Modeling

Software Architecture
Chapter 6

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

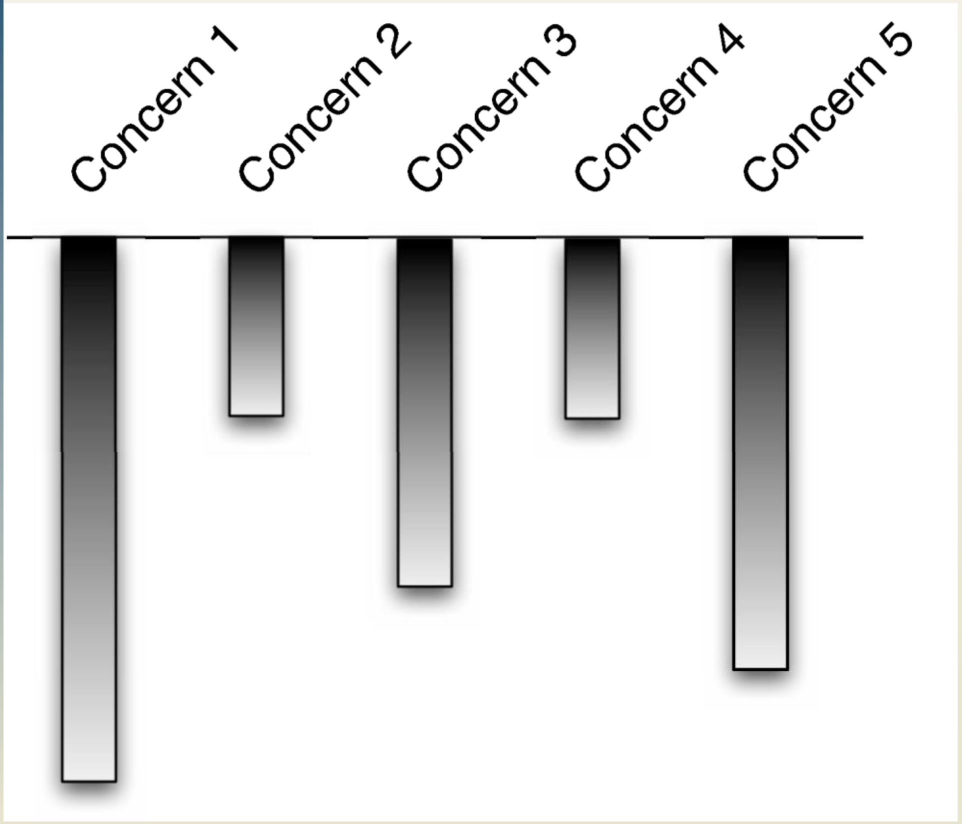
What is Architectural Modeling?

- 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

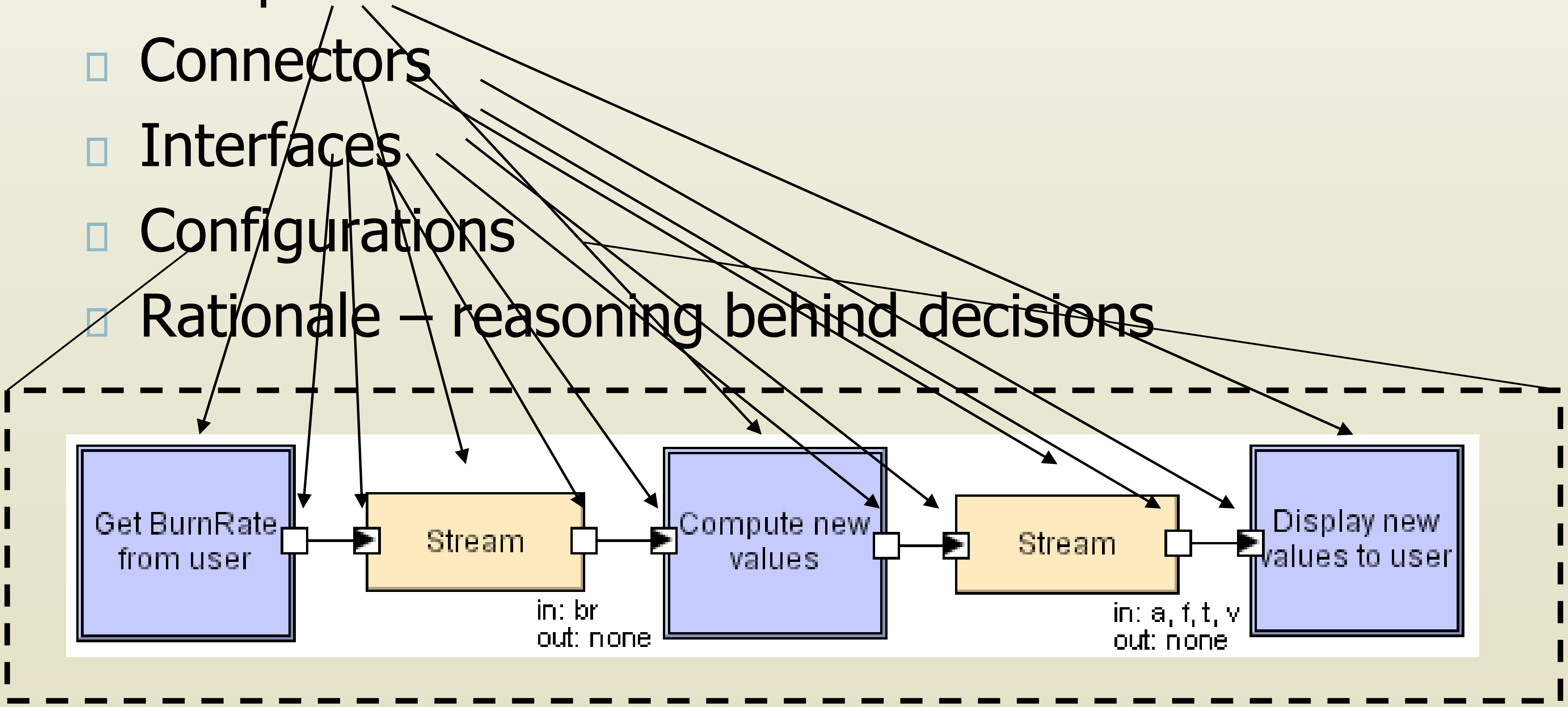
From Maier and Rechtin, "The Art of Systems Architecting" (2000)

Basic Activities Behind Stakeholder-Driven Modeling

1. Identify relevant aspects of the software to model
2. Roughly categorize them in terms of importance
3. Identify the goals of modeling for each aspect
    1. Communication
    2. Bug finding
    3. Quality analysis
    4. Generation of other artifacts
    5. Etc.
4. Select modeling notations that will model the selected aspects at appropriate levels of depth to achieve the modeling goals
5. Create the models
6. Use the models in a manner consistent with the modeling goals

What do We Model?

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



What do We Model? (cont’d)

- Components
  - The architectural building blocks that encapsulate a subset of the system’s functionality and/or data, and restrict access to them via an explicitly defined interface
- Connectors
  - Architectural building blocks that affect and regulate interactions among components
- Interfaces
  - Points at which components and connectors interact with the outside world–in general, other components and connectors

What do We Model? (cont'd)

- Configurations
  - A set of specific associations between the components and connectors of a software system's architecture; such associations may be captured via graphs whose nodes represent components and connectors, and whose edges represent their interconnectivity
- Rationale
  - The information that explains why particular architectural decisions were made, and what purpose various elements serve

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)

- Elements of the architectural style
  - *Specific Elements*
    - A style may prescribe that particular components, connectors or interfaces be included in architectures or used in specific situations
  - *Component, Connector, and Interface Types*
    - Specific types of elements may be permitted, required or prohibited in the architecture
  - *Constraints on Interaction*
    - Temporal ("calling components may call `init()` before any other method")
    - Topological ("only components in the *client* layer are allowed to invoke components in the *server* layer")
    - Specify particular protocols (e.g., FTP or HTTP)

What do We Model? (cont'd)

- Elements of the architectural style (cont'd)
  - *Behavioral Constraints*
    - They define how architectural elements behave and can range from simple rules to complete behavioral specifications of components
  - *Concurrency Constraints*
    - Constraints on which elements perform their functions concurrently and how they synchronize access to shared resources

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, host and network configurations or mapping of architectural elements to code or binary artifacts
  - Dynamic aspects *do* change as a system runs
    - E.g., state of individual components or connectors over time (behavioral models) or state of a data flow through a system over time

What do We Model? (cont'd)

- Static and Dynamic Aspects (cont'd)
  - The static/dynamic distinction is often unclear
    - Consider a system whose topology is relatively stable but changes several times during system startup
    - Or changes due to component failure, the use of flexible connectors or architectural dynamism
  - In such cases, models that capture both static and dynamic system aspects may be employed
    - E.g., a static base topology may be accompanied by a set of transitions that describe a limited set of changes that may occur to that topology during execution

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)
- The former refer to properties of the system being modeled
- The latter refer to changes to the models themselves
- A *model of a dynamic aspect* of a system describes how the system changes as it executes
- A *dynamic model* actually changes itself

What do We Model? (cont'd)

- Functional and non-functional aspects of a system
  - Functional aspects relate to *what* a system does
    - "The system prints medical records"
  - Non-functional aspects relate to *how* a system performs its functions
    - "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 specific, detailed, and exact

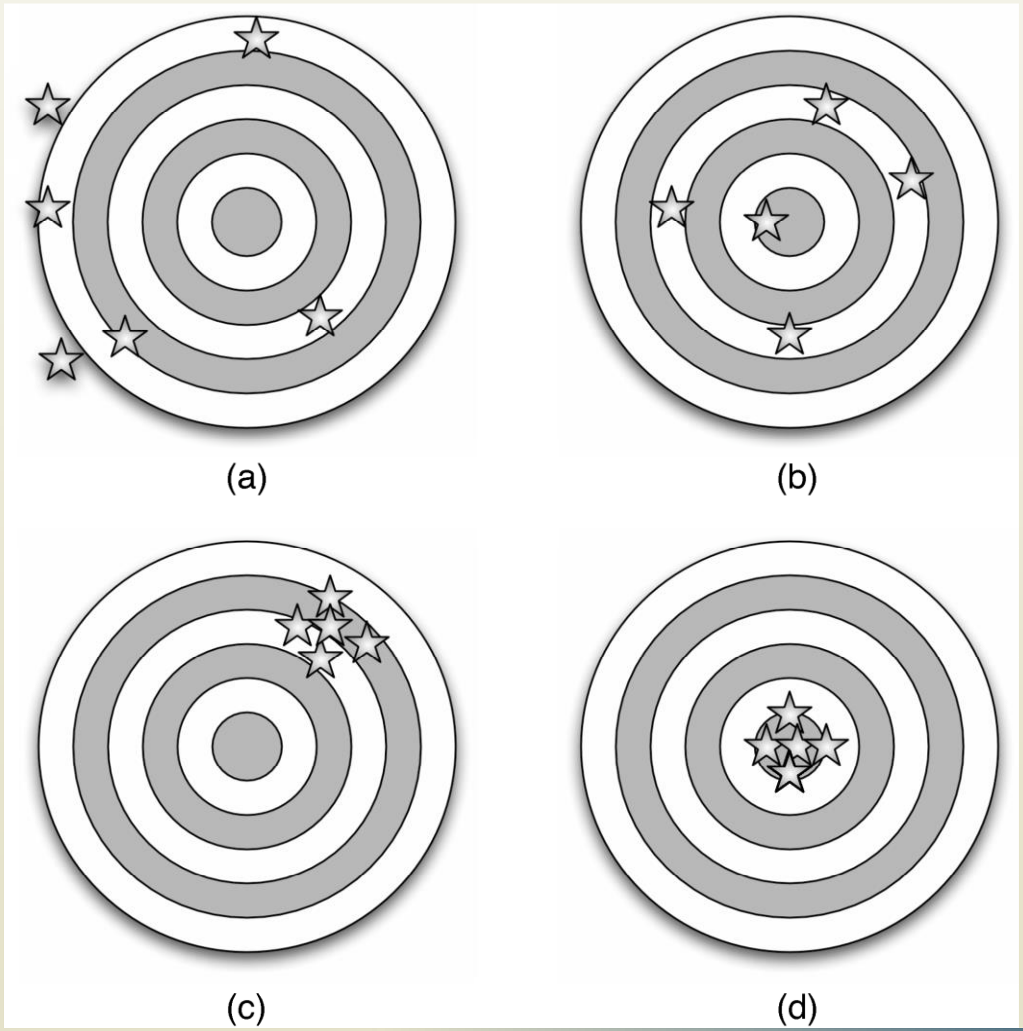
Accuracy vs. Precision

Inaccurate and imprecise: incoherent or contradictory assertions

Accurate but imprecise: ambiguous or shallow assertions

Inaccurate but precise: detailed assertions that are wrong

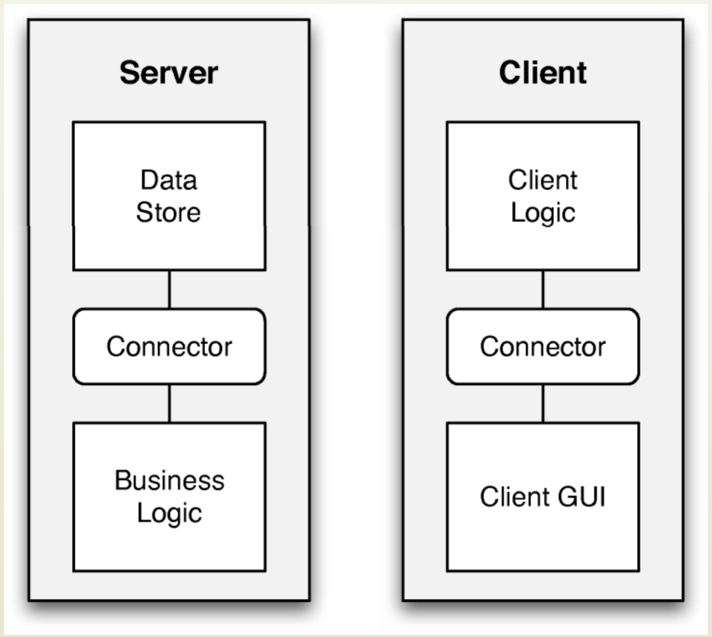
Accurate and precise: detailed assertions that are correct

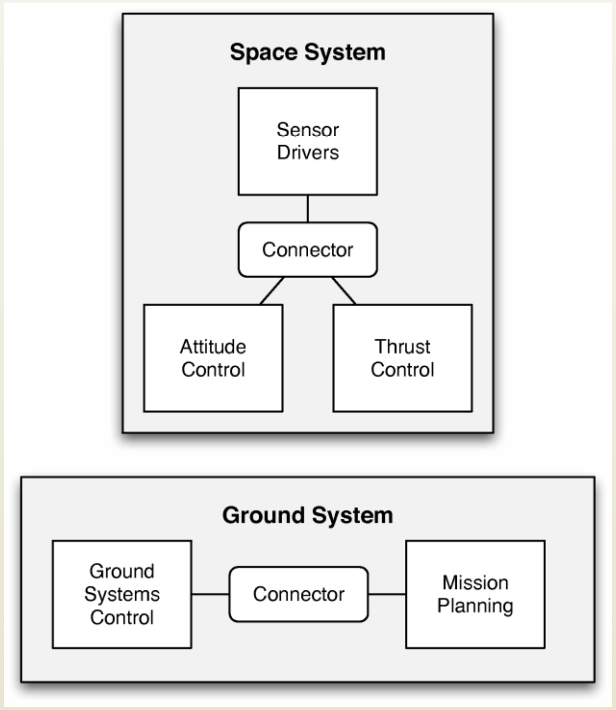


Views and Viewpoints

- 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’
- **A *view* is a set of design decisions related by a common concern (or set of concerns)**
- **A *viewpoint* defines the perspective from which a view is taken**

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
- It is also possible for multiple views to be taken from the same viewpoint for the same system
  - One view might show only top-level components, while another view might show additionally subcomponents and internal structure

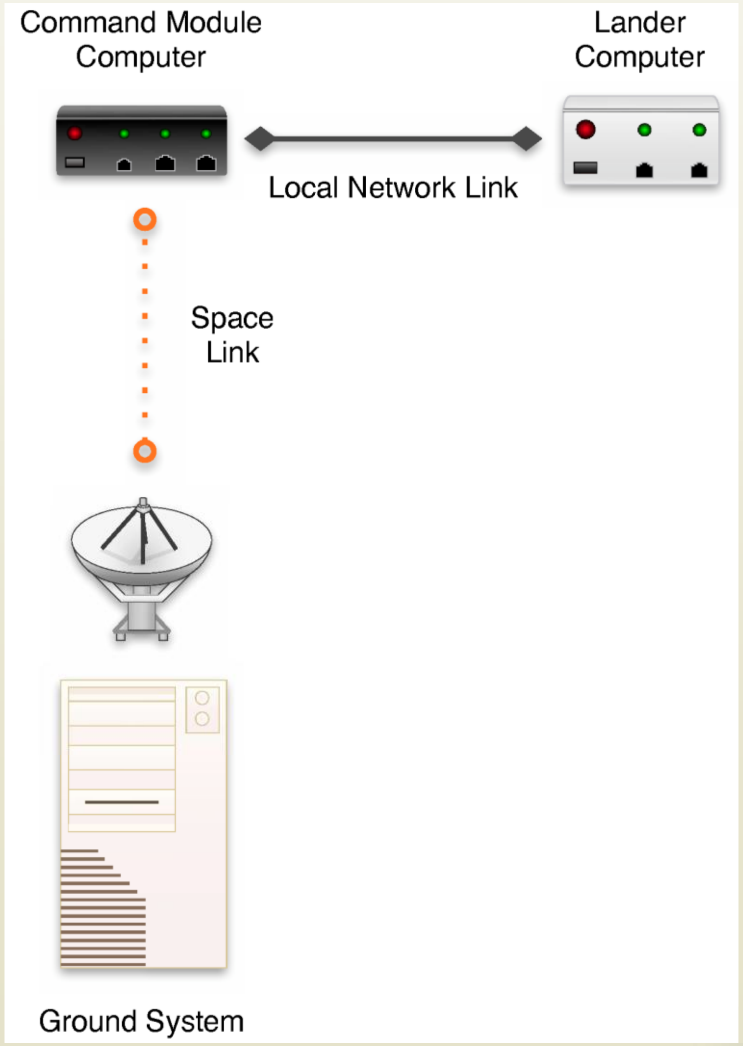
Commonly-Used Viewpoints (cont'd)

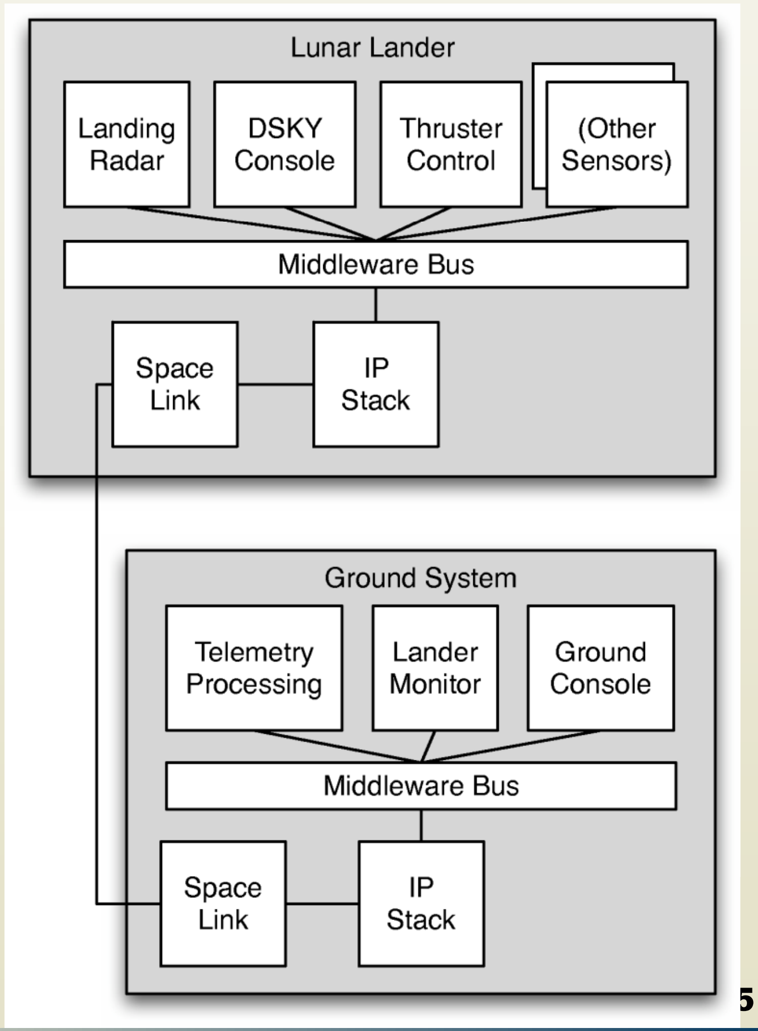
- Importance of views and viewpoints
  - They provide a way to limit presented information to a cognitively manageable subset of the architecture
  - They display related concepts simultaneously
  - They can be tailored to the needs of specific stakeholders
  - They can be used to display the same data at various levels of abstraction

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





Example of View Inconsistency (cont’d)

- The previous figure shows two hypothetical views of a distributed LL system
- The physical view (a) depicts three hosts (`Ground System`, `Command Module Computer`, `Lander Computer`)
- However, the deployment view (b) shows components assigned to only two hosts (`Ground System`, `Lunar Lander`)
- This inconsistency is relatively easy to spot; other, more subtle inconsistencies (e.g., between different behavioral specifications of an architecture) are harder to detect and more costly to fix

Common Types of Inconsistencies

- *Direct inconsistencies*
  - When two views assert directly contradictory propositions
    - 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
    - E.g., a “top level” structural view contains a component that is absent from a structural view that includes subarchitectures

Common Types of Inconsistencies (cont'd)

- *Static vs. dynamic aspect inconsistencies*
  - Dynamic aspects (e.g., behavioral specifications) conflict with static aspects (e.g., topologies)
  - E.g., a message sequence chart view might depict the handling of messages by a component that is not contained in the structural view
- *Dynamic vs. dynamic aspect inconsistencies*
  - Different descriptions of dynamic aspects of a system conflict
  - E.g., a message sequence chart depicts a specific interaction between components that is not allowed by the relevant behavioral specifications

Common Types of Inconsistencies (cont’d)

- *Functional vs. non-functional inconsistencies*
  - When a non-functional property of a system prescribed by a non-functional view is not met by the design expressed by functional views
  - E.g., a non-functional view of a client-server system may express that the system should be robust, but the physical view of the system may show only a single server with no evidence of failure-handling machinery

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 or concepts (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 allow) 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?
- View Consistency
  - How does the approach help you assess or maintain consistency among different views expressed in a model?

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

Lunar Lander in Natural Language

“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.”*

Natural Language Example (cont'd)

- The structure of the components and their dependencies is explicitly stated, as well as a description of their behaviors, inputs, outputs, and general responsibilities
- However, the description does not explain the algorithm the calculation component uses, the particular formats of the data values, anything about the connectors between the components, or what the user interface should look like
- The ambiguity of the natural language may result in the generation of many different implementations that satisfy this architectural description and they may not all function identically

Related Alternatives

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

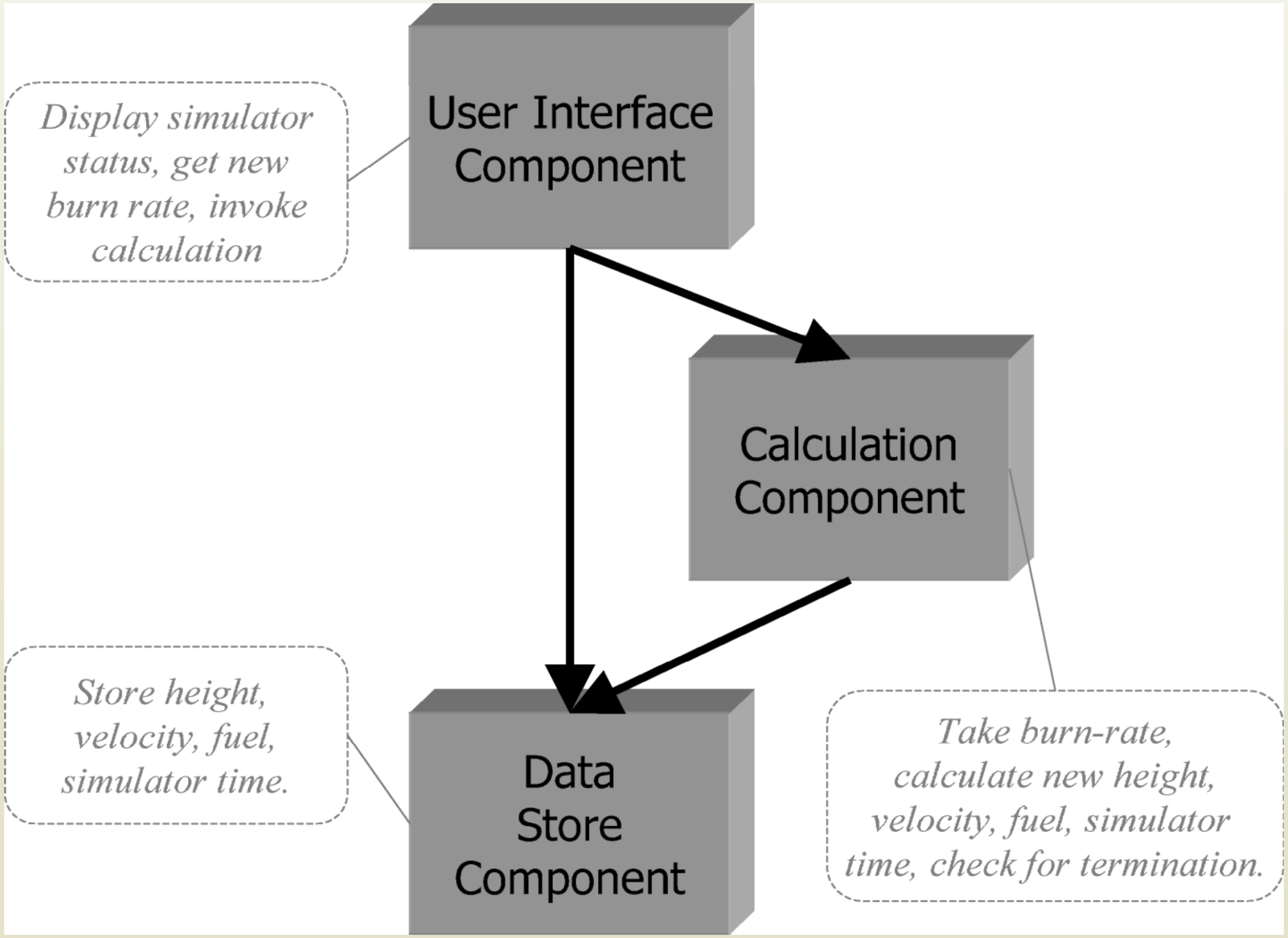
Natural Language Evaluation

- 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

Lunar Lander as an Informal Graphical Model


Display simulator status, get new burn rate, invoke calculation
User Interface Component
Calculation Component
Store height, velocity, fuel, simulator time.
Data Store Component
Take burn-rate, calculate new height, velocity, fuel, simulator time, check for termination.

Informal Graphical Model Example (cont’d)

- The particular symbology used is not directly explained– there is no underlying semantic model through which to interpret this diagram
- The intended interpretation is that the 3D boxes are software components, the arrows indicate invocation dependencies, and the round rectangles are commentary on the intended behavior and responsibilities of the components
- Somewhat better than the natural language model, in that the componentization of the system and the component dependencies are immediately obvious, and the behaviors are visually connected to the components

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

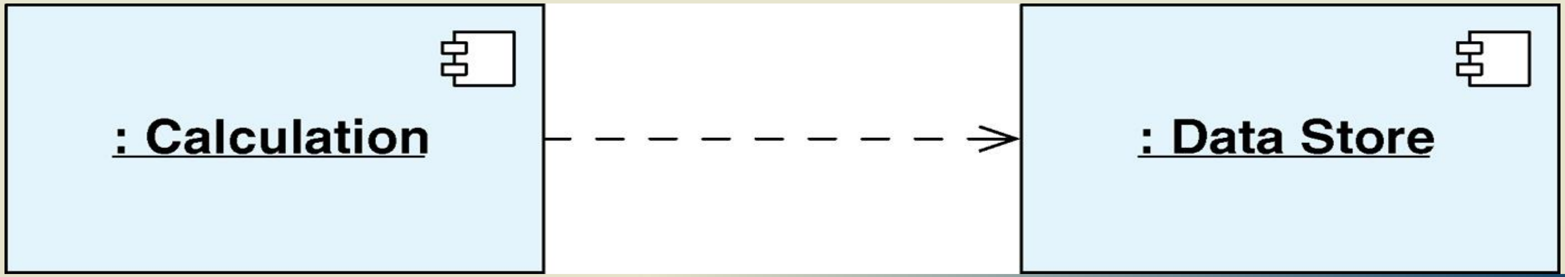
- Scope and purpose
  - Arbitrary diagrams consisting of symbols and text
- Basic elements
  - Geometric shapes, splines, clip-art, text segments
- Style
  - In general, no support
- Static & Dynamic Aspects
  - Any aspect can be modeled, but no semantics behind models
- Dynamic Models
  - Rare, although 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 (what in this course we call 'viewpoints') 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

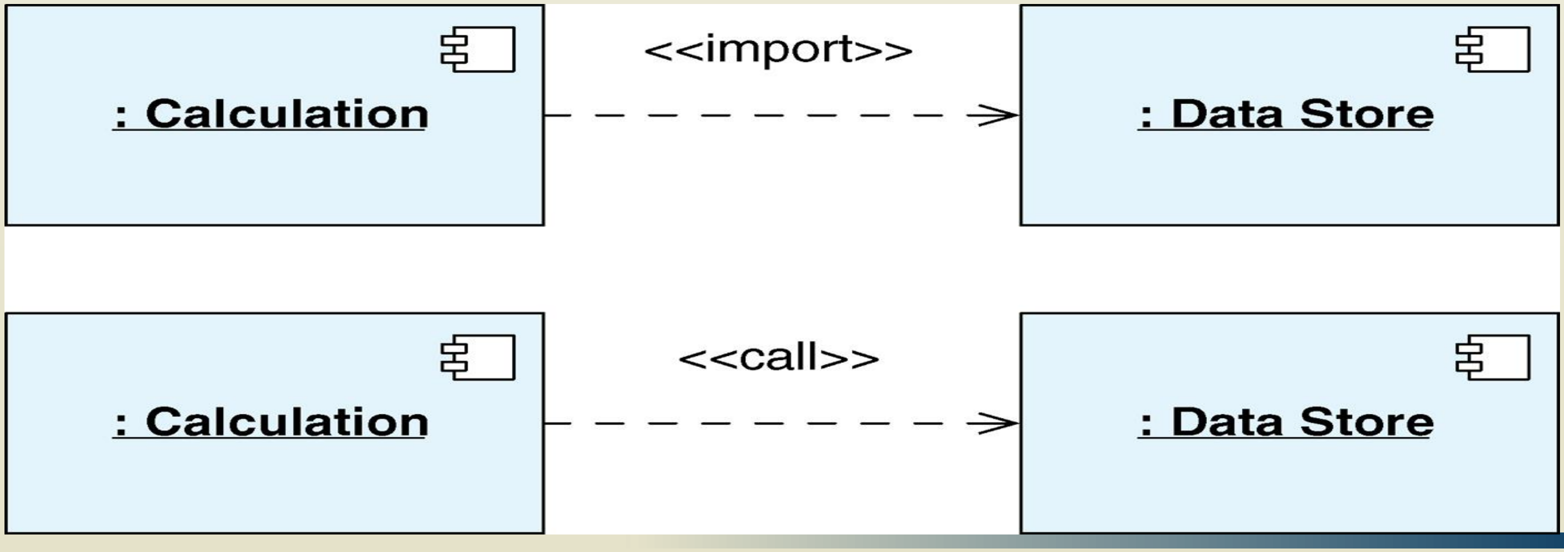
UML – the Unified Modeling Language (cont'd)

- The figure below shows two components with a dashed arrow showing a 'dependency': `Calculation` is dependent on `Data Store`; however, it is unclear what kind of dependency this is
  - Some element of `Calculation` calls `Data Store`?
  - Instances of `Calculation` contain a pointer to an instance of `Data Store`?
  - `Calculation` requires `Data Store` to compile?
  - `Calculation` can send messages to `Data Store`?
  - `Calculation`'s implementation has a method that takes an instance of `Data Store`'s implementation as a parameter?

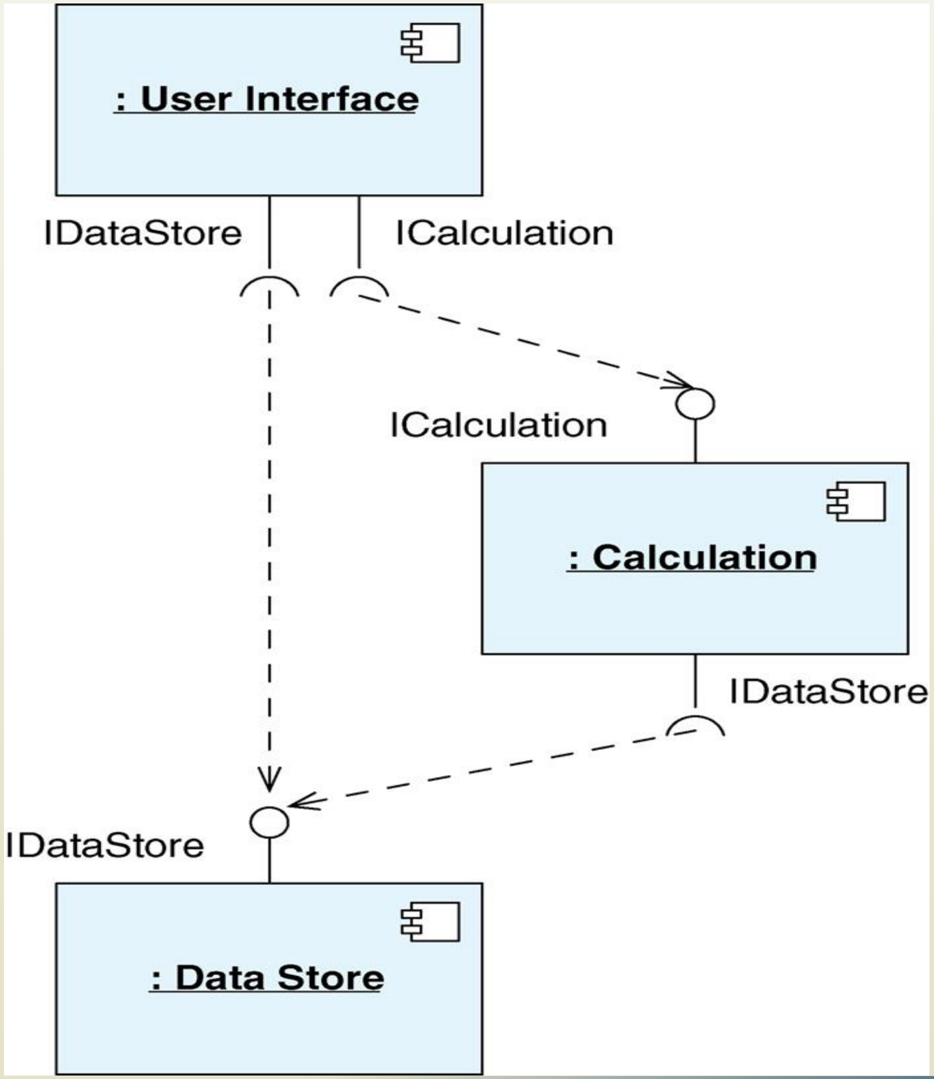


UML – the Unified Modeling Language (cont'd)

- UML offers *stereotypes* or *tagged values* to provide more info
- In the figure below, the stereotypes used give more details about the dependency between the two components
- In the first case, `Calculation` imports `Data Store` and in the second `Calculation` calls `Data Store`
- But what exactly does it mean to "import" or "call"?



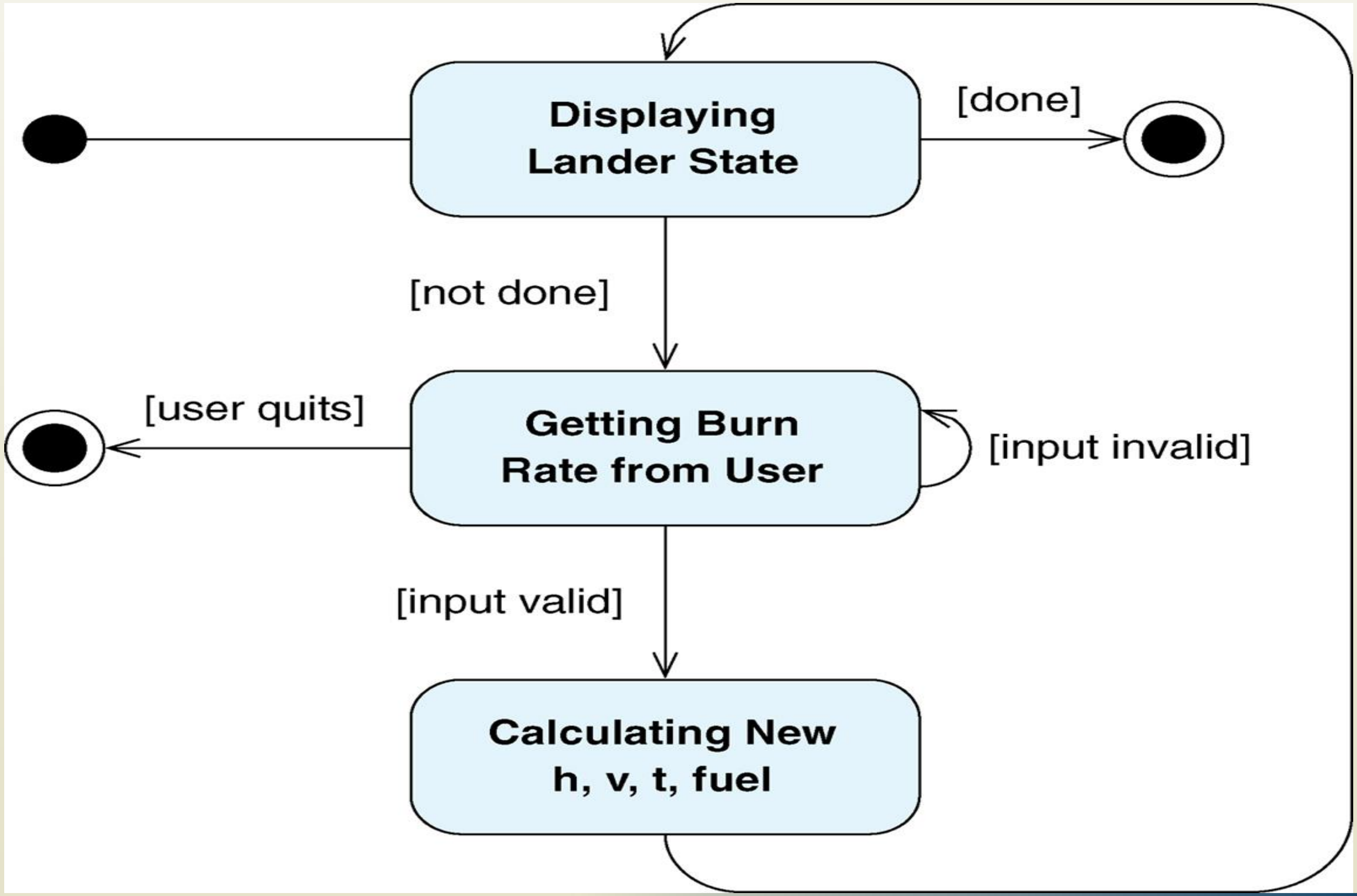
Lunar Lander in UML



Lunar Lander in UML (cont'd)

- The component diagram of LL might look like the one in the previous slide and it looks very similar to the informal graphical diagram
- However, this diagram has a rigorous syntax and some underlying semantics
- The symbols used are documented in the UML specification
- Informal boxes in the graphical diagram are replaced by the well-defined 'component' symbol
- However, the diagram is not completely unambiguous
  - What kind of components are involved?
  - When and how are calls among components made?

Lunar Lander in UML (cont'd)



Lunar Lander in UML (cont’d)

- The behavior of the system might be specified using a UML statechart diagram, as the one in the previous slide
- The start state is indicated by the plain dark circle and the end state is indicated by the outlined dark circle
- Each rounded rectangle represents a state of the system
- Arrows represent transitions between the states
- The conditions in square brackets indicate guards that constrain when state transitions may occur

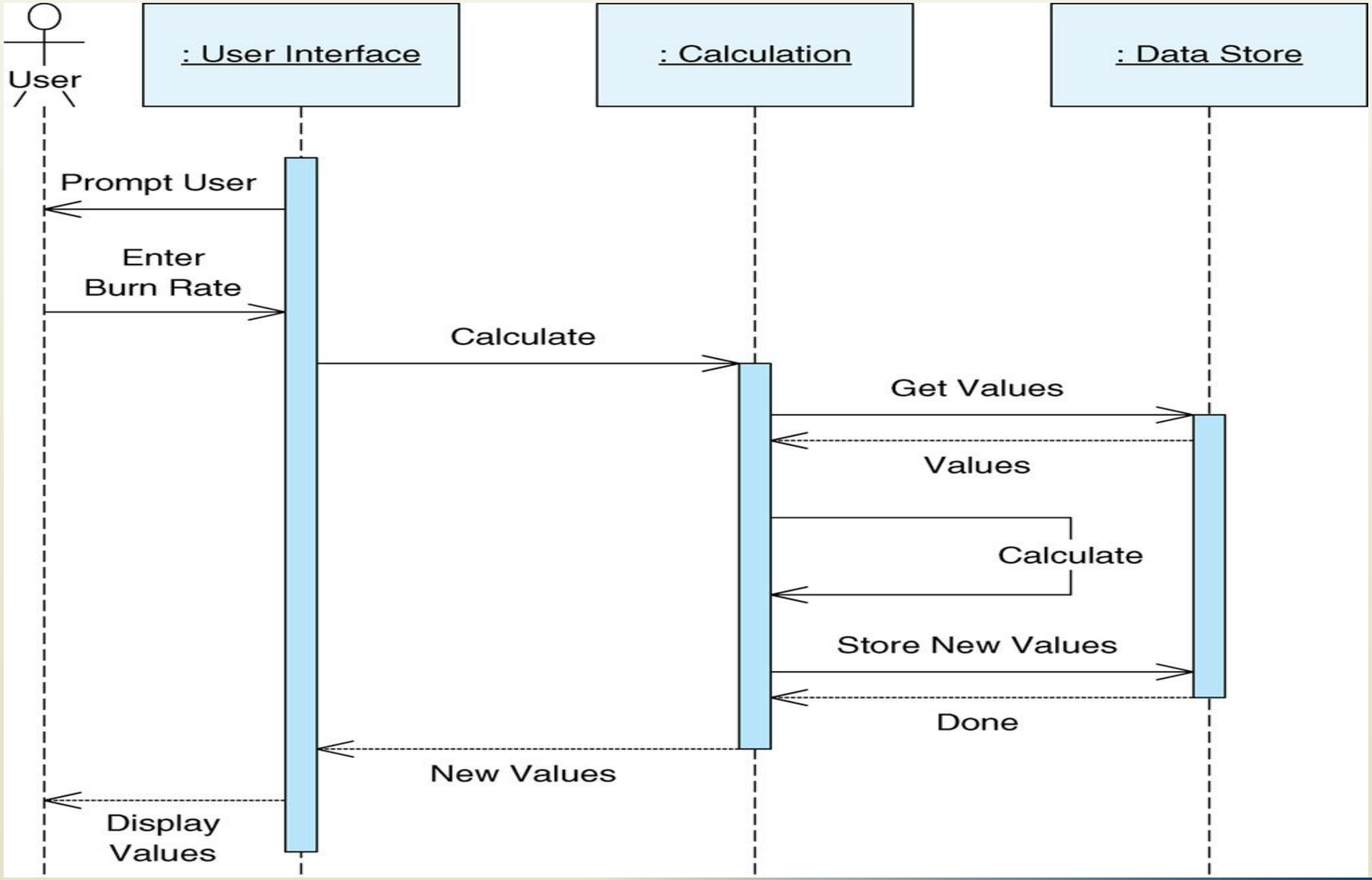
Lunar Lander in UML (cont'd)

- According to this statechart:
  - The LL system begins by displaying the lander state
  - Then, either the simulation is done, or the system will request a burn rate from the user
  - The user may choose to end the program or provide a burn rate and, if this is valid, the program will then calculate the new simulation state and display it
  - The loop will repeat until the simulation is done

Lunar Lander in UML (cont'd)

- The statechart provides a more rigorous and formal description of the system behavior than either the natural language or informal graphical architecture description
- However, it leaves out an important detail:
  - Which components perform the specified actions?
- This is captured in another type of a UML diagram, the sequence diagram

Lunar Lander in UML (cont’d)



Lunar Lander in UML (cont'd)

- The sequence diagram depicts a particular sequence of operations that can be performed by the LL components
- `User Interface` gets a burn rate from the user, `Calculation` retrieves the state of the lander from the `Data Store` and updates it, and then returns the termination state of the lander to `User Interface`
- The previous three diagrams capture both the static (structural) and dynamic (behavioral) aspects of the system
- Are these different views consistent with each other?

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

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

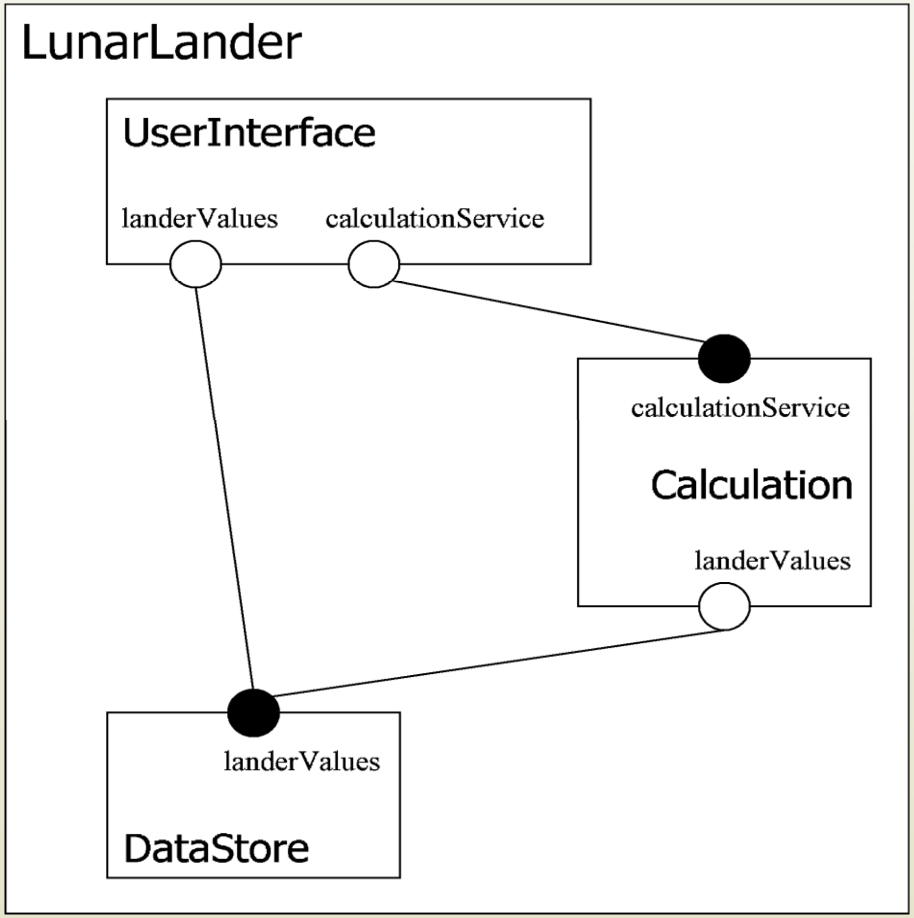
LL in Darwin

```
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;
}
```



Canonical Textual Visualization

Graphical Visualization

LL in Darwin (cont'd)

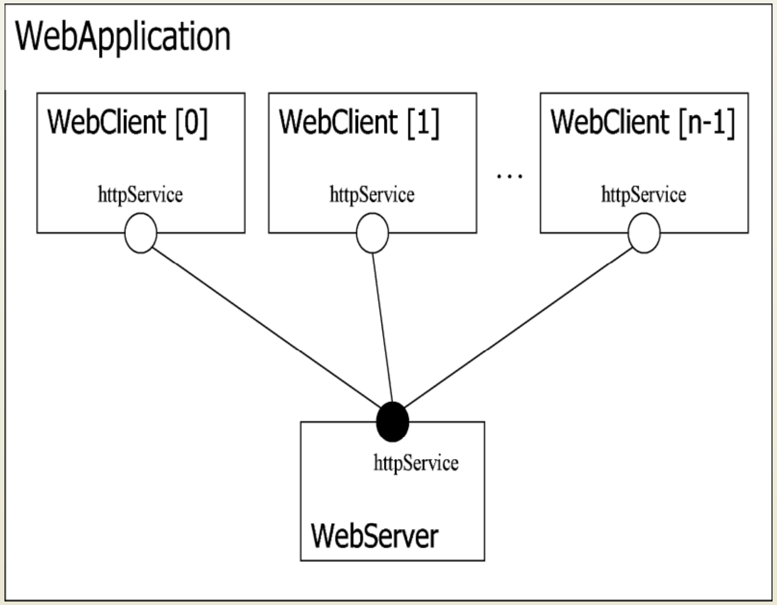
- Each component is described with explicit provided and required interfaces
- The overall application structure is defined using a top-level component with an internal structure
  - I.e., the LL application itself is a component containing the `UserInterface`, `Calculation`, and `DataStore` components

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;
    }
}
```



- Many declarative ADLs simply enumerate all the components and bindings in an architecture one by one
- Darwin, in addition, supports the creation of configurations using programming-language-like constructs, such as loops
- In the above code the number of clients is parameterizable

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
  - Architecture specifications in Rapide are interesting because they are *executable*
  - 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

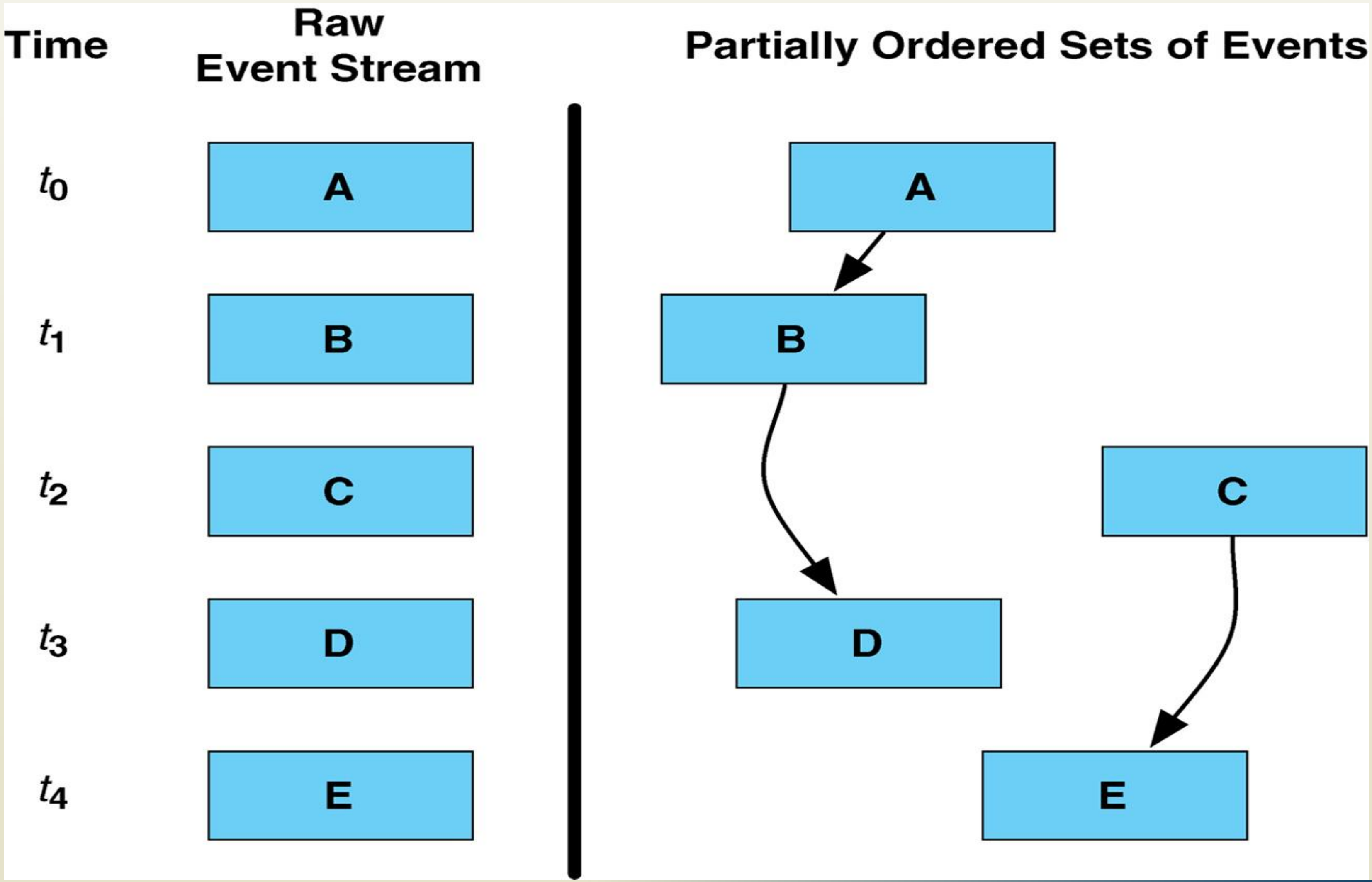
POSETs

- The power of Rapide comes from its organization of events into partially ordered sets, called POSETs
- Rapide components work concurrently, emitting and responding to events
- There are causal relationships between some events, e.g., if a component receives event `A` and responds by emitting event `B`, then there is a causal relationship from `A`→`B`

POSETs (cont'd)

- Causal relationships between two events `A` and `B` in Rapide exist when any of the following are true:
  - `A` and `B` are generated by the same process
  - A process is triggered by `A` and then generates `B`
  - A process generated `A` and then assigns to a variable `v`, another process reads `v` and then generates `B`
  - `A` triggers a connection that generates `B`
  - `A` precedes `C` which precedes `B` (transitive closure)
- As a program runs, its components generate a stream of events over time and some of these events will be causally related by one of the above relationships

POSETs (cont’d)



POSETs (cont'd)

- In the previous figure, the left portion shows a raw event stream over time: components in some software architecture modeled in Rapide send events `A`, `B`, `C`, `D`, `E` at times `t0` through `t4`, respectively
  - These events are temporally ordered but not causally ordered
- The right portion of the figure shows the causal ordering of these events; in fact, there are two partial orders:
  - `A`, `B`, and `D`
  - `C` and `E`
- We call these orderings partial because not all the events are ordered with respect to one another

LL in Rapide

```
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;
```

LL in Rapide (cont'd)

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

LL in Rapide (cont'd)

```
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;
```

LL in Rapide (cont'd)

- The Rapide description looks on the surface similar to a Darwin one but it also includes behavioral information
- In this version of LL there are two players
- The code starts by defining the types of available components and their interfaces
- Each interface has a number of events it can receive (`in`) and send out (`out`)
- In addition, each component has a behavioral specification, defining how it reacts to different events

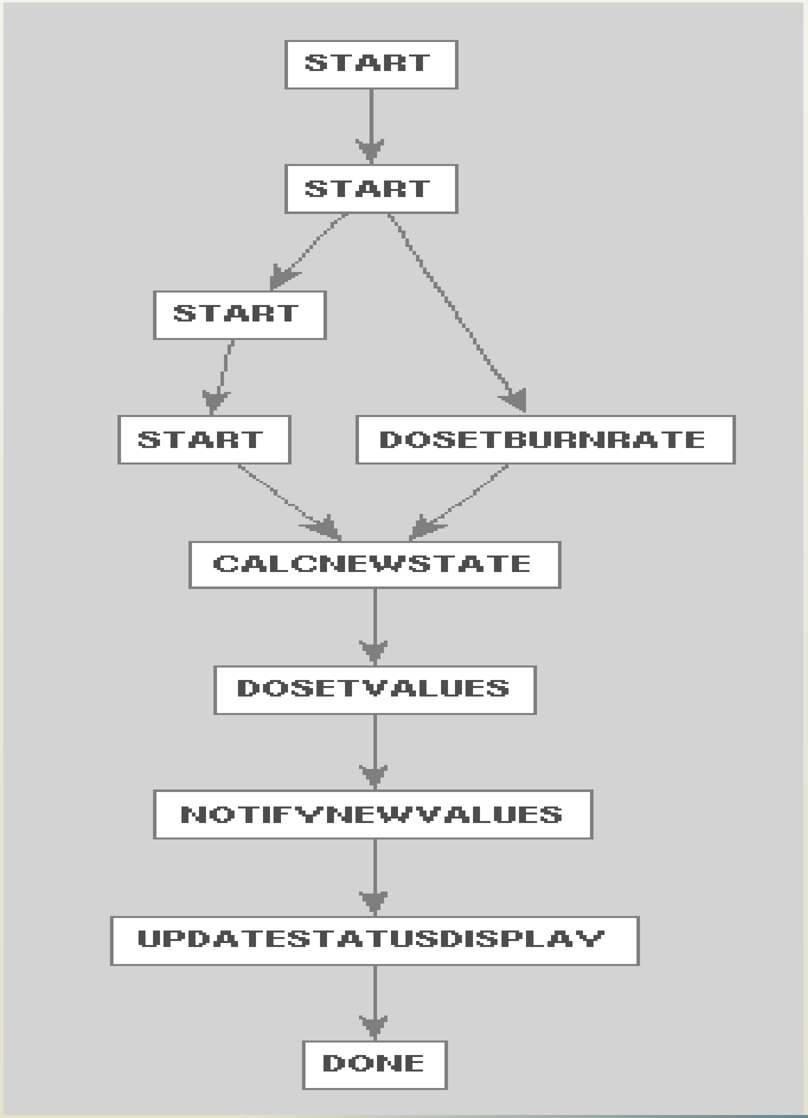
LL in Rapide (cont'd)

- At the end of the specification, the system's structure is defined
    - First components that implement the different interface types
    - Then links between the component interfaces
- The players start off by sending an updated burn rate and then they wait for the display to be updated with the new status before making another move
- In this version of the game, players are limited to three moves, in order for the game not to continue indefinitely as there is no other end-game condition

LL in Rapide (cont'd)

- The `Calculation` component waits for a `SetBurnerRate` event
- When it receives it, it will fire the internal event `CalcNewState` and then fire a `DoSetValues` message to the `DataStore` component to update the game state
- When the game state is updated, `DataStore` fires a `NotifyNewValues` event which causes the players' displays to be updated, thus prompting them to make their next moves
- Now let's see how the events are generated in such a program

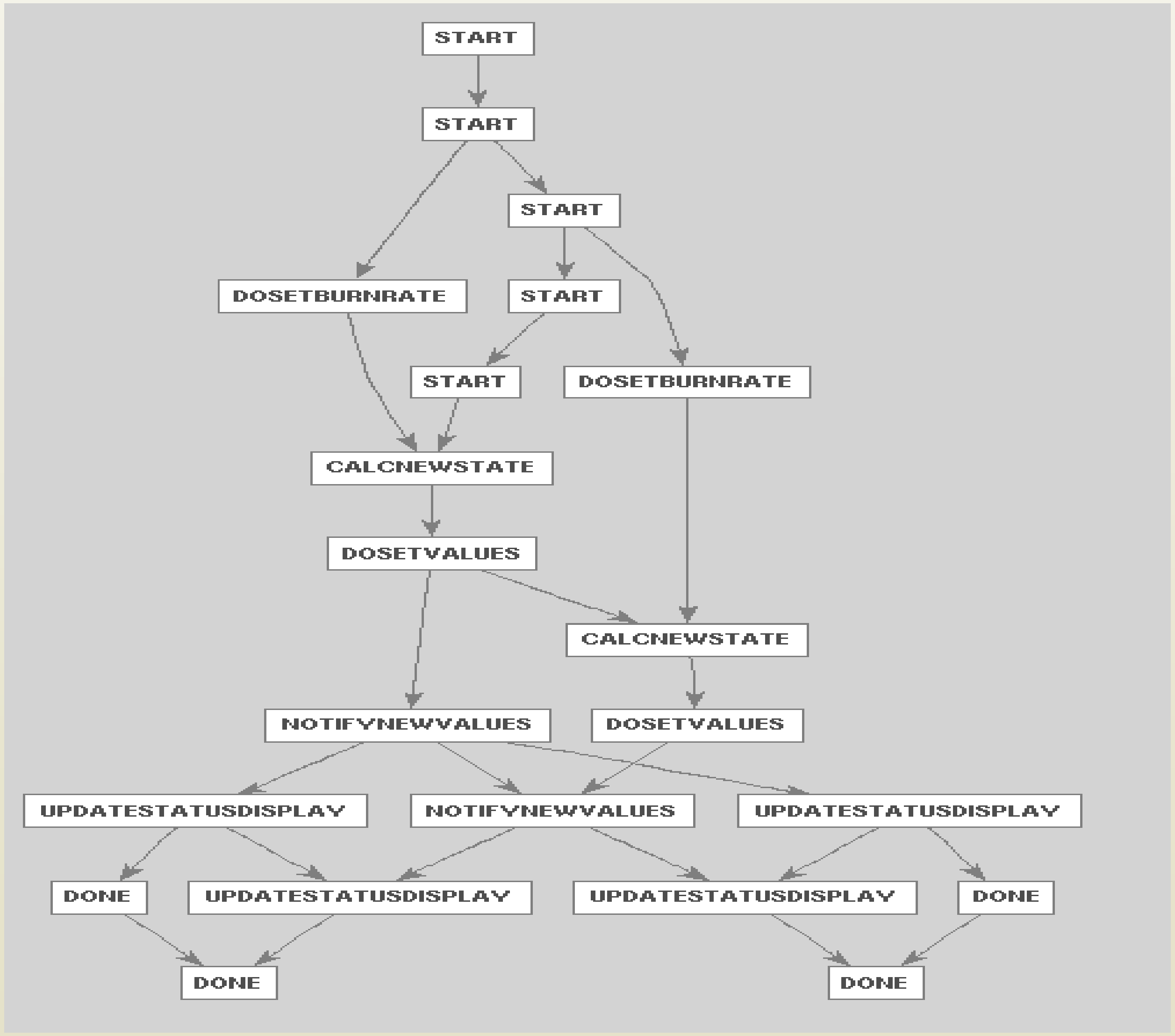
Simulation Output With One Player



Simulation Output With One Player (cont'd)

- The only counterintuitive aspect of the previous graph is the group of 'start' events that are fired initially
- As one of Rapide's primary focus areas is concurrent architectures, it attempts to trigger simultaneous processing by loading the simulation with a number of 'start' events at the beginning
- Other than this, this trace of events for a one-player version of the game looks reasonable
- Let us now see how events are generated for a two-player version of the game

Simulation Output With Two Players



Simulation Output With Two Players (cont'd)

- By examining the causality arrows, we observe that the two pathways are intertwined
  - Requests are getting intermingled, since there is no locking or transaction support in this design
- Also, we see the fan-out of display updates at the bottom
  - Each user's display is getting updated twice–once for their own move and once for the other player
  - This means that both (or all in case of more than two) players will try to move at the same time
- So, this specific modeling of LL is buggy; without the Rapide events' graphs it would be difficult to detect

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

LL in Wright

```
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)

Connector Call
```

Role Caller = $\overline{call} \rightarrow return \rightarrow Caller[\,]\S$

Role Callee = $call \rightarrow \overline{return} \rightarrow Callee[\,]\S$

$$\begin{aligned} \text{Glue} = \; & Caller.call \rightarrow \overline{Callee.call} \rightarrow Glue \\ & [\,]Callee.return \rightarrow \overline{Caller.return} \rightarrow Glue \\ & [\,]\S \end{aligned}$$

LL in Wright (cont'd)

```
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.calulate as UItoC.Callee

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

End LunarLander.
```

LL in Wright (cont'd)

- The structural aspects of the Wright specification of LL resemble those we have seen earlier in UML and Darwin
- The distinguishing feature here is the CSP-based formal specifications of components/connectors interfaces and behavior
- The value of these specifications is that properties such as freedom from deadlock can be analyzed, something difficult or impossible to detect if a system is implemented in a traditional programming language

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

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

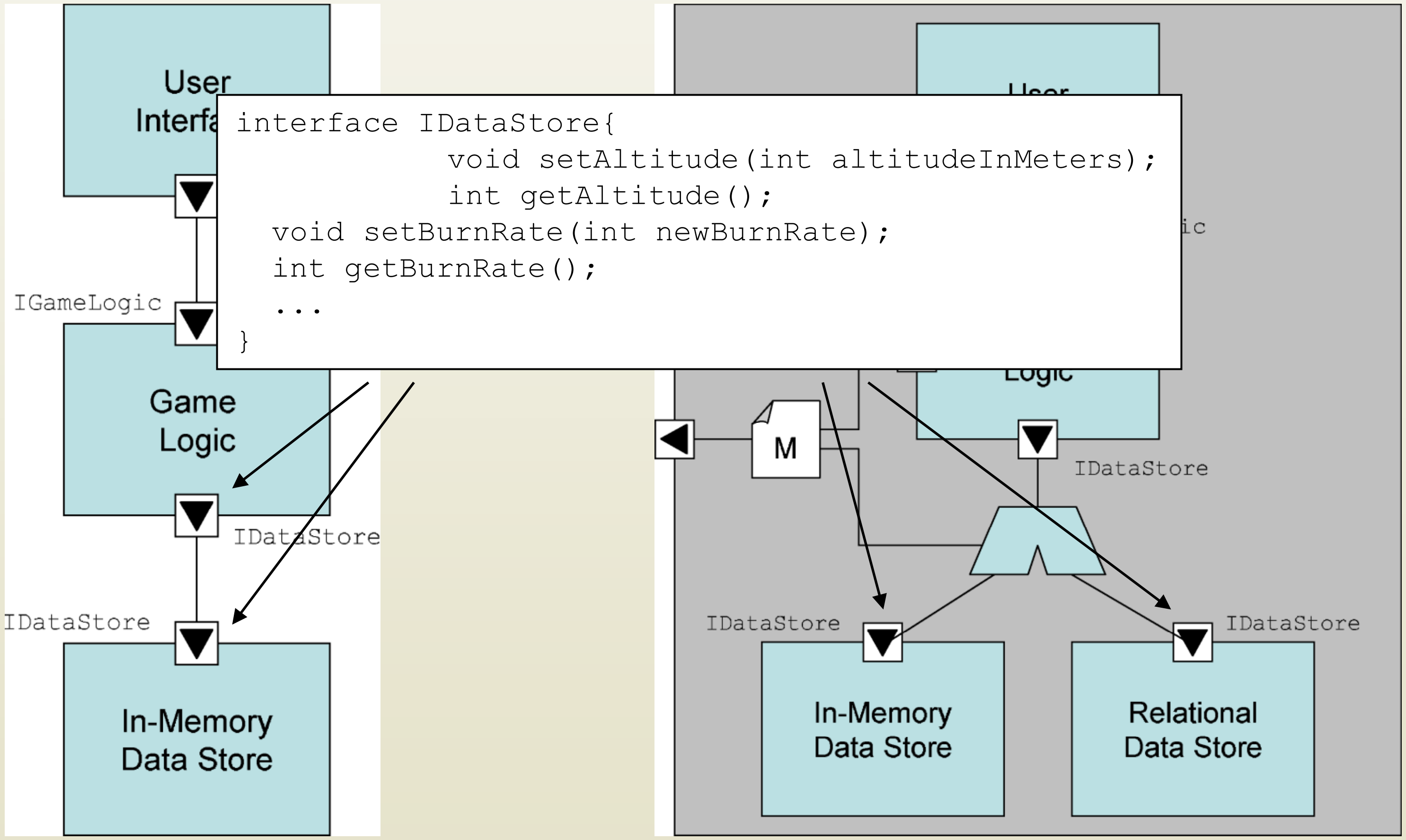
Domain- and Style-Specific ADLs

- Notations we have surveyed thus far have been generically applicable to many types of software systems; however, some ADLs are domain-specific or style-specific, or at least optimized for describing architectures in a particular domain or style
- These types of ADLs are important for several reasons:
  - Their scope is better tailored to stakeholder needs, since they target a particular group of them
  - They are able to leave out unnecessary details and excessively verbose constructs because there is little need for genericity; assumptions about the domain or style can be directly encoded into the ADL semantics
    - E.g., if a particular style mandates the use of a single kind of connector, there is no need to have a notion of connector in this ADL; users assume that all links use this connector

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

LL in Koala



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

Single system

Product line of two systems

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

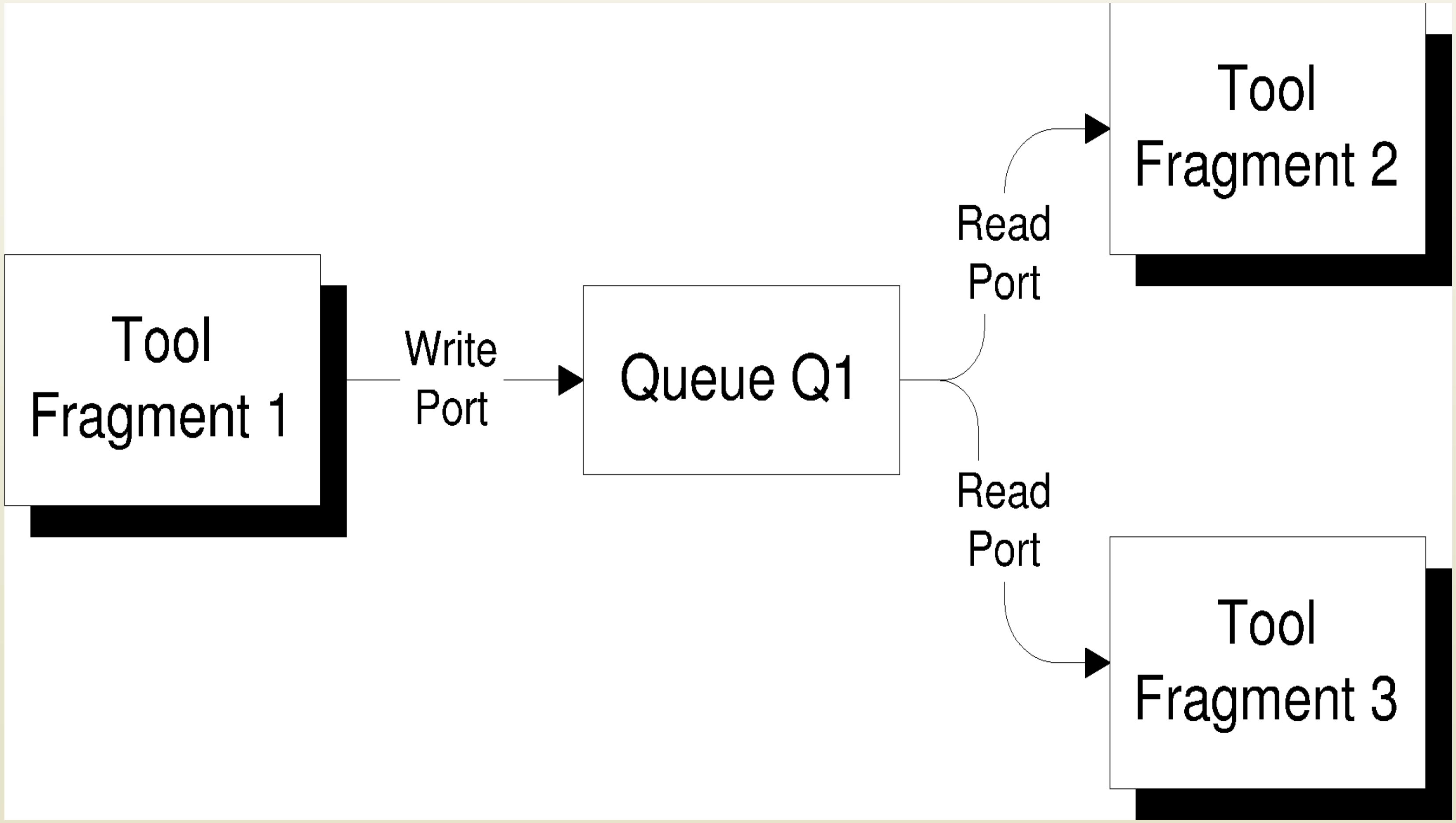
Weaves

- An architectural style and notation for modeling systems of small-grain 'tool fragments' that communicate through data flows of objects
- Can be seen as a variant of the pipe-and-filter style with three significant differences:
  - Weaves tool fragments process object streams instead of pipe-and-filter's byte streams
  - Weaves connectors are explicitly sized object queues, whereas pipe-and-filter connectors are implicit pipes
  - Weaves tools can have multiple inputs and outputs, whereas pipe-and-filter components have one input and one output

Weaves (cont'd)

- Advantages
  - Extremely optimized notation
    - Even simpler than Darwin diagrams
  - Close mapping to implemented systems
- Disadvantages
  - Addresses structure and data flows only

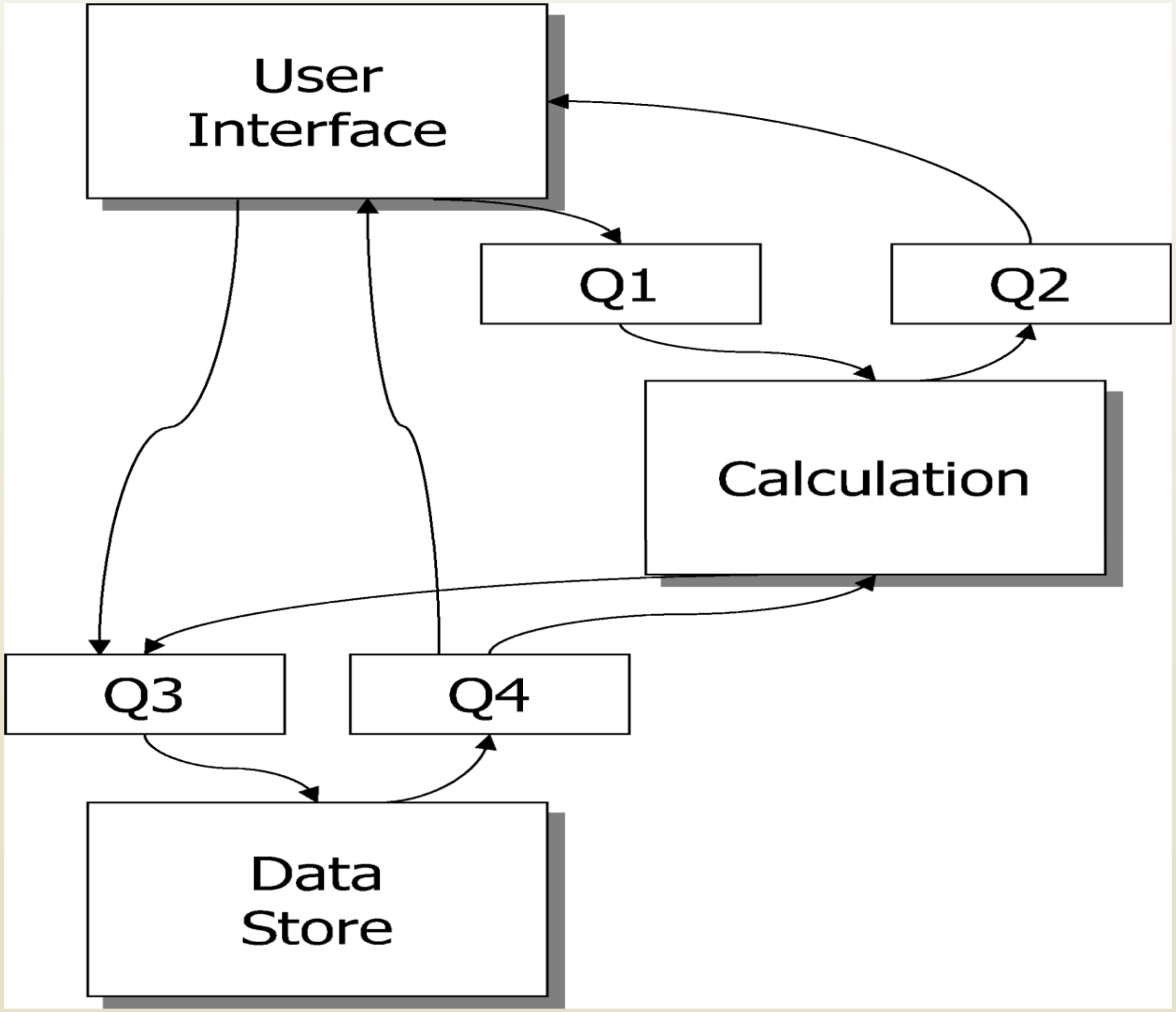
Weaves Example



Weaves Example (cont'd)

- The component `Tool Fragment 1` outputs a stream of objects to an explicit queue connector `Q1`, which forks the stream and forwards the objects to both `Tool Fragment 2` and `Tool Fragment 3`
- The notation is graphical and minimalistic
  - Components are represented by shadowed boxes and queue connectors are represented by plain boxes
  - Configurations are expressed using directed arrows connecting components and connectors
- This minimal notation is adequate to serve as a description notation for the Weaves style and the semantic interpretation of the few elements are provided by the style itself

LL in Weaves



LL in Weaves(cont'd)

- Although the Weaves model is almost identical to the Darwin one, the meaning is different because a component in Weaves is not the same as a component in Darwin
- The components in Weaves don't communicate by means of request-response procedure calls, but instead through streams of objects
- One notable difference from other models is the explicit presence of return channels
  - The fact that a request travels from `User Interface` to `Calculation` (through `Q1`) doesn't imply that a response comes back along the same path; this response's path must be explicitly specified and have its own queue (`Q2`)

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
  - This detail is what gives AADL its power and analyzability
  - 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

LL in AADL

```
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;
```

LL in AADL (cont’d)

```
connections
  bus access network -> the_calculation_processor.network;
  event data port response_get ->
              the_calculation_process.response_get;
  event port the_calculation_process.request_get ->
              request_get;
  event data port response_store ->
              the_calculation_process.response_store;
properties
  Actual_Processor_Binding => reference
              the_calculation_processor applies to
              the_calculation_process;
end calculation_type.calculation;

processor calculation_processor_type
features
  network : requires bus access
              lan_bus.calculation_to_datastore;
end calculation_processor_type;

process calculation_process_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;
end calculation_process_type;
```

LL in AADL (cont'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;
```

LL in AADL 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

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

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

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

LL in Acme

```
//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;
  }
```

LL in Acme (cont'd)

```
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;
   };
}
```

LL in Acme

- The model is verbose, because:
  - Acme is domain neutral and its semantics make few assumptions that allow information to be conveyed implicitly
  - The intention is that the user doesn’t write code but the latter is generated from a graphical environment (AcmeStudio)
- The basic model has only one property
  - Each procedure call connector is annotated with a property indicating whether or not the operation has a return value
  - This may be useful if the designer wants to use asynchronous communication: if a caller doesn't require a return value, it can continue executing

LL in Acme (cont'd)

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

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

LL in Acme

- Additional properties can add detail to the basic model
- In the previous slide, an extended description of `DataStore` is defined, indicating that this component should store its data in a non-replicated table called `LanderTable` in a relational database
- However, these particular names and values don't have a universal meaning
  - Tools and stakeholders must be informed about which properties to expect and how to process their values

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
  - Did not take advantage of XML extension mechanisms and instead provides extensibility by simply encoding Acme’s name-value pair properties in XML

LL in ADML

- 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 (cont'd)

- The syntax in xADL is defined in a set of XML schemas
- XML schemas are similar to DTDs but the latter define document syntax through production rules and the former define syntax through a set of data types
- xADL is the composition of all the xADL schemas, where each such schema adds a set of features to the language; this has the following advantages:
  - *Incremental adoption*–users can use as few or as many features make sense in their domain
  - *Divergent extension*–users can extend the language to tailor it to their own purposes
  - *Feature reuse*–schemas can be shared among projects that need common features

xADL (cont'd)

- Because xADl can be extended with unforeseen constructs, it requires its own tools to cope with a notation whose syntax may change from project to project
- *The xADL Binding Library*
  - A data binding library consisting of a set of Java classes that correspond to xADL data types
  - A program can query and manipulate instances of these classes to explore and change an xADL document
- *Apigen*
  - A xADL data binding library generator: given a set of XLM schemas, it generates the complete data binding library with support for these schemas

xADL and xADLite

- The native storage format of xADl is in an XML format which is very verbose
- This is also because xADL makes extensive use of XML namespaces and multiple schemas that add a significant amount of “housekeeping” data to xADL documents
- A simple component in xADL XML format would look like the code in the following slide

xADL and xADLite (cont'd)

```
<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>
```

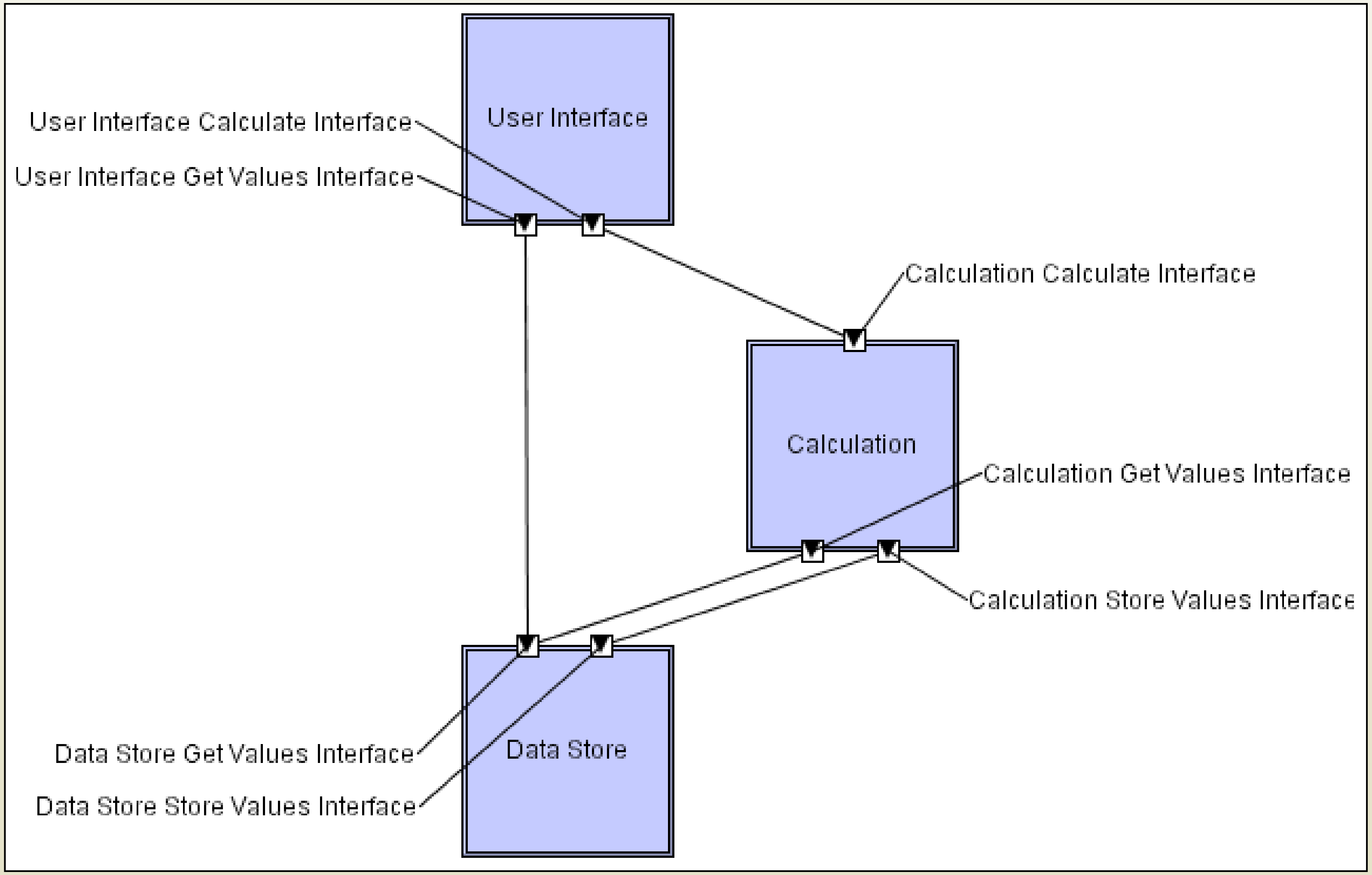
xADL and xADLite (cont’d)

- Even if we remove namespace and XML-typing information, the result is still verbose
- There is an alternative, more syntactically terse format, called xADLite
- The transformation between xADL and xADLite is lossless and no information is lost in translating a document from one formalism to the other
- The same component in xADLite looks now like the code in the following slide

xADL and xADLite (cont'd)

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

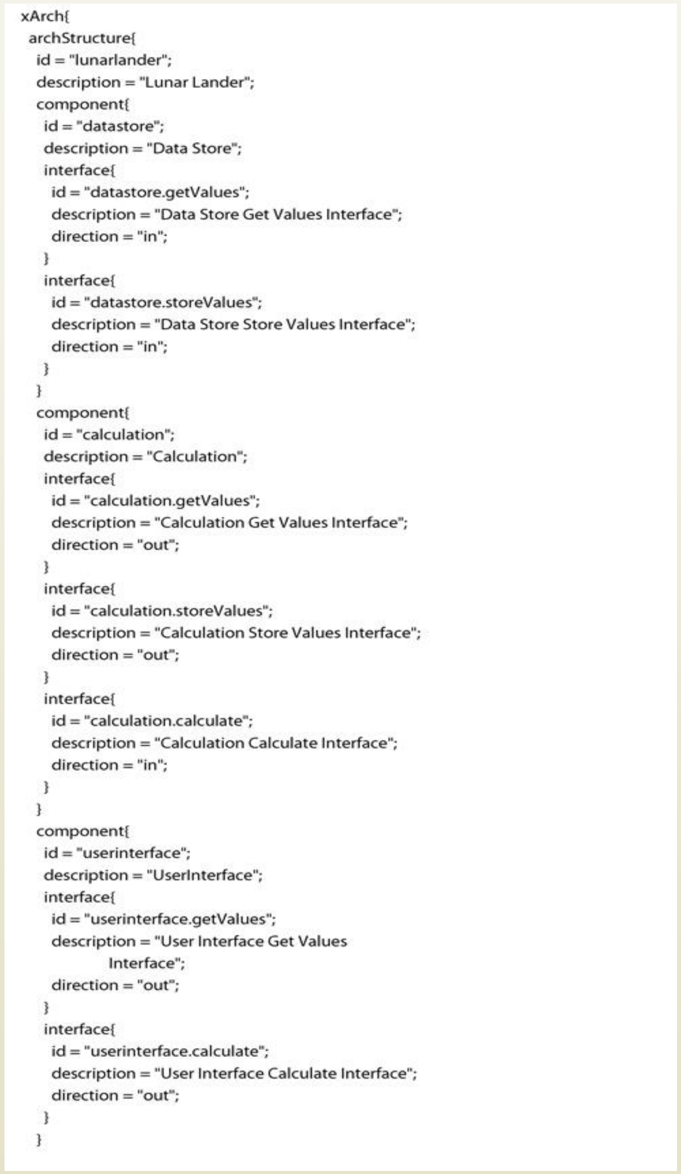
LL in xADL


User Interface
User Interface Calculate Interface
User Interface Get Values Interface
Calculation Calculate Interface
Calculation
Calculation Get Values Interface
Calculation Store Values Interface
Data Store Get Values Interface
Data Store Store Values Interface
Data Store

LL in xADL (cont’d)

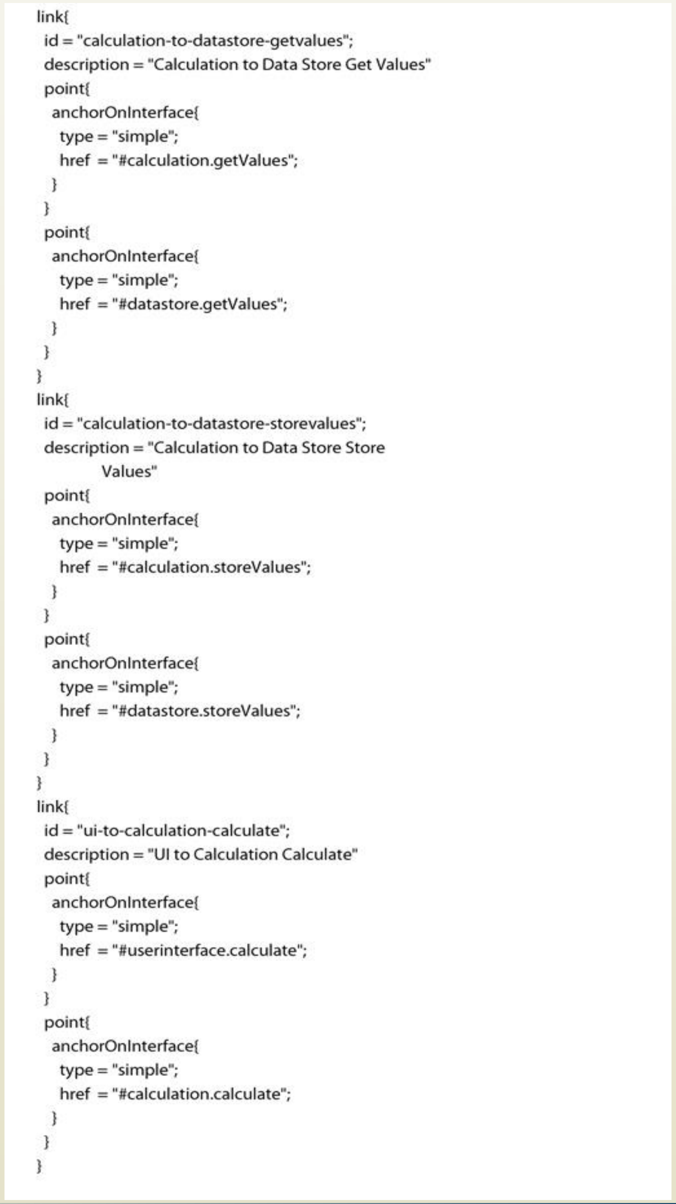
- The visual modeling of the Lunar Lander in xADL using the graphical editor Archipelago is shown in the previous slide
- As with Acme, the key notational contribution of xADL lies in its extensibility, which goes beyond adding properties to the core constructs
- xADL has no fundamental separation between core concepts and extensions
    - It allows the addition of completely new syntactic elements as well as structural extensions to existing elements
- The textual form of LL in xADl might look like the one in the following slides

LL in xADL (cont'd)



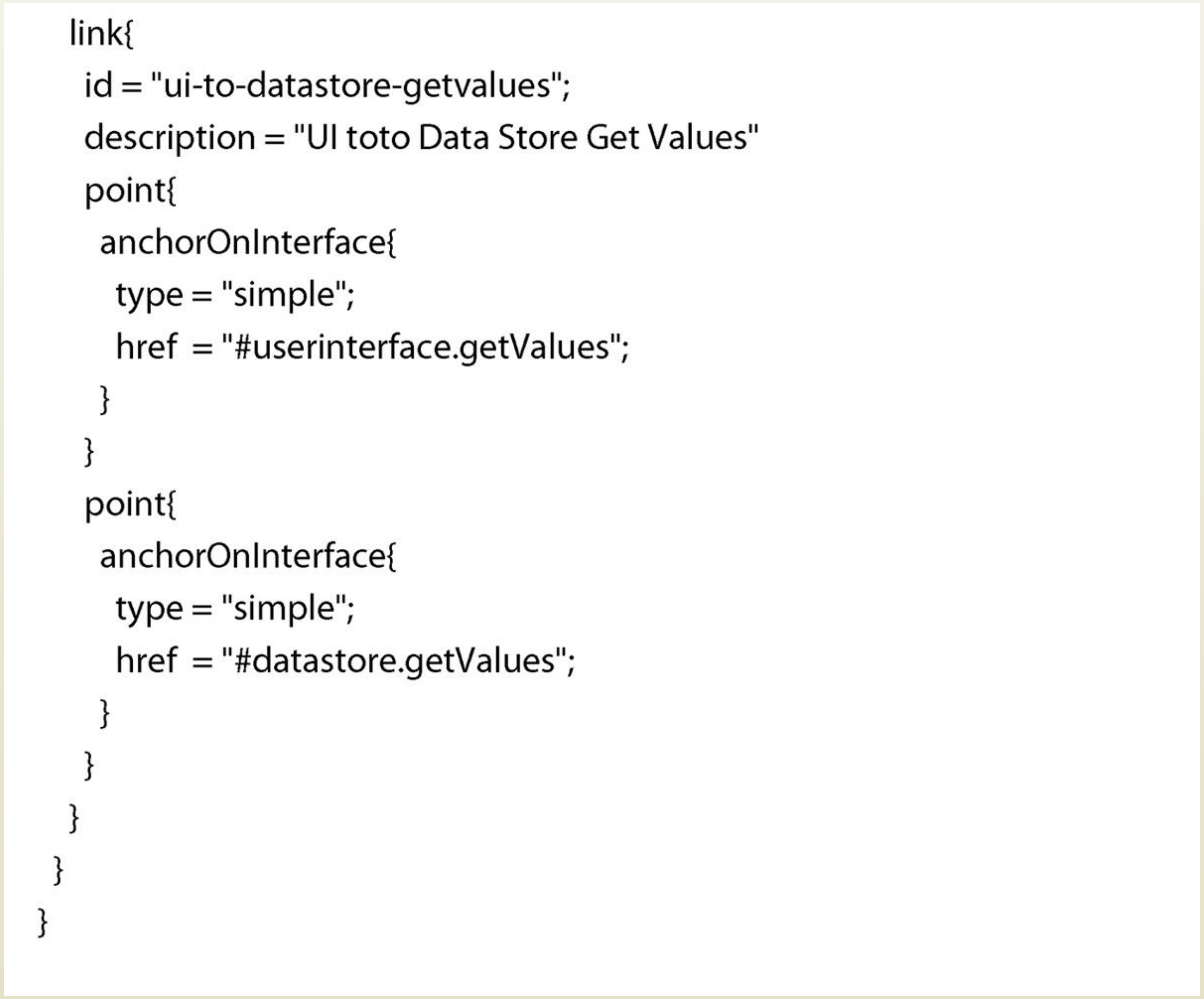
```
xArch{
 archStructure{
  id = "lunarlander";
  description = "Lunar Lander";
  component{
   id = "datastore";
   description = "Data Store";
   interface{
    id = "datastore.getValues";
    description = "Data Store Get Values Interface";
    direction = "in";
   }
   interface{
    id = "datastore.storeValues";
    description = "Data Store Store Values Interface";
    direction = "in";
   }
  }
  component{
   id = "calculation";
   description = "Calculation";
   interface{
    id = "calculation.getValues";
    description = "Calculation Get Values Interface";
    direction = "out";
   }
   interface{
    id = "calculation.storeValues";
    description = "Calculation Store Values Interface";
    direction = "out";
   }
   interface{
    id = "calculation.calculate";
    description = "Calculation Calculate Interface";
    direction = "in";
   }
  }
  component{
   id = "userinterface";
   description = "UserInterface";
   interface{
    id = "userinterface.getValues";
    description = "User Interface Get Values
               Interface";
    direction = "out";
   }
   interface{
    id = "userinterface.calculate";
    description = "User Interface Calculate Interface";
    direction = "out";
   }
  }
```

LL in xADL (cont'd)



```
link{
 id = "calculation-to-datastore-getvalues";
 description = "Calculation to Data Store Get Values"
 point{
  anchorOnInterface{
   type = "simple";
   href  = "#calculation.getValues";
  }
 }
 point{
  anchorOnInterface{
   type = "simple";
   href  = "#datastore.getValues";
  }
 }
}
link{
 id = "calculation-to-datastore-storevalues";
 description = "Calculation to Data Store Store
          Values"
 point{
  anchorOnInterface{
   type = "simple";
   href  = "#calculation.storeValues";
  }
 }
 point{
  anchorOnInterface{
   type = "simple";
   href  = "#datastore.storeValues";
  }
 }
}
link{
 id = "ui-to-calculation-calculate";
 description = "UI to Calculation Calculate"
 point{
  anchorOnInterface{
   type = "simple";
   href  = "#userinterface.calculate";
  }
 }
 point{
  anchorOnInterface{
   type = "simple";
   href  = "#calculation.calculate";
  }
 }
}
```

LL in xADL (cont'd)



```
  link{
   id = "ui-to-datastore-getvalues";
   description = "UI toto Data Store Get Values"
   point{
    anchorOnInterface{
     type = "simple";
     href  = "#userinterface.getValues";
    }
   }
   point{
    anchorOnInterface{
     type = "simple";
     href  = "#datastore.getValues";
    }
   }
  }
 }
}
```

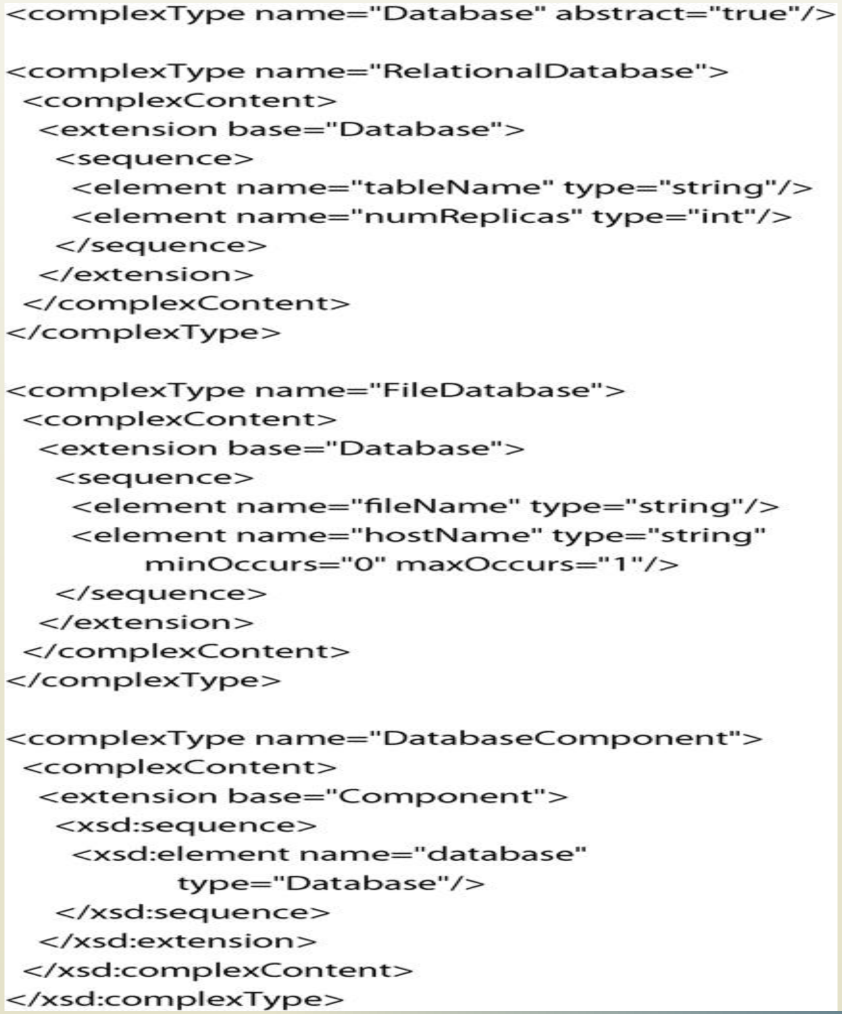
Demonstrating the Extension of a Schema in xADL

```
<complexType name="Component">
 <sequence>
  <element name="description"
       type="Description"/>
  <element name="interface" type="Interface"
       minOccurs="0" maxOccurs="unbounded"/>
  <element name="type"
       type="XMLLink"
       minOccurs="0" maxOccurs="1"/>
 </sequence>
 <attribute name="id" type="Identifier"/>
</complexType>
```

Demonstrating the extension of a schema in xADL(cont’d)

- The specification in the previous slide says that a component has the following:
  - One identifier (a string attribute)
  - One description (a string element)
  - Zero or more interfaces
  - An optional link to its type
- By adding another schema, we can create an extended form of a component that can be used to capture more information about where the component will store data
- This is shown on the next slide

Demonstrating the Extension of a Schema in xADL (cont'd)



```
<complexType name="Database" abstract="true"/>

<complexType name="RelationalDatabase">
 <complexContent>
  <extension base="Database">
   <sequence>
    <element name="tableName" type="string"/>
    <element name="numReplicas" type="int"/>
   </sequence>
  </extension>
 </complexContent>
</complexType>

<complexType name="FileDatabase">
 <complexContent>
  <extension base="Database">
   <sequence>
    <element name="fileName" type="string"/>
    <element name="hostName" type="string"
         minOccurs="0" maxOccurs="1"/>
   </sequence>
  </extension>
 </complexContent>
</complexType>

<complexType name="DatabaseComponent">
 <complexContent>
  <extension base="Component">
   <xsd:sequence>
    <xsd:element name="database"
           type="Database"/>
   </xsd:sequence>
  </xsd:extension>
 </xsd:complexContent>
</xsd:complexType>
```

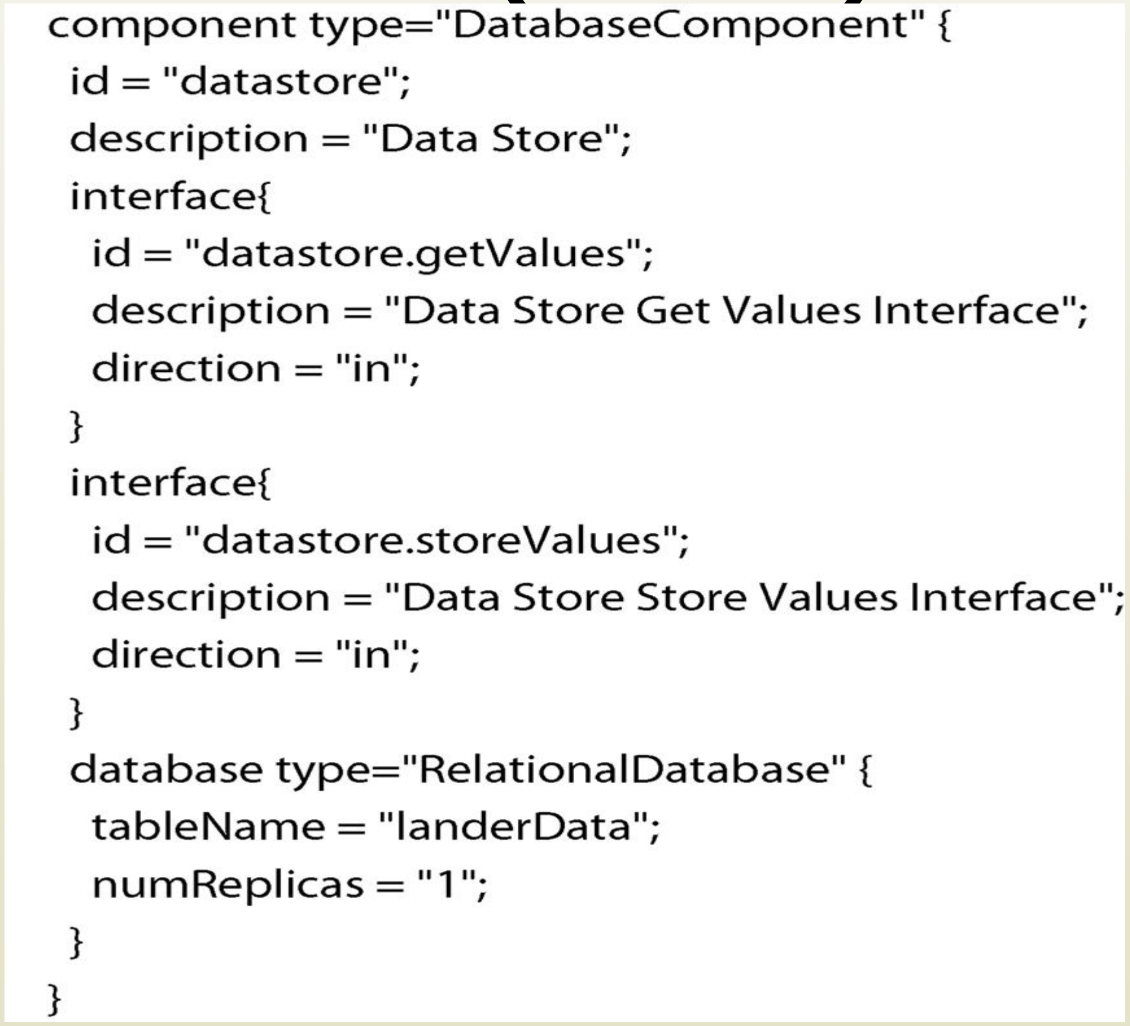
Demonstrating the Extension of a Schema in xADL(cont'd)

- This extension adds several new capabilities and defines:
  - A new ADT called `Database`, indicating we are going to define many different subtypes of `Database` that will be substitutable whenever a `Database` is needed
  - Two concrete subtypes of `Database`
    - `RelationalDatabase` with a table name and a number of replicas
    - `FileDatabase` with a file name and an optional host name on which the file resides
  - An extension to the plain xADL `Component` datatype
    - A `DatabaseComponent` has everything that a component does, plus a `Database` element
    - Because both `RelationalDatabase` and `FileDatabase` are `Databases`, either one can be used here

Demonstrating the Extension of a Schema in xADL(cont'd)

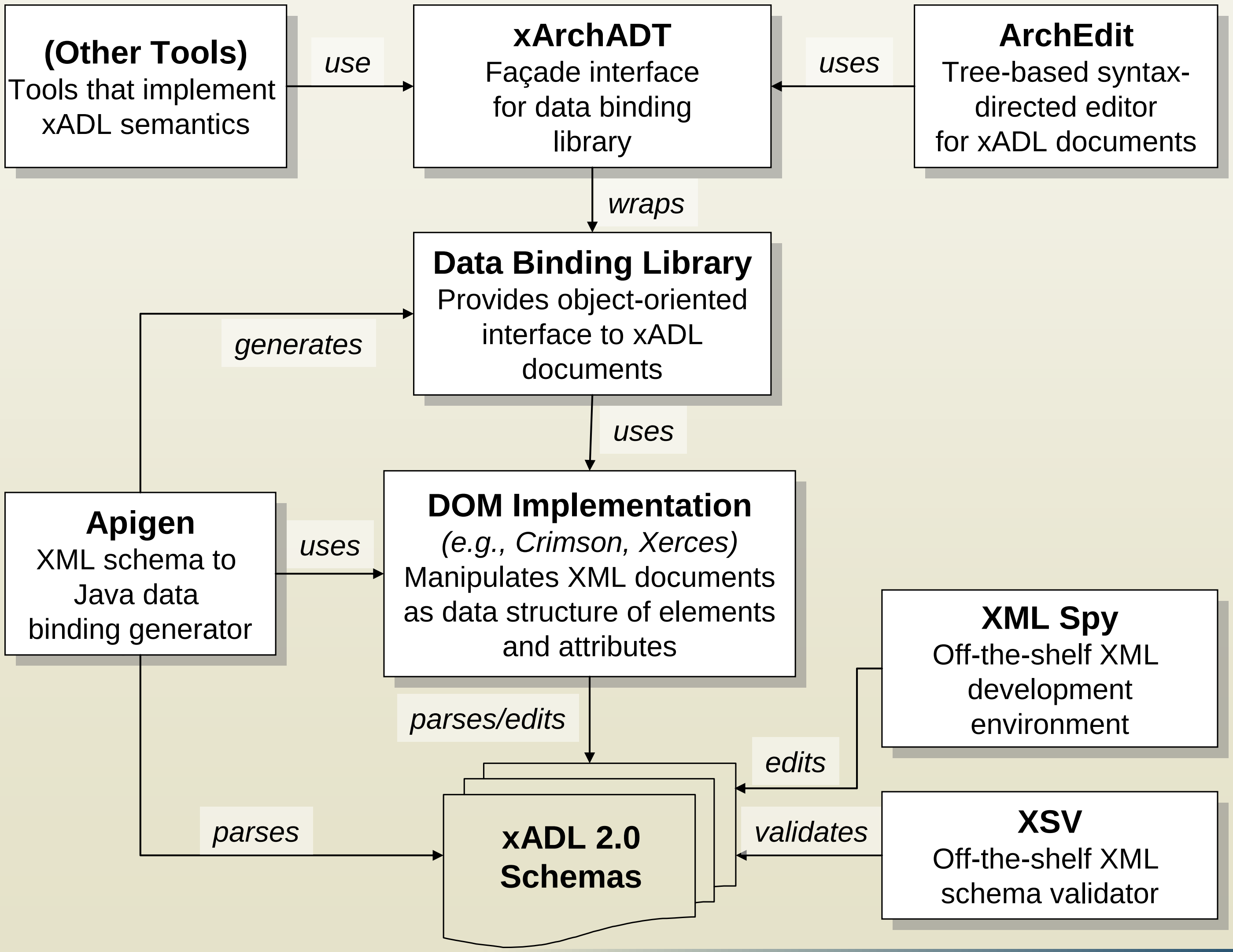
- We can now extend the original xADL description of `Data Store` in Lunar Lander
- When the new `DatabaseComponent` schema is ready, the Apigen tool can generate a new data binding library that supports the new constructs
- Sharing of schemas are possible and encouraged
  - The `DatabaseComponent` schema can be reused in different projects
- The extended Lunar Lander `Data Store` component is shown on the next slide

Demonstrating the Extension of a Schema in xADL (cont’d)

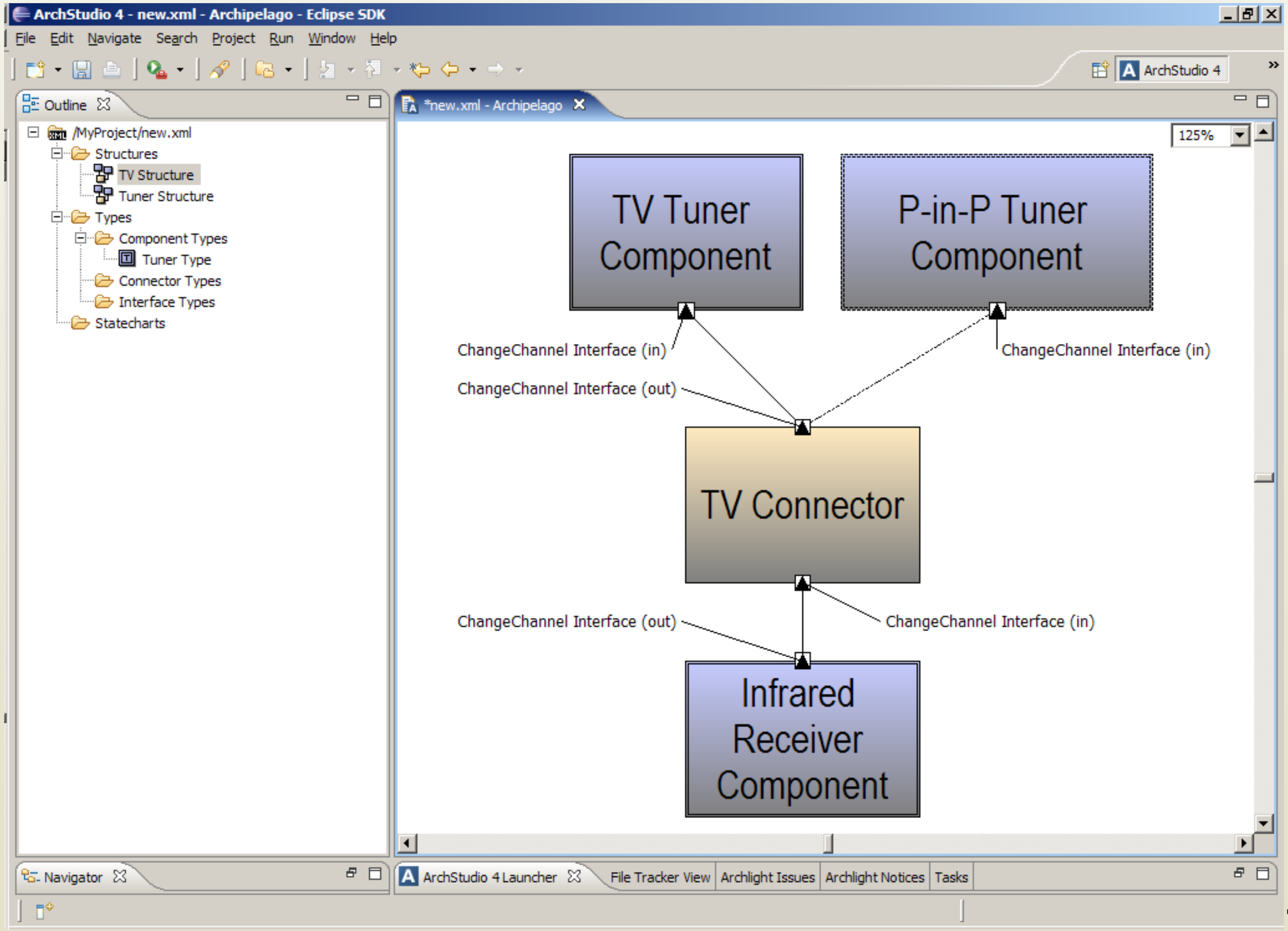


```
component type="DatabaseComponent" {
  id = "datastore";
  description = "Data Store";
  interface{
    id = "datastore.getValues";
    description = "Data Store Get Values Interface";
    direction = "in";
  }
  interface{
    id = "datastore.storeValues";
    description = "Data Store Store Values Interface";
    direction = "in";
  }
  database type="RelationalDatabase" {
    tableName = "landerData";
    numReplicas = "1";
  }
}
```

xADL Tools


(Other Tools)
Tools that implement xADL semantics
use
xArchADT
Façade interface for data binding library
uses
ArchEdit
Tree-based syntax-directed editor for xADL documents
wraps
Data Binding Library
Provides object-oriented interface to xADL documents
generates
uses
Apigen
XML schema to Java data binding generator
uses
DOM Implementation
(e.g., Crimson, Xerces)
Manipulates XML documents as data structure of elements and attributes
XML Spy
Off-the-shelf XML development environment
parses/edits
edits
parses
xADL 2.0 Schemas
validates
XSV
Off-the-shelf XML schema validator

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

When Systems Become Too Complex to Model

- The LL architecture is a relatively simple one with a finite number of well-known components that run on a single host
- However, gigantic and diverse applications cannot be modeled with some of the techniques we saw
  - ‘Agile’ systems that are not explicitly designed above the level of code modules
  - Extremely large, complex, or dynamic systems (e.g., the Web)
- The approach to be taken here is to abstract away aspects of the complexity to reach a point where modeling is feasible but also still useful

When Systems Become Too Complex to Model (cont'd)

- Strategies to consider in this case include the following:
  - Model limited aspects of the architecture
    - E.g., you cannot model the Web but you can model specific interaction patterns
  - Model an instance
    - E.g., you cannot model the Web but you can model only the portion of the system that is relevant
  - Exploit regularity
    - Often extremely large systems have low heterogeneity, where large portions of the system look almost exactly like other portions of the same system; these portions can be modeled once (as in the case of the example in Darwin)

When Systems Become Too Complex to Model (cont'd)

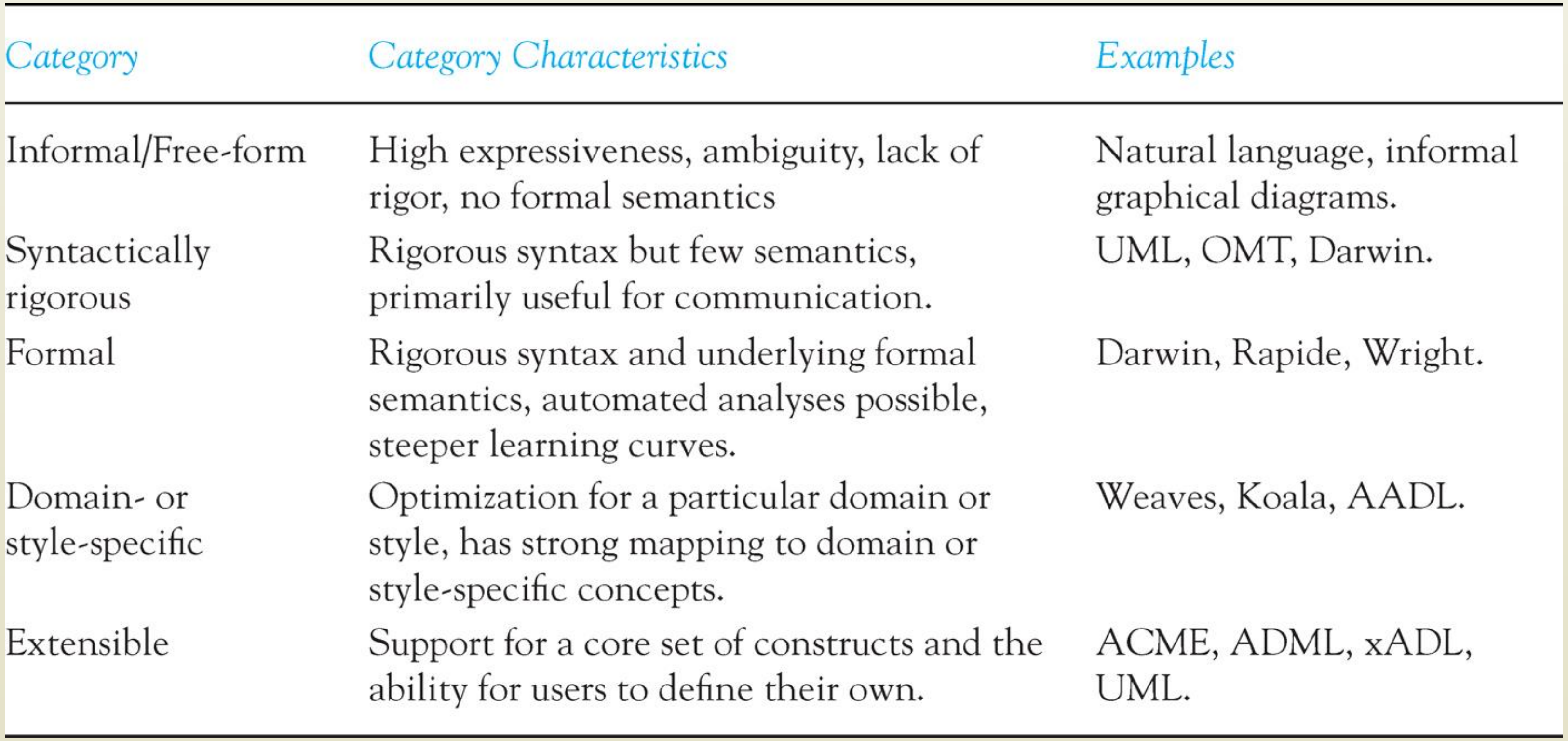
- Strategies to consider in this case include the following (cont'd)
  - Model the style
    - The Web is based on the REST architecture; instead of modeling the Web as an application, consider modeling the REST style instead
  - Model the protocol
    - The Web is, in large measure, characterized by adherence to the HTTP protocol
    - Some notations we examined in this chapter (such as Wright) can be used to to model protocol details

Summary

- We have introduced architecture modeling and identified many of the issues that occur in the modeling process
- We have also introduced and provided examples of many architecture description notations
- The preceding overview optimized for breadth rather than depth
  - Semantics and capabilities of many of these notations are quite deep and subtle
  - Some even have entire books written about them
  - You are encouraged to investigate individual notations more deeply

Summary (cont'd)

- No single notation–even an extensible notation–is sufficient to capture all the aspects of an architecture
- The following table groups the various approaches



Category	Category Characteristics	Examples
Informal/Free-form	High expressiveness, ambiguity, lack of rigor, no formal semantics	Natural language, informal graphical diagrams.
Syntactically rigorous	Rigorous syntax but few semantics, primarily useful for communication.	UML, OMT, Darwin.
Formal	Rigorous syntax and underlying formal semantics, automated analyses possible, steeper learning curves.	Darwin, Rapide, Wright.
Domain- or style-specific	Optimization for a particular domain or style, has strong mapping to domain or style-specific concepts.	Weaves, Koala, AADL.
Extensible	Support for a core set of constructs and the ability for users to define their own.	ACME, ADML, xADL, UML.