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

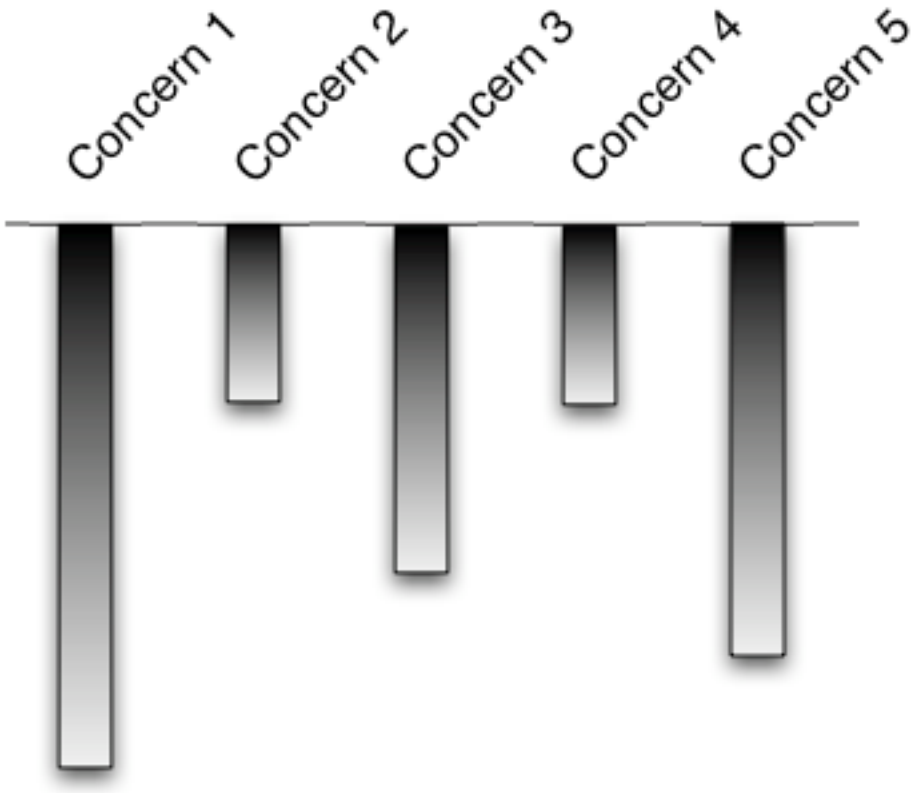
What is Architectural Modeling?

- 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

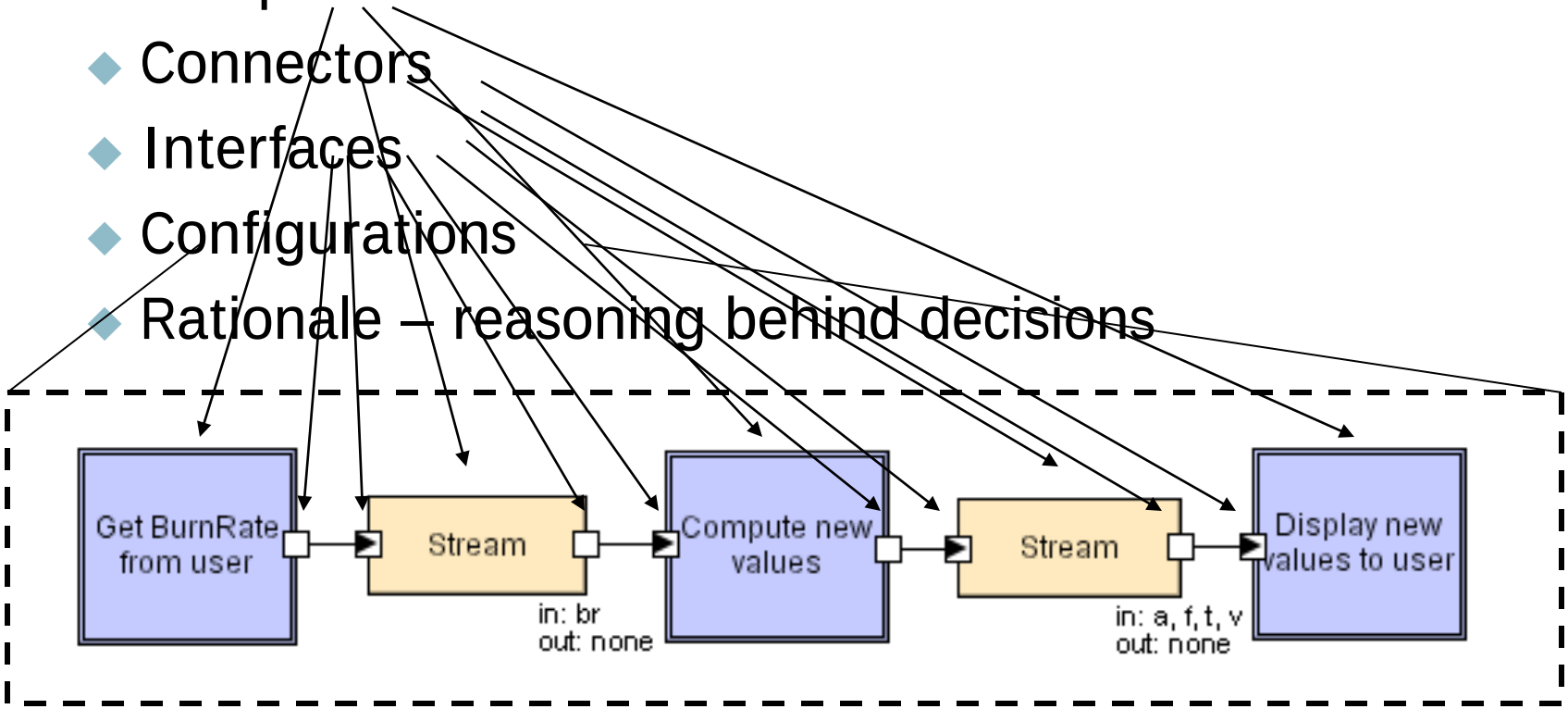


- Stakeholders identify aspects of the system they are concerned about
- Stakeholders decide the relative importance of these concerns
- Modeling depth should roughly mirror the relative importance of concerns

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

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)

Accurate but imprecise: ambiguous or shallow assertions

Inaccurate but precise: detailed assertions that are wrong

(c)

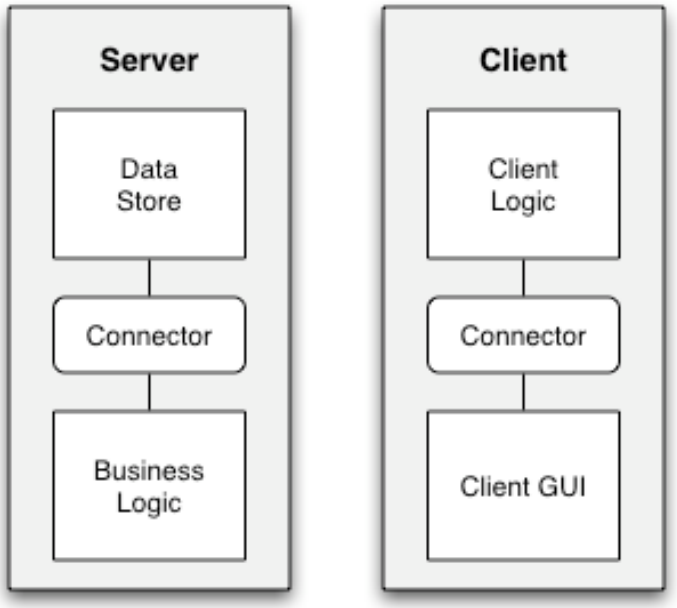
(d)

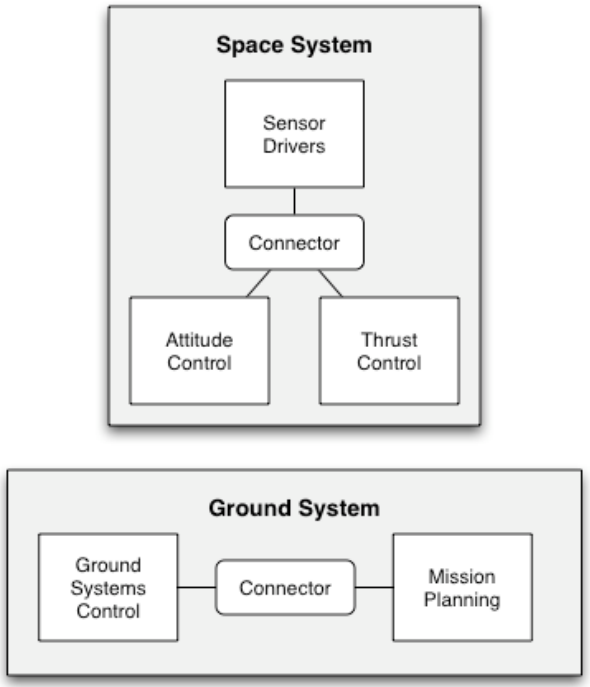
Accurate and precise: detailed assertions that are correct

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.

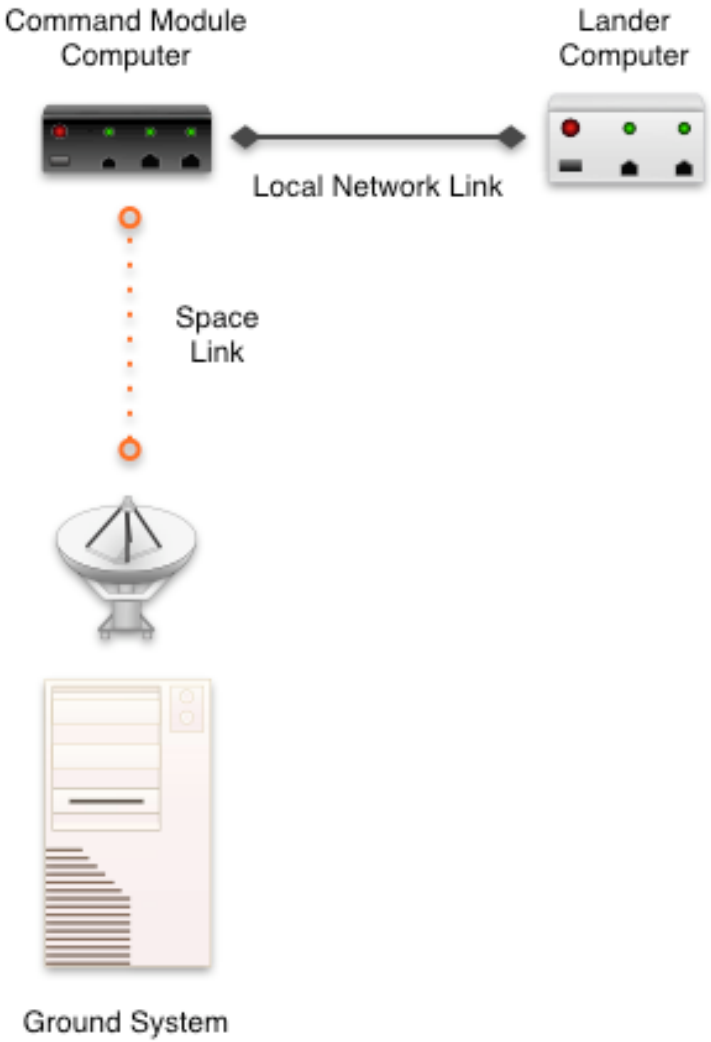
Commonly-Used Viewpoints (cont'd)

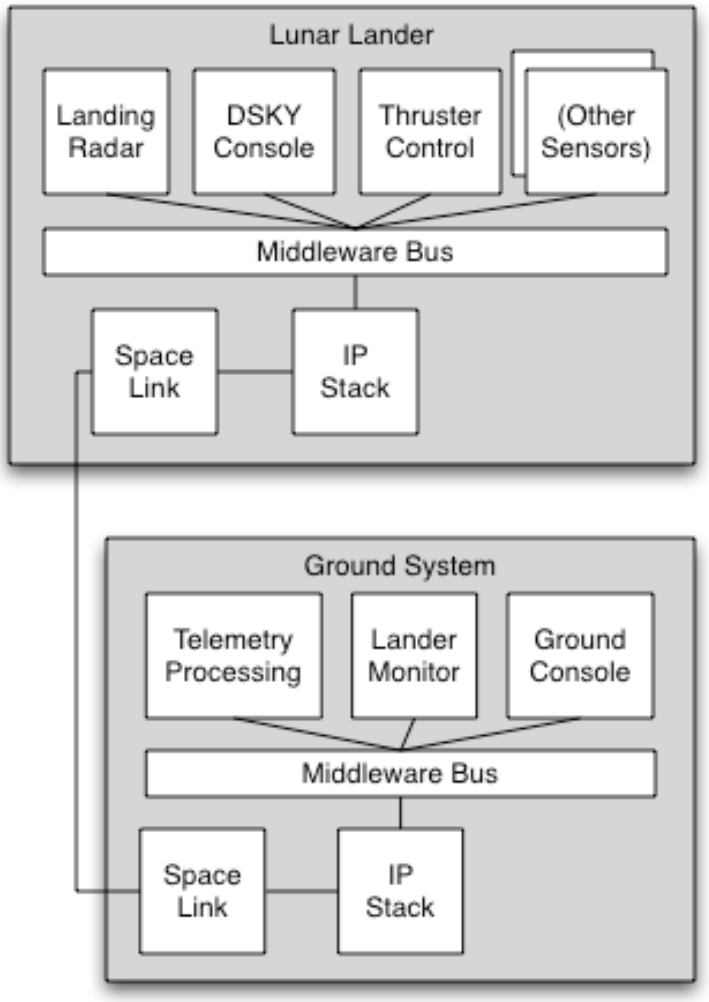
- Concurrency Viewpoints
  - Capture how concurrency and threading will be managed in a system.
- Behavioral Viewpoints
  - Capture the expected behavior of (parts of) a system.

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 views?

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

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

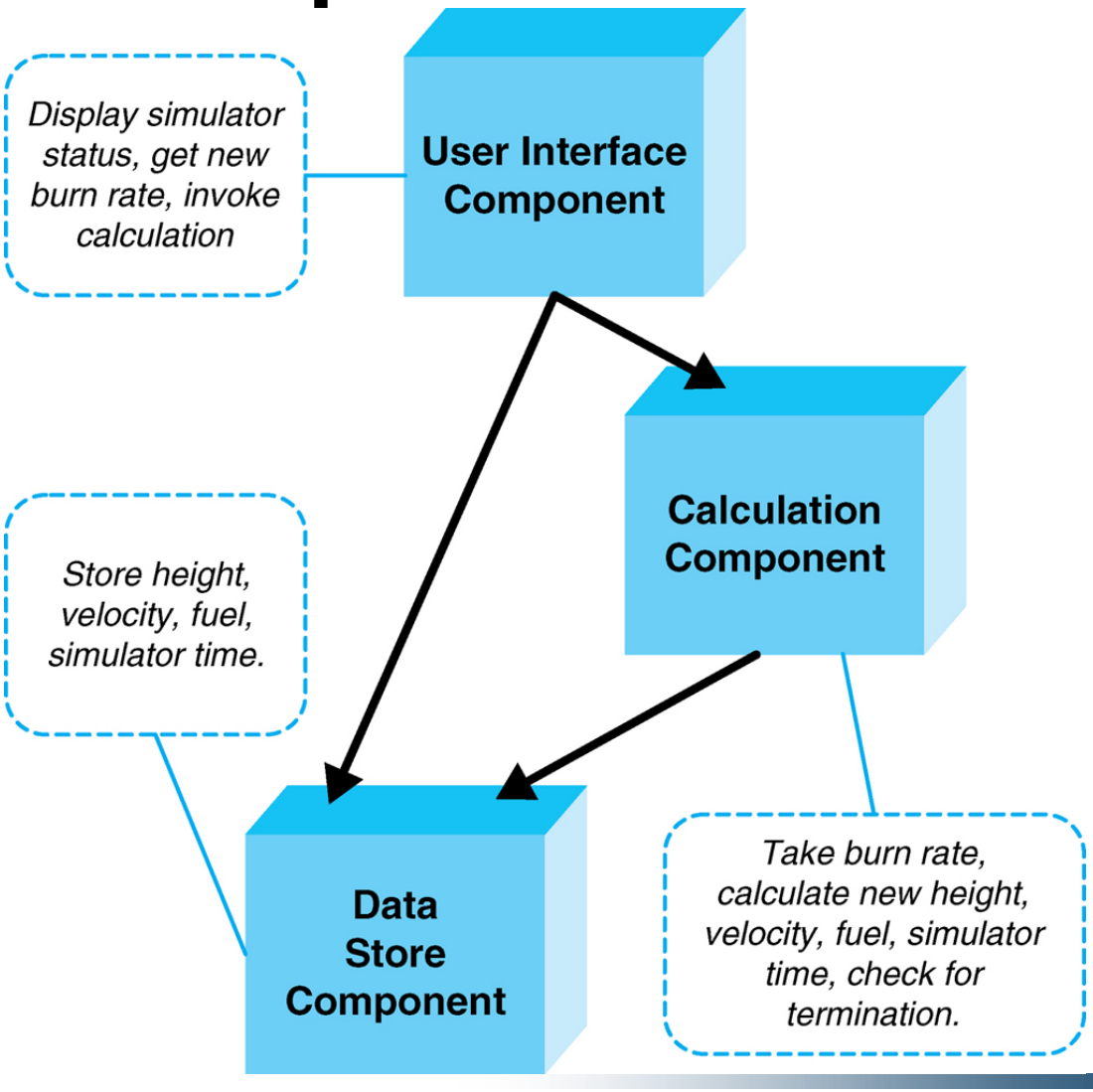
Natural Language Evaluation

- 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)
- View 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

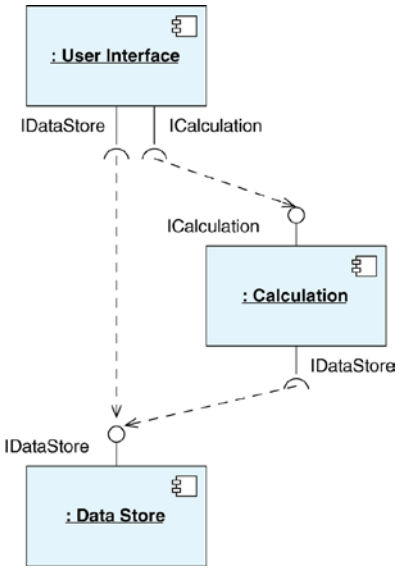
Informal Graphical Evaluation

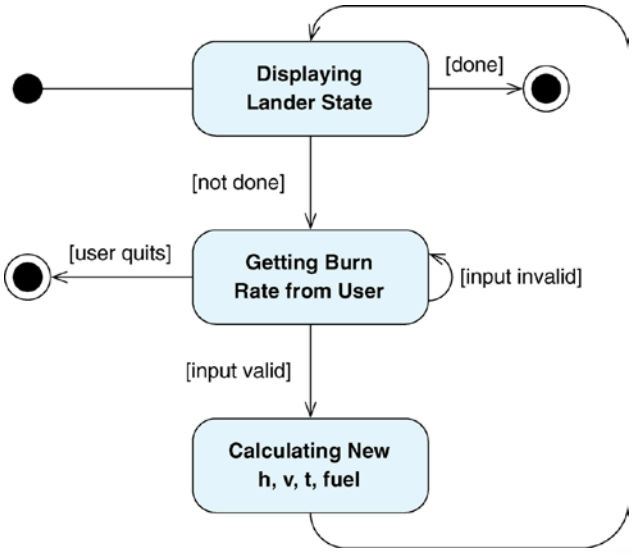
- Scope and purpose
  - Arbitrary diagrams consisting of symbols and text
- Basic elements
  - Geometric shapes, splines, clip-art, text segments
- Style
  - In general, no support
- Static & Dynamic Aspects
  - Any aspect can be modeled, but no semantics behind models
- Dynamic Models
  - Rare, although 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)
- View consistency
  - Manual reviews and inspection

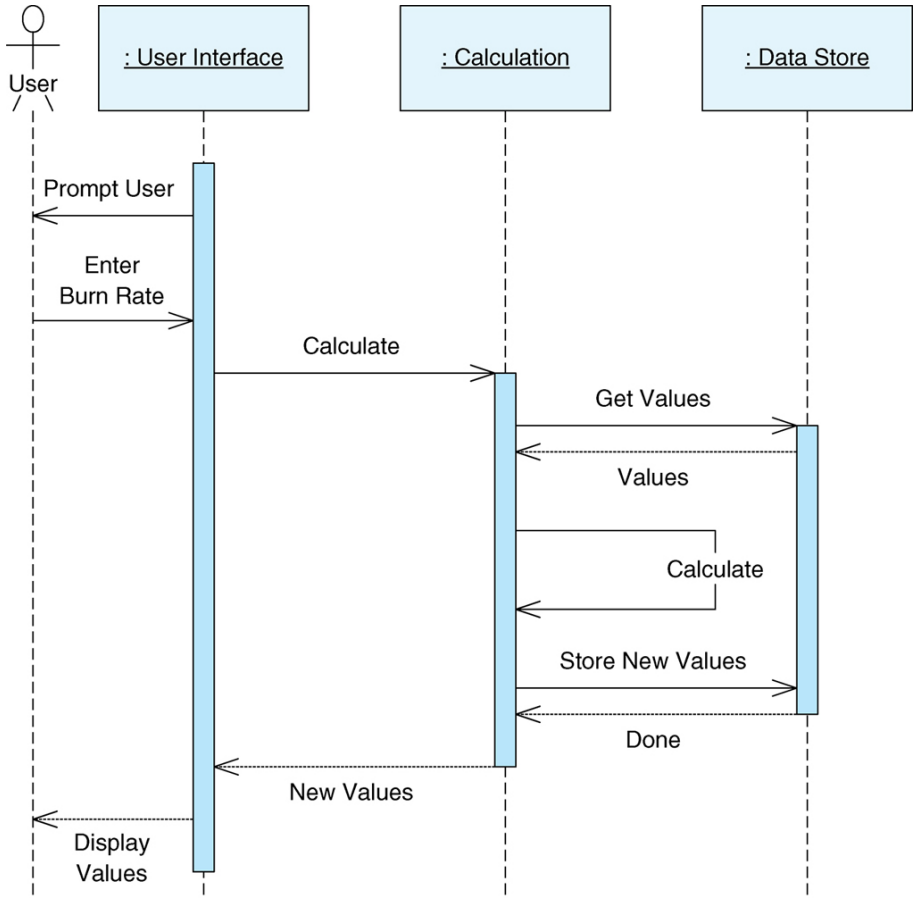
UML – the Unified Modeling Language

- 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
- View consistency
  - Constraint checking, ersatz tool methods, manual