Software Architecture and Design Thinking 116U01C701

Module 3

Modelling, Analysis

3.1 Modeling Concepts

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?

Architectural Modelling

Architectures are characterized 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 (presenting abstract idea)*** 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.

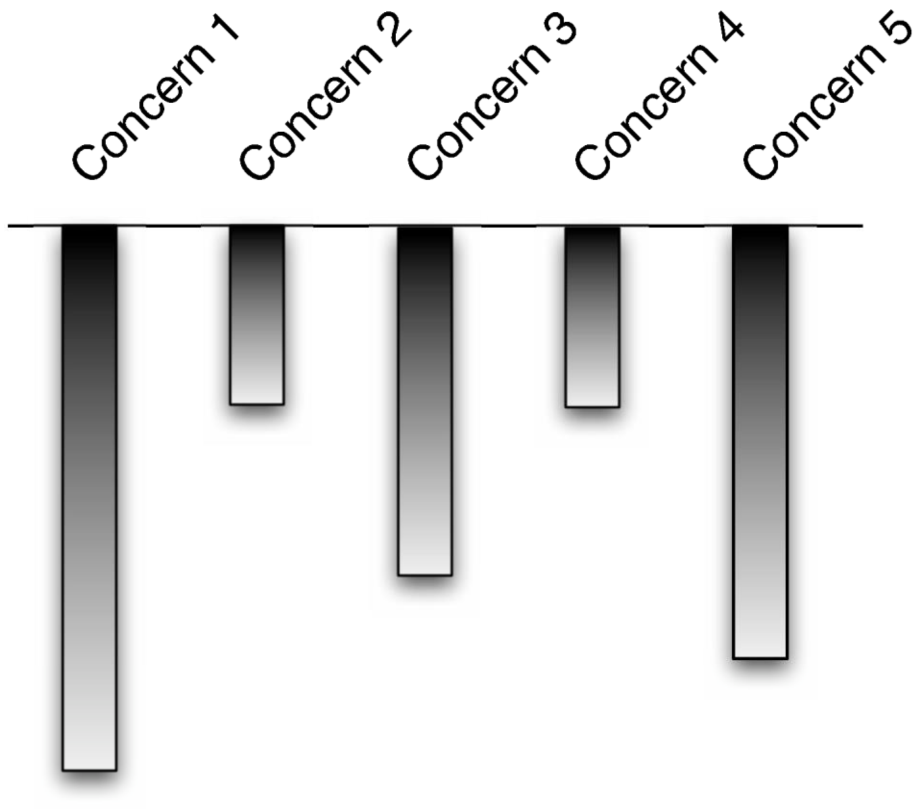
Choose What to Model

Architects and other stakeholders must make critical decisions:

- What architectural decisions and concepts should be modeled
- At what level of detail
- 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

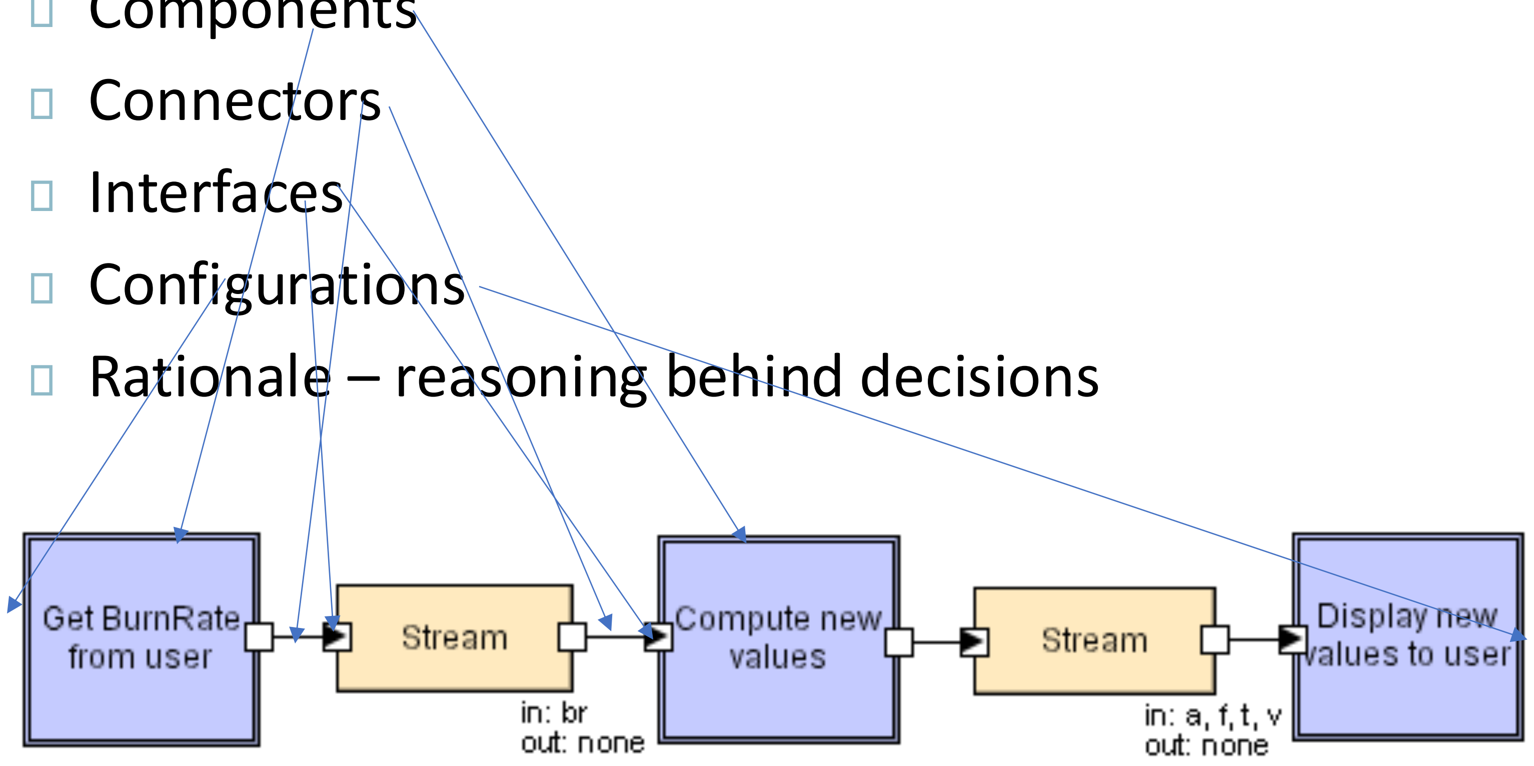
Stakeholder-Driven Modelling



- 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

What to Model

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



What do we model

- 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

- 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

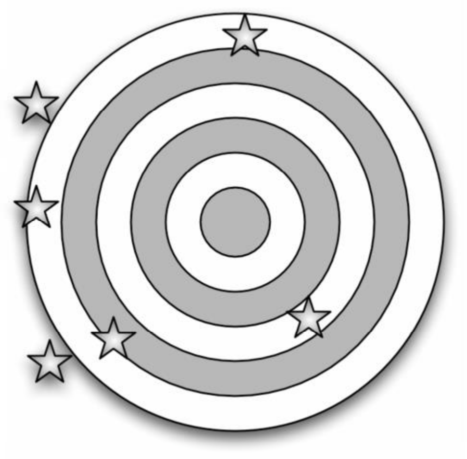
- Functional and non-functional aspects of a system
  - Functional
    - "The system prints medical records"
  - Non-functional
    - "The system prints medical records *quickly* and *confidentially*."
- Architectural models tend to be functional, but like rationale it is often important to capture non-functional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

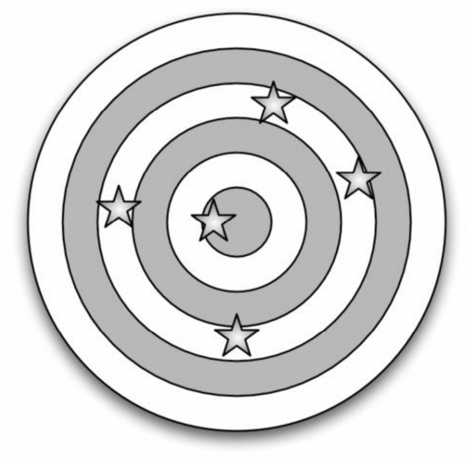
Important Characteristics of Models

- 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 v/s Precision

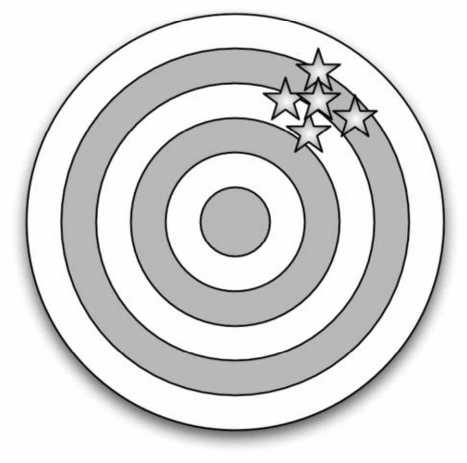
Inaccurate and imprecise: incoherent or contradictory assertions

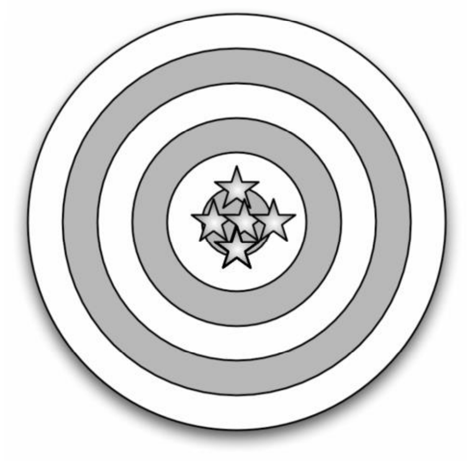

(a)


(b)

Accurate but imprecise: ambiguous or shallow assertions

Inaccurate but precise: detailed assertions that are wrong


(c)

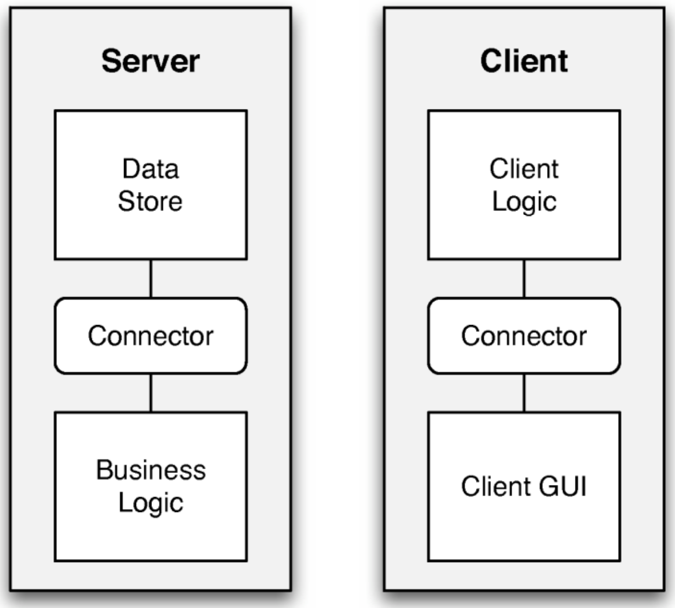

(d)

Accurate and precise: detailed assertions that are correct

3.2 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 View-point



Deployment view of a 3-tier application

Instance of a view is view-point

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

- **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

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

Analysis Goals

- Goals may include early estimation of system size, complexity, cost
- Adherence of architectural model to design guidelines and constraints
- Satisfaction of system functional and non functional requirements
- Assessment of the implemented system 's correctness with respect to it's documented architecture
- Evaluation of opportunities for reusing existing functionality when implementing parts of the modelled system

Architectural Analysis Goals

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

Architectural Analysis Goals – Completeness

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

Architectural Analysis Goals – Consistency

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

Name Consistency

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

Interface Consistency

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

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

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

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

Behavioral Consistency

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

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

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

QueueClient

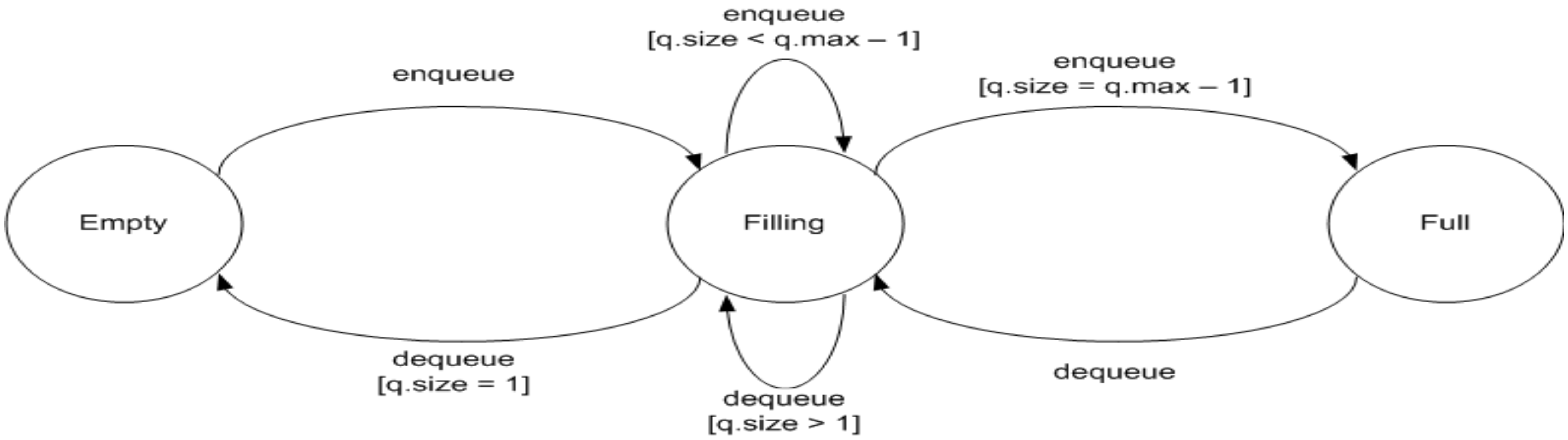
```
precondition  q.size > 0;
postcondition ~q.size = q.size;
```

QueueServer

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

Interaction Consistency

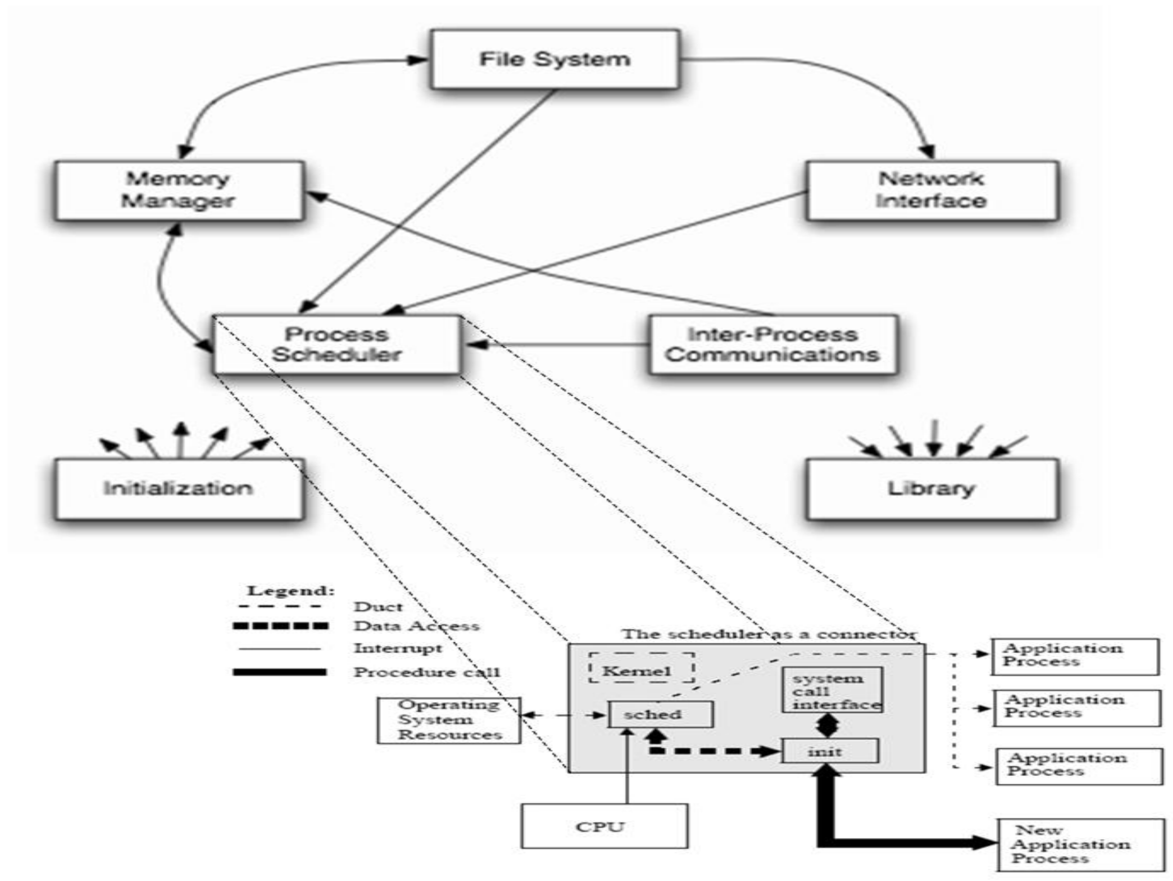
- Names, interfaces, and behaviors of interacting components may match, yet they may still be unable to interact properly
- Example: QueueClient and QueueServer components



Refinement Consistency

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

Refinement Consistency Example



Compatibility

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

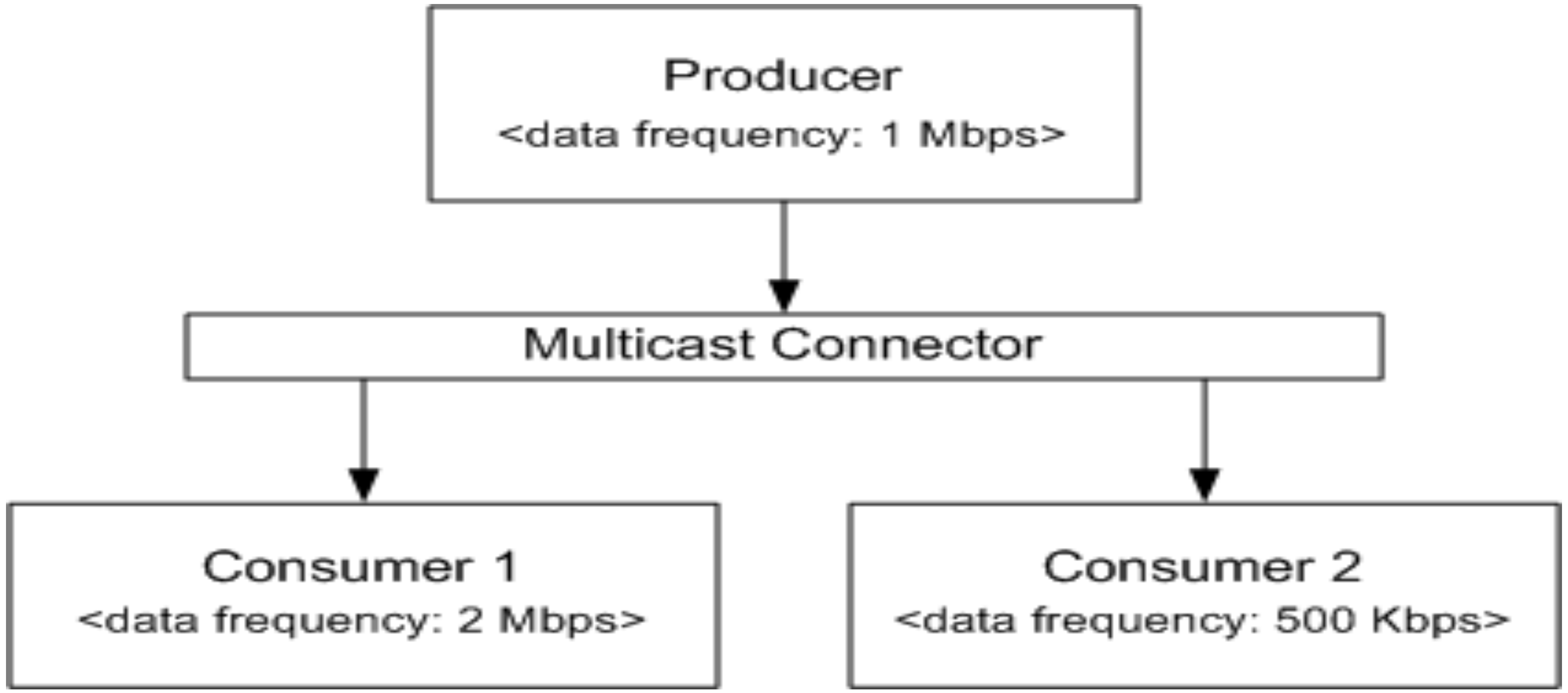
Correctness

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

Scope of Analysis

- Component- and connector-level Analysis
  - Component-application dependent
  - Connector-application independent
- Subsystem- and system-level
  - System is collection of components and connectors
  - Beware of the “honey-baked ham” syndrome
- Data exchanged in a system or subsystem
  - Data structure – types or untyped, discrete or streamed
  - Data flow – point to point or broadcast
  - Properties of data exchange-consistency, security and latency
  - Data is properly modeled implemented and exchanged
  - Web application, e-commerce and multimedia
- Architectures at different abstraction levels
- Comparison of two or more architectures
  - Processing
  - Data
  - Interaction
  - Configuration
  - Non-functional properties

Data Exchange Example



Architectural Concern Being Analyzed

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

3.3 Level of Formality

- Informal models
- Semi-formal models
- Formal models

Type of Analysis

- **Static analysis:**
  - Inferring the properties of a software system from one or more of its models ***without actually executing those models.***
  - E.g. syntactic analysis (checks only if the syntax is right, used appropriate notations, use of architectural description language, design diagram notations)
  - Can be automated by compilation or manual by inspection
- **Dynamic analysis:**
- Involves actual execution or simulation of a model
- Performed only after semantic analysis (static)
- State transition diagram
- Scenario-driven analysis
  - Can be both static and dynamic

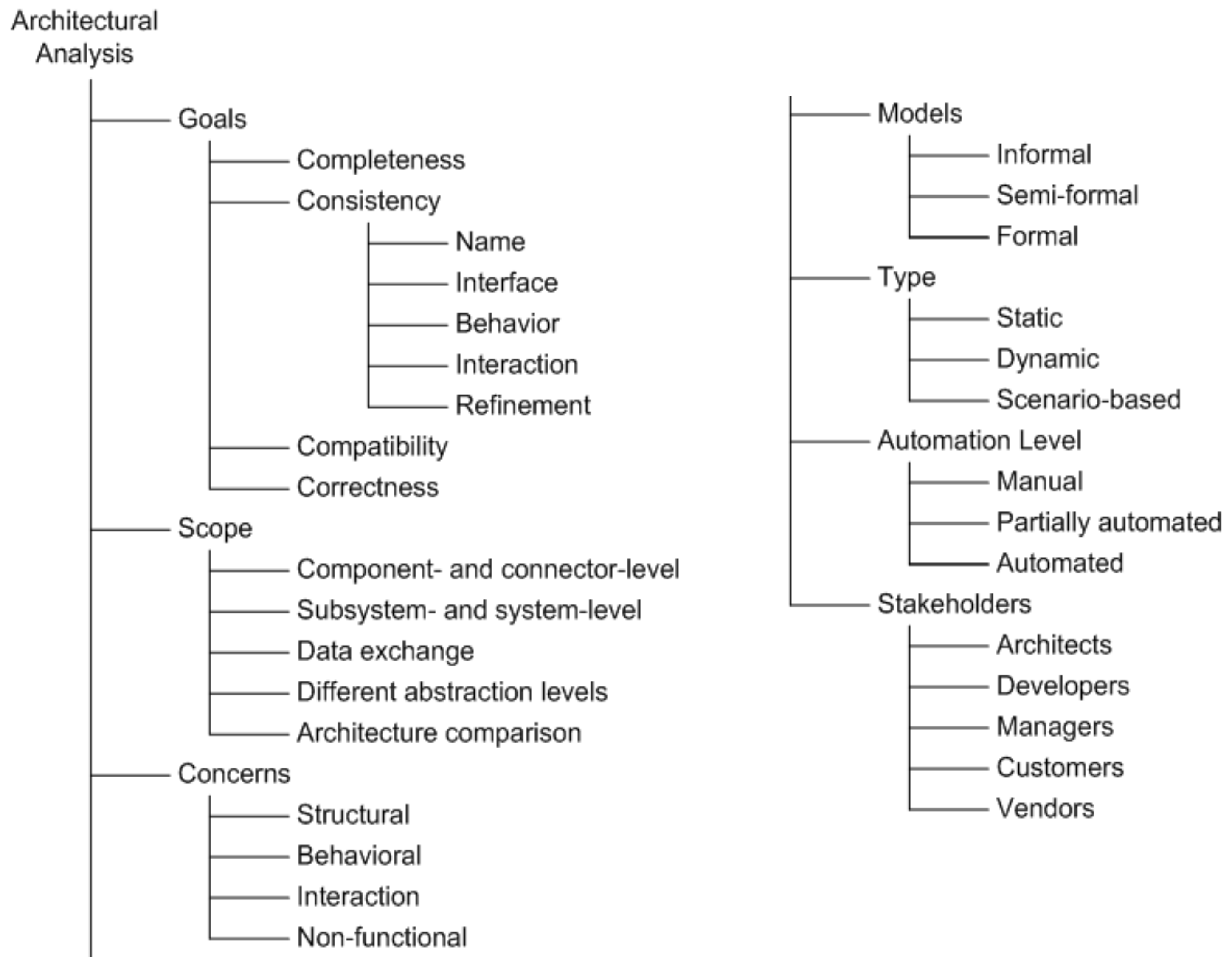
Type of Analysis

- **Scenario based Analysis**
    - **Difficult to analyze big complex system**
    - **Use case based analysis**
    - **May contain both static and Dynamic**

Level of Automation

- Manual – significant human involvement
- Partially Automated – Tools and Human
- Fully Automated - Tools

3.3 Analysis Techniques



Analysis Techniques Categories

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

Analysis Techniques Categories

- **Inspection- and review-based:**
- Architectural models studied by human stakeholders for specific properties
- The stakeholders define analysis objective
- Can fulfill any of the four Goals - Cs
- Manual techniques
  - Can be expensive
- Useful in the case of informal architectural descriptions
- Useful in establishing "soft" system properties
  - E.g., scalability or adaptability
- Able to consider multiple stakeholders' objectives and multiple architectural properties

Analysis Techniques Categories

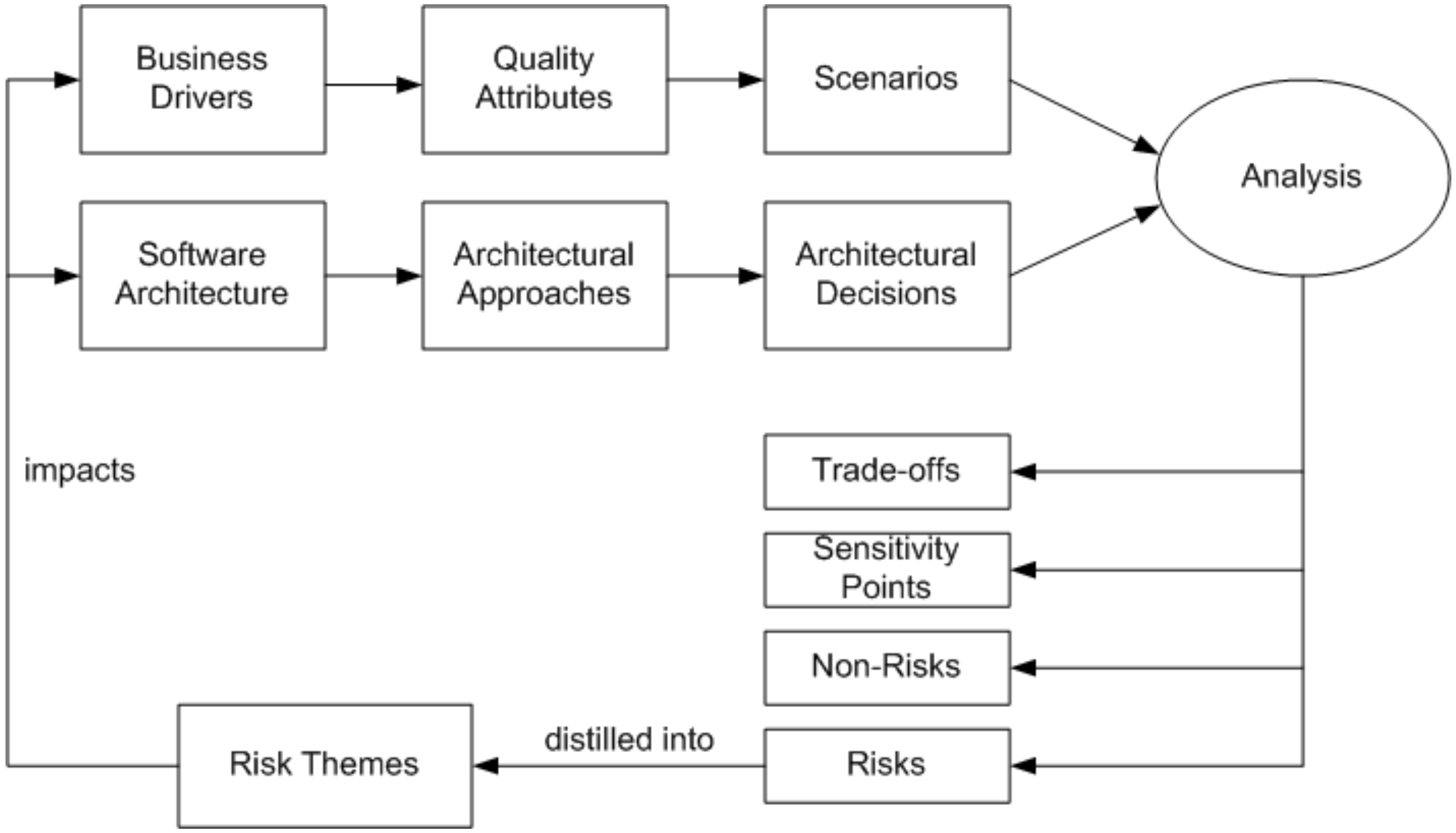
- **Inspection- and review covers:**
  - *Analysis Goals* – any
  - *Analysis Scope* – any
  - *Analysis Concern* – any, but particularly suited for non-functional properties
  - *Architectural Models* – any, but must be geared to stakeholder needs and analysis objectives
  - *Analysis Types* – mostly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

Analysis Techniques Categories

- **Architectural Trade-off Analysis Method (ATAM) :**
- Human-centric process for identifying risks early on in software design
- Focuses specifically on four quality attributes (NFPs)
    - Modifiability
    - Security
    - Performance
    - Reliability
- Reveals how well an architecture satisfies quality goals and how those goals trade-off

Analysis Techniques Categories

- **Architectural Trade-off Analysis Method (ATAM) :**



Analysis Techniques Categories

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

Analysis Techniques Categories

- **ATAM Scenarios:**

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

Scenarios are prioritized based on importance to stakeholders

Analysis Techniques Categories

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

Analysis Techniques Categories

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

Analysis Techniques Categories

- **ATAM summary:**

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

Analysis Techniques Categories

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

Analysis Techniques Categories

- **Model based Analysis:**

Analysis Goals – consistency, compatibility, internal correctness

Analysis Scope – any

Analysis Concern – structural, behavioral, interaction, and possibly non-functional properties

Architectural Models – semi-formal and formal

Analysis Types – static

Automation Level – partially and fully automated

Stakeholders – mostly architects and developers

Analysis Techniques Categories

- **Model based Analysis summery:**

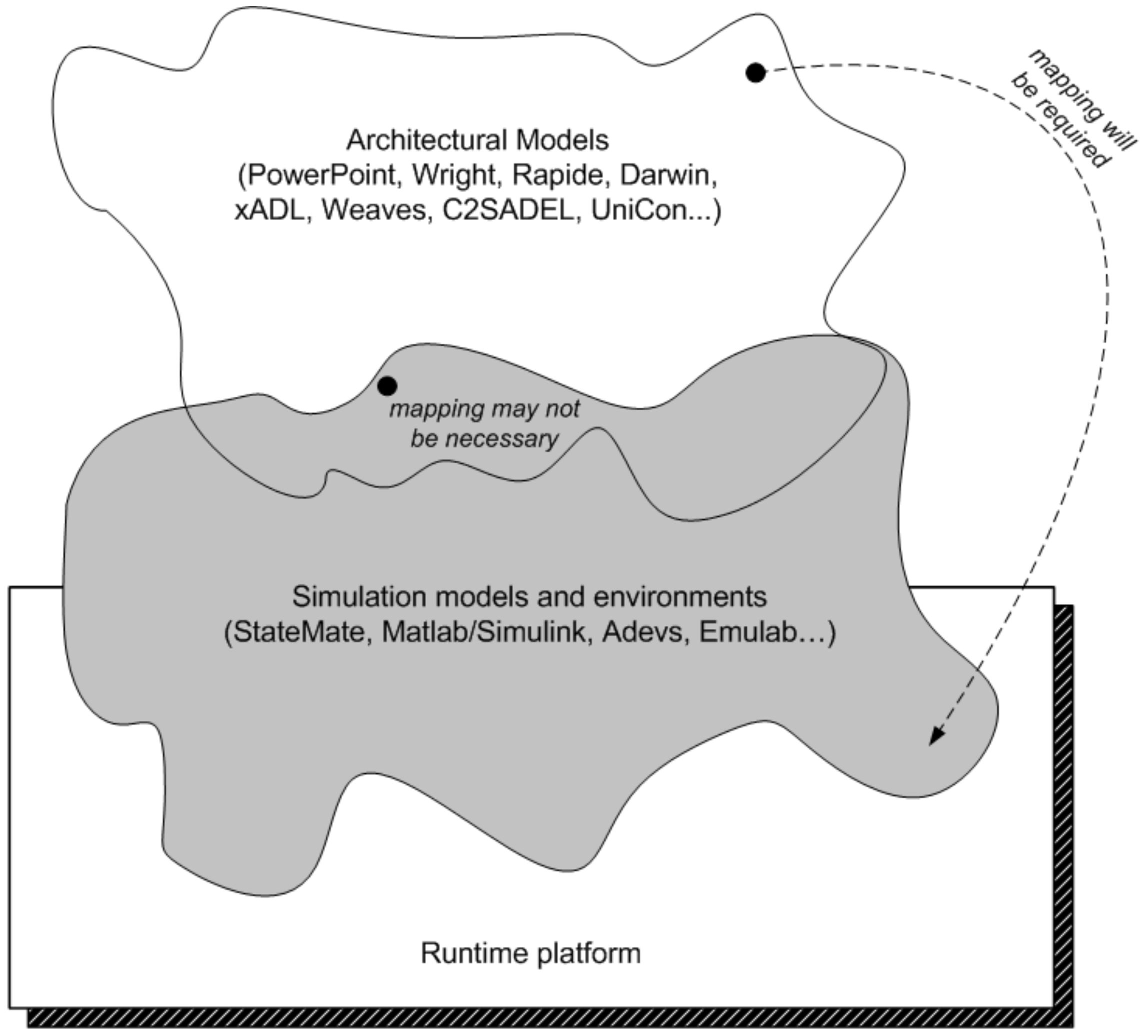
Goals	Consistency Compatibility Completeness (internal)
Scope	Component- and connector-level Subsystem- and system-level Data exchange Different abstraction levels Architecture comparison
Concern	Structural Behavioral Interaction Non-functional
Models	Semi-formal Formal
Type	Static
Automation Level	Partially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

Analysis Techniques Categories

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

Analysis Techniques Categories

- **Simulation based Analysis:**


Architectural Models
(PowerPoint, Wright, Rapide, Darwin,
xADL, Weaves, C2SADEL, UniCon...)
mapping will be required
mapping may not be necessary
Simulation models and environments
(StateMate, Matlab/Simulink, Adevs, Emulab…)
Runtime platform

Analysis Techniques Categories

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

Analysis Techniques Categories

- **Simulation based Analysis summery:**

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

Designing for Non Functional Properties

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

Designing for FPs

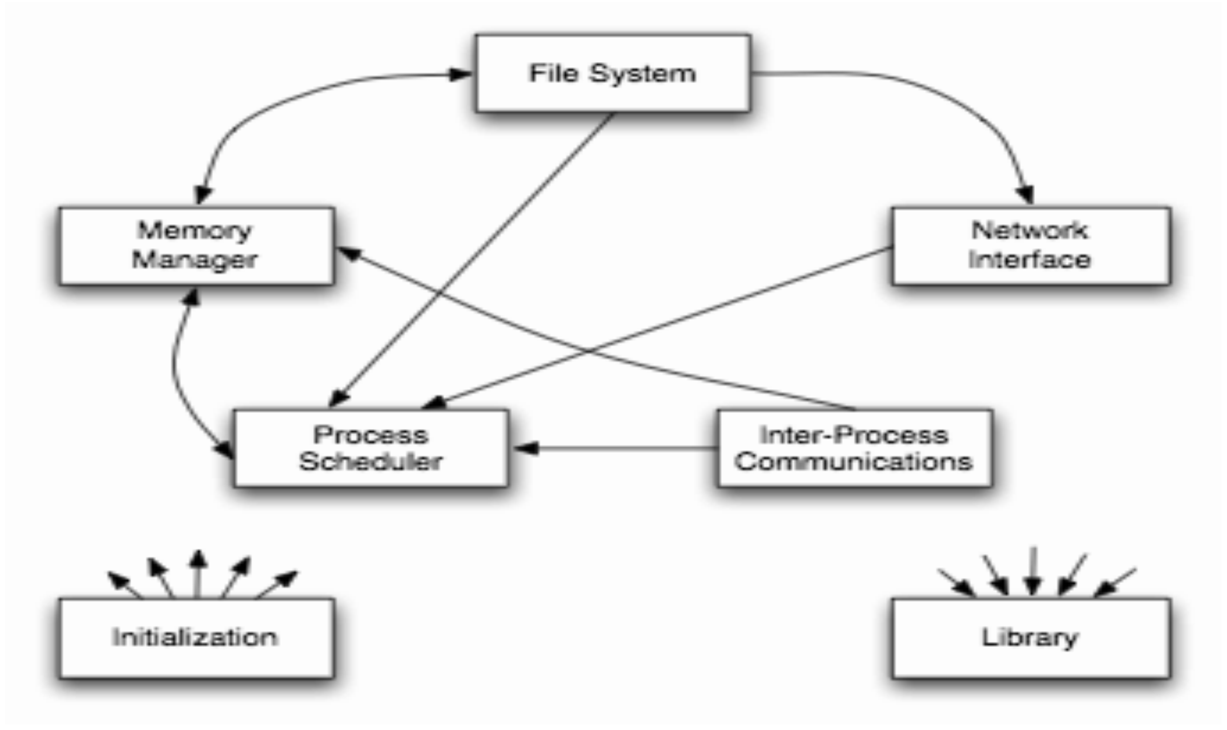
- Any engineering product is sold based on its functional properties (FPs)
  - TV set, DVD player, stereo, mobile telephone
- Providing the desired functionality is often quite challenging
  - Market demands
  - Competition
  - Strict deadlines
  - Limited budgets
- However, the system’s success will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

FPs vs. NFPs – An Example

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

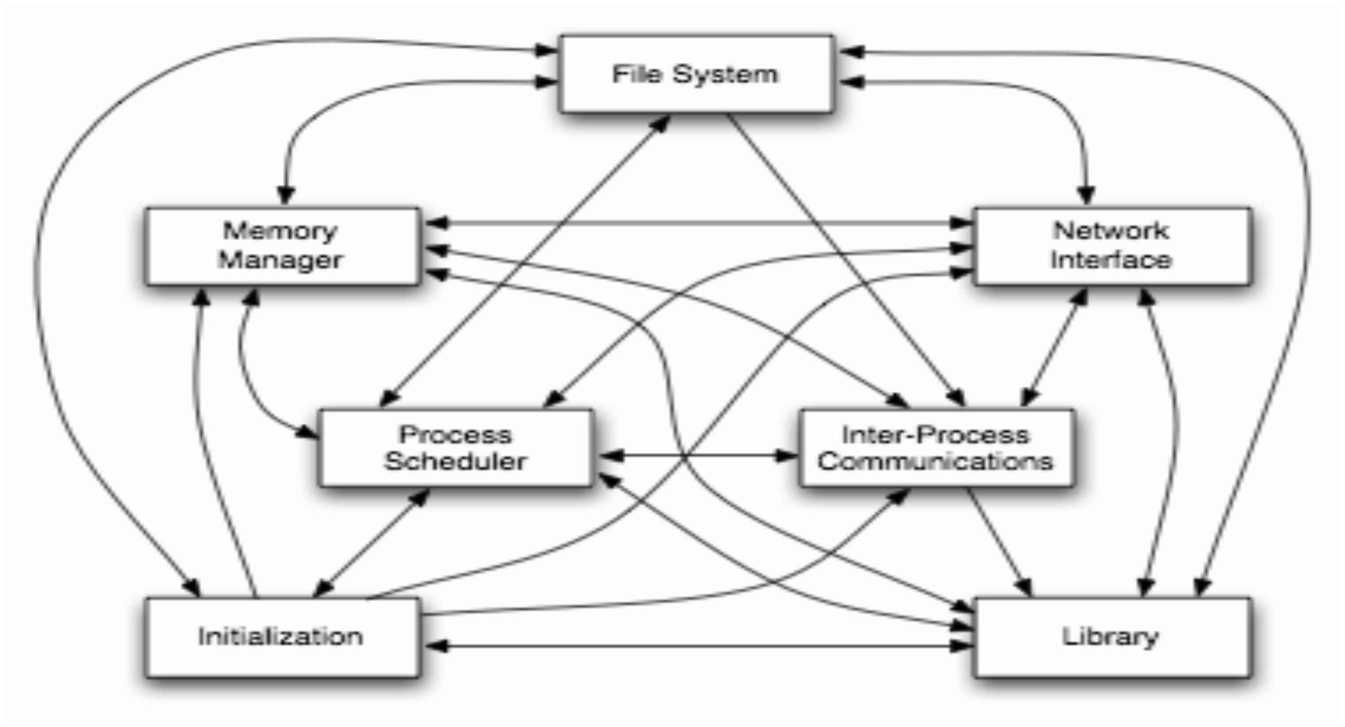
FPs vs. NFPs – Another Example

- Linux – “as-documented” architecture



FPs vs. NFPs – Another Example

- Linux – “as-implemented” architecture



Challenges of Designing for NFPs

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

Design Guidelines for Ensuring NFPs

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

Overarching Objective

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

Efficiency

- **Efficiency** is a quality that reflects a software system's ability to meet its performance requirements while minimizing its usage of the resources in its computing environment
  - Efficiency is a measure of a system's resource usage *economy*

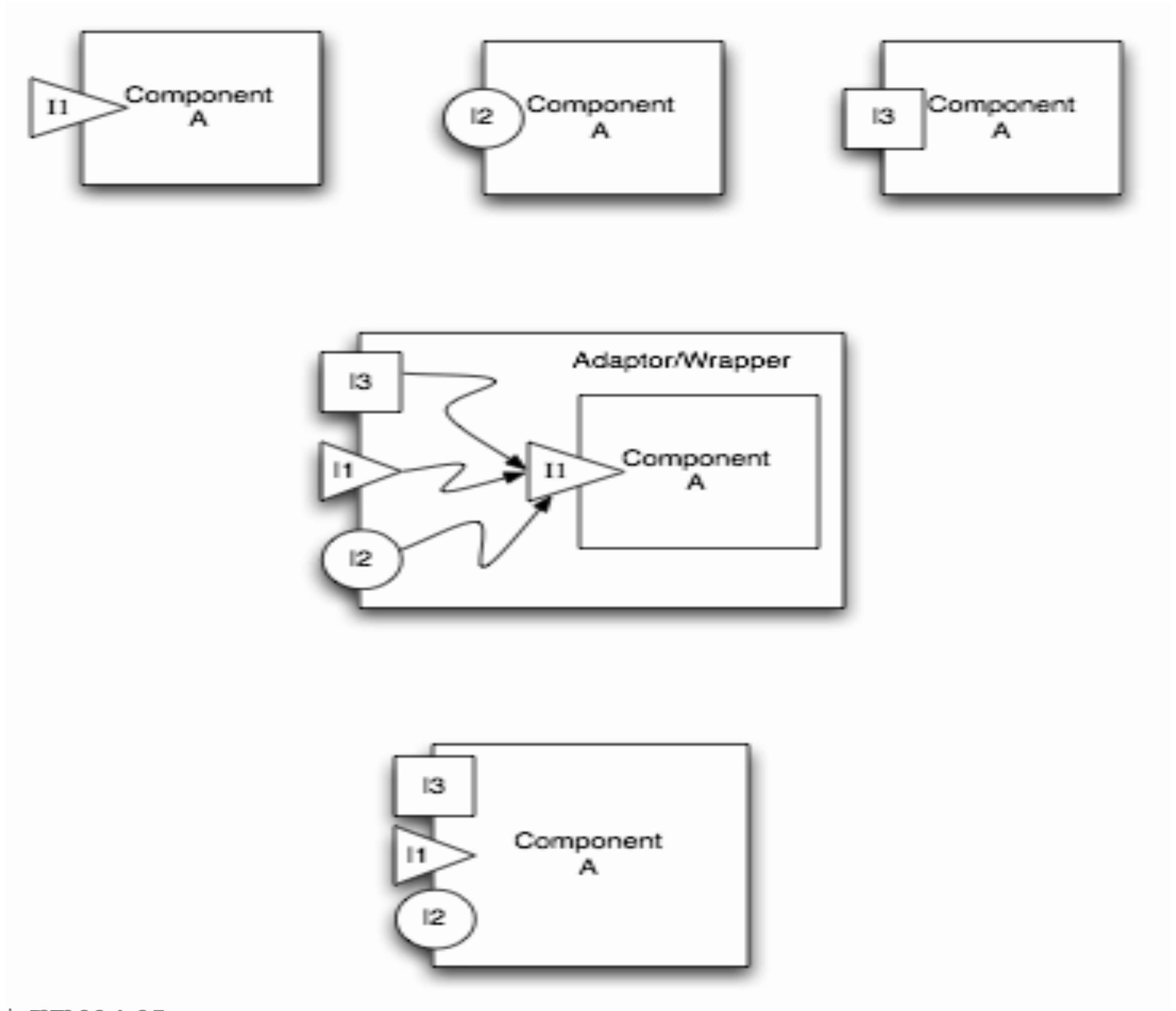
- What can software architecture say about efficiency?
  - Isn't efficiency an implementation-level property?

➢ *Efficiency starts at the architectural level!*

Software Components and Efficiency

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

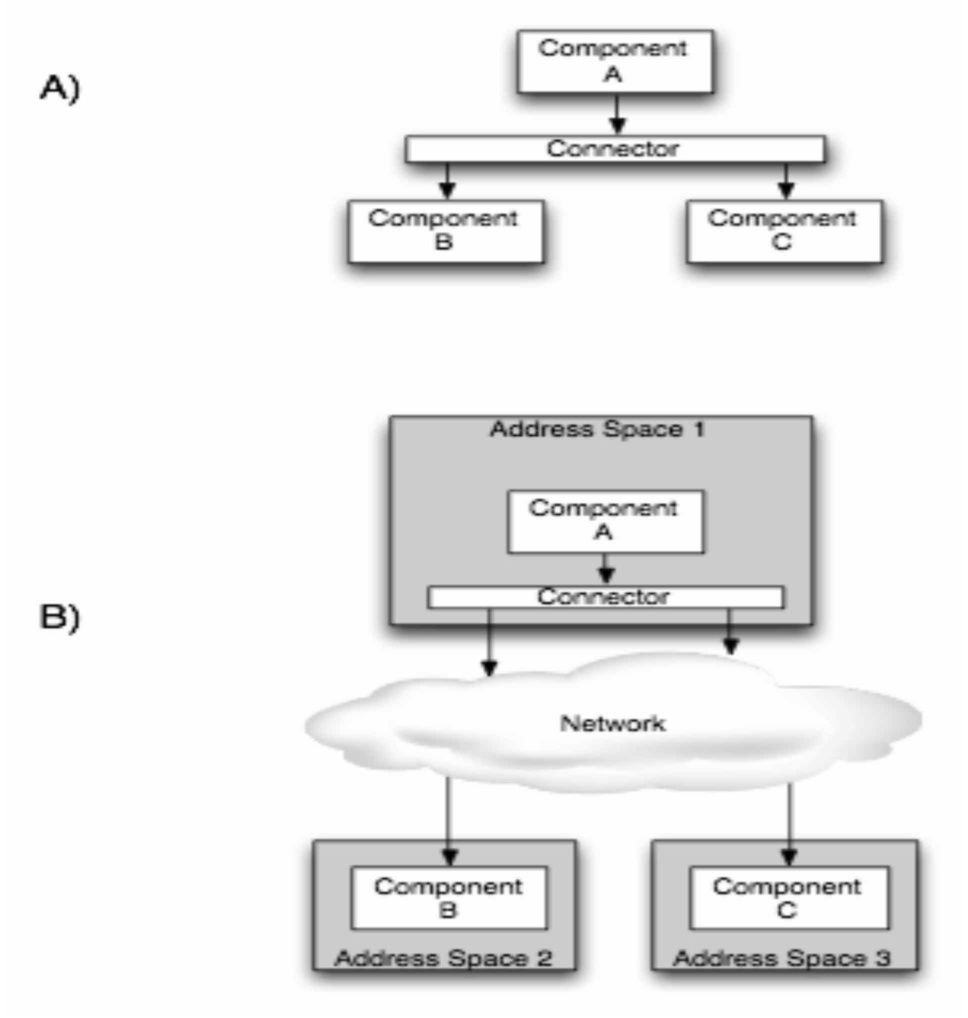
Multiple Interfaces to the Same Functionality



Software Connectors and Efficiency

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

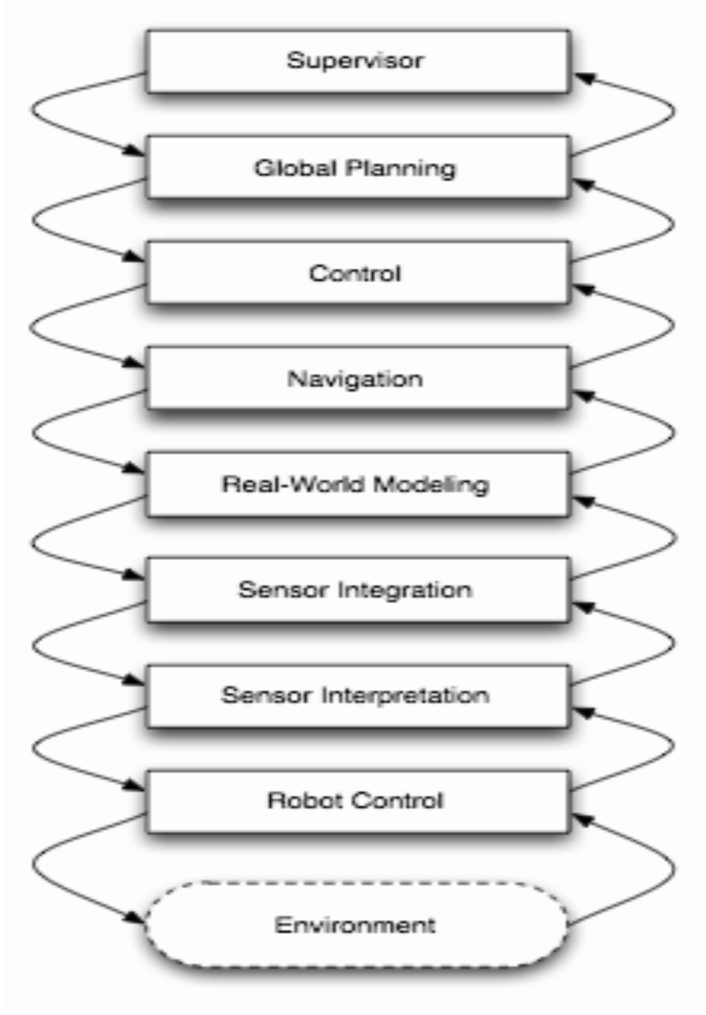
Distribution Transparency



Architectural Configurations and Efficiency

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

Performance Penalty Induced by Distance



NFP Design Techniques

Software Architecture
Lecture 20

Complexity

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

Software Components and Complexity

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

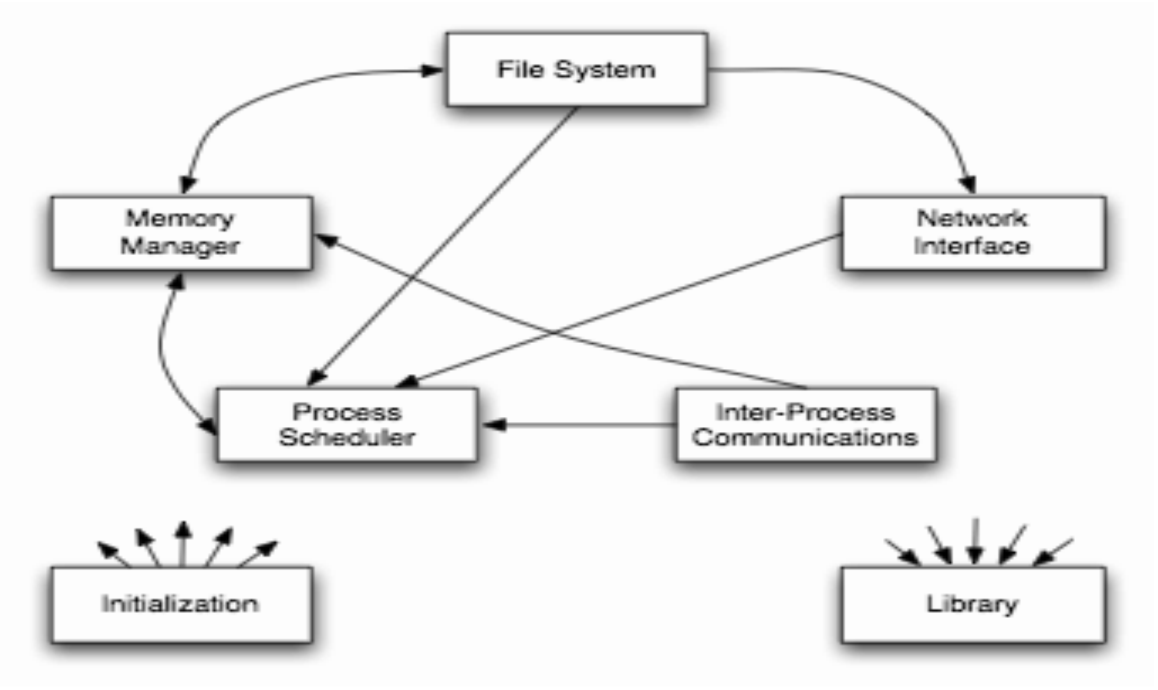
Software Connectors and Complexity

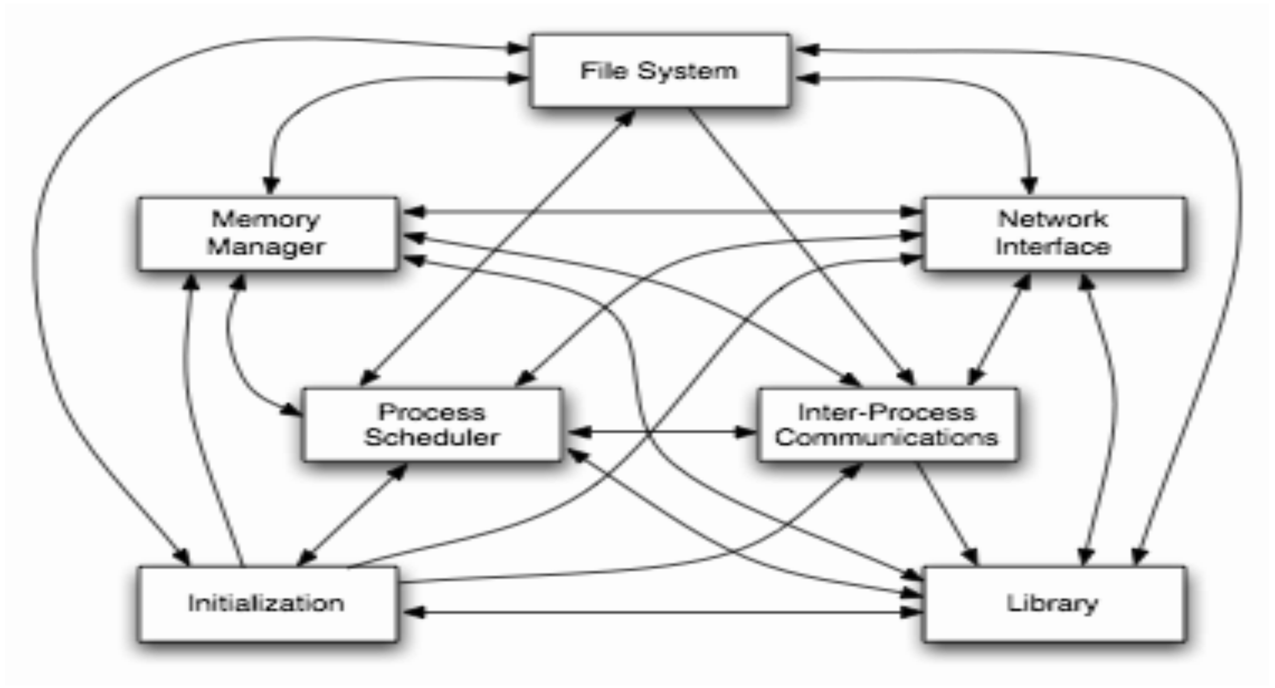
- Treat connectors explicitly
- Keep only interaction facilities inside connectors
- Separate interaction concerns into different connectors
- Restrict interactions facilitated by each connector
- Be aware of the impact of off-the-shelf connectors on complexity

Architectural Configurations and Complexity

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

Complexity in Linux





Scalability and Heterogeneity

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

Software Components and Scalability

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

Software Connectors and Scalability

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

Architectural Configurations and Scalability

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

Adaptability

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

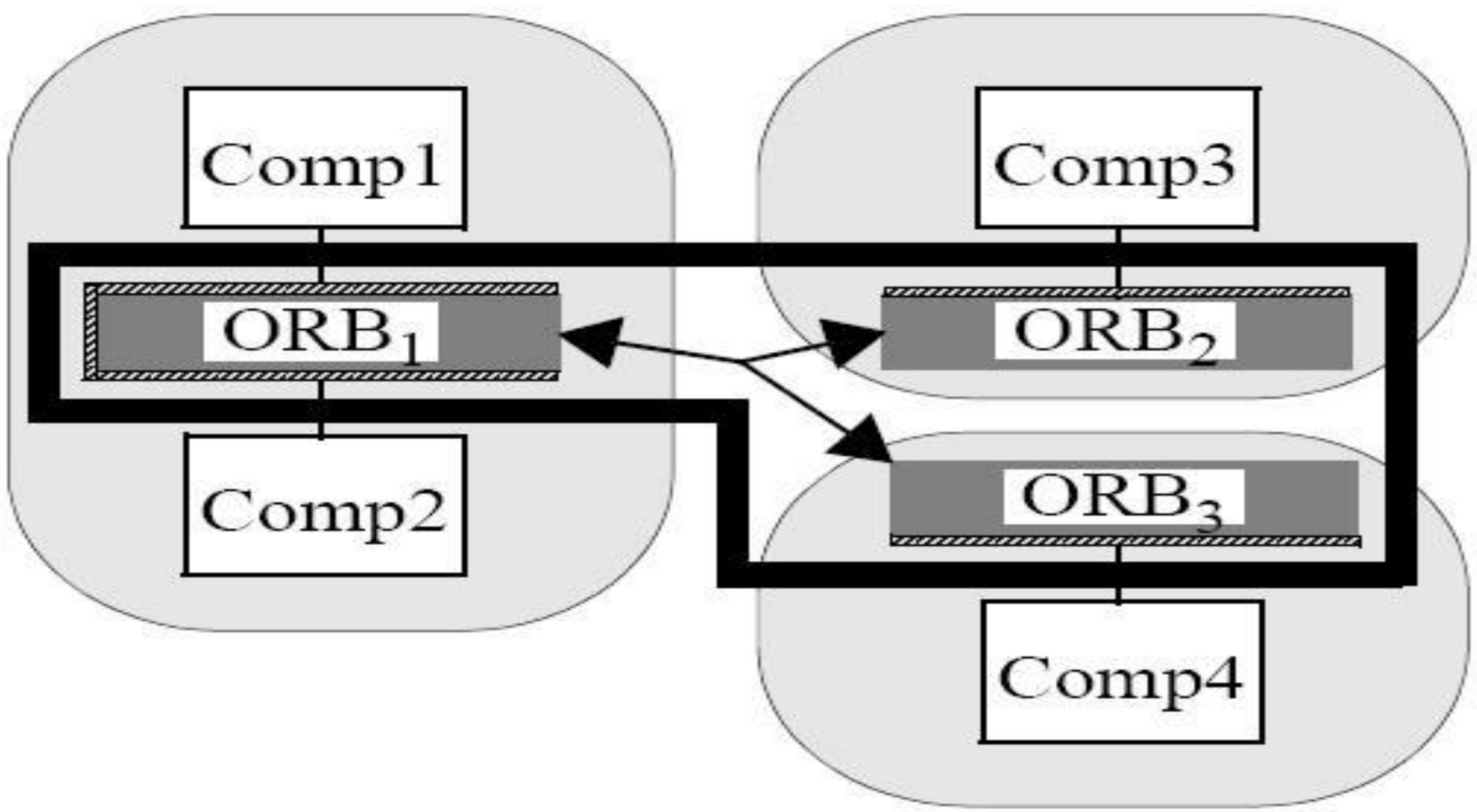
Software Components and Adaptability

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

Software Connectors and Adaptability

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

Composable Connectors



Architectural Configurations and Adaptability

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

Dependability

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

Software Components and Dependability

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

Software Connectors and Dependability

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

Architectural Configurations and Dependability

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