard input
 - ◆ Format them for human reading and send them to standard output

DisplayValues Filter

```
import java.io.*;  
  
public class DisplayValues{  
  
    public static void main(String[] args){  
        try{  
            BufferedReader inputReader = new  
                BufferedReader(new InputStreamReader(System.in));  
  
            String inputLine = null;  
            do{  
                inputLine = inputReader.readLine();  
                if((inputLine != null) &&  
                    (inputLine.length() > 0)){  
  
                    if(inputLine.startsWith("#")){  
                        //This is a status line of text, and  
                        //should be passed down the pipeline with  
                        //the pound-sign stripped off  
                        System.out.println(inputLine.substring(1));  
                }  
            }  
        }  
    }  
}
```

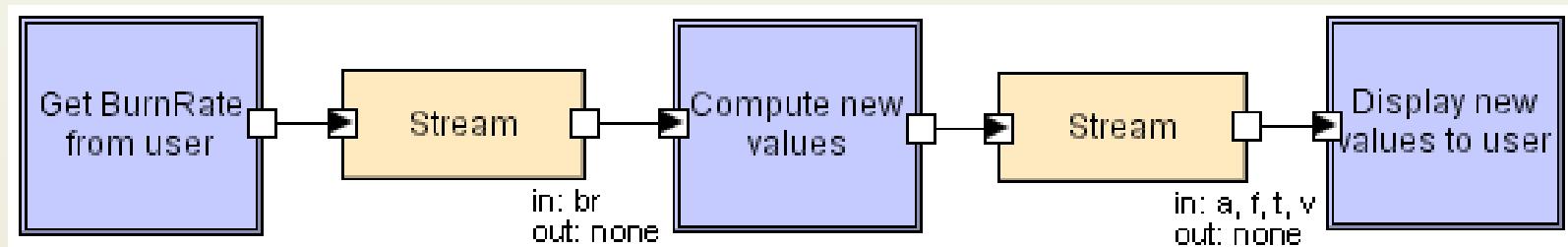
```
import java.util.Scanner;
public class InputProcessor {
    public void processInput() {
        Scanner inputLine = new Scanner(System.in);
        String inputLine = inputLine.nextLine();
        if(inputLine.startsWith("q")) {
            System.out.println("Quitting");
            return;
        }
        else if(inputLine.startsWith("%")) {
            //This is a value to display
            if(inputLine.length() > 1) {
                try{
                    char valueType = inputLine.charAt(1);
                    int value = Integer.parseInt(inputLine.substring(2));

                    switch(valueType) {
                        case 'a':
                            System.out.println("Altitude: " + value);
                            break;
                        case 'f':
                            System.out.println("Fuel remaining: " + value);
                            break;
                        case 'v':
                            System.out.println("Current Velocity: " + value);
                            break;
                        case 't':
                            System.out.println("Time elapsed: " + value);
                            break;
                    }
                }
                catch(NumberFormatException nfe) {
                }
            }
        }
    }
}
```

```
import java.io.*;
public class TimeElapsed {
    public static void main(String[] args) {
        String inputLine;
        BufferedReader inputReader = new BufferedReader(new InputStreamReader(System.in));
        try {
            while ((inputLine = inputReader.readLine()) != null) {
                if (inputLine.startsWith("%")) {
                    System.out.println("Time elapsed: " + value);
                } else {
                    System.out.println("Time elapsed: " + value);
                }
            }
        } catch (IOException ioe) {
            ioe.printStackTrace();
        }
    }
}
```

Implementing Pipe and Filter

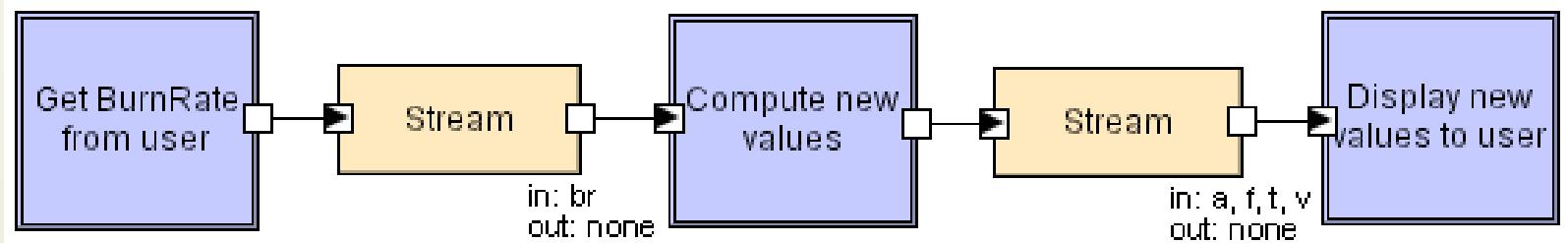
Lunar Lander



- Instantiating the application
 - ◆ `java GetBurnRate | java CalcNewValues | java DisplayValues`

Implementing Pipe and Filter

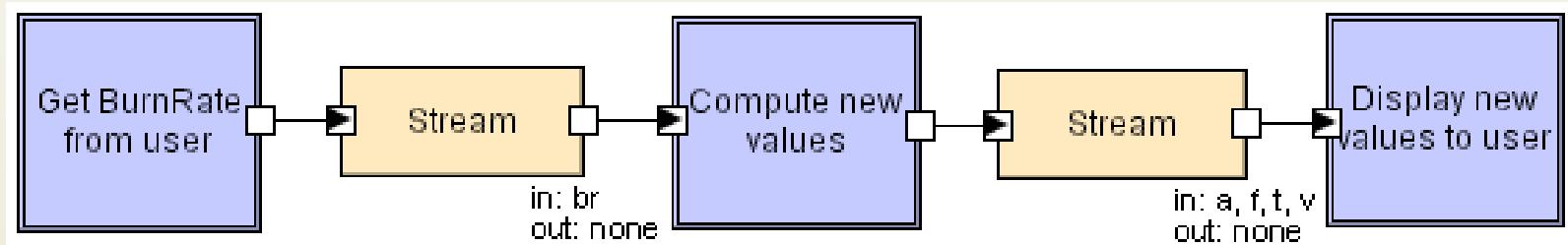
Lunar Lander



```
C:\>C:\WINDOWS\system32\cmd.exe
C:\>Projects\lander-pf\bin>java GetBurnRate | java CalcNewValues | java DisplayValues
Altitude: 1000
Fuel remaining: 500
Current Velocity: 70
Time elapsed: 0
Welcome to Lunar Lander
Enter burn rate or <0 to quit:
```

Implementing Pipe and Filter

Lunar Lander



A screenshot of a Windows command prompt window titled "C:\WINDOWS\system32\cmd.exe". The window shows the execution of three Java classes in sequence:

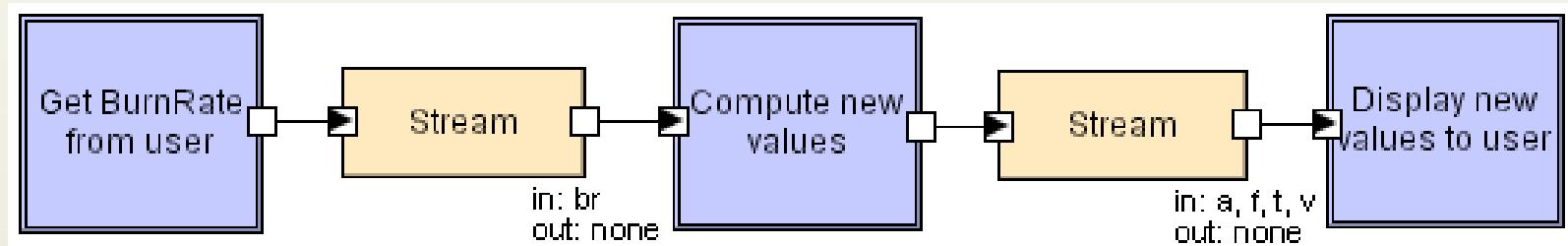
```
C:\Projects\lander-pf\bin>java GetBurnRate | java CalcNewValues | java DisplayUa
```

The application's output is displayed in the window:

```
lues
Altitude: 1000
Fuel remaining: 500
Current Velocity: 70
Time elapsed: 0
Welcome to Lunar Lander
Enter burn rate or <0 to quit:
50
Altitude: 930
Fuel remaining: 450
Current Velocity: 62
Time elapsed: 1
Enter burn rate or <0 to quit:
-
```

Implementing Pipe and Filter

Lunar Lander

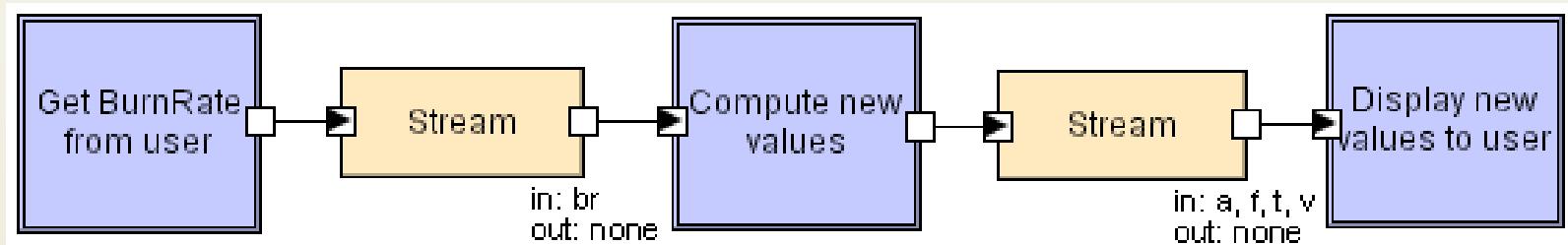


A screenshot of a Windows command prompt window titled "C:\WINDOWS\system32\cmd.exe". The window shows the execution of three Java classes in sequence: "java GetBurnRate | java CalcNewValues | java DisplayUa". The output of the application is displayed in the window, showing the state of a lunar lander over time as burn rates are entered.

```
C:\Projects\lander-pf\bin>java GetBurnRate | java CalcNewValues | java DisplayUa
lues
Altitude: 1000
Fuel remaining: 500
Current Velocity: 70
Time elapsed: 0
Welcome to Lunar Lander
Enter burn rate or <0 to quit:
50
Altitude: 930
Fuel remaining: 450
Current Velocity: 62
Time elapsed: 1
Enter burn rate or <0 to quit:
100
Altitude: 868
Fuel remaining: 350
Current Velocity: 44
Time elapsed: 2
Enter burn rate or <0 to quit:
```

Implementing Pipe and Filter

Lunar Lander



```
C:\WINDOWS\system32\cmd.exe
0
Altitude: 57
Fuel remaining: 0
Current Velocity: 22
Time elapsed: 26
Enter burn rate or <0 to quit:
0
Altitude: 35
Fuel remaining: 0
Current Velocity: 24
Time elapsed: 27
Enter burn rate or <0 to quit:
0
Altitude: 11
Fuel remaining: 0
Current Velocity: 26
Time elapsed: 28
Enter burn rate or <0 to quit:
0
You have crashed.
Altitude: 0
Fuel remaining: 0
Current Velocity: 28
Time elapsed: 29
```

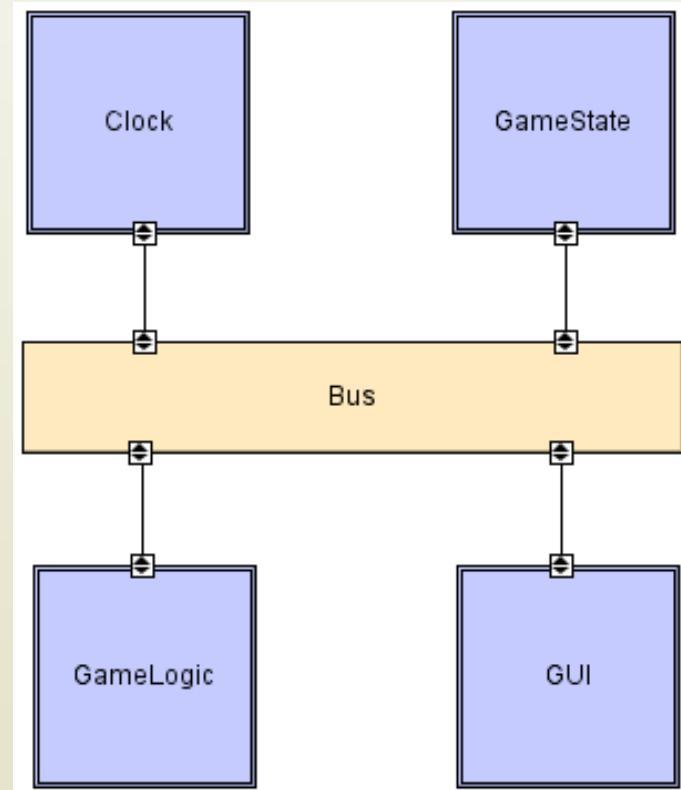
A screenshot of a Windows command prompt window titled "C:\WINDOWS\system32\cmd.exe". The window displays the output of a Lunar Lander simulation. The simulation starts at an altitude of 57, fuel remaining 0, current velocity 22, and time elapsed 26. It prompts the user to enter a burn rate or type <0 to quit. The user enters 0, resulting in an altitude of 35, fuel 0, velocity 24, and time 27. Another 0 entry leads to an altitude of 11, fuel 0, velocity 26, and time 28. A third 0 entry results in the message "You have crashed." and the final state of 0 altitude, 0 fuel, 28 velocity, and 29 time.

Takeaways

- java.io provides a number of useful facilities
 - ◆ Stream objects (System.in, System.out)
 - ◆ Buffering wrappers
- OS provides some of the facilities
 - ◆ Pipes
 - ◆ Concurrency support
 - Note that this version of the application would not work if it operated in batch-sequential mode
- We had other communication mechanisms available, but did not use them to conform to the P&F style
- We had to develop a new (albeit simple) protocol to get the correct behavior

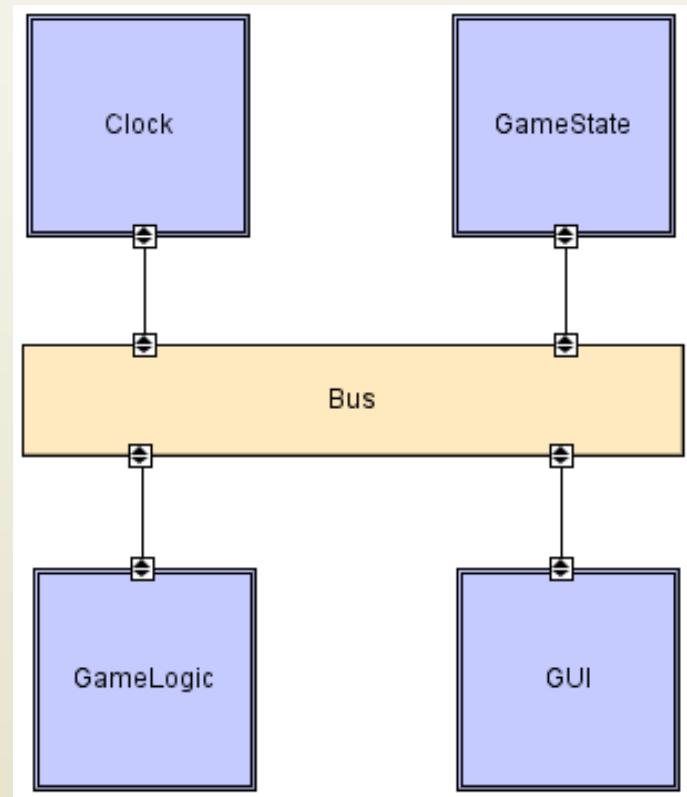
Implementing Lunar Lander in C2

- Framework: Lightweight C2 framework
- Each component has its own thread of control
- Components receive requests or notifications and respond with new ones
- Message routing follows C2 rules
- This is a real-time, clock-driven version of Lunar Lander



Implementing Lunar Lander in C2 (cont'd)

- First: Clock component
- Sends out a 'tick' notification periodically
- Does not respond to any messages

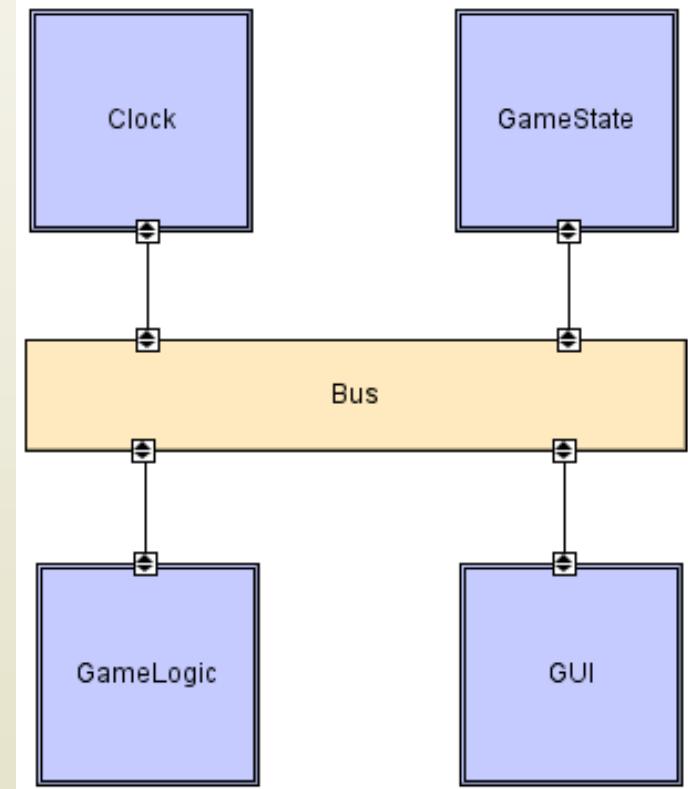


```
import c2.framework.*;  
  
public class Clock extends ComponentThread{  
    public Clock(){  
        super.create("clock", FIFOPort.class);  
    }  
  
    public void start(){  
        super.start();  
  
        Thread clockThread = new Thread(){  
            public void run(){  
                //Repeat while the application runs  
                while(true){  
                    //Wait for five seconds  
                    try{  
                        Thread.sleep(5000);  
                    }  
                    catch(InterruptedException ie){}  
  
                    //Send out a tick notification  
                    Notification n = new Notification("clockTick");  
                    send(n);  
                }  
            }  
        };  
    }  
};
```

```
import c2.framework.*;  
  
public class Clock extends ComponentThread{  
    public Clock() {  
        super.create("clock", FIFOPort.class);  
    }  
  
    public void start(){  
        super.start();  
        clockThread.start();  
    }  
    Thread clockThread = new Thread(this);  
    protected void handle(Notification n){  
        //This component does not handle notifications  
    }  
  
    protected void handle(Request r){  
        //This component does not handle requests  
    }  
}  
  
//Send out a tick notification  
Notification n = new Notification("clockTick");  
send(n);  
}  
}  
};
```

Implementing Lunar Lander in C2

- Second: GameState Component
- Receives request to update internal state
- Emits notifications of new game state on request or when state changes
- Does NOT compute new state
 - ◆ Just a data store



GameState Component

```
import c2.framework.*;

public class GameState extends ComponentThread{

    public GameState() {
        super.create("gameState", FIFOPort.class);
    }

    //Internal game state and initial values
    int altitude = 1000;
    int fuel = 500;
    int velocity = 70;
    int time = 0;
    int burnRate = 0;
    boolean landedSafely = false;
```

```
protected void handle(Request r) {
    if(r.name().equals("updateGameState")) {
        //Update the internal game state
        if(r.hasParameter("altitude")){
            this.altitude = ((Integer)r.getParameter("altitude")).intValue();
        }
        if(r.hasParameter("fuel")){
            this.fuel = ((Integer)r.getParameter("fuel")).intValue();
        }
        if(r.hasParameter("velocity")){
            this.velocity = ((Integer)r.getParameter("velocity")).intValue();
        }
        if(r.hasParameter("time")){
            this.time = ((Integer)r.getParameter("time")).intValue();
        }
        if(r.hasParameter("burnRate")){
            this.burnRate = ((Integer)r.getParameter("burnRate")).intValue();
        }
        if(r.hasParameter("landedSafely")){
            this.landedSafely = ((Boolean)r.getParameter("landedSafely"))
                .booleanValue();
        }
        //Send out the updated game state
        Notification n = createStateNotification();
        send(n);
    }
}
```

```
protected void handle(Notification n) {
    if(n.getName().equals("getGameState")) {
        //If a component requests the game state
        //without updating it, send out the state

        Notification n = createStateNotification();
        send(n);
    }
}

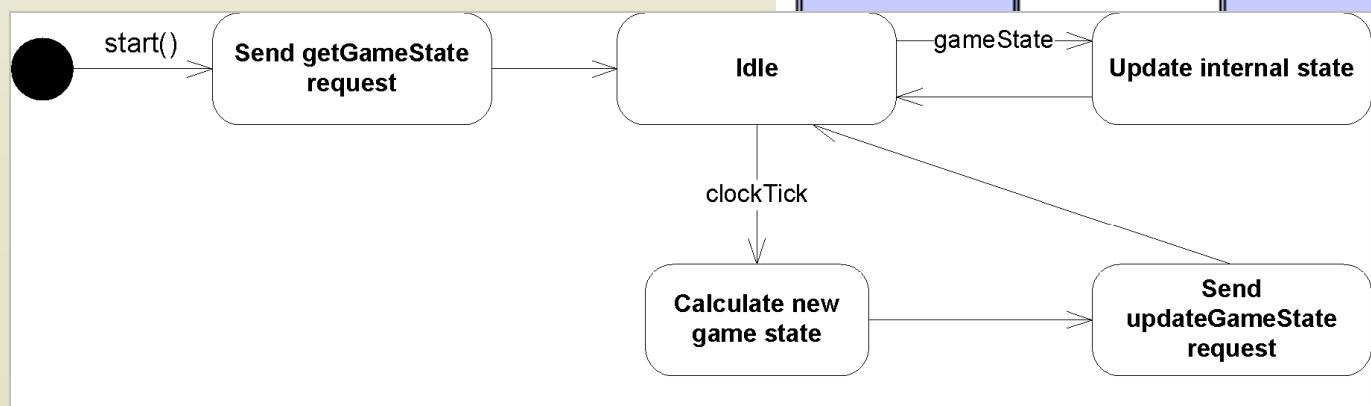
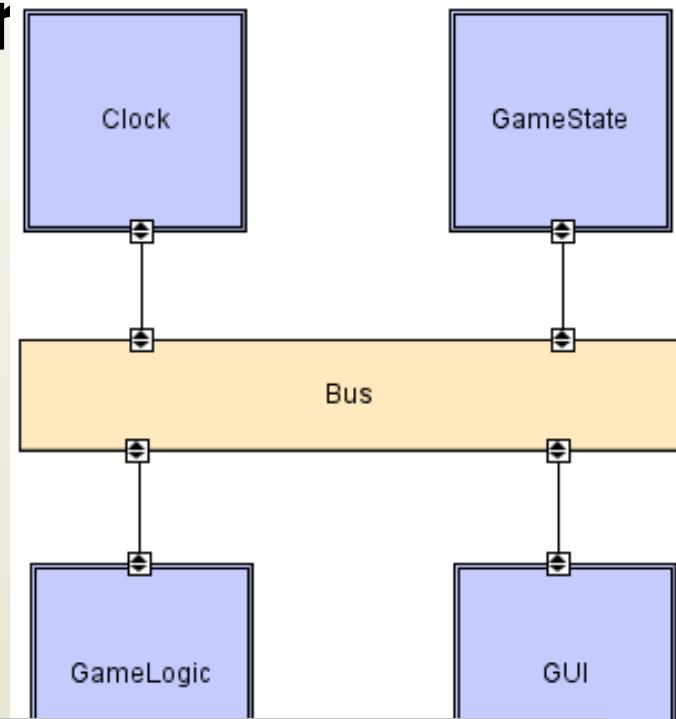
protected Notification createStateNotification() {
    //Create a new notification comprising the
    //current game state

    Notification n = new Notification("gameState");
    n.addParameter("altitude", altitude);
    n.addParameter("fuel", fuel);
    n.addParameter("velocity", velocity);
    n.addParameter("time", time);
    n.addParameter("burnRate", burnRate);
    n.addParameter("landedSafely", landedSafely);
    return n;
}

protected void handle(Notification n) {
    //This component does not handle notifications
}
```

Implementing Lunarcraft

- Third: GameLogic Component
- Receives notifications of game state changes
- Receives clock ticks
 - ◆ On clock tick notification, calculates new state and sends request up



GameLogic Component

```
import c2.framework.*;  
  
public class GameLogic extends ComponentThread{  
    public GameLogic(){  
        super.create("gameLogic", FIFOPort.class);  
    }  
  
    //Game constants  
    final int GRAVITY = 2;  
  
    //Internal state values for computation  
    int altitude = 0;  
    int fuel = 0;  
    int velocity = 0;  
    int time = 0;  
    int burnRate = 0;  
  
    public void start(){  
        super.start();  
        Request r = new Request("getGameState");  
        send(r);  
    }  
}
```

Game

```
protected void handle(Notification n) {
    if(n.name().equals("gameState")) {
        if(n.hasParameter("altitude")) {
            this.altitude =
                ((Integer)n.getParameter("altitude")).intValue();
        }
        if(n.hasParameter("fuel")) {
            this.fuel =
                ((Integer)n.getParameter("fuel")).intValue();
        }
        if(n.hasParameter("velocity")) {
            this.velocity =
                ((Integer)n.getParameter("velocity")).intValue();
        }
        if(n.hasParameter("time")) {
            this.time =
                ((Integer)n.getParameter("time")).intValue();
        }
        if(n.hasParameter("burnRate")) {
            this.burnRate =
                ((Integer)n.getParameter("burnRate")).intValue();
        }
    }
}
```

Game Simulation

```
protected void handle(Notification n) {
    if(n.name().equals("gameState")) {
        else if(n.name().equals("clockTick")) {
            //Calculate new lander state values
            int actualBurnRate = burnRate;
            if(actualBurnRate > fuel) {
                //Ensure we don't burn more fuel than we have
                actualBurnRate = fuel;
            }

            time = time + 1;
            altitude = altitude - velocity;
            velocity = ((velocity + GRAVITY) * 10 -
                        actualBurnRate * 2) / 10;
            fuel = fuel - actualBurnRate;

            //Determine if we landed (safely)
            boolean landedSafely = false;
            if(altitude <= 0) {
                altitude = 0;
                if(velocity <= 5) {
                    landedSafely = true;
                }
            }
        }
    }
}
```

Game

```
protected void handle(Notification n) {
    if(n.name().equals("gameState")) {
        else if(n.name().equals("clockTick")) {
            //Calculate new lander state values
            int actualBurnRate = burnRate;

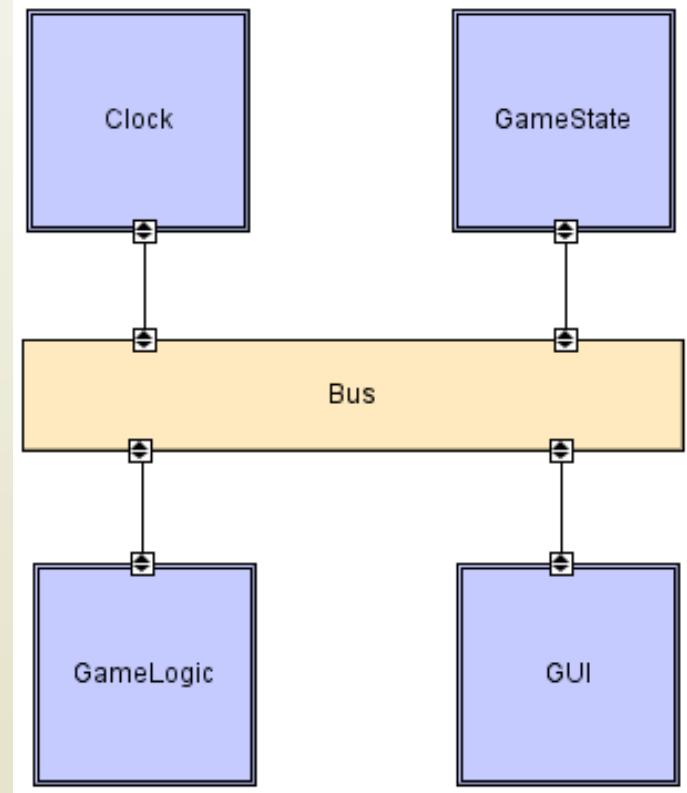
            Request r = new Request("updateGameState");
            r.addParameter("time", time);
            r.addParameter("altitude", altitude);
            r.addParameter("velocity", velocity);
            r.addParameter("fuel", fuel);
            r.addParameter("landedSafely", landedSafely);
            send(r);
        }
    }

    protected void handle(Request r) {
        //This component does not handle requests
    }
}

if(velocity <= 5) {
    landedSafely = true;
}
```

Implementing Lunar Lander in C2

- Fourth: GUI Component
- Reads burn rates from user and sends them up as requests
- Receives notifications of game state changes and formats them to console



GUI Component

```
import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;

import c2.framework.*;

public class GUI extends ComponentThread{
    public GUI(){
        super.create("gui", FIFOPort.class);
    }

    public void start(){
        super.start();
        Thread t = new Thread(){
            public void run(){
                processInput();
            }
        };
        t.start();
    }
}
```

Gu

```
import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;
import java.util.StringTokenizer;
public class LunarLander {
    public static void main(String[] args) {
        try {
            System.out.println("Welcome to Lunar Lander");
            BufferedReader inputReader = new BufferedReader(
                new InputStreamReader(System.in));
            int burnRate = 0;
            do{
                System.out.println("Enter burn rate or <0 to quit:");
                try{
                    String burnRateString = inputReader.readLine();
                    burnRate = Integer.parseInt(burnRateString);
                    Request r = new Request("updateGameState");
                    r.addParameter("burnRate", burnRate);
                    send(r);
                }
                catch(NumberFormatException nfe){
                    System.out.println("Invalid burn rate.");
                }
            }while(burnRate >= 0);
            inputReader.close();
        }
        catch(IOException ioe){
            ioe.printStackTrace();
        }
    }
}
```

Software Architecture: Foundations, Theory, and Practice

```
public void processInput() {  
    System.out.println("Welcome to Lunar Lander");  
  
    protected void handle(Notification n) {  
        if(n.name().equals("gameState")) {  
            System.out.println();  
            System.out.println("New game state:");  
  
            if(n.hasParameter("altitude")) {  
                System.out.println(" Altitude: " + n.getParameter("altitude"));  
            }  
            if(n.hasParameter("fuel")) {  
                System.out.println(" Fuel: " + n.getParameter("fuel"));  
            }  
            if(n.hasParameter("velocity")) {  
                System.out.println(" Velocity: " + n.getParameter("velocity"));  
            }  
            if(n.hasParameter("time")) {  
                System.out.println(" Time: " + n.getParameter("time"));  
            }  
            if(n.hasParameter("burnRate")) {  
                System.out.println(" Burn rate: " + n.getParameter("burnRate"));  
            }  
        }  
        System.out.println();  
        System.out.println("Stack trace:");  
        System.out.printStackTrace();  
    }  
}
```

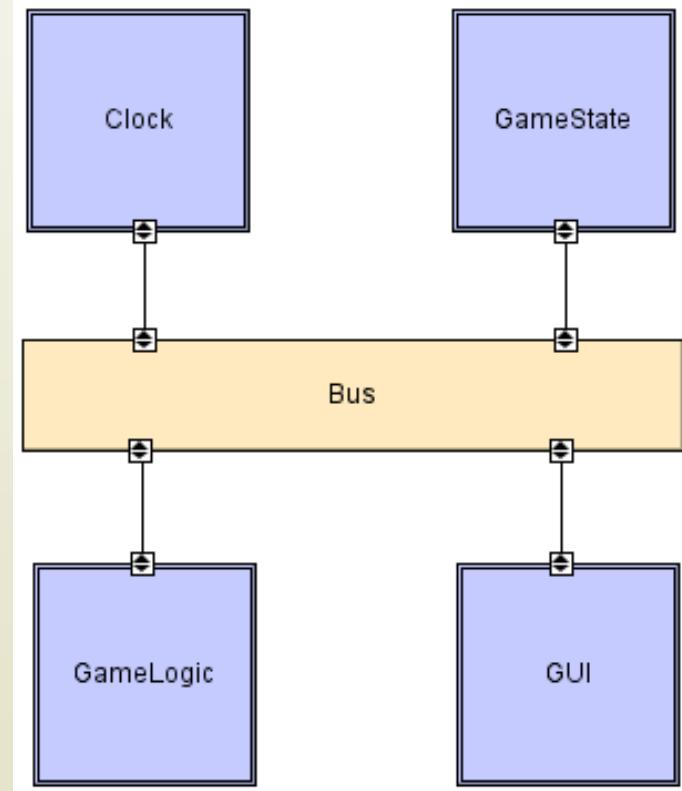
```
public void processInput() {
    System.out.println("Welcome to Lunar Lander");

    if(n.hasParameter("altitude")){
        int altitude =
            ((Integer)n.getParameter("altitude")).intValue();
        if(altitude <= 0){
            boolean landedSafely =
                ((Boolean)n.getParameter("landedSafely"))
                .booleanValue();
            if(landedSafely){
                System.out.println("You have landed safely.");
            }
            else{
                System.out.println("You have crashed.");
            }
            System.exit(0);
        }
    }
}

protected void handle(Request r){
    //This component does not handle requests
}
```

Implementing Lunar Lander in C2

- Lastly, main program
- Instantiates and connects all elements of the system



```
import c2.framework.*;  
  
public class LunarLander{  
  
    public static void main(String[] args) {  
        //Create the Lunar Lander architecture  
        Architecture lunarLander = new  
            SimpleArchitecture("LunarLander");  
  
        //Create the components  
        Component clock = new Clock();  
        Component gameState = new GameState();  
        Component gameLogic = new GameLogic();  
        Component gui = new GUI();  
  
        //Create the connectors  
        Connector bus = new ConnectorThread("bus");  
  
        //Add the components and connectors to the architecture  
        lunarLander.addComponent(clock);  
        lunarLander.addComponent(gameState);  
        lunarLander.addComponent(gameLogic);  
        lunarLander.addComponent(gui);  
  
        lunarLander.addConnector(bus);  
    }  
}
```

```
import c2.framework.*;  
  
public class LunarLander{  
  
    public static void main(String[] args) {  
        //Create the Lunar Lander architecture  
          
        //Add components  
          
        //Create the welds (links) between components and  
        //connectors  
          
        //Start the application  
          
        //Add the components and connectors to the architecture  
          
          
    }  
}  
  
lunarLander.addConnector(bus);
```

Takeaways

- Here, the C2 framework provides most all of the scaffolding we need
 - ◆ Message routing and buffering
 - ◆ How to format a message
 - ◆ Threading for components
 - ◆ Startup and instantiation
- We provide the component behavior
 - ◆ Including a couple new threads of our own
- We still must work to obey the style guidelines
 - ◆ Not everything is optimal: state is duplicated in Game Logic, for example

Applied Architectures

**Software Architecture
Lecture 17**

Objectives

- Illustrate how principles have been used to solve challenging problems
 - ◆ Usually means combining elements
- Highlight some critical issues
 - ◆ I.e., ignore them at your peril
- Show how architecture can be used to explain and analyze common commercial systems

Outline

- Distributed and networked architectures
 - ◆ Limitations
 - ◆ REST
 - ◆ Commercial Internet-scale applications
- Decentralized applications
 - ◆ Peer-to-peer
 - ◆ Web services
- Some interesting domains
 - ◆ Robotics
 - ◆ Wireless sensors
 - ◆ Flight simulators

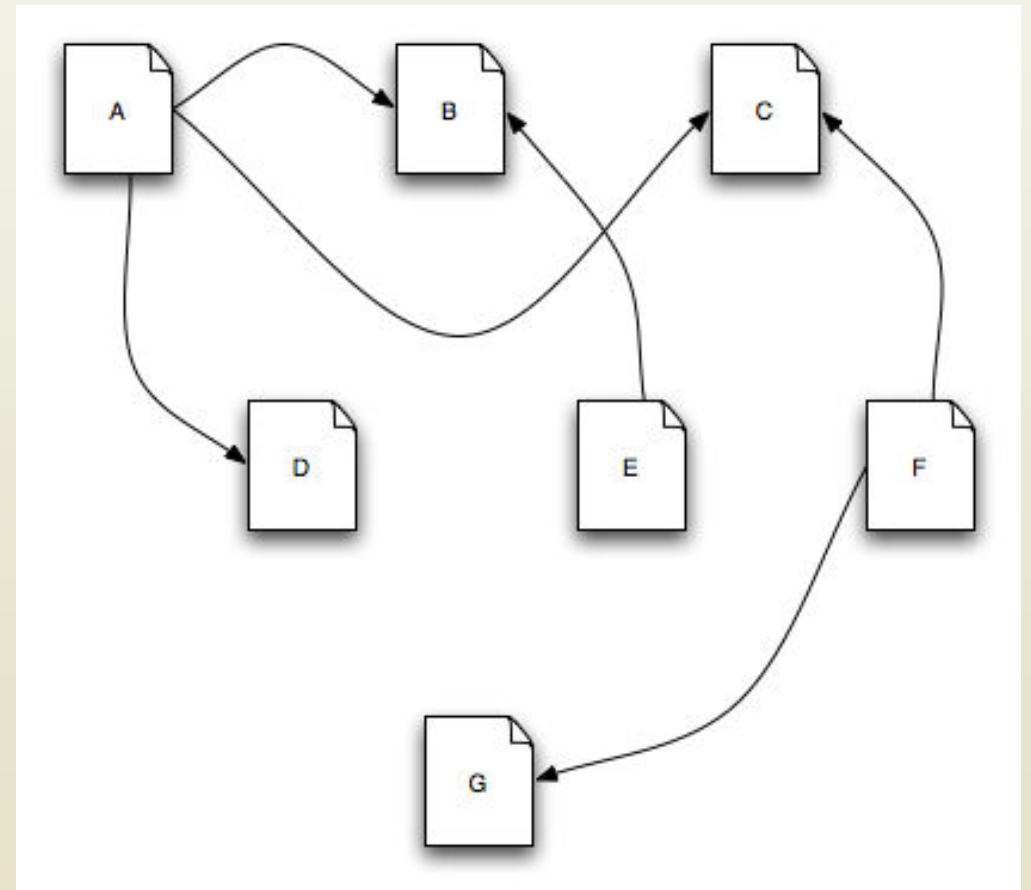
Limitations of the Distributed Systems Viewpoint

- The network is reliable
- Latency is zero
- Bandwidth is infinite
- The network is secure
- Topology doesn't change
- There is one administrator
- Transport cost is zero
- The network is homogeneous

-- Deutsch & Gosling

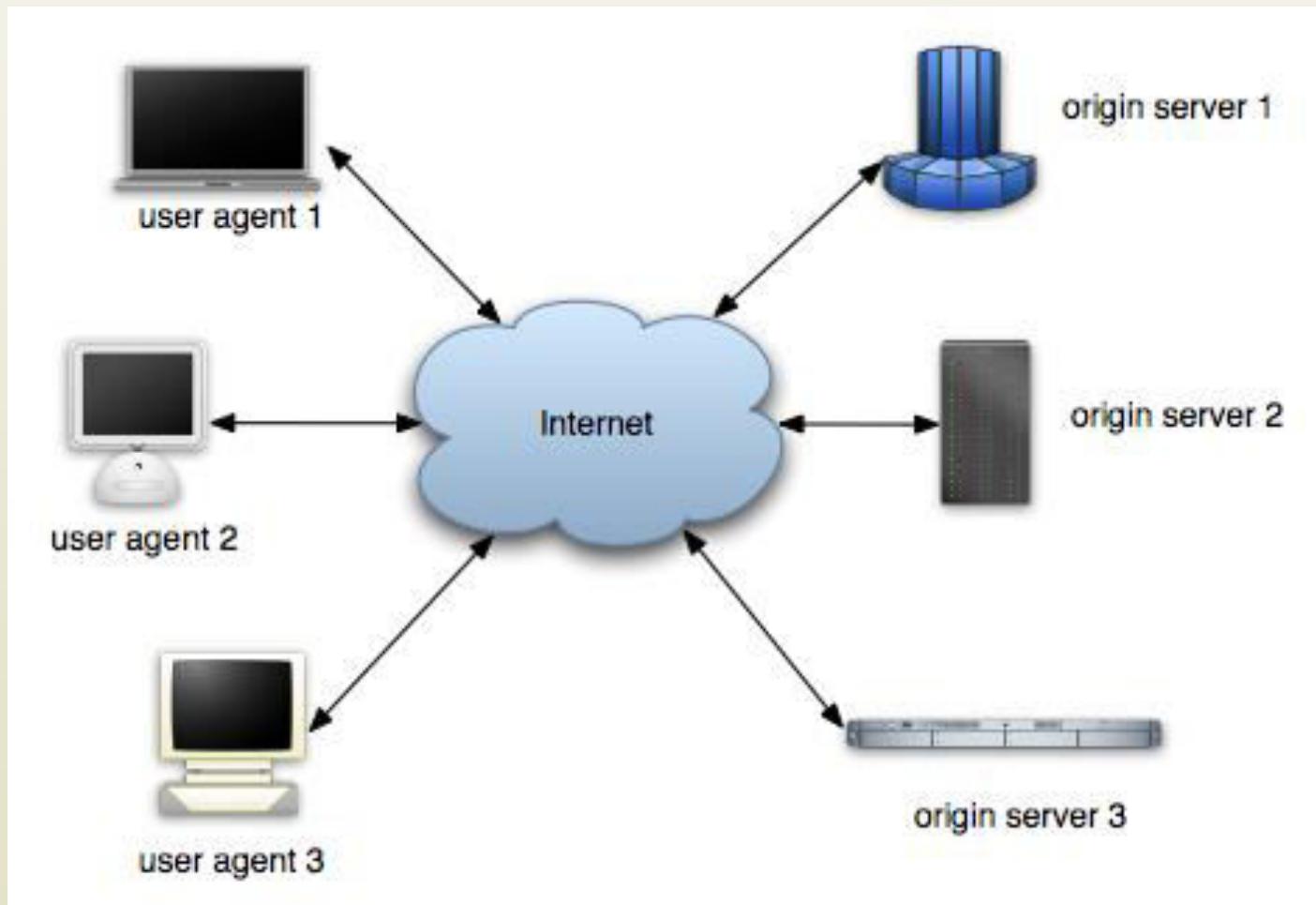
Architecture in Action: WWW

- (From lecture #1) This is the Web



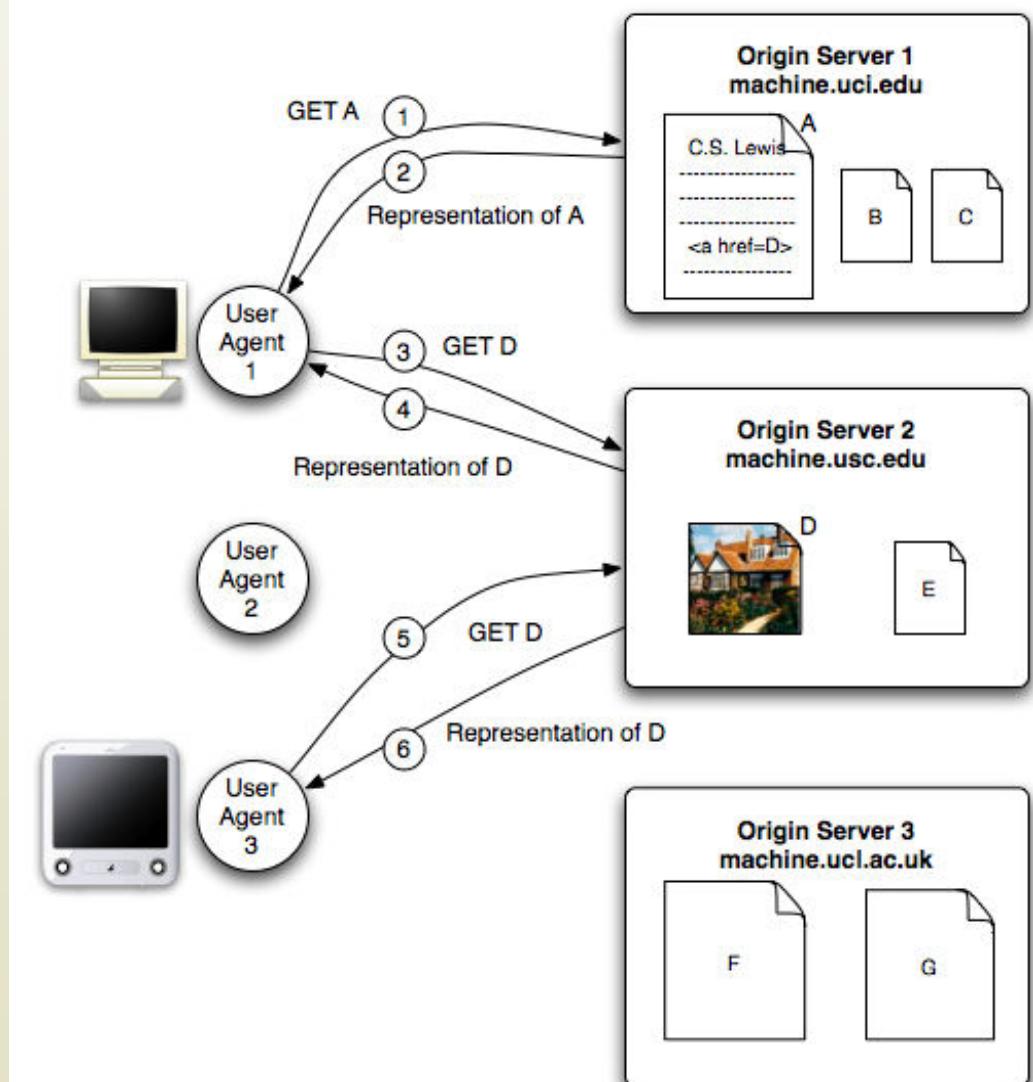
Architecture in Action: WWW (cont'd)

- So is this



Architecture in Action: WWW

- And this



WWW's Architecture

- **The application is distributed (actually, decentralized) hypermedia**
- Architecture of the Web is wholly separate from the code
- There is no single piece of code that implements the architecture.
- There are multiple pieces of code that implement the various components of the architecture.
 - ◆ E.g., different Web browsers
- Stylistic constraints of the Web's architectural style are not apparent in the code
 - ◆ The effects of the constraints are evident in the Web
- One of the world's most successful applications is only understood adequately from an architectural vantage point.

REST Principles

- [RP1] The key abstraction of information is a resource, named by an URL. Any information that can be named can be a resource.
- [RP2] The representation of a resource is a sequence of bytes, plus representation metadata to describe those bytes. The particular form of the representation can be negotiated between REST components.
- [RP3] All interactions are context-free: each interaction contains all of the information necessary to understand the request, independent of any requests that may have preceded it.

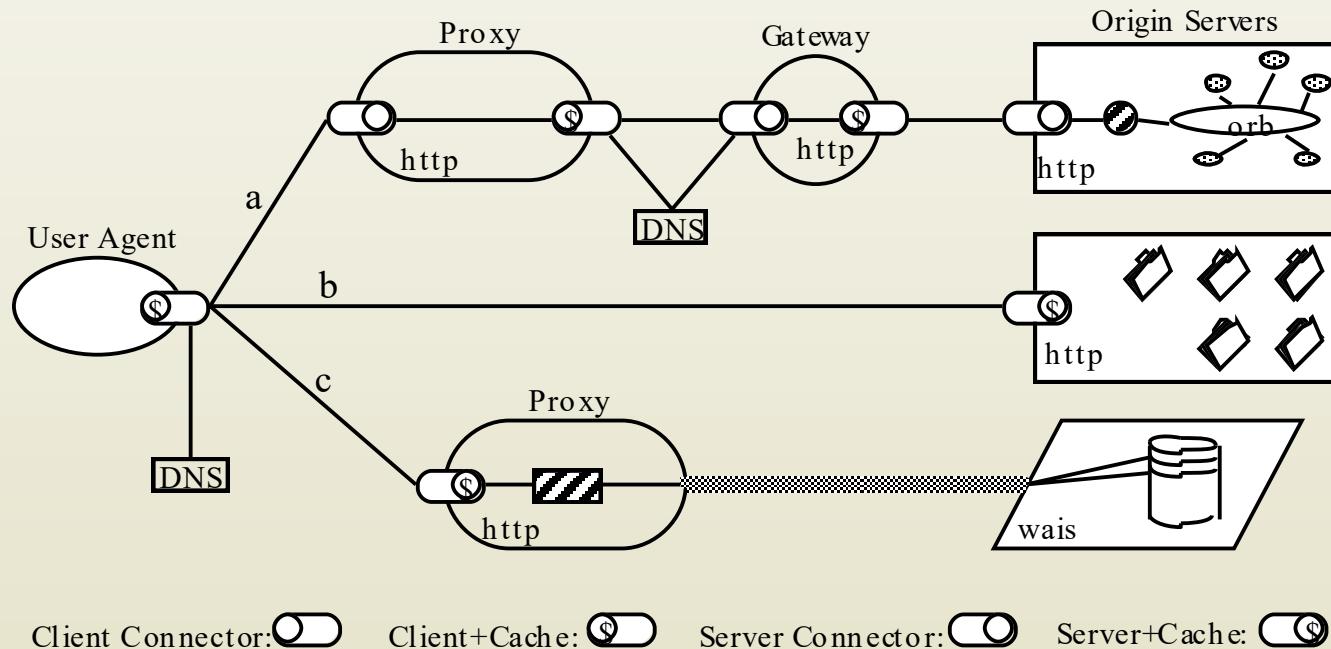
REST Principles (cont'd)

- [RP4] Components perform only a small set of well-defined methods on a resource producing a representation to capture the current or intended state of that resource and transfer that representation between components. These methods are global to the specific architectural instantiation of REST; for instance, all resources exposed via HTTP are expected to support each operation identically.

REST Principles (cont'd)

- [RP5] Idempotent operations and representation metadata are encouraged in support of caching and representation reuse.
- [RP6] The presence of intermediaries is promoted. Filtering or redirection intermediaries may also use both the metadata and the representations within requests or responses to augment, restrict, or modify requests and responses in a manner that is transparent to both the user agent and the origin server.

An Instance of REST



REST – Data Elements

- Resource
 - ◆ Key information abstraction
- Resource ID
- Representation
 - ◆ Data plus metadata
- Representation metadata
- Resource metadata
- Control data
 - ◆ e.g., specifies action as result of message

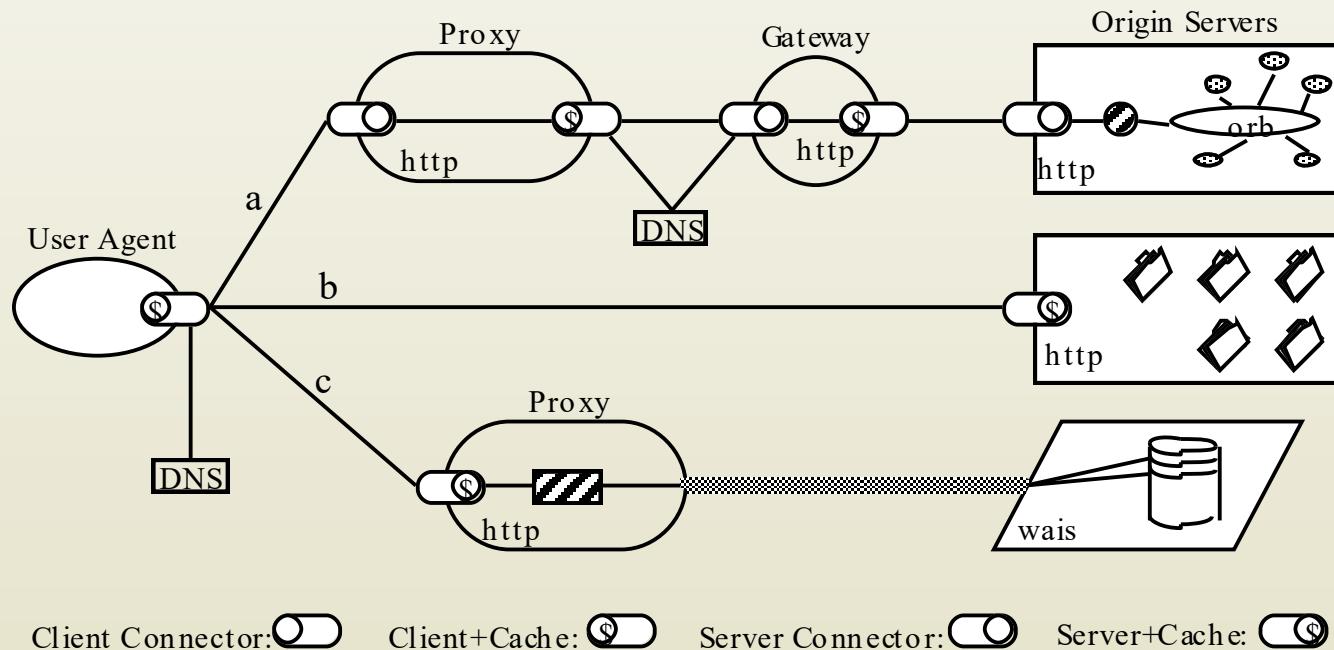
REST – Connectors

- Modern Web Examples
- client libwww, libwww-perl
- server libwww, Apache API, NSAPI
- cache browser cache, Akamai cache network
- resolver bind (DNS lookup library)
- tunnel SOCKS, SSL after HTTP CONNECT

REST – Components

- User agent
 - ◆ e.g., browser
- Origin server
 - ◆ e.g., Apache Server, Microsoft IIS
- Proxy
 - ◆ Selected by client
- Gateway
 - ◆ Squid, CGI, Reverse proxy
 - ◆ Controlled by server

An Instance of REST



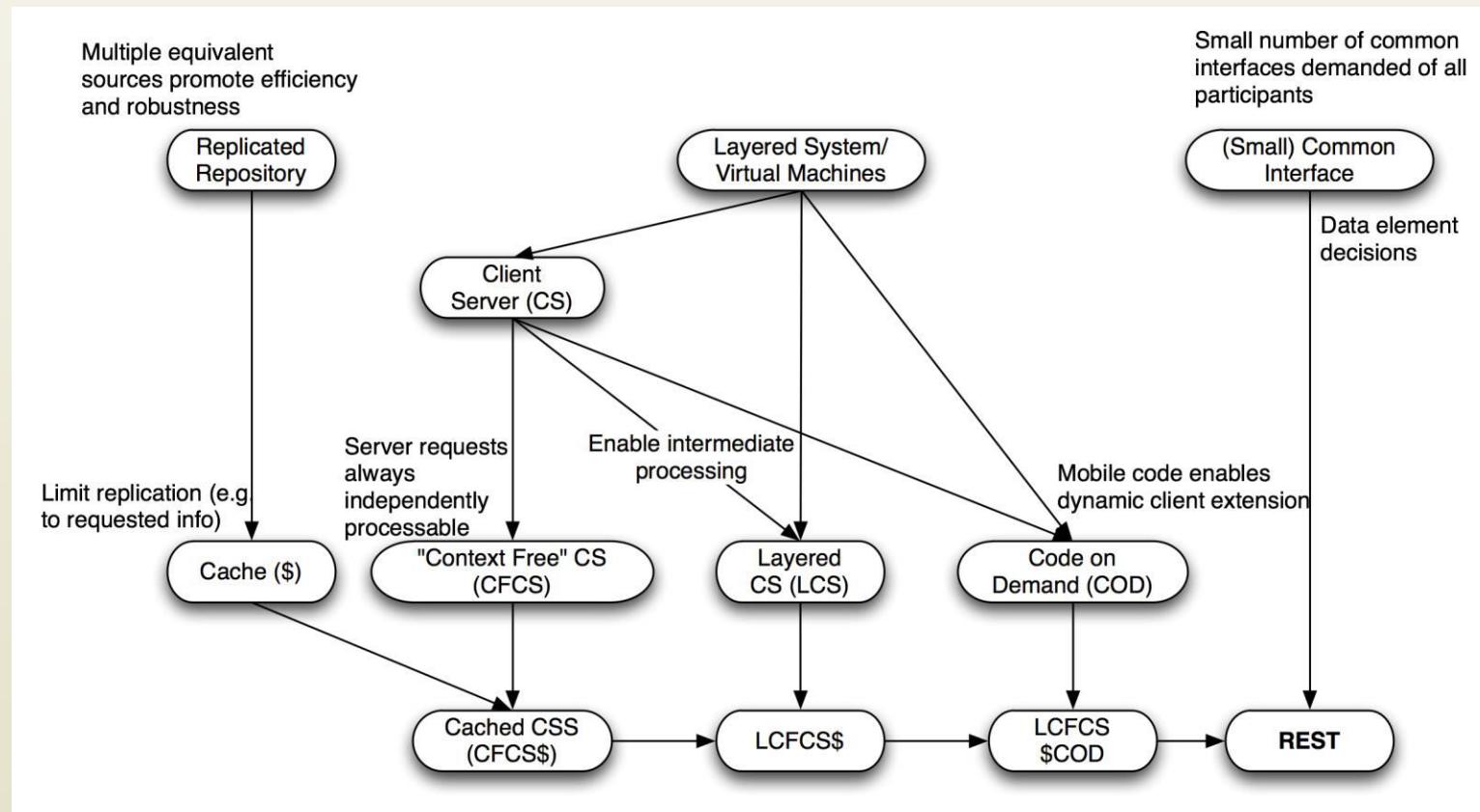
Derivation of REST

Key choices in this derivation include:

- Layered Separation (a theme in the middle portion of diagram) is used to increase efficiencies, enable independent evolution of elements of the system, and provide robustness;
- Replication (left side of the diagram) is used to address latency and contention by allowing the reuse of information;
- Limited commonality (right side) addresses the competing needs for universally understood operations with extensibility.

The derivation is driven by the application(!)

Derivation of REST (cont'd)



REST: Final Thoughts

- REpresentational S~~t~~tate Transfer
- Style of modern web architecture
 - ◆ Web architecture one of many in style
- Web diverges from style on occasion
 - ◆ e.g., Cookies, frames
 - ◆ Doesn't explain mashups

Commercial Internet-Scale Applications

- Akamai
 - ◆ Caching to the max
- Google
 - ◆ Google distributed file system (GFS)
 - ◆ MapReduce
 - Data selection and reduction
 - All parallelization done automatically

Architectural Lessons from Google

- **Abstraction layers abound:** GFS hides details of data distribution and failure, for instance; MapReduce hides the intricacies of parallelizing operations;
- By designing, from the outset, for **living with failure** of processing, storage, and network elements, a highly robust system can be created;
- **Scale is everything:** Google's business demands that everything be built with scaling issues in mind;

Architectural Lessons from Google (cont'd)

- By **specializing the design to the problem domain**, rather than taking the generic “industry standard” approach, high performance and very cost-effective solutions can be developed;
- By **developing a general approach** (MapReduce) to the data extraction/reduction problem, a highly reusable service was created.