Building Software Architecture

A terminology review

What is an architectural view?

- Views reflect three broad ways an architect looks at a system:
  - Units of implementation (module views)
  - Runtime units (C&C views)
  - Relation to non-software structures (allocation views)

A terminology review

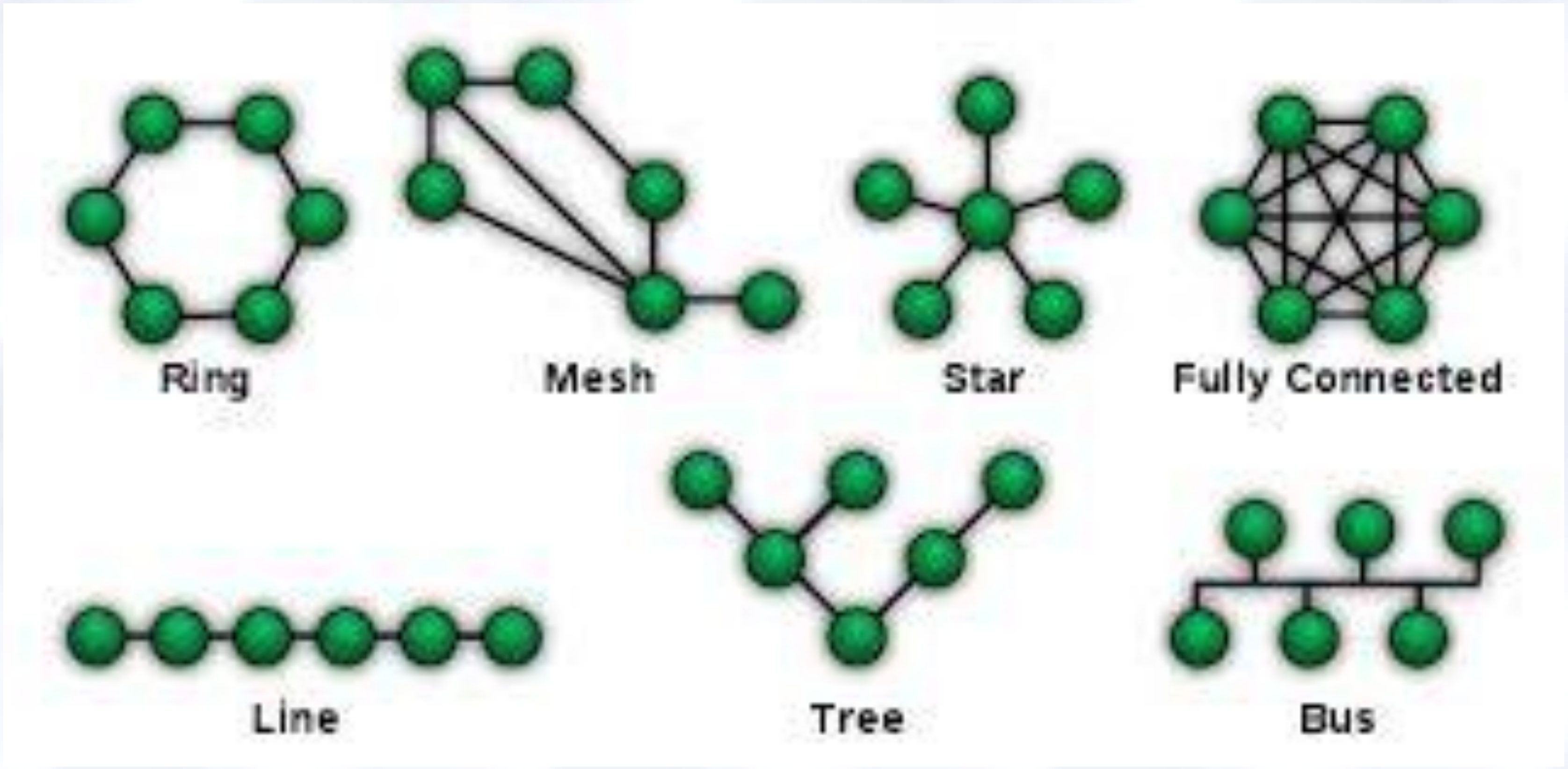
Architectural styles vs. architectural views?

- Within a view, many **choices still remain**:
  - What kinds of **relationships** are allowed?
  - What kinds of **constraints** are enforced?
  - **Different choices** lead to **different architectural styles**
- Accordingly, an architectural style can represent ***modules*** or ***runtime units*** or ***software & non-software elements.***

Basic Properties of Styles

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

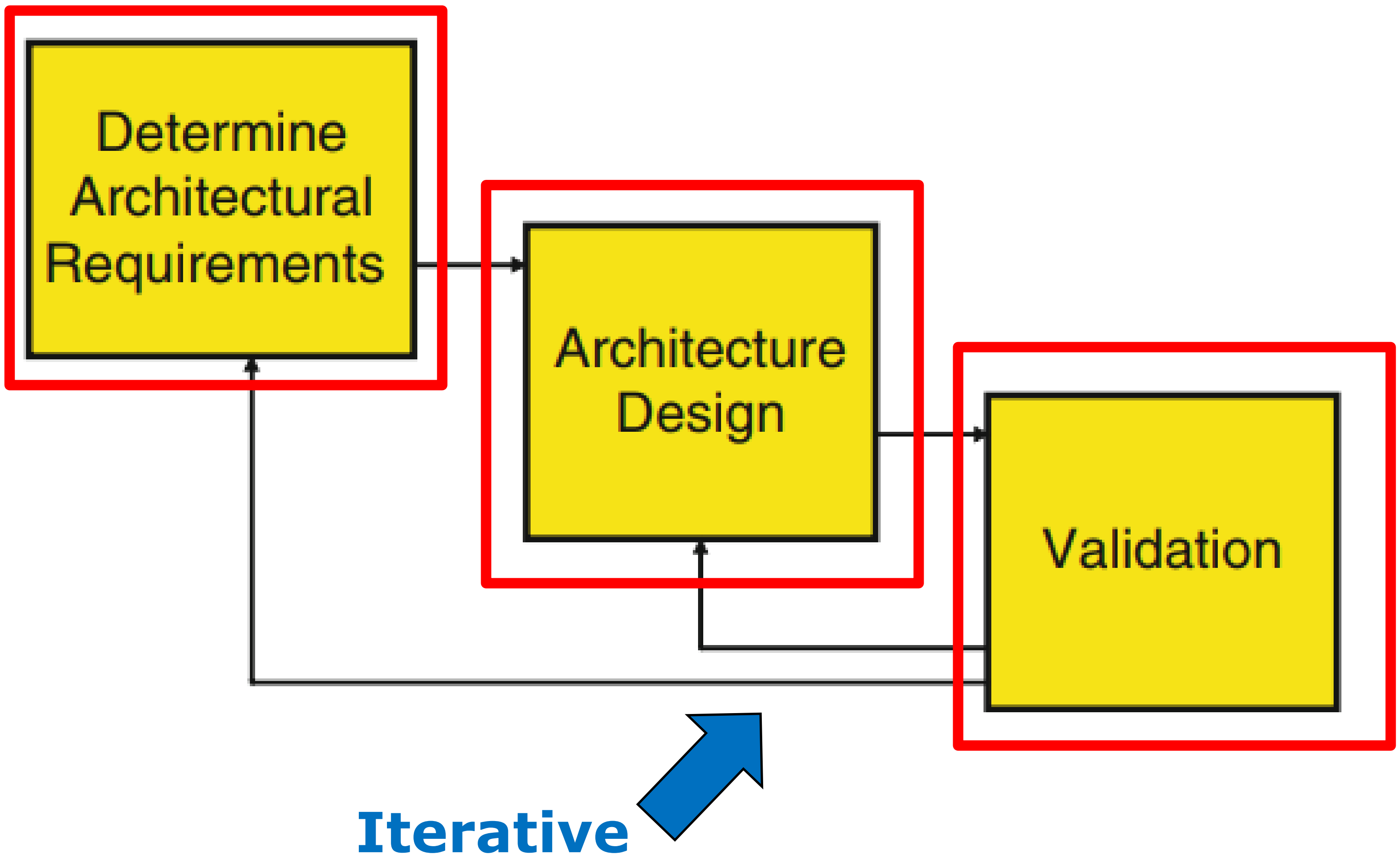
A Topology?



What is the role of a software architect?

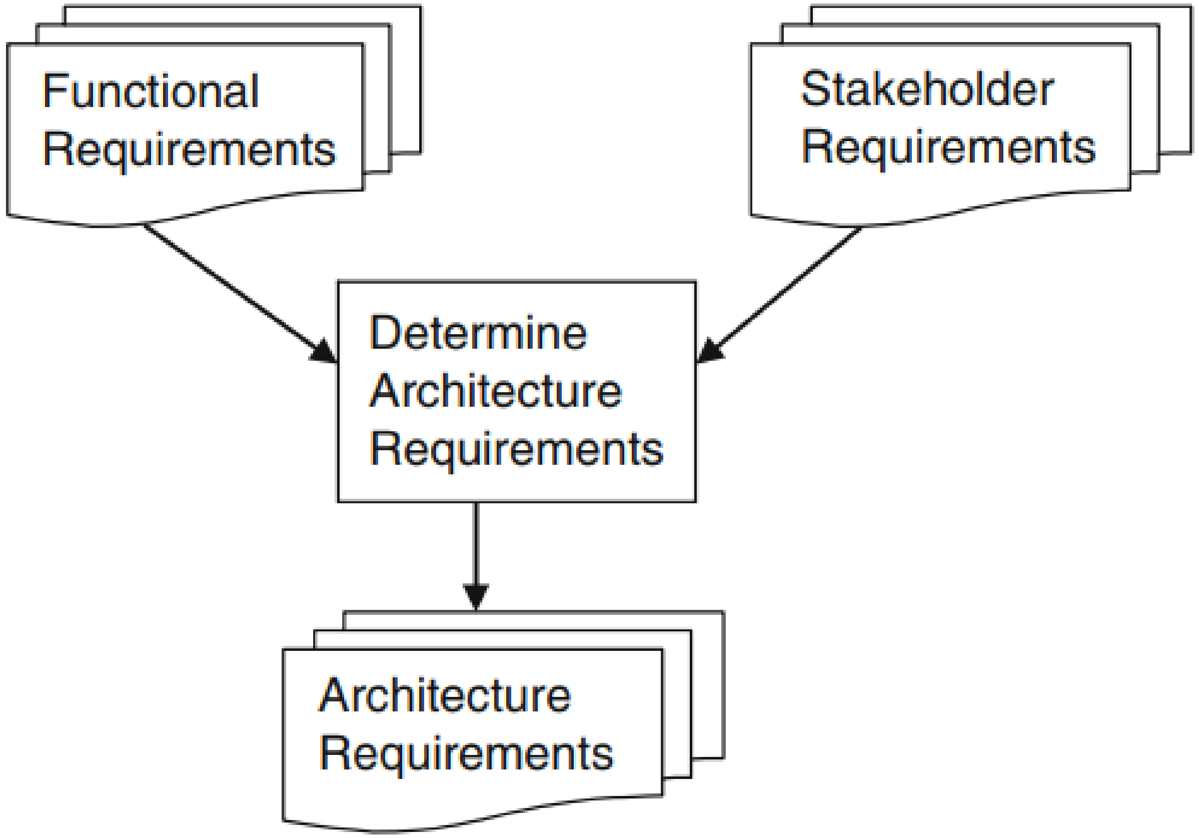
- The role of the software architect is to:
  - Work with the requirements team (explicit requirements?)
  - Work with various application stakeholders (Why?)
  - Lead the technical design team
  - Work with the project team

The Architecture Process Outline

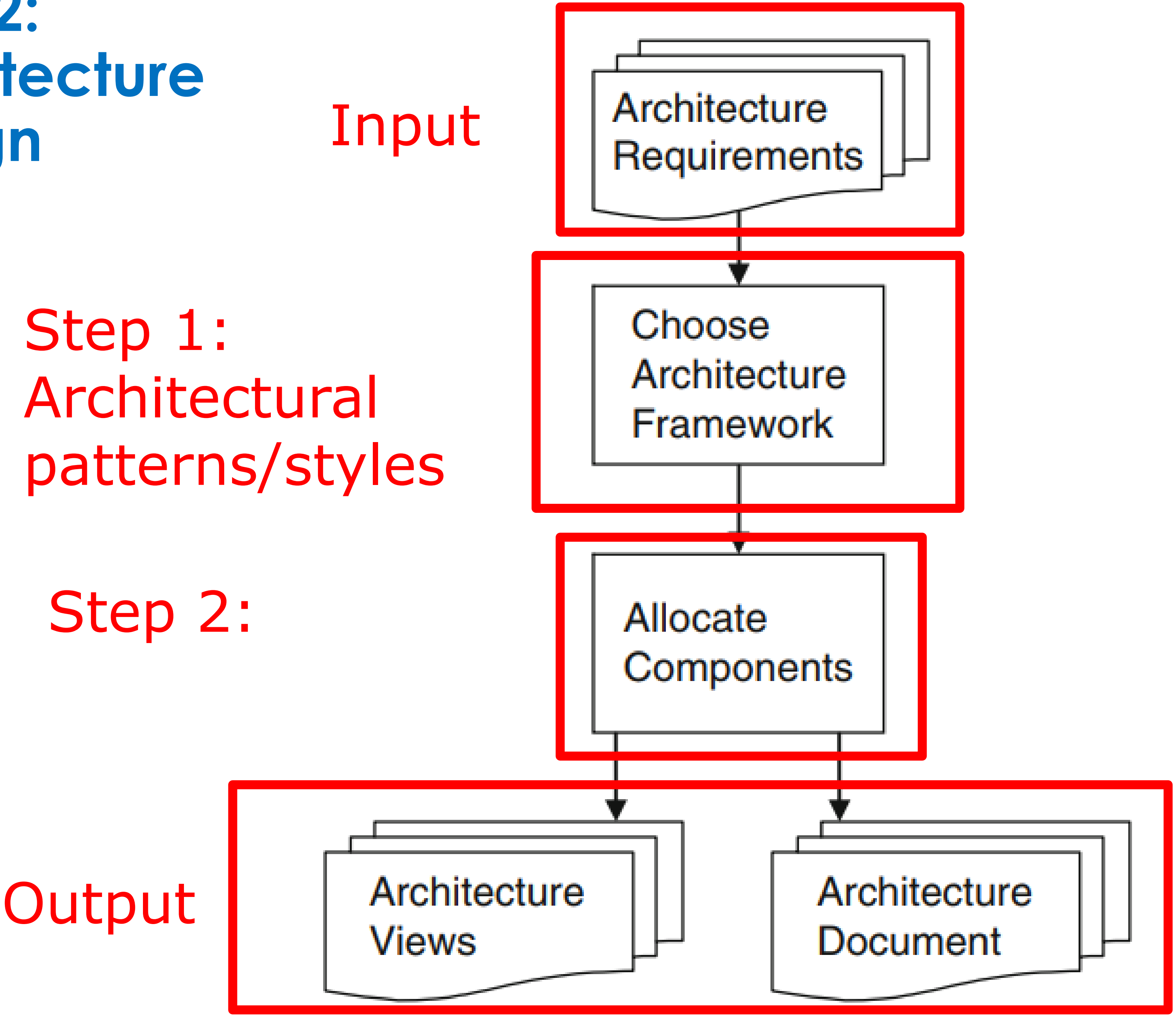


Step 1: Determine Architectural Requirements

- Architecture requirements (also called architecturally significant requirements or architecture use cases) are essentially the quality and nonfunctional requirements for the application.


Functional Requirements
Stakeholder Requirements
Determine Architecture Requirements
Architecture Requirements

Step 2: Architecture Design


Input
Architecture Requirements
Step 1: Architectural patterns/styles
Choose Architecture Framework
Step 2:
Allocate Components
Output
Architecture Views
Architecture Document

Step 2: Architecture Design

The design stage has two steps:

- The first involves choosing an overall strategy for the architecture, based around proven architecture patterns (a.k.a. architectural styles). (Step 2a)
- The second involves specifying the individual components that make up the application, showing how they fit into the overall framework and allocating them responsibilities. (Step 2b)

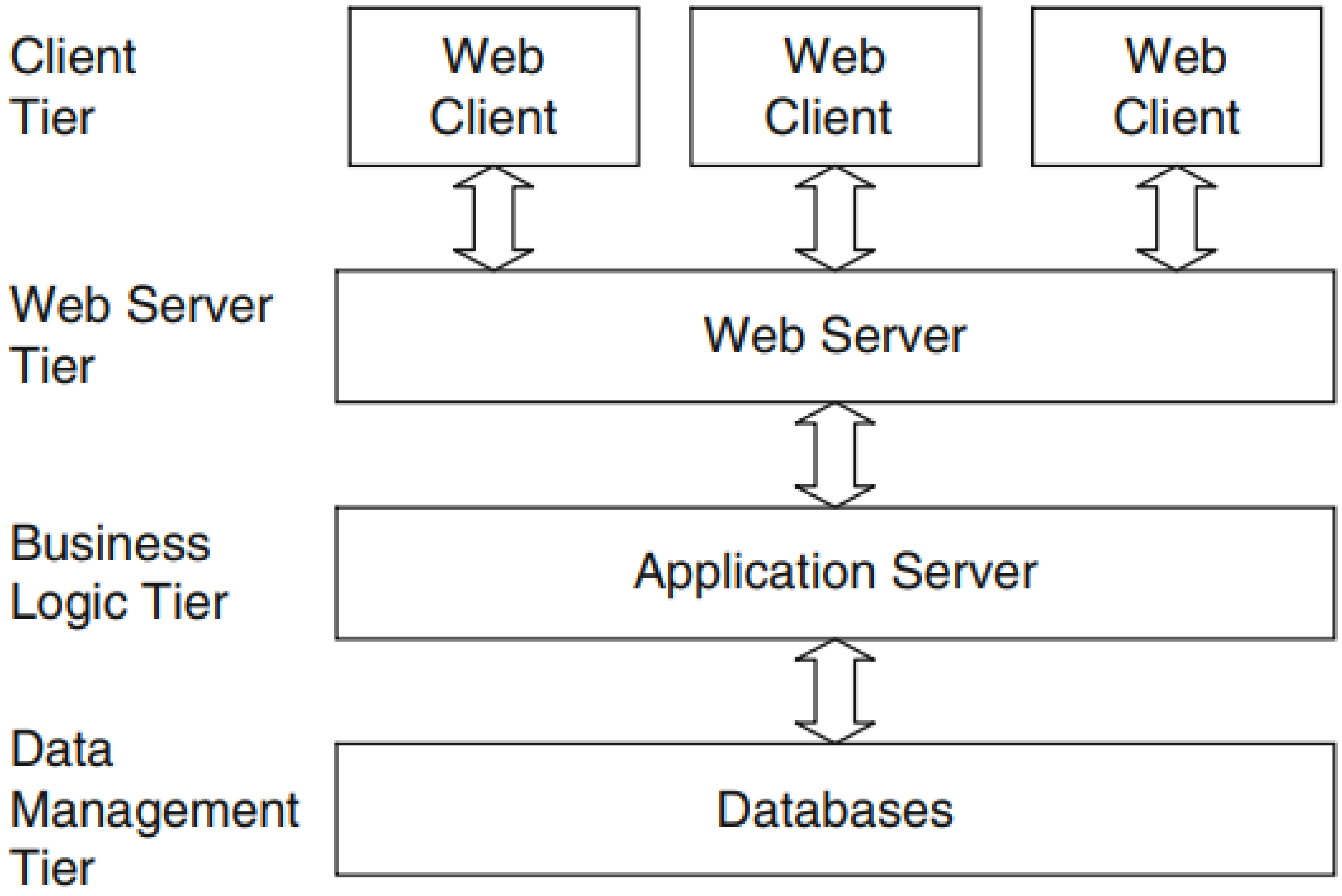
Step 2a: Choosing the Architecture Framework

- Leveraging known solutions minimizes the risks that an application will fail due to an inappropriate architecture
- Architectural patterns have clear pros/cons with respect to specific quality attributes.
- For small applications, a single architecture pattern like n-tier client-server may suffice.
- For more complex applications, the design will incorporate one or more known patterns, with the architect specifying how these patterns integrate to form the overall architecture

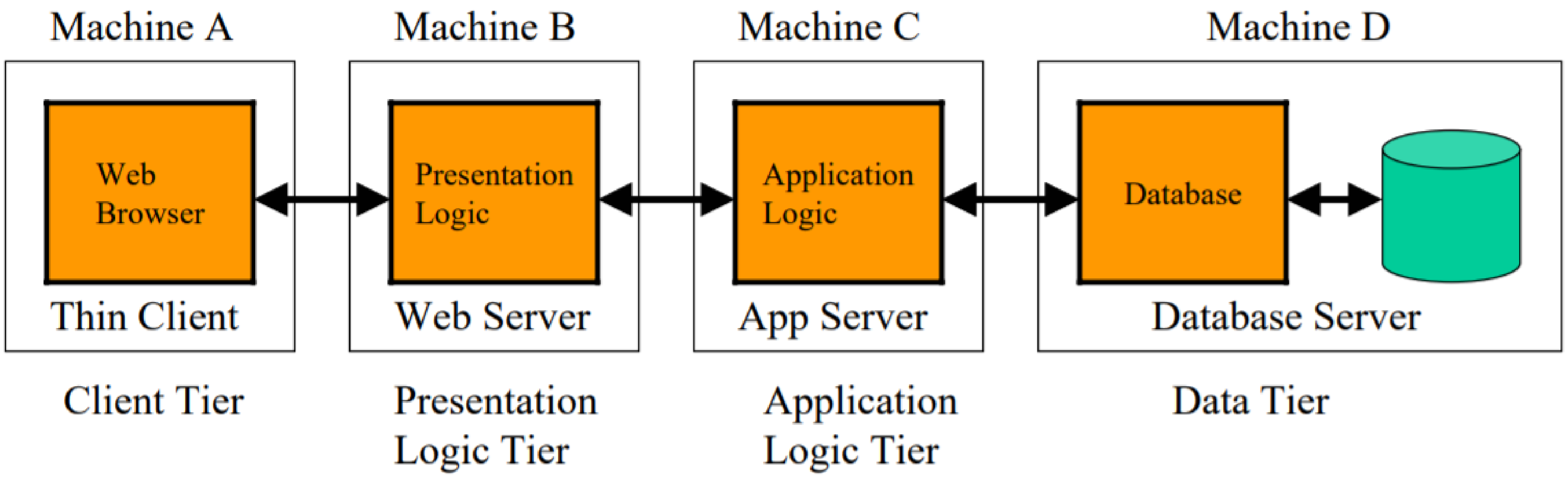
Step 2a: Architectural Styles

- **Layered architecture (N-tier)**
  - Client-server
- Peer-to-peer
- Pipe-and-filter
- Messaging
- Implicit Invocation
  - Publish-subscribe (Example: Spotify)
  - Event based (Eclipse IDE)
- Traditional language-influenced styles
  - Main program and sub-routines
  - Object-oriented (Abstract Data Types)
- Shared Data

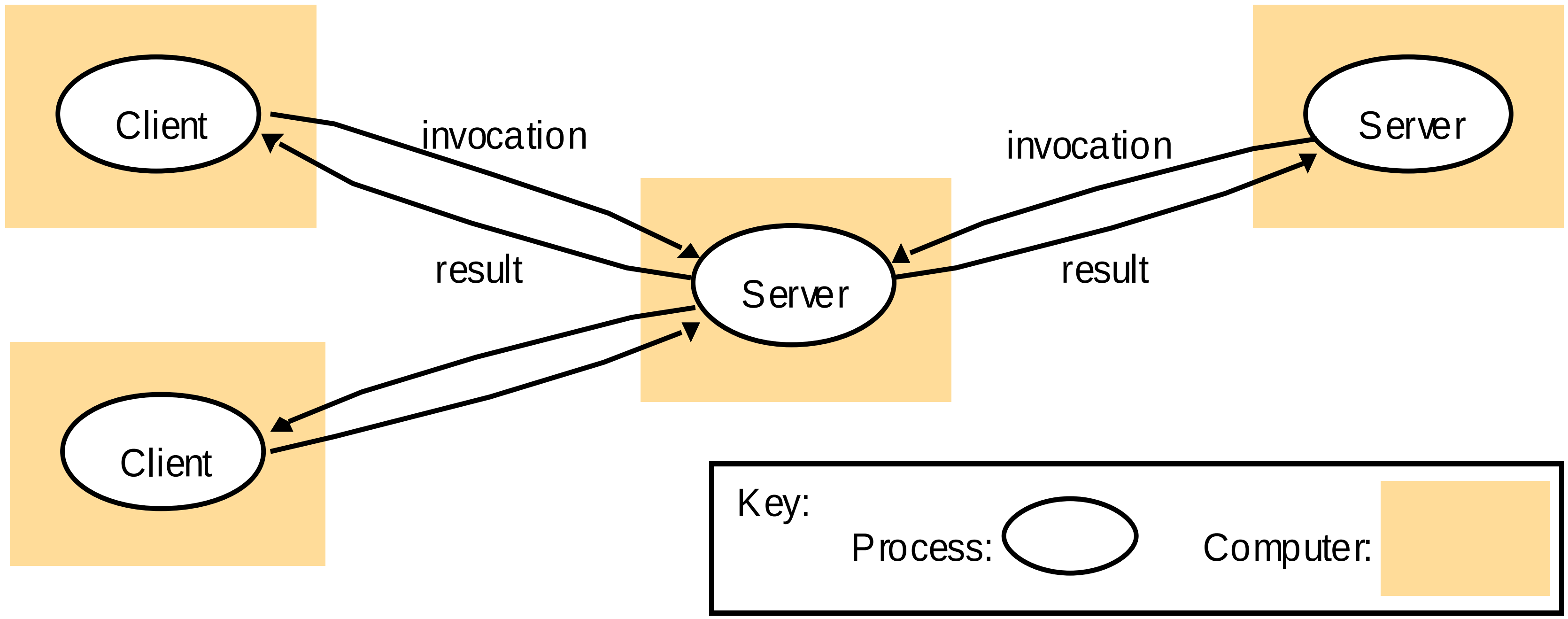
Step 2a: N-Tier Client Server Pattern



Step 2a: N-Tier Client Server Pattern



Step 2a: Client-Server



- Client processes interact with individual server processes in a separate computer in order to access data or resource. The server in turn may use services of other servers.
- Example?

Step 2a: N-Tier Client Server Pattern

- Key properties?
  - Separation of concerns
  - Synchronous communication
    - Requests emanate in a single direction from the client tier, through the web and business logic tiers to the data management tier.
  - Flexible deployment:
    - No restrictions on how a multi-tier application is deployed.
    - All tiers could run on the same machine, or at the other extreme, each tier may be deployed on its own machine.

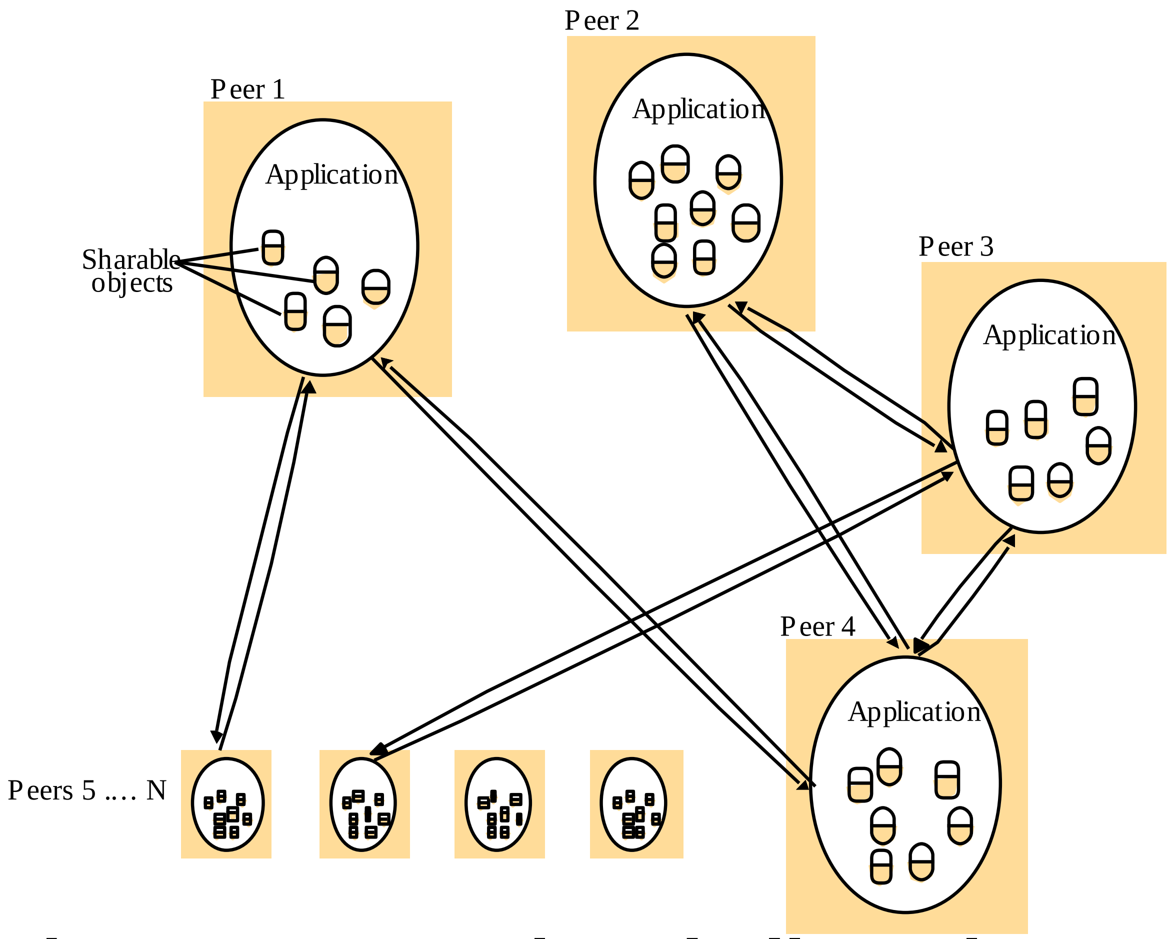
Step 2a: N-Tier Client Server Pattern

- Addressing Quality Attributes?
  - Availability?
    - Servers in each tier could be replicated.
  - Modifiability?
    - Separation of concerns!
  - Performance?
    - Depends on the speed of connections between different tiers
    - Depends on the amount of data transferred across the different tiers
  - Scalability?
    - Does the tier type matter?

Step 2a: Architectural Styles

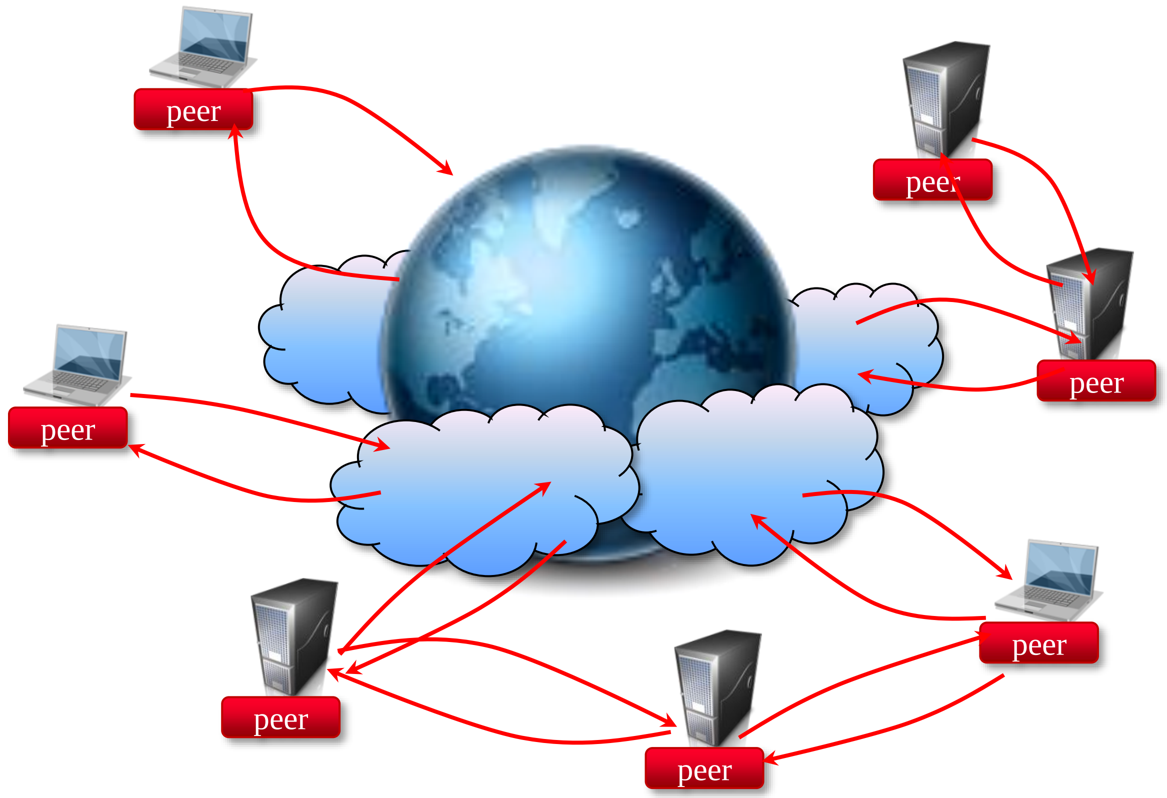
- Layered architecture (N-tier)
  - Client-server
- **Peer-to-peer**
- Pipe-and-filter
- Messaging
- Implicit Invocation
  - Publish-subscribe (Example: Spotify)
  - Event based (Eclipse IDE)
- Traditional language-influenced styles
  - Main program and sub-routines
  - Object-oriented (Abstract Data Types)
- Shared Data

Step 2a: Peer-to-Peer



- **All of the processes play similar roles,** interacting cooperatively as peers to perform distributed activities or computations without distinction between clients and servers. (Examples?)
- Pros/Cons?

P2P with a Centralized Index Server (e.g. Napster Architecture)



Step 2a: Architectural Styles

- Layered architecture (N-tier)
  - Client-server
- Peer-to-peer
- **Pipe-and-filter**
- Messaging
- Implicit Invocation
  - Publish-subscribe (Example: Spotify)
  - Event based (Eclipse IDE)
- Traditional language-influenced styles
  - Main program and sub-routines
  - Object-oriented (Abstract Data Types)
- Shared Data

Step 2a: Pipe and Filter Style

- Streams of data are passed from filter program to another
- The filters can operate concurrently, with no requirement for a producing component to finish before a component that consumes the producer's output begins.
- Components are filters
  - Transform input data streams into output data streams
  - Possibly incremental production of output

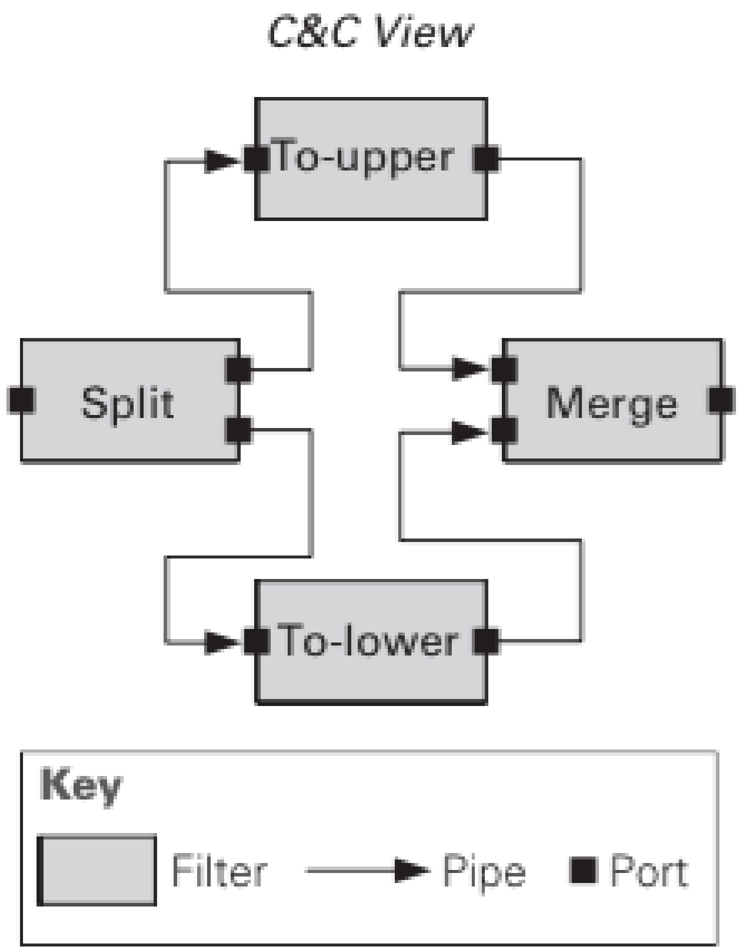
Step 2a: Pipe and Filter Style

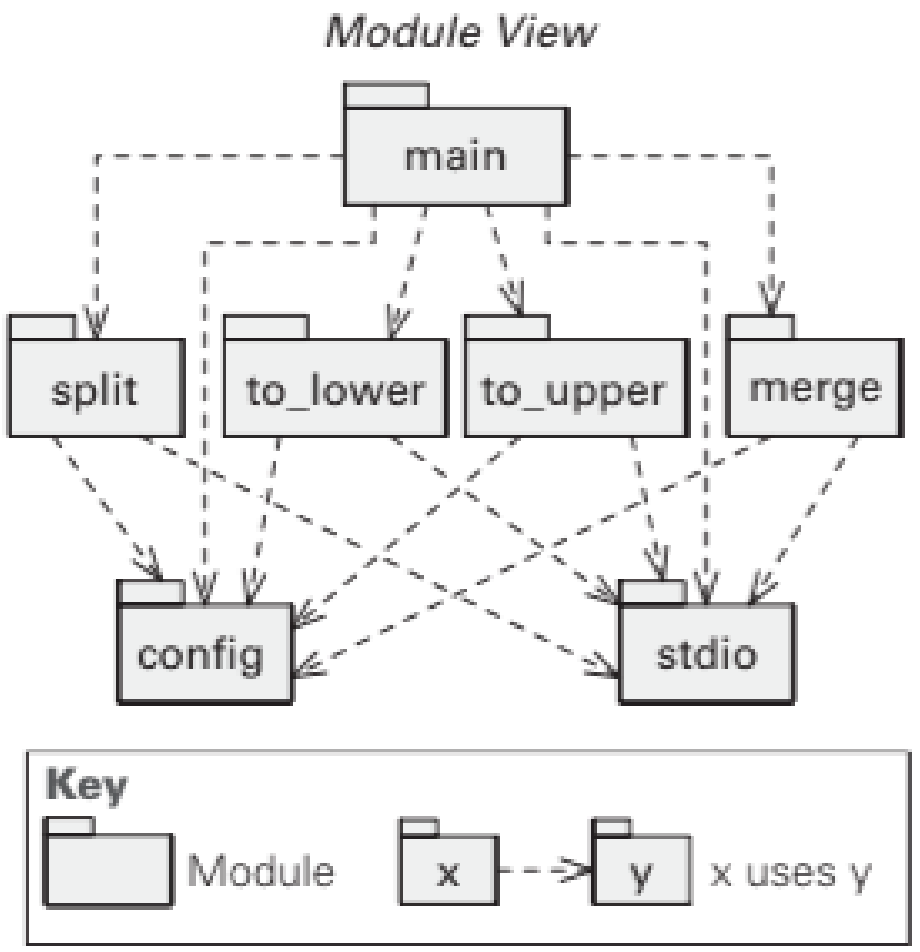
- Connectors are pipes
  - Conduits for data streams
- Style invariants
  - Filters are independent (no shared state)
- Examples
  - UNIX shell
  - Example: `ls invoices | grep -e August | sort`

Step 2a: Pipe and Filter Style

- Disadvantages?
- Cautions?
- Qualities?

Step 2a: Pipe and Filter Style

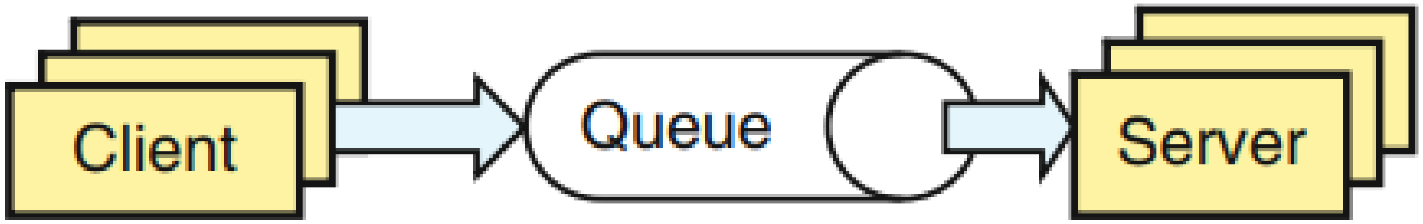




Step 2a: Architectural Styles

- Layered architecture (N-tier)
  - Client-server
- Peer-to-peer
- Pipe-and-filter
- **<u>Messaging</u>**
- Implicit Invocation
  - Publish-subscribe (Example: Spotify)
  - Event based (Eclipse IDE)
- Traditional language-influenced styles
  - Main program and sub-routines
  - Object-oriented (Abstract Data Types)
- Shared Data

Step 2a: Messaging



Key properties?

- Asynchronous communications:
  - Clients send requests to the queue, where the message is stored until an application removes it.
  - After the client has written the message to the queue, it continues without waiting for the message to be removed.

Step 2a: Messaging

- Addressing Quality Attributes?
  - Availability?
    - Queues can be replicated across messaging servers.
  - Modifiability?
    - Asynchronous communication
    - Message format

Step 2a: Architectural Styles

- Layered architecture (N-tier)
  - Client-server
- Peer-to-peer
- Pipe-and-filter
- Messaging
- **<u>Implicit Invocation</u>**
  - Publish-subscribe (Example: Spotify)
  - Event based (Eclipse IDE)
- Traditional language-influenced styles
  - Main program and sub-routines
  - Object-oriented (Abstract Data Types)
- Shared Data

Implicit Invocation

- Styles within implicit invocation category are characterized by calls that are invoked indirectly and implicitly as a response to a notification or an event.
  - Publish-subscribe: The publisher periodically creates information, and the subscriber obtains a copy of the information, or at least is notified of its availability.
  - Event-based is characterized by independent components communicating solely by sending events through an event bus.

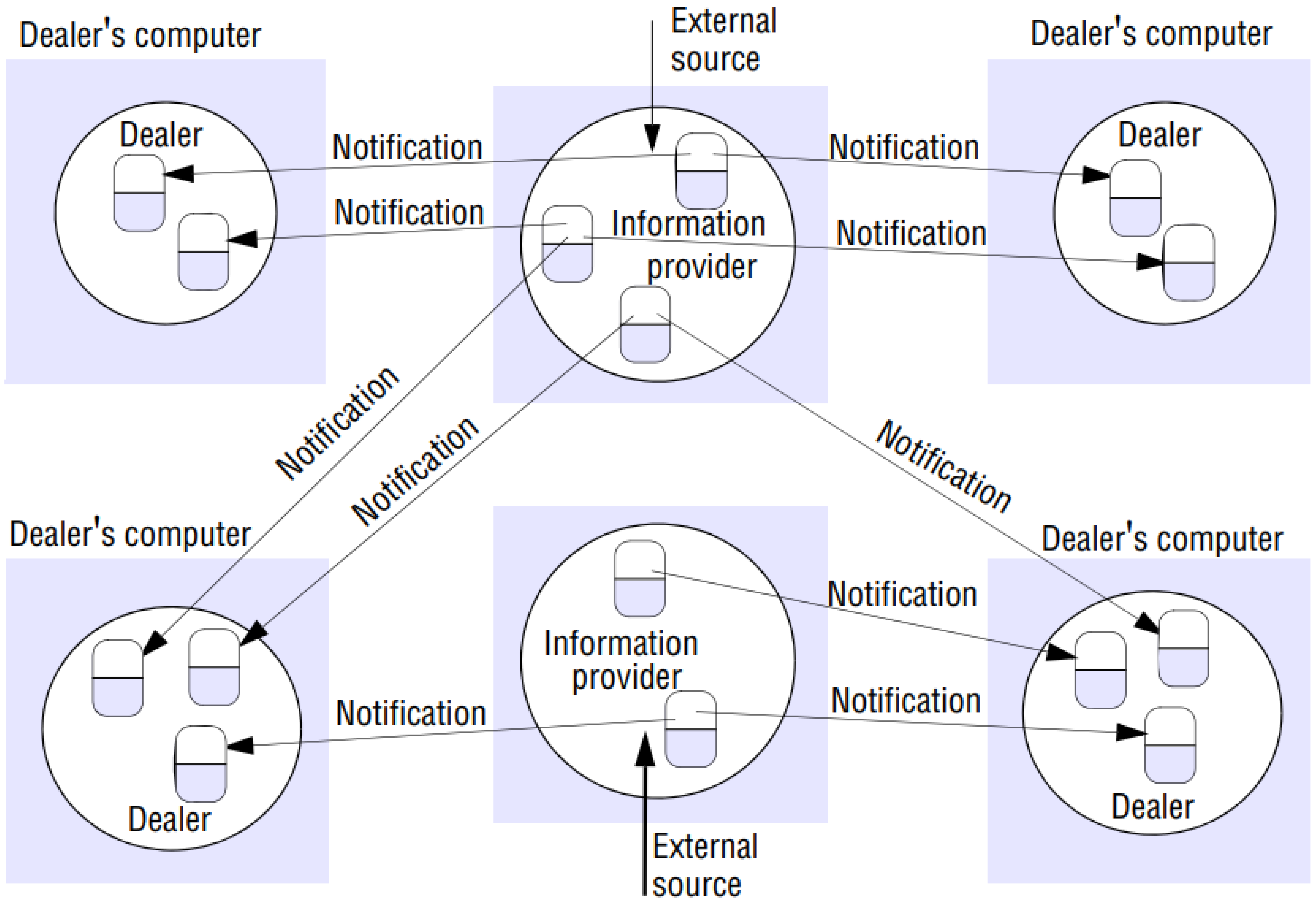
Publish-Subscribe Systems

- A publish-subscribe system is a system where *publishers* publish structured events to an event service and *subscribers* express interest in particular events through *subscriptions* which can be arbitrary patterns over the structured events.
- A given event will be delivered to potentially many subscribers, and hence publish-subscribe is fundamentally a one-to-many communication paradigm.
- Applications?

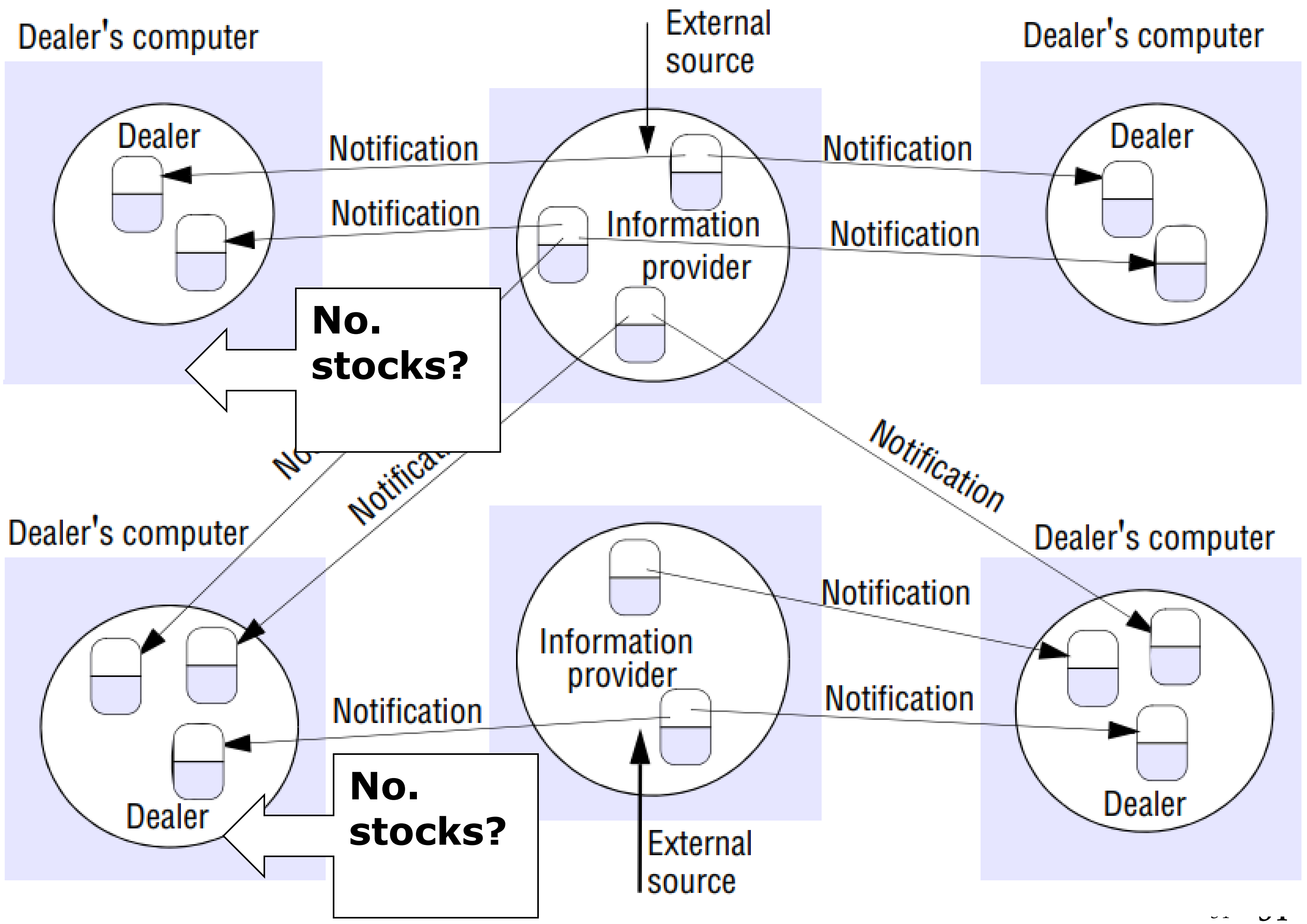
Example: Dealing Room System

- Consider a simple dealing room system whose task is to allow dealers using computers to see the latest information about the market prices of the stocks they deal in.
- The market price for a single named stock is represented by an associated object.
- The information arrives in the dealing room from several different external sources in the form of updates to some or all of the stocks.
- Dealers are typically interested only in their own specialist stocks.

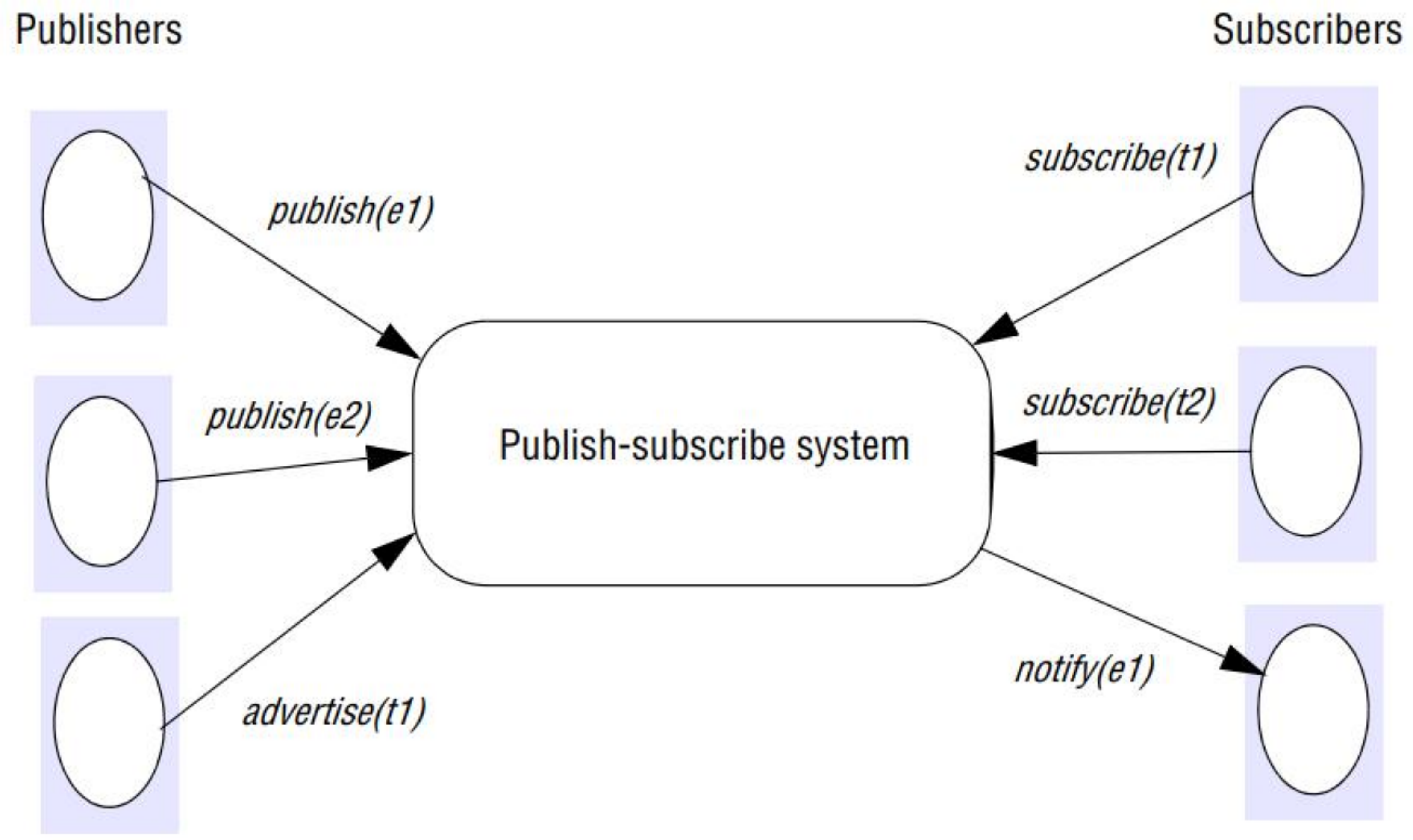
Example: Dealing Room System



Example: Dealing Room System



Publish-Subscribe Programming Model



Publish-Subscribe Programming Model Filter Model

1) Channel-based: publishers publish events to named channels and subscribers then subscribe to one of these named channels to receive all events sent to that channel.
2) Topic-based: each notification is expressed in terms of a number of fields, with one field denoting the topic.
3) Content-based: Content-based approaches are a generalization of topic-based over a range of fields in an event notification.

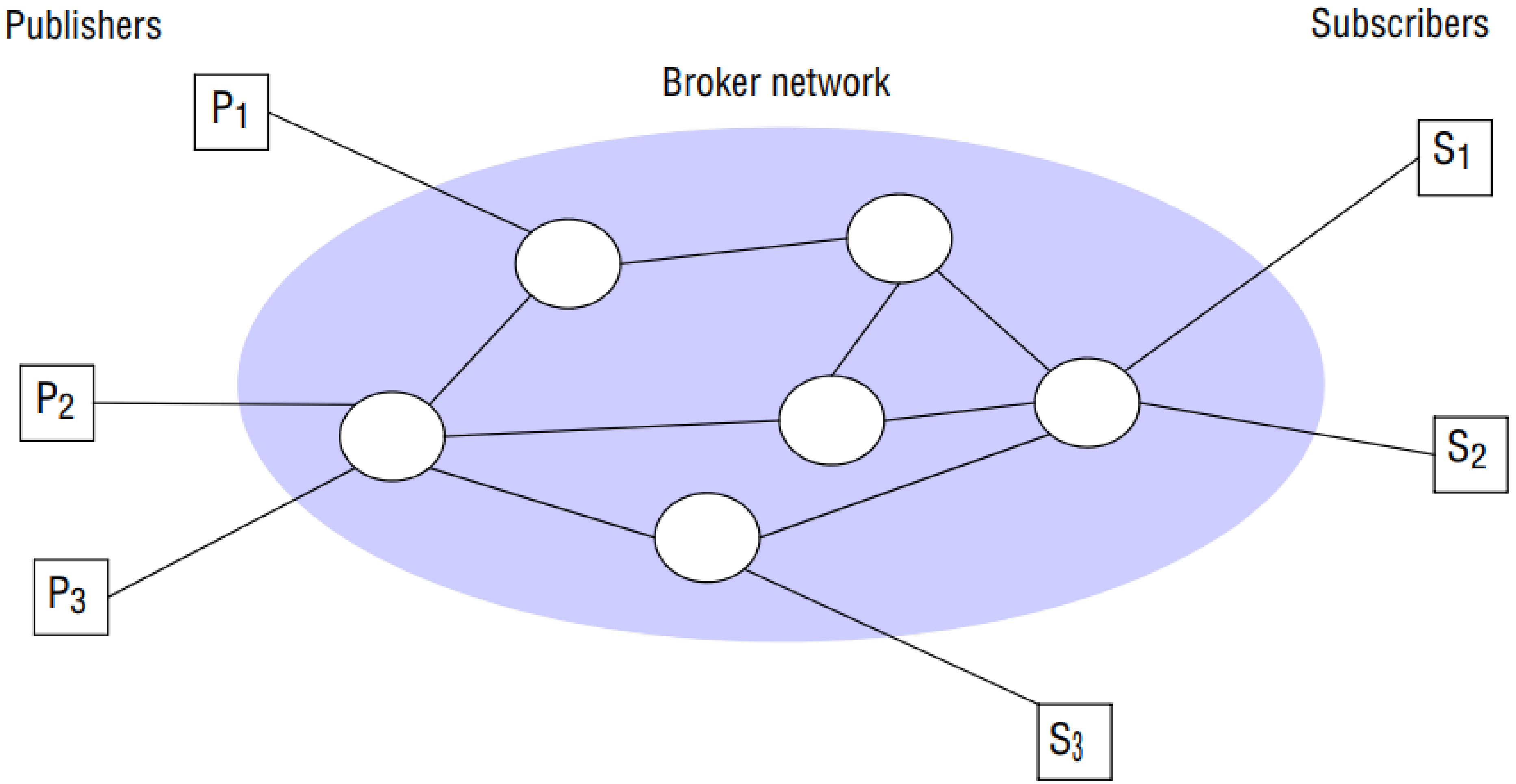
Publish-Subscribe Programming Model Filter Model

- Example: Alexander is interested in the topic of publish-subscribe systems, where the system in question is the ‘CORBA Event Service’ and where the author is ‘Tim Kindberg’ or ‘Gordon Blair’. (Filter?)

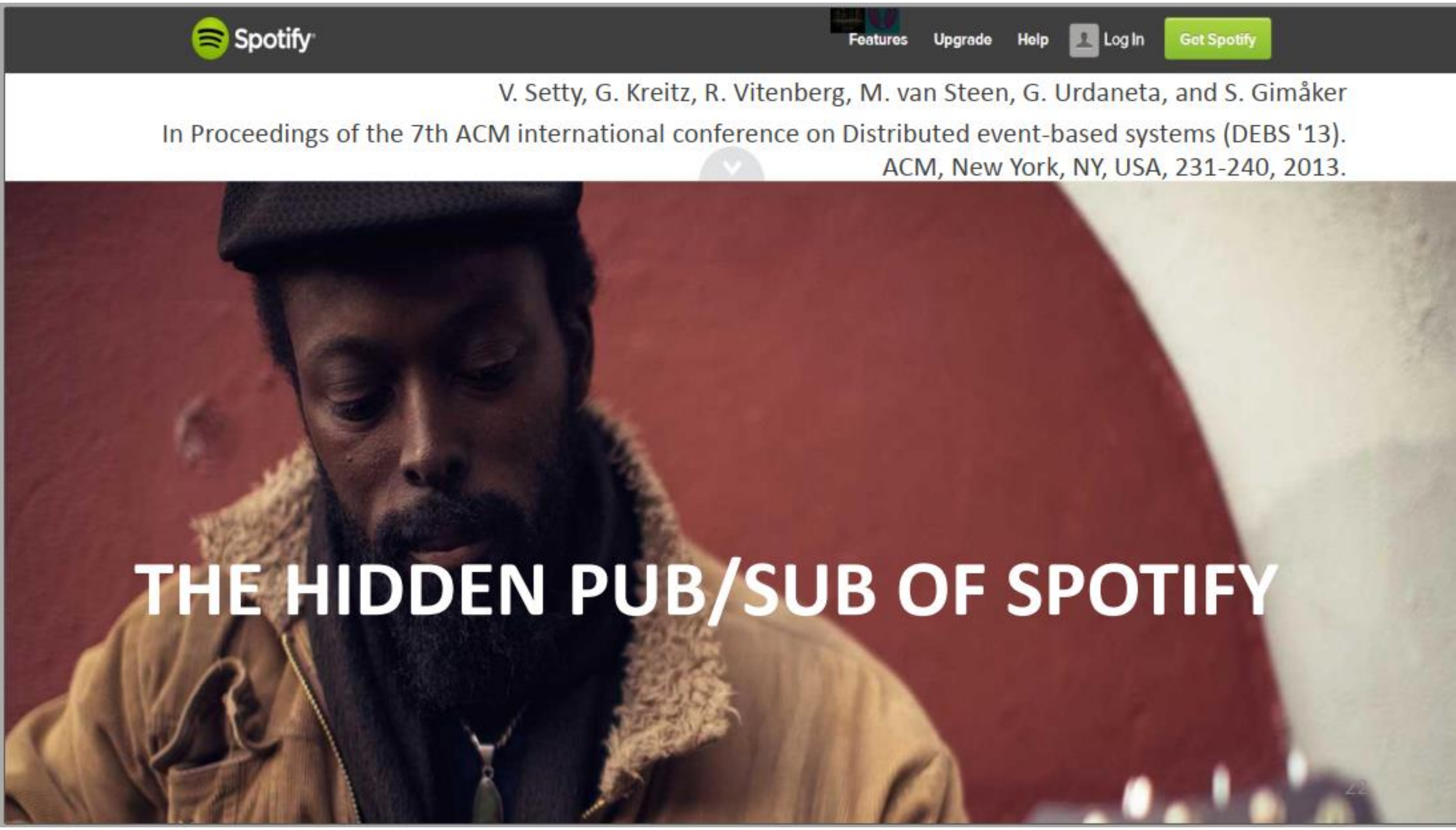
Publish-Subscribe Programming Model Concerns?

- The main concern is to deliver events efficiently to all subscribers
- Centralized implementation using **brokers** (How?)
  - Scalability
  - Failure handling
- Distributed implementation

Distributed Implementation



Case Study: Spotify



V. Setty, G. Kreitz, R. Vitenberg, M. van Steen, G. Urdaneta, and S. Gimåker

In Proceedings of the 7th ACM international conference on Distributed event-based systems (DEBS '13). ACM, New York, NY, USA, 231-240, 2013.

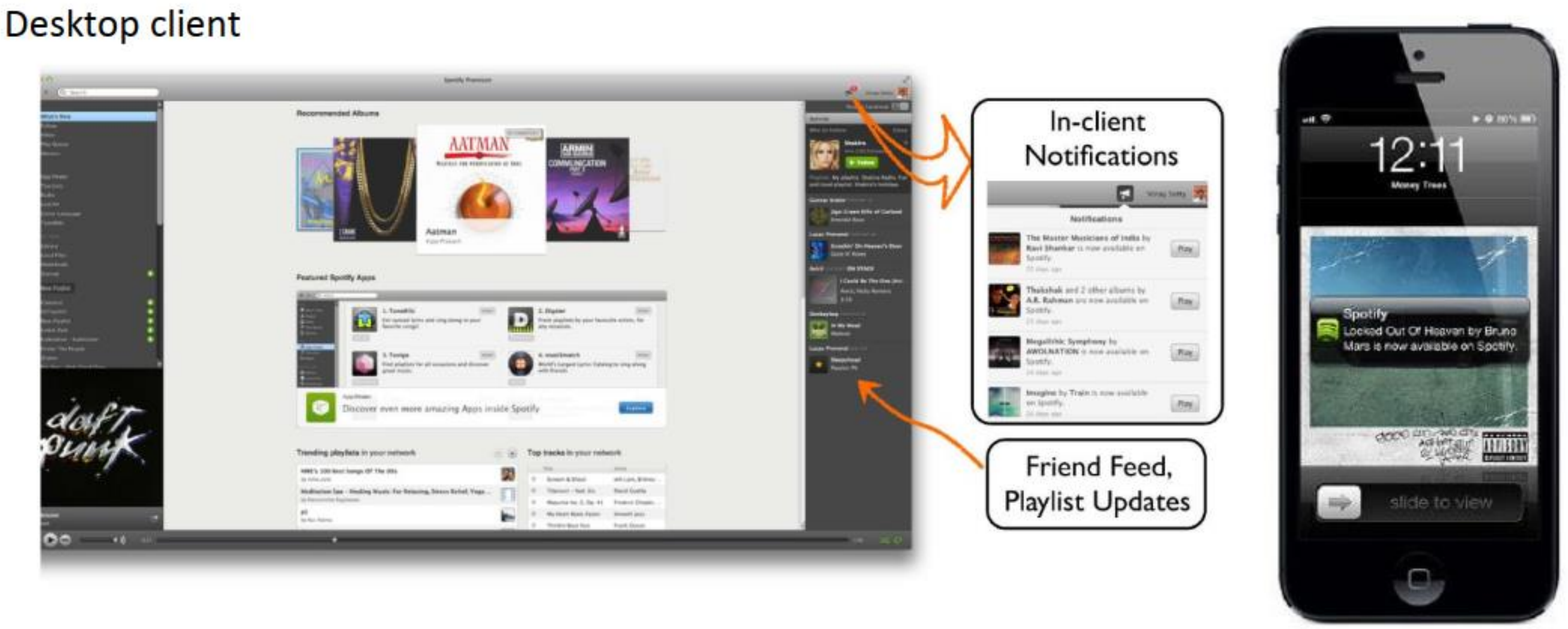
THE HIDDEN PUB/SUB OF SPOTIFY

Those slides are from a previous offering of the course “Distributed Systems (5DV147)”

Case Study: Spotify

- Topic-based subscriptions
- Hybrid engine
  - Relay events to online users in real time
  - Store and forward selected events to offline users
- Brokers are organized as a DHT (distributed hash table) overlay that spans three sites in Sweden, UK, and USA.
- Design to scale
  - Stores approx., 600 million subscriptions at any given time
  - Matches billions of publication events every day

Case Study: Spotify


Desktop client
In-client Notifications
Friend Feed, Playlist Updates
12:11
Spotify
Locked Out Of Heaven by Bruno Mars is now available on Spotify.
slide to view

Those slides are from a previous offering of the course "Distributed Systems (5DV147)"

Case Study: Spotify

Topic-based subscription

subscription(user_name, topic_name)

- Types of topics
  - Friends (Spotify + Facebook): FB friends who are Spotify users and by sharing music
  - Playlists (URI): other users playlists (updates), “Collaborative” playlists or only modifiable by creator
  - Artists pages (follow artist): new albums or news related to artist

Those slides are from a previous offering of the course “Distributed Systems (5DV147)”

Case Study: Spotify

Publication events

- All events delivered in real time (best effort and guaranteed delivery) to online users
- Some notifications are sent by email to retrieve in the future
- Example, new album from famous artist added
  - Instant notification sent to online followers
  - Email notification to offline followers
  - Event persisted so that (new) followers can retrieve it in the future (e.g., from another device)

Case Study: Spotify

Publication events

- Friend feed
  - Event notification to all friends following user
    - Play a track, create or modify playlist, add a favorite(artist, track, album)
    - Publish event on Facebook wall (optional)

Case Study: Spotify

Publication events

- Playlist updates
- – Event notifications when
  - A playlist is modified (adding or removing track, renaming playlist) via friend feed
- Synchronize playlist across all devices of all subscribers of the playlist

Those slides are from a previous offering of the course “Distributed Systems (5DV147)”

Case Study: Spotify

Publication events

- Artist pages
- – Notification sent to followers of artist when
  - New album added in Spotify
  - Playlist created by artist

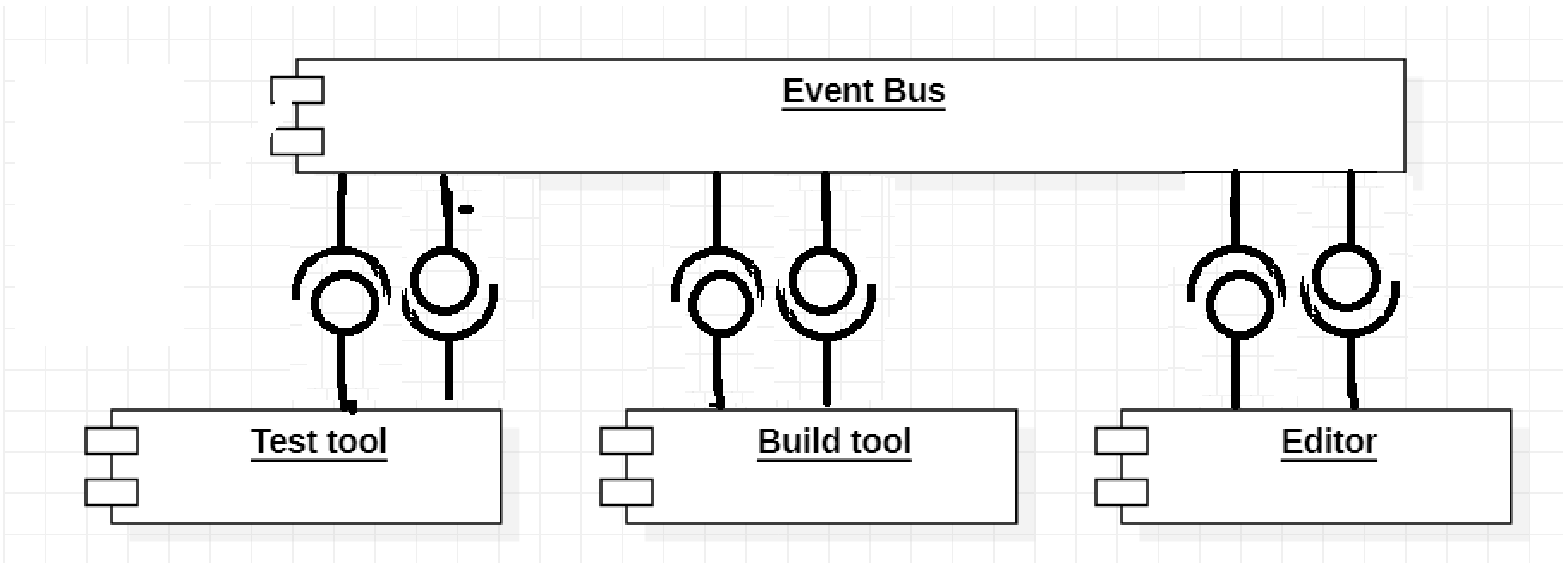
Those slides are from a previous offering of the course “Distributed Systems (5DV147)”

Event-based Systems

- Components emit events to the event-bus, which then transmits them to every other component.
- Components may react in response to receipt of an event, or may ignore it.
- How does event-based differ from publish-subscribe?
  - Within event-based, all components can emit and consume events. There is no classification of components into publishers/subscribers.
  - Within event-based, the distribution of the events is the responsibility of the connectors.

Event-based Systems - Example

- Example retrieved from Coursera's University of Alberta's course on Software Architecture.
- Consider a code editor that includes three main components connected through an event bus.



Step 2a: Architectural Styles

- Layered architecture (N-tier)
  - Client-server
- Peer-to-peer
- Pipe-and-filter
- Messaging
- Implicit Invocation
  - Publish-subscribe (Example: Spotify)
  - Event based (Eclipse IDE)
- **<u>Traditional language-influenced styles</u>**
  - Main program and sub-routines
  - Object-oriented (Abstract Data Types)
- Shared Data

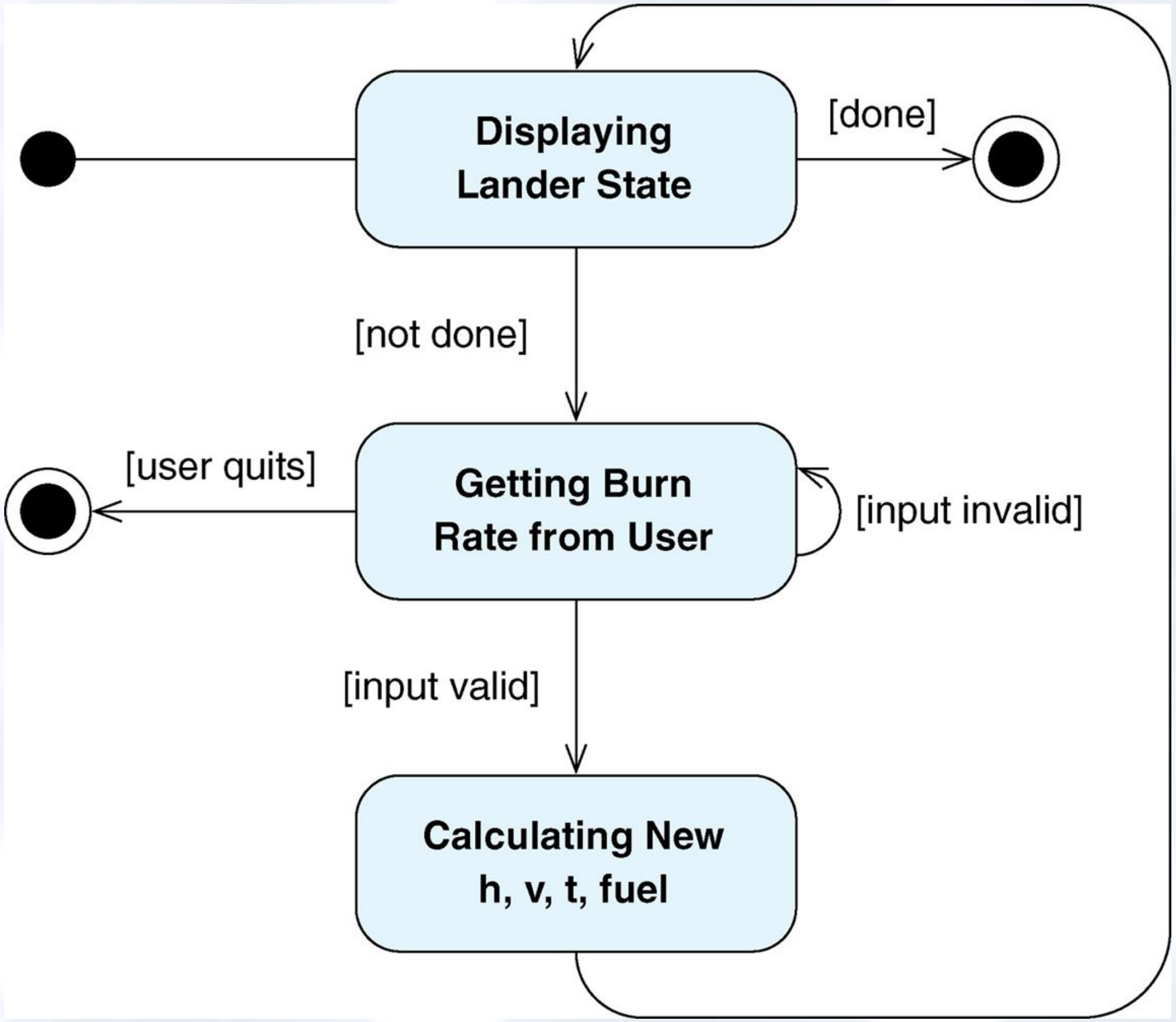
The Lunar Lander Game

- A simple computer game that first appeared in the 1960’s
- A player controls a spaceship as it falls towards the surface of the moon.
- The objective of the player is to land safely on the moon **without** running out of fuel, and without crashing (i.e., to land with small enough velocity so as not to crash).
- The lunar lander state has several fields: height, velocity, fuel, and moon gravity.

The Lunar Lander's Workflow

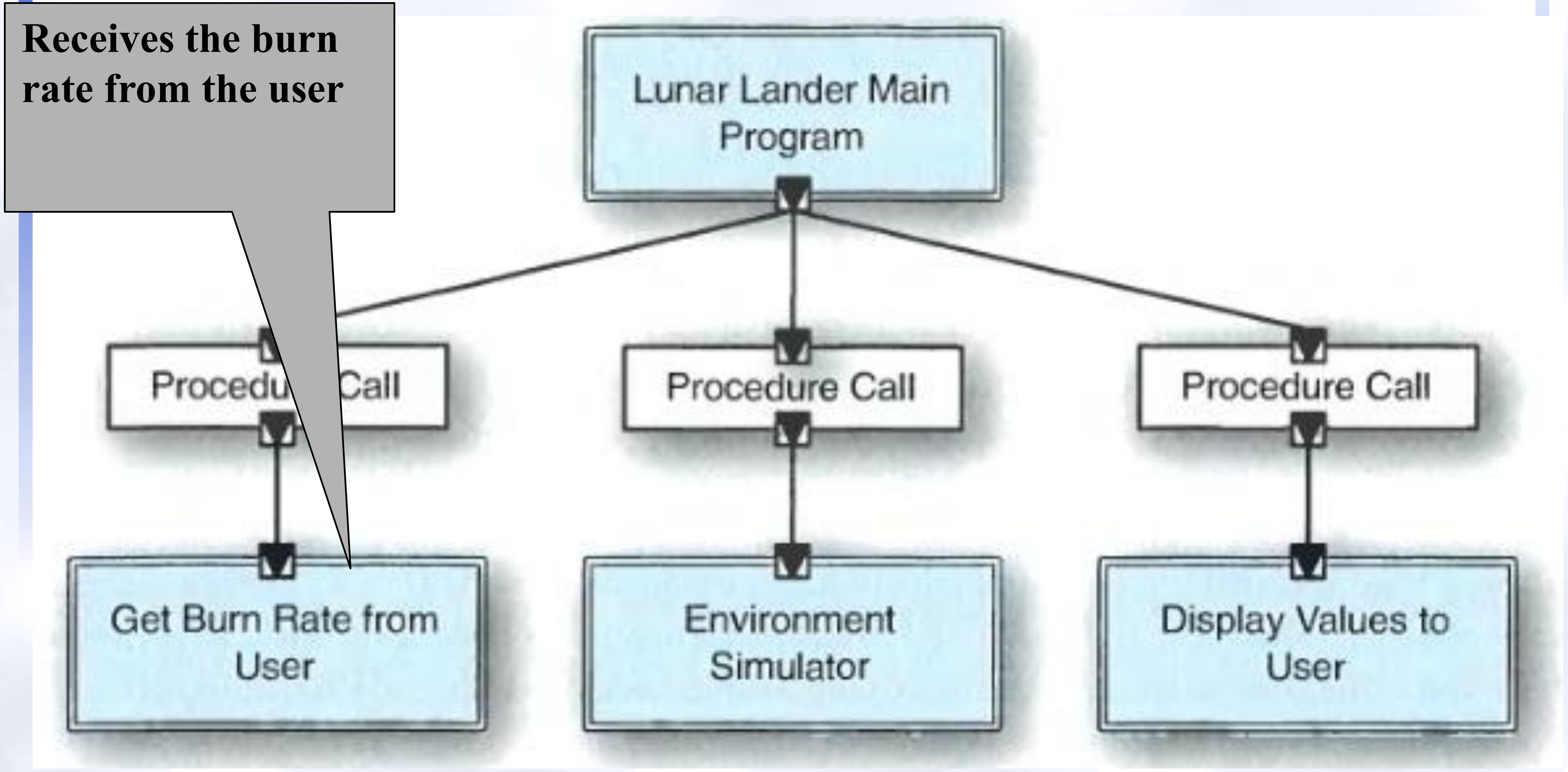
1. The user inputs a burn rate (i.e. rate at which the fuel is to be burned over the next time interval **t**)
2. The system:
   a) Retrieves the current **state** (height, velocity, fuel, and gravity) of the lunar lander
   b) Calculates the new **state** (height, velocity, fuel) of the lunar lander after time **t**.
   c) Displays the new state on the screen to the user.
   d) Checks if the ship has landed.
      i. If yes, end the game.
      ii. If no, go to step 1 (new burn rate)

The Lunar Lander’s Workflow



Main Program and Subroutines

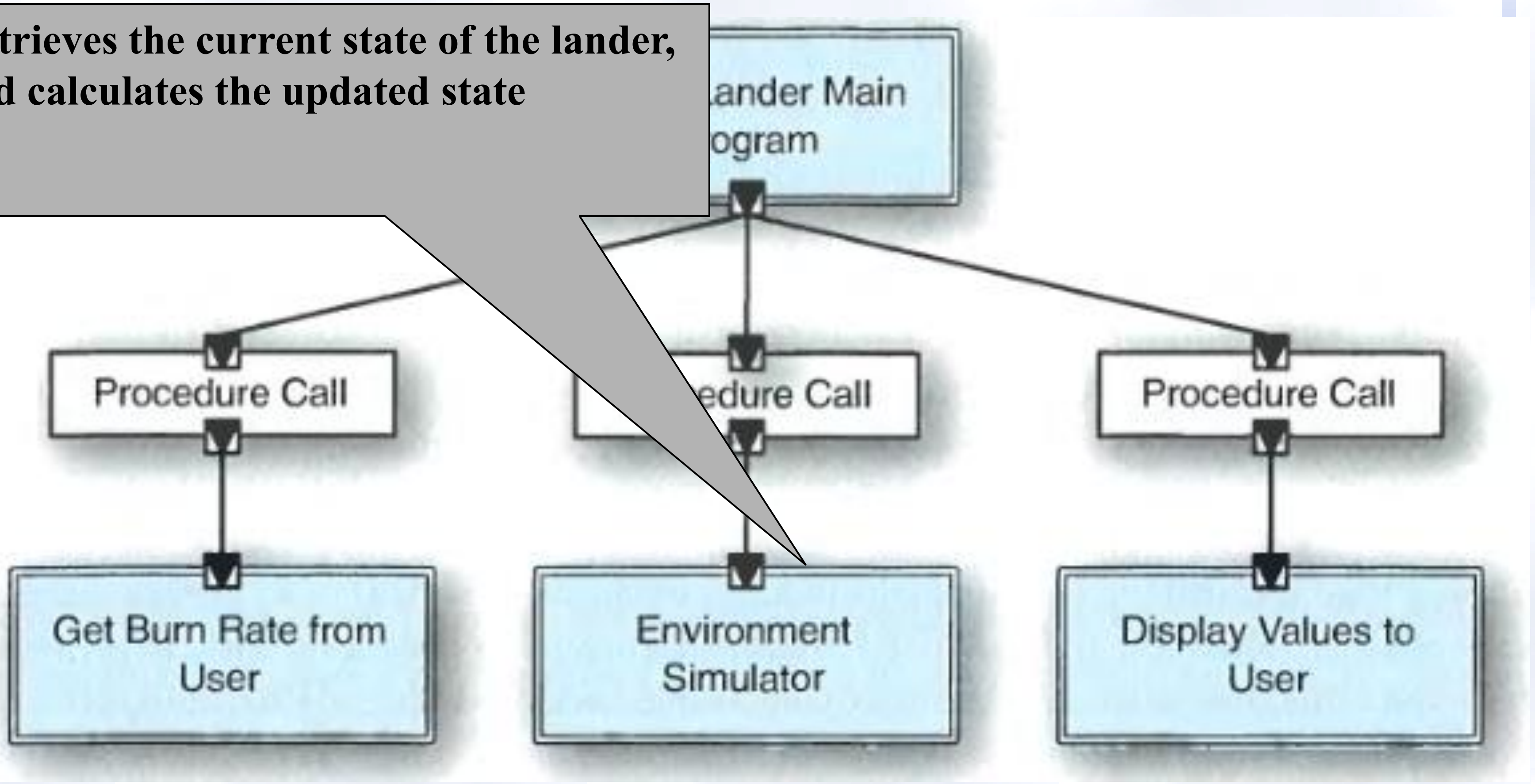
- Provides a functional decomposition into steps based on the user’s interaction with the application.



Main Program and Subroutines

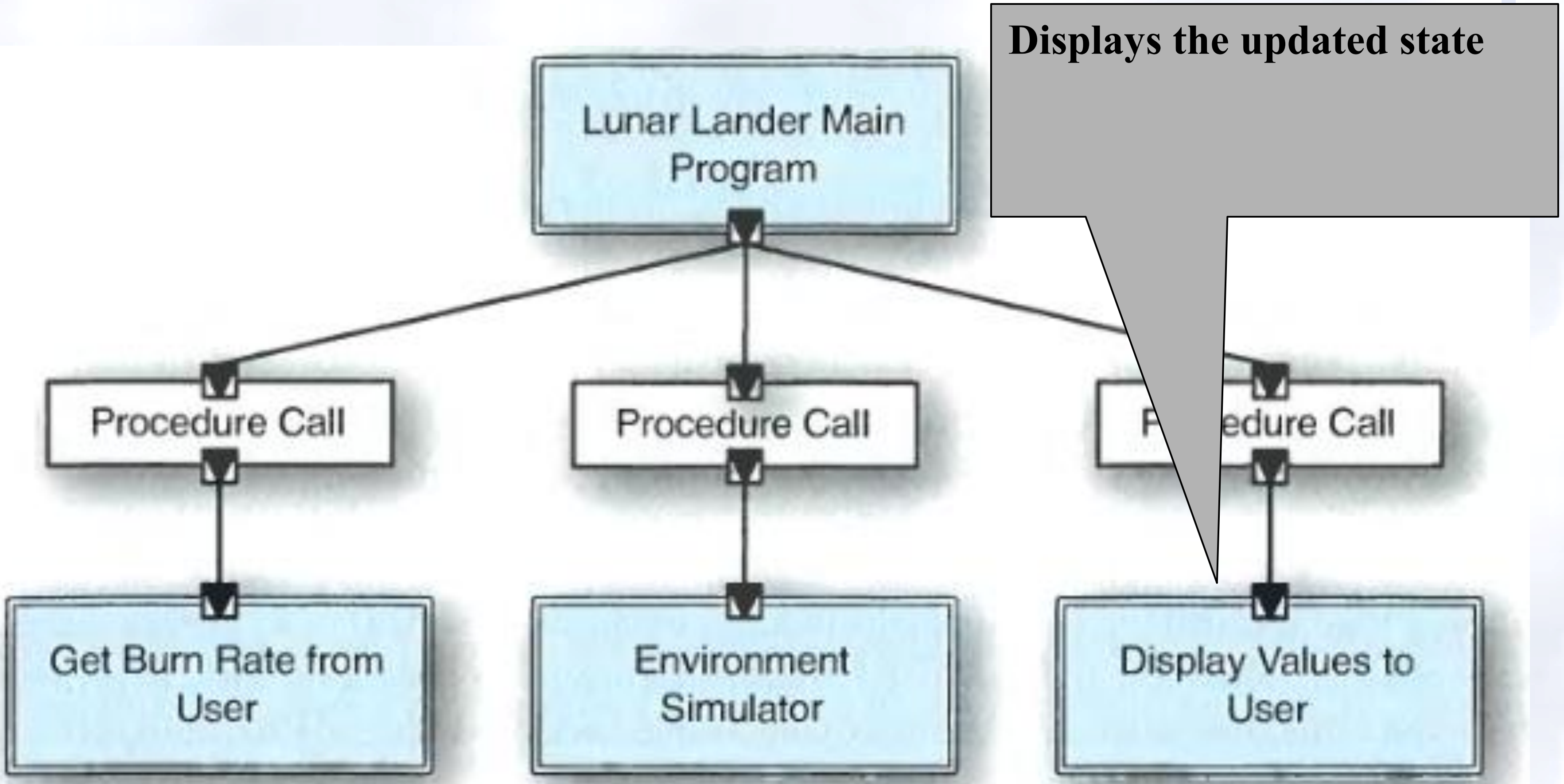
- Provides a functional decomposition into steps based on the user's interaction with the application.

Retrieves the current state of the lander, and calculates the updated state



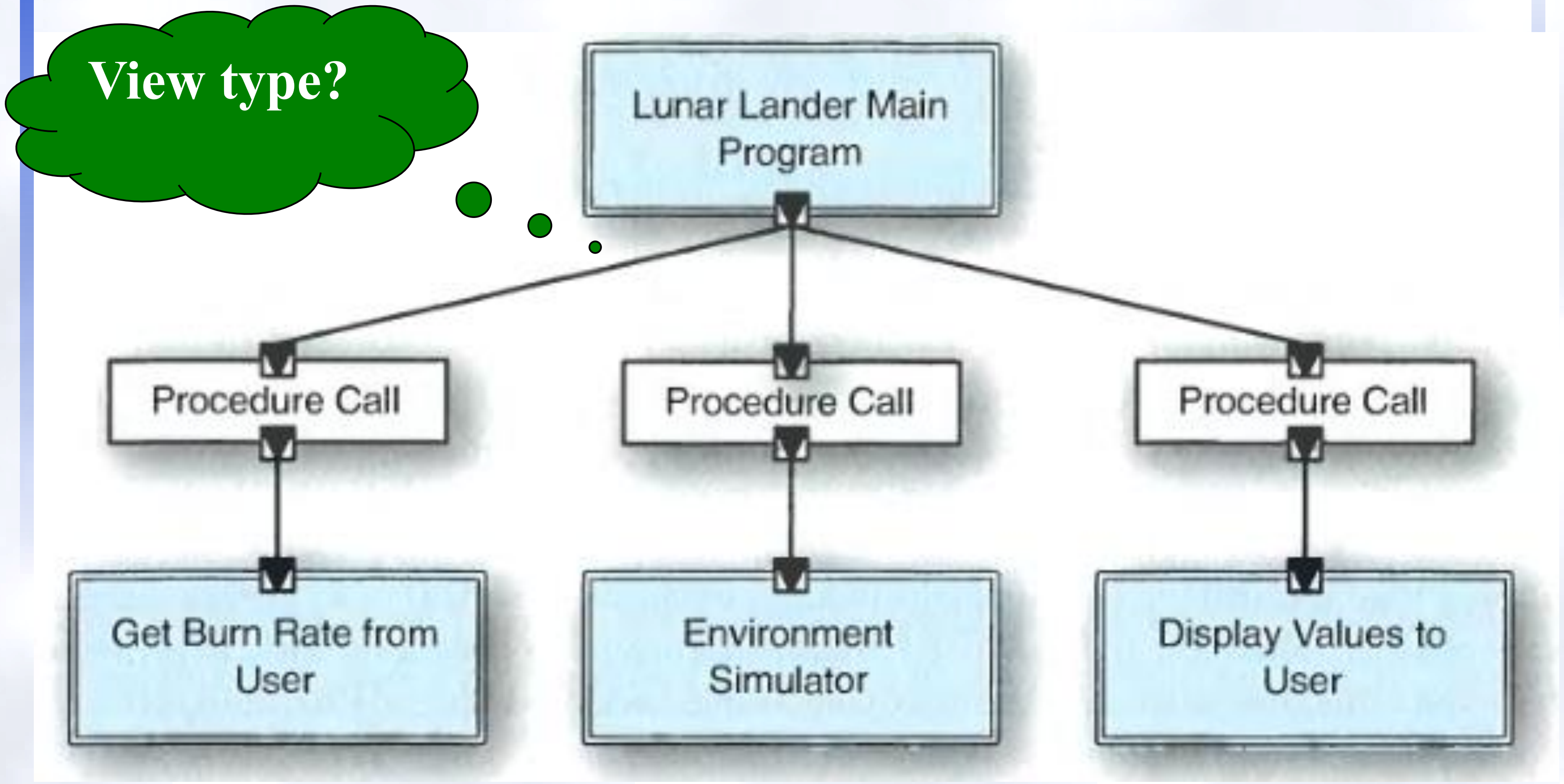
Main Program and Subroutines

- Provides a functional decomposition into steps based on the user's interaction with the application.



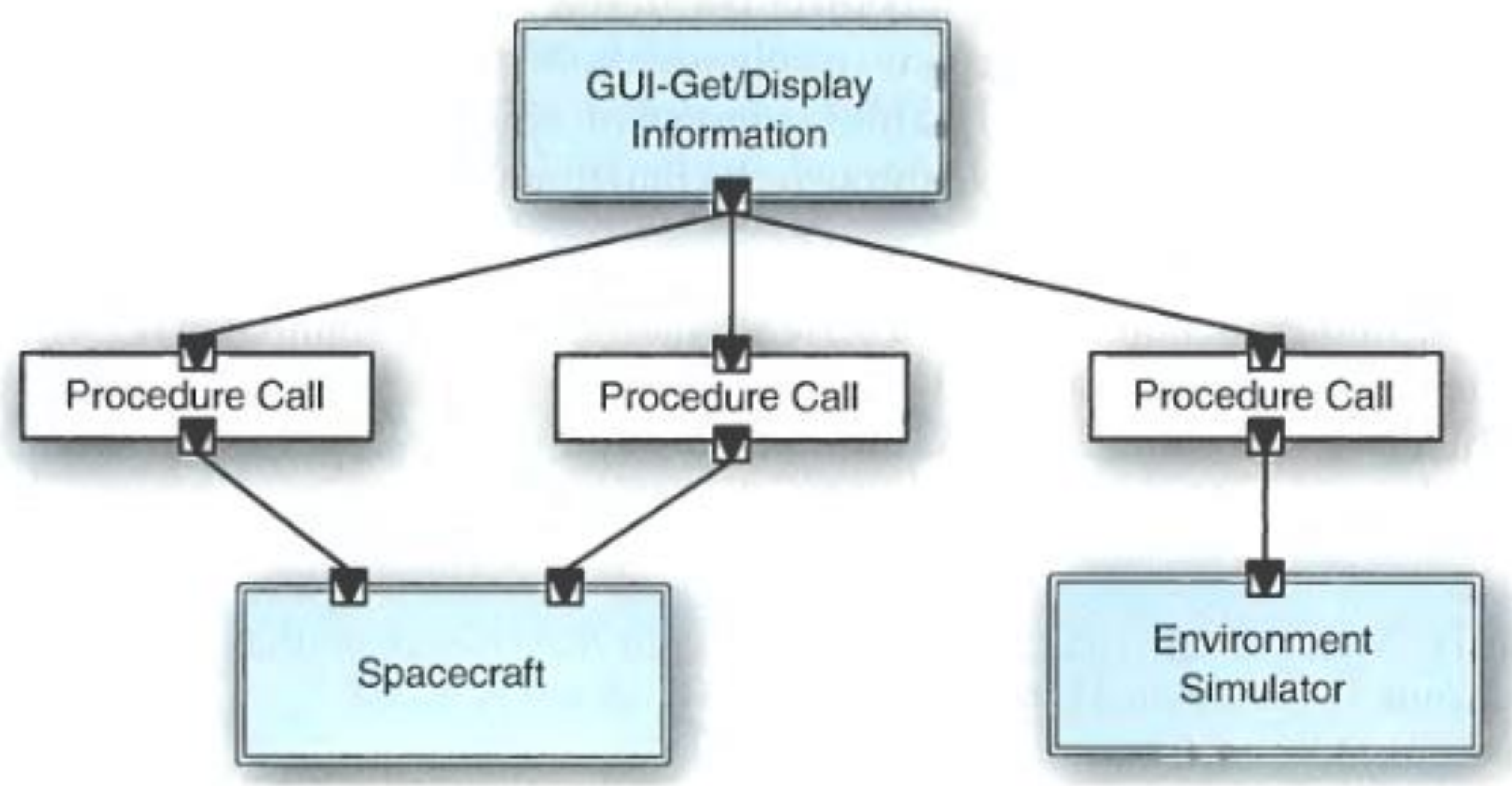
Main Program and Subroutines

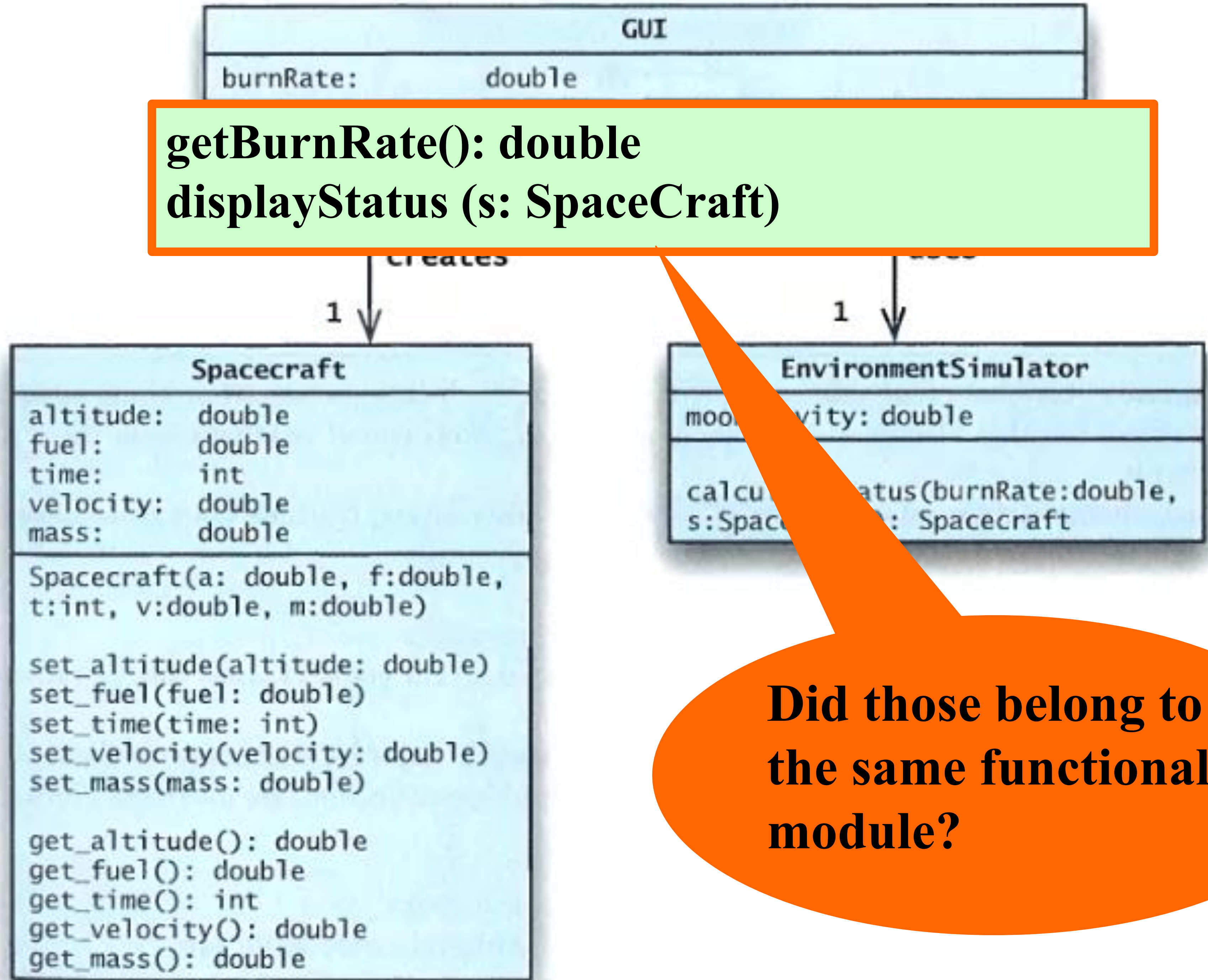
- Provides a functional decomposition into steps based on the user's interaction with the application.



Object-Oriented Lunar Lander

- State strongly encapsulated with functions that operate on that state as objects.





getBurnRate(): double
displayStatus (s: SpaceCraft)

Did those belong to the same functional module?

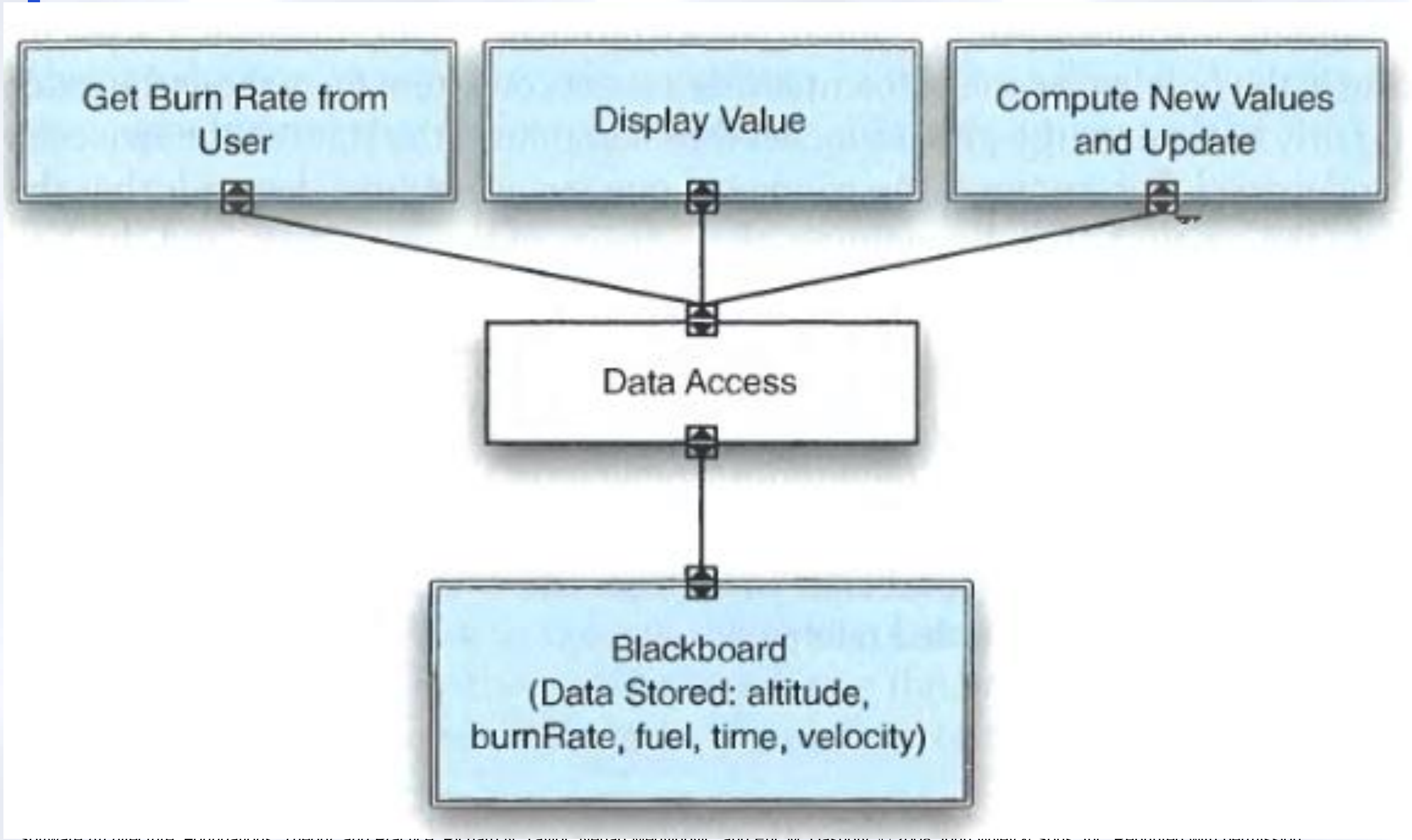
Object-Oriented Lunar Lander

- Qualities gained:
  - **Integrity of data operations:** (data manipulated only by appropriate functions).
  - **Abstraction** (implementation details hidden).
- Cautions?
  - Use in distributed applications requires extensive middleware to provide access to remote objects.
- When to use it?
  - Close mapping between external/internal entities.
  - Complex data structures.
- When to avoid it?

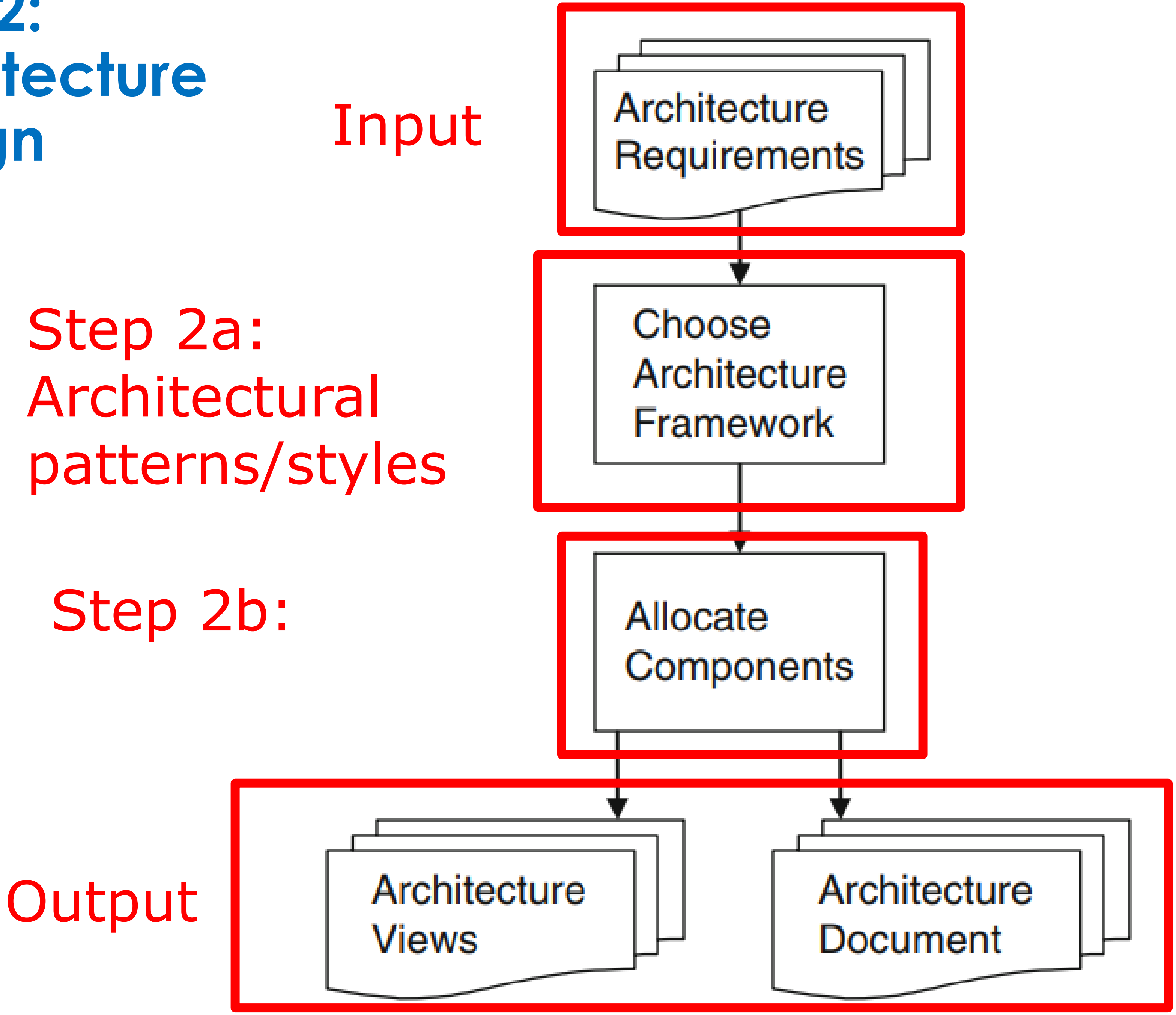
Shared Memory

- Multiple components have access to the same data store, and communicate through that data store.
- The center of design attention is explicitly on these structured, shared repositories, and that consequently they are well ordered and carefully managed.

Shared Memory


Get Burn Rate from User
Display Value
Compute New Values and Update
Data Access
Blackboard (Data Stored: altitude, burnRate, fuel, time, velocity)

Step 2: Architecture Design


Input
Architecture Requirements
Step 2a: Architectural patterns/styles
Choose Architecture Framework
Step 2b:
Allocate Components
Output
Architecture Views
Architecture Document

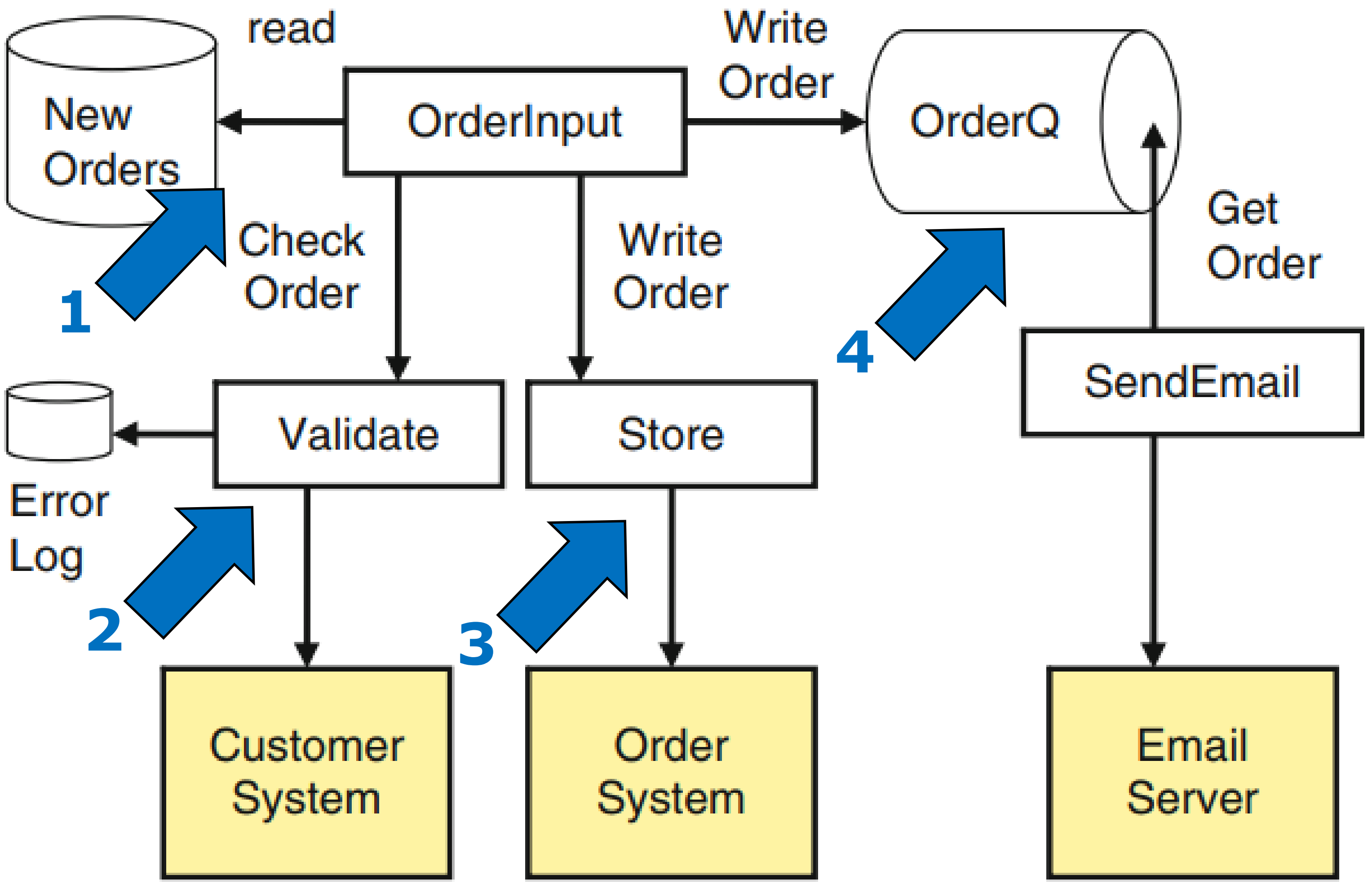
Step 2b: Allocate Components

- Once an overall architecture pattern has been selected, the next task is to define the major components that will comprise the design.
- The chosen pattern must be augmented by:
  - Identifying the major application components, and how they plug into the framework.
  - Identifying the interface or services that each component supports.
  - Identifying the responsibilities of the component.
  - Identifying dependencies between components.
  - Identifying partitions in the architecture that are candidates for distribution over servers in a network.

Step 2b: Order Processing Example Architecture

- New orders are received (from where is irrelevant) and loaded into a database.
- Each order must be validated against an existing customer details system to check the customer information and that valid payment options exist.
- Once validated, the order data is simply stored in the order processing database
- An email is generated to the customer to inform them that their order is being processed.

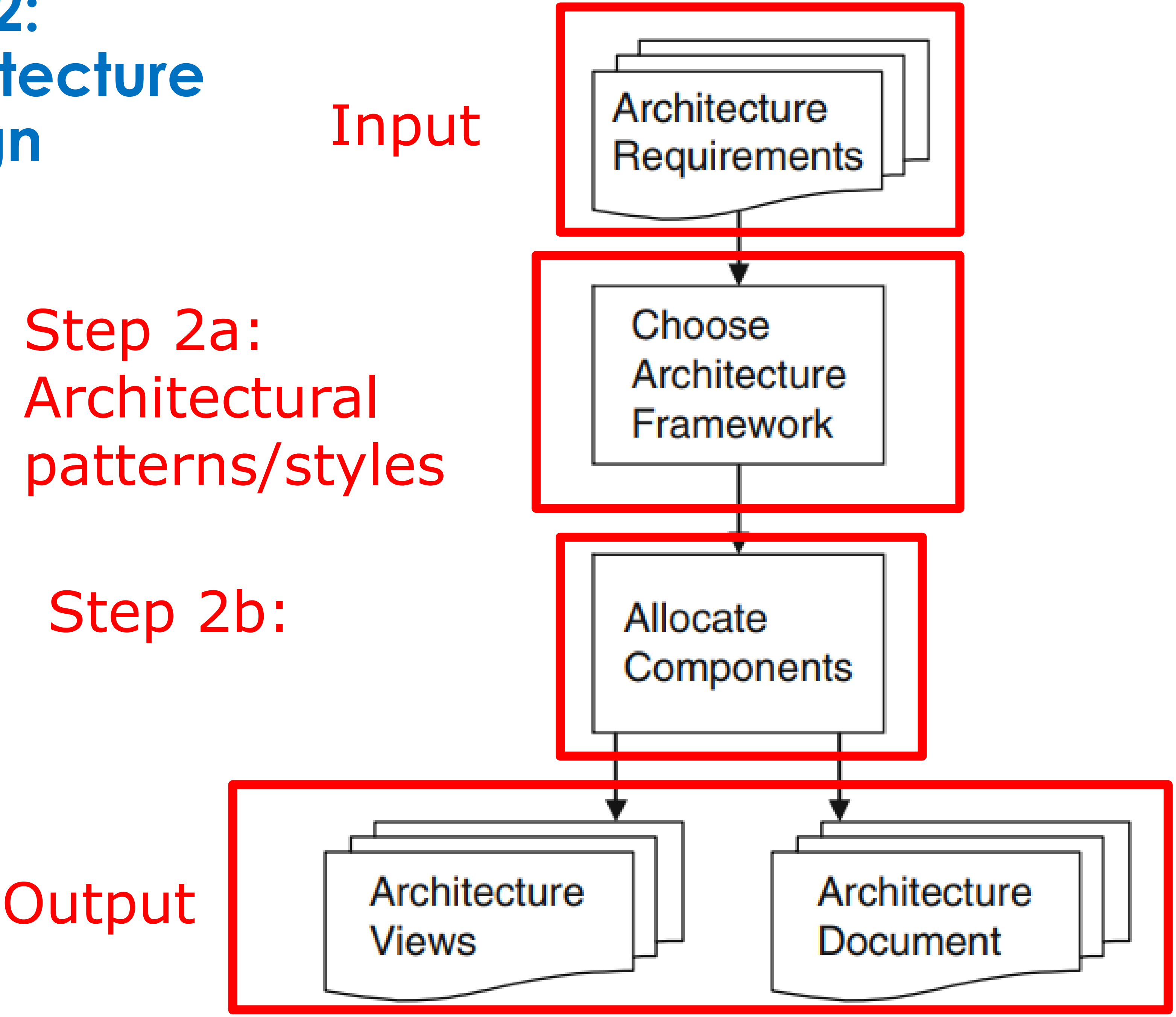
Step 2b: Order Processing Example Architecture



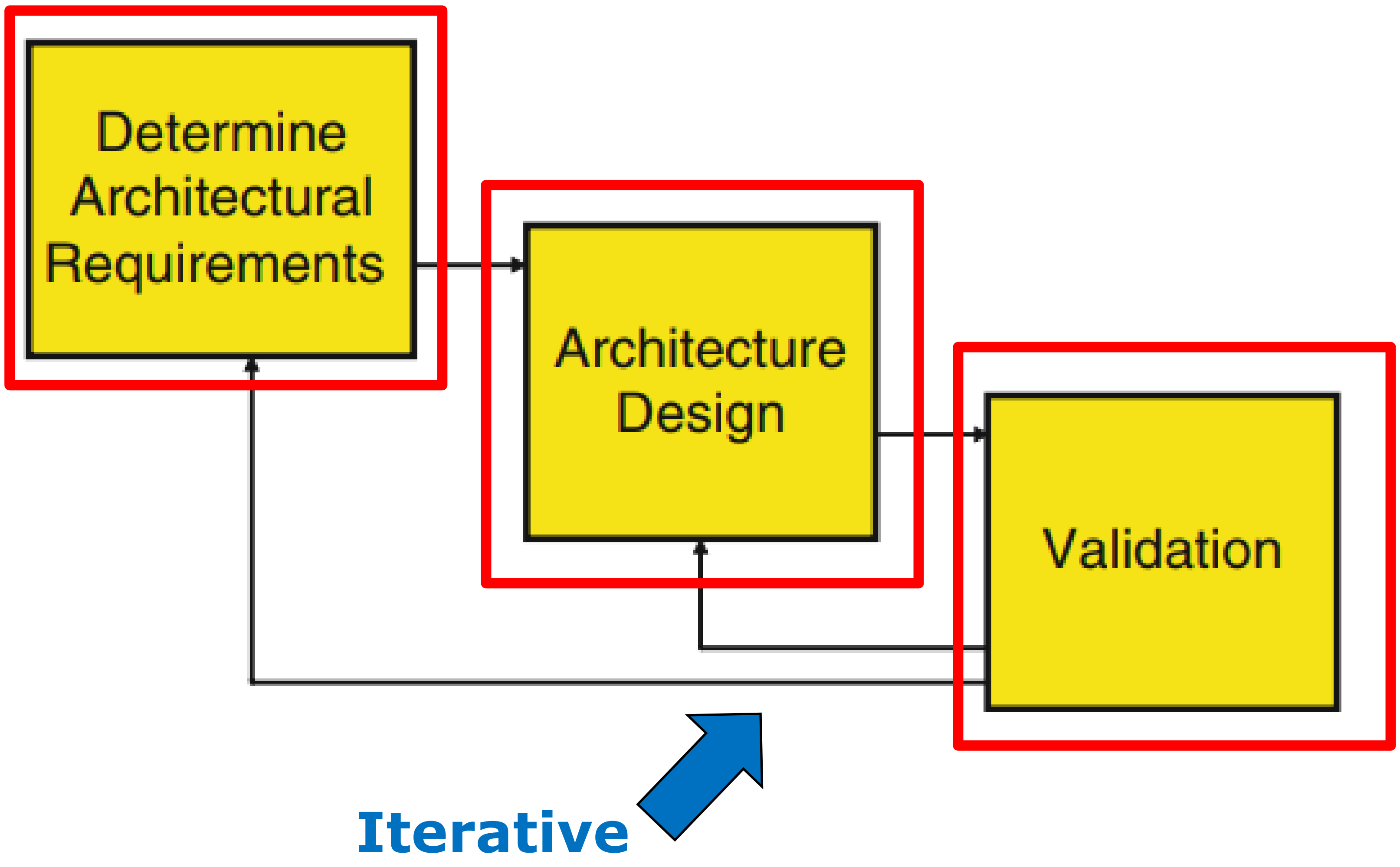
Step 2b: Order Processing Example Architecture

- The general architecture framework is based on straightforward messaging.
- Information about each valid order is removed from the queue, formatted as an email and sent to the customer using the mail server.
- Using a message queue this architecture decouples the order processing from the email formatting and delivery.

Step 2: Architecture Design


Input
Architecture Requirements
Step 2a: Architectural patterns/styles
Choose Architecture Framework
Step 2b:
Allocate Components
Output
Architecture Views
Architecture Document

The Architecture Process Outline



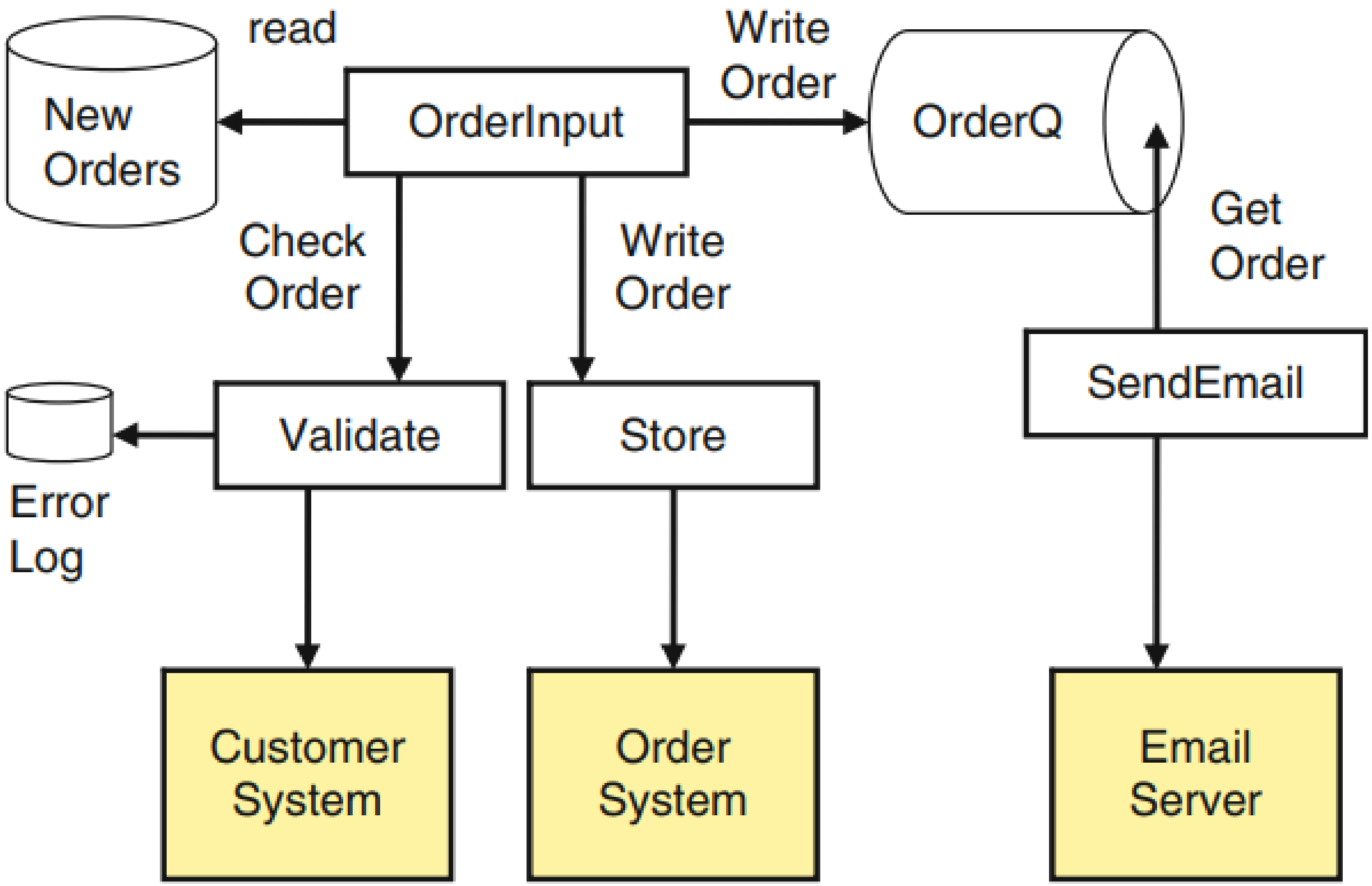
Step 3: Architecture Validation

- During the architecture process, the aim of the validation phase is to increase the confidence of the design team that the architecture is fit for purpose.
- Challenges?
- Whether it's the architecture for a new application, or an evolution of an existing system, the proposed design is just a design.
- It can't be executed or tested to see that it fulfills its requirements.
- It will also likely consist of new components that still have to be built

Step 3: Architecture Validation

- What can be done?
- Two main techniques:
  - Manual testing using test scenarios (SEI ATAM)
  - Construction of a prototype that creates a simple archetype of the desired application (prototype testing)

Step 3: Architecture Validation



Step 3: Architecture Validation – Scenarios

- Scenarios are a technique developed at the SEI to tease out issues concerning an architecture through manual evaluation and testing.
- Scenarios are related to architectural concerns such as quality attributes.
- Scenarios aim to highlight the consequences of the architectural decisions that are encapsulated in the design.

Step 3: Architecture Validation – Scenarios

- Scenarios involve:
- Defining some kind of stimulus that will have an impact on the architecture.
- The scenario then involves working out how the architecture responds to this stimulus.
- If the response is desirable, then a scenario is deemed to be satisfied by the architecture.
- If the response is undesirable, or hard to quantify, then a flaw or at least an area of risk in the architecture may have been uncovered.

Step 3: Architecture Validation – Scenarios

- Example:
  - Quality attribute: Availability
  - Stimulus: The network connection to the message consumers fails
  - Response: Messages are stored on the MOM server until the connection is restored. Messages will only be lost if the server fails before the connection comes back up
  - Is such response desirable or not?
    - Example?

Step 3: Order Processing Architecture Validation through Scenarios

- Scenario 1
  - Quality attribute: Modifiability
  - Stimulus: The Customer System packaged application is updated to an Oracle database
  - Response: The Validate component must be rewritten to interface to the Oracle system
  - *Is such response desirable?*

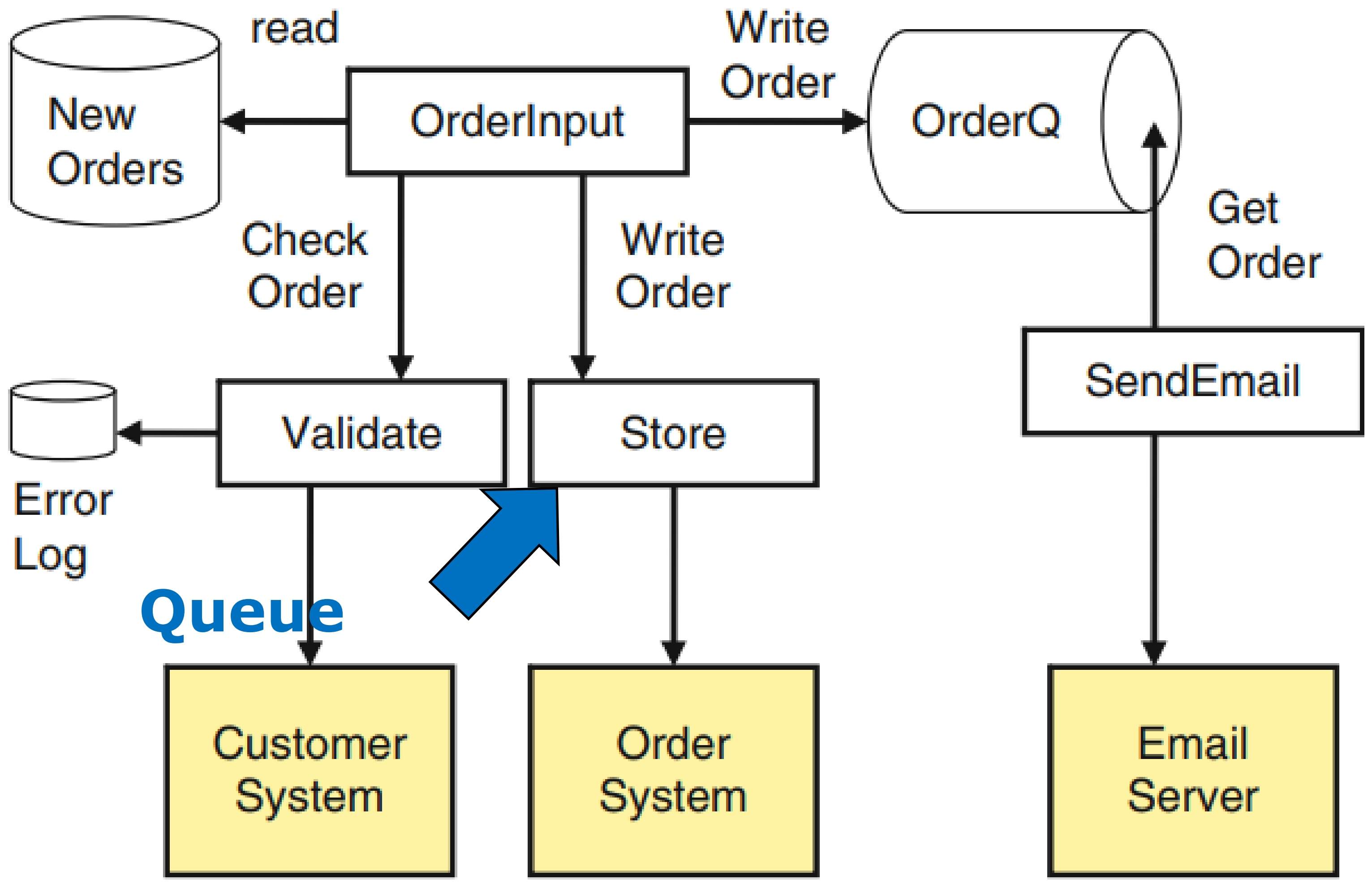
Step 3: Order Processing Architecture Validation through Scenarios

- Scenario 2
  - Quality attribute: Availability
  - Stimulus: The email server fails
  - Response:
    - Messages build up in the OrderQ until the email server restarts.
    - Messages are then sent by the SendEmail component to remove the backlog. Order processing is not affected.
  - *Is such response desirable?*

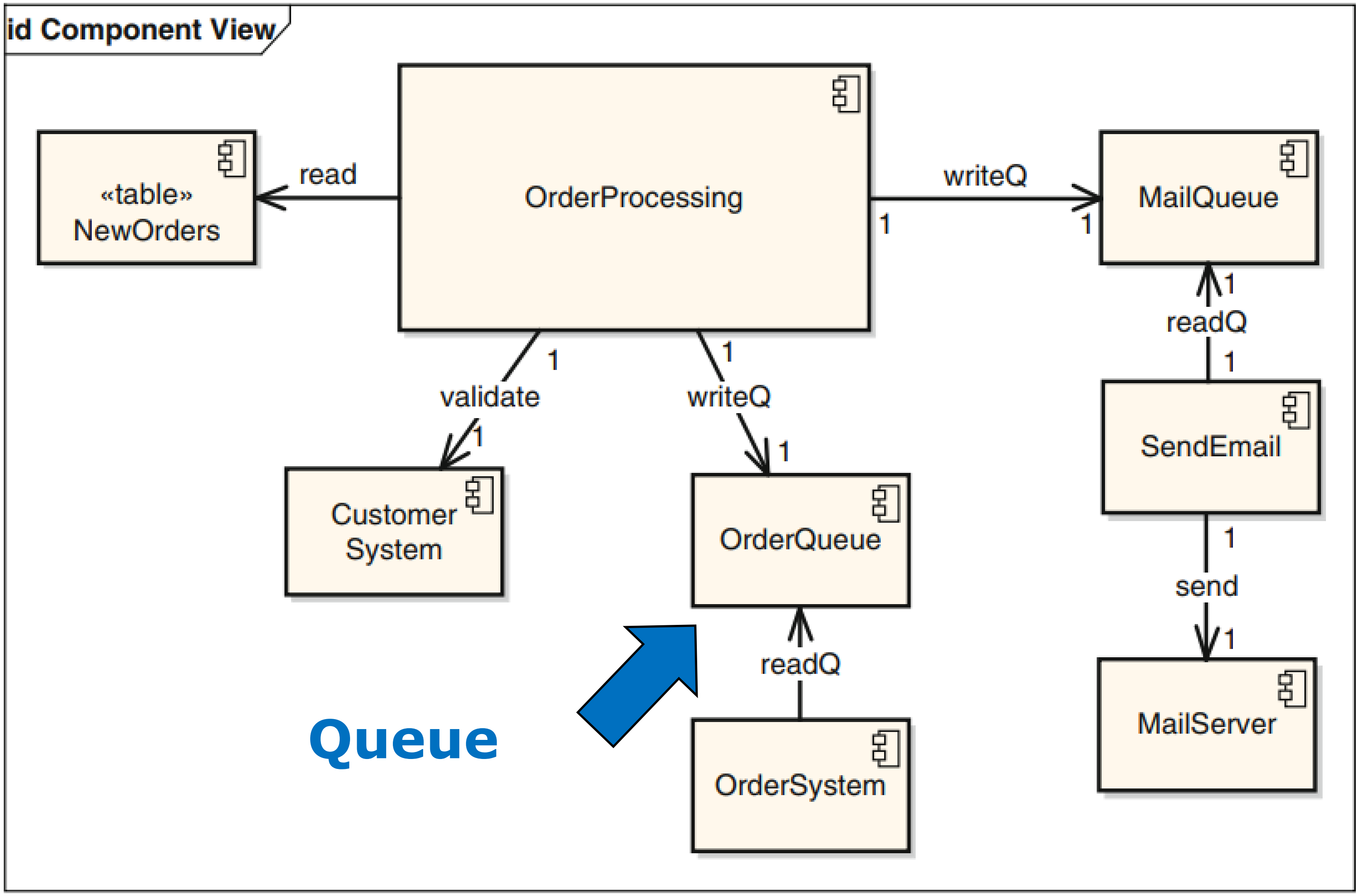
Step 3: Order Processing Architecture Validation through Scenarios

- Scenario 3
  - Quality attribute: Reliability
  - Stimulus: The Customer or Order systems are unavailable
  - Response:
    - If either fails, order processing halts and alerts are sent to system administrators so that the problem can be fixed
      Is such response desirable?

Step 3: Architecture Validation

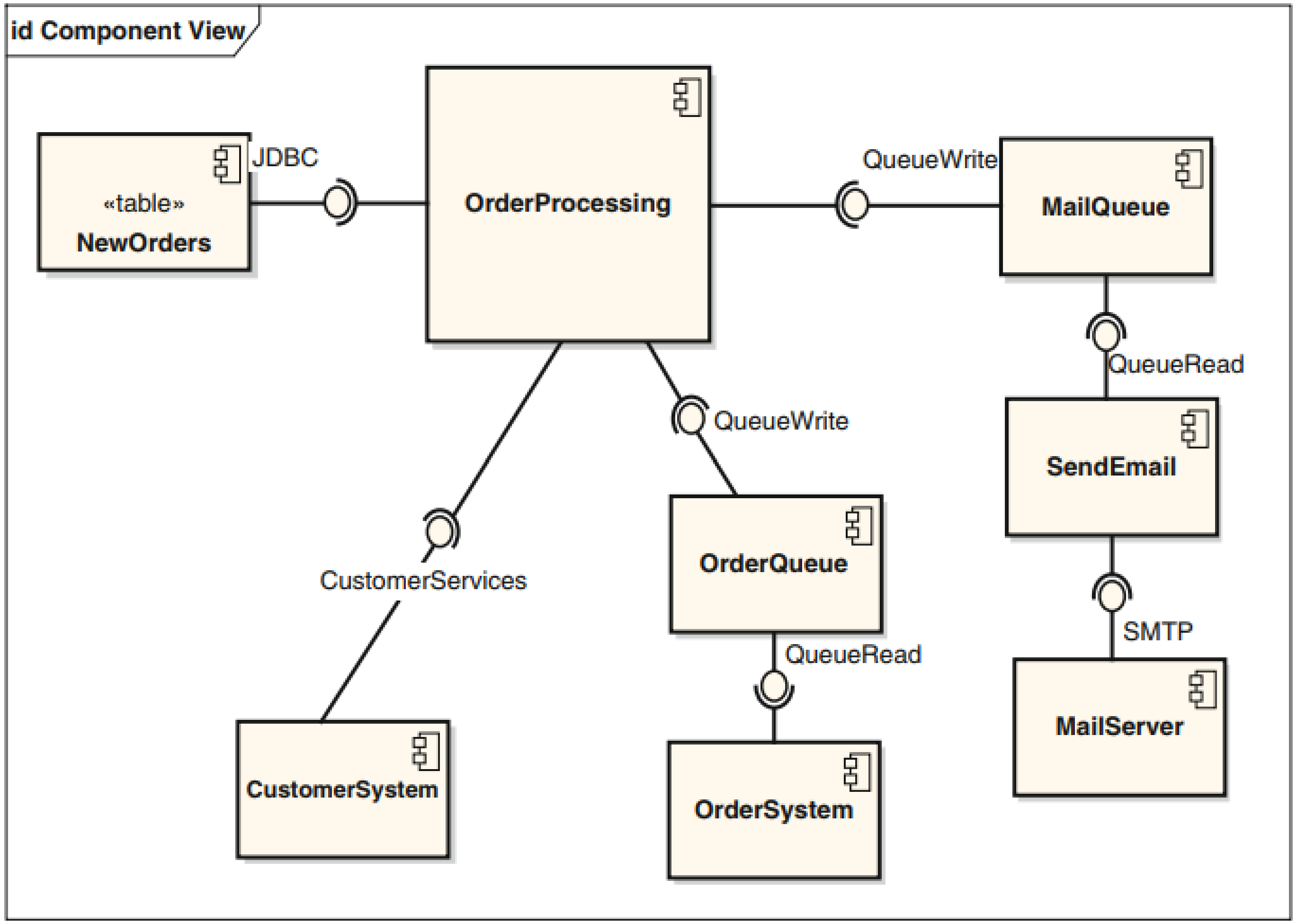


Step 3: Architecture Validation



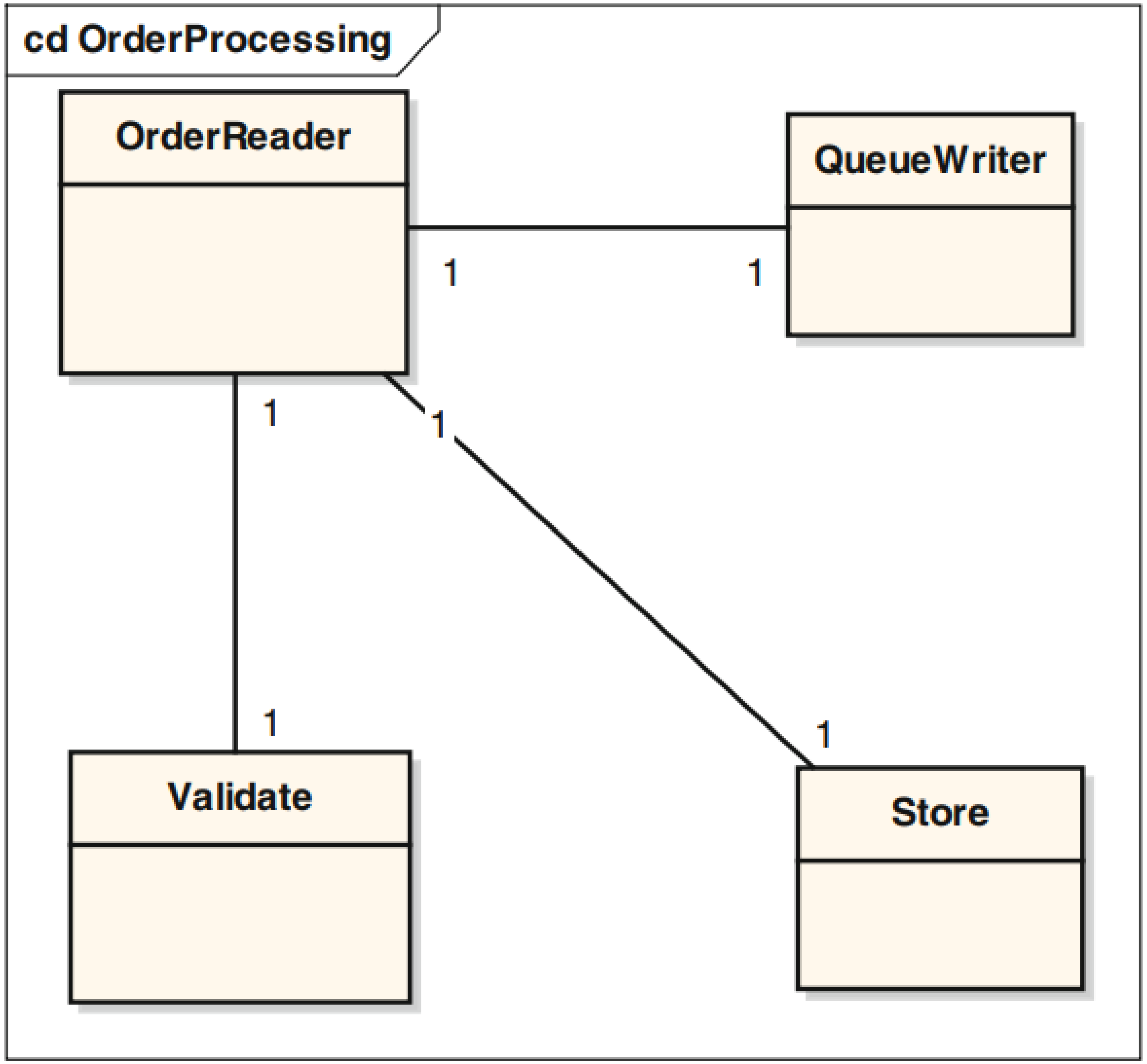
UML-based Architectural Views

UML component diagram for the order processing example



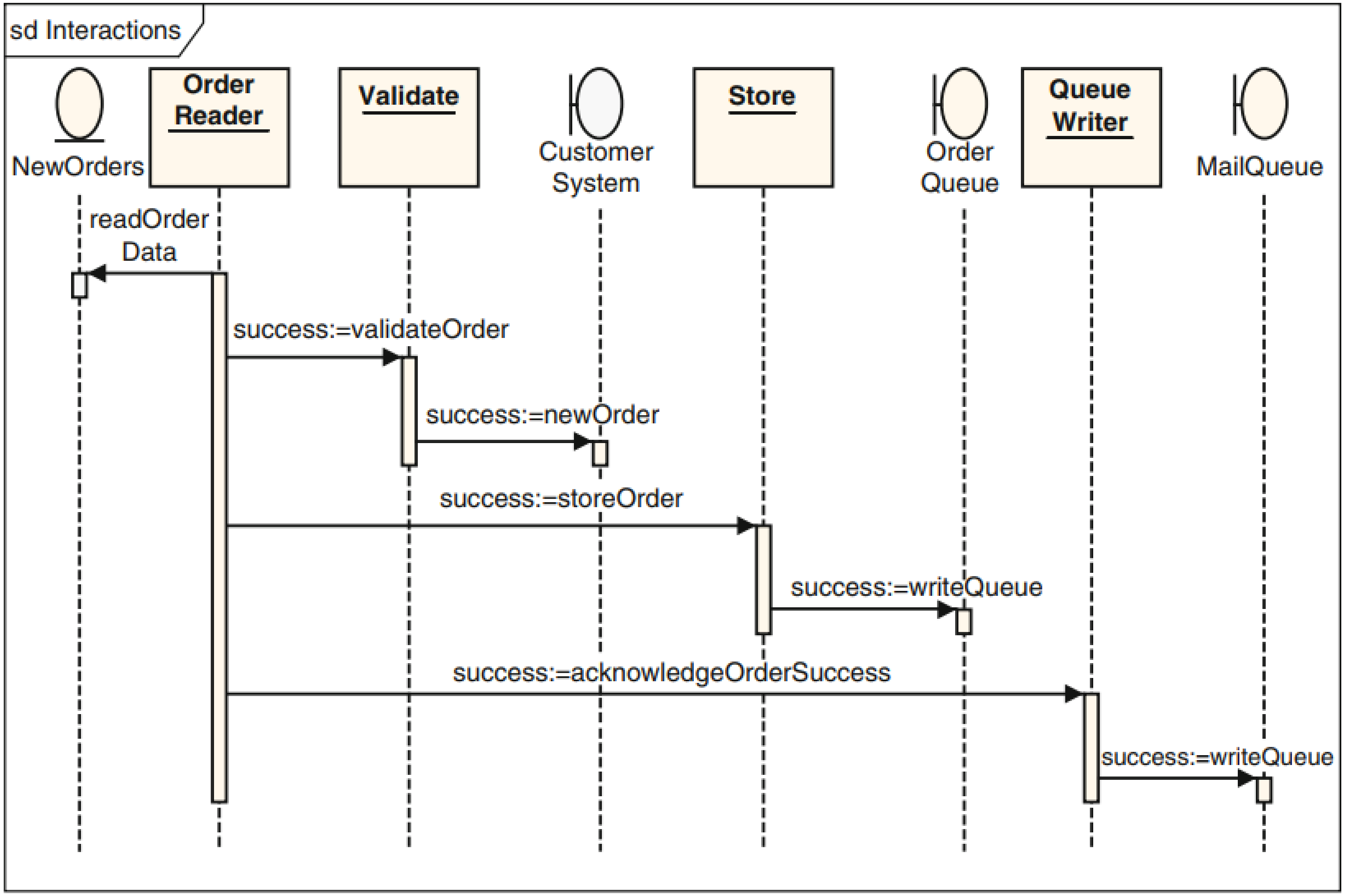
UML-based Architectural Views

Classes for the order processing component

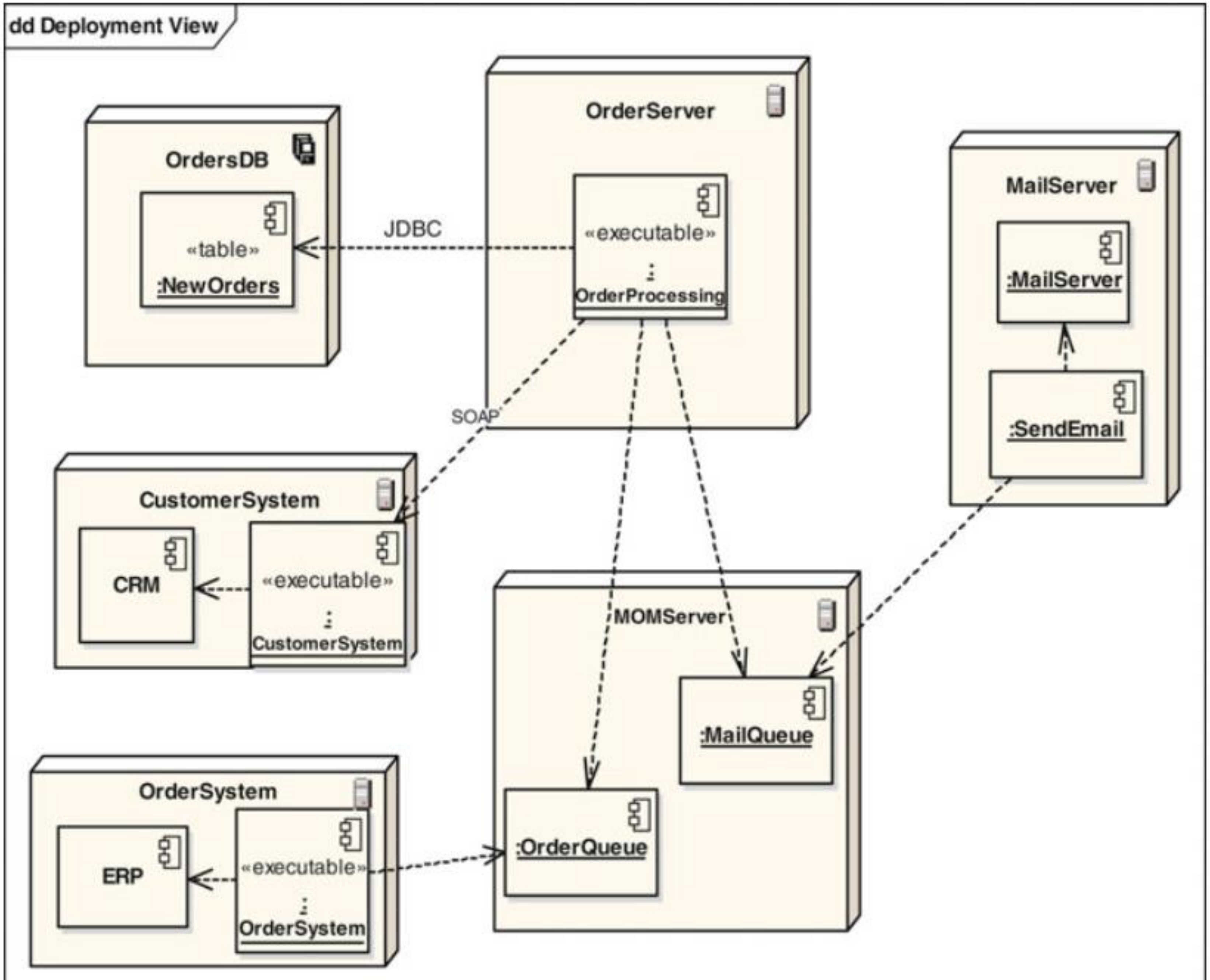


UML-based Architectural Views

Sequence diagram for the order processing system



UML Deployment diagram



Required Readings

- Chapters 7 and 8 from “Essential Software Architecture” textbook of Ian Gorton, 2011 Edition.