Viewpoint: Functional

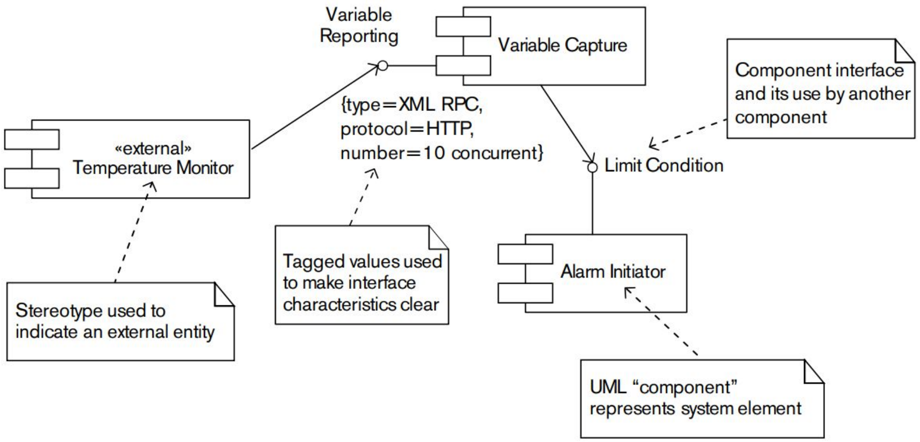
CSCE 742 -
Lecture 8/9 - 09/27 and 10/02/2018

The Functional Viewpoint

- The **functional view** of a system defines the architectural elements that deliver the functions of the system being described.
- Documents the system's functional structure:
  - Key functional elements and their responsibilities.
  - The interfaces they expose (internal/external).
  - The interactions between them.
- This view demonstrates how the system will perform the functions required of it.

The Functional Viewpoint

- Cornerstone of the architectural description.
  - Drives the definition of Information, Concurrency, Development, and Deployment Views.
- Should offer structure to guide design without placing too many constraints.
  - Does not detail physical infrastructure.



Key Attributes

- Concerns:
  - Functional capabilities, external interfaces, internal structure, and functional design philosophy.
- Models: Functional structure model
- Problems and Pitfalls:
  - Poorly defined interfaces, poorly understood responsibilities, infrastructure modeled as functional elements, overloaded view, diagrams without definitions, reconciling stakeholders, wrong level of detail, and too many dependencies.
- Stakeholders: All

Today's (and Next) Class

- Introduce the Functional Viewpoint.
  - How to specify and document.
  - Introduce the building blocks: elements, interfaces, and connectors.
    - The structures, what they can do, and how they interact.
  - Visualization using UML Context Diagrams
  - Pitfalls of functional architectural design.

Concern: Functional Capabilities

- Define what the system is - and is not - required to do.
- Some projects have a agreed set of requirements.
  - Functional View can focus on showing how the elements provide functionality.
- Some projects will require a clear statement of system capabilities.

Concern: External Interfaces

- Define data, event, and control flow between your system and others.
- Data can flow in or out.
  - Causing or caused by internal state change.
- Events can be consumed or emitted.
  - Notifications for your system or issued to others.
- Control can be inbound or outbound.
  - Requests to or made by your system.
- Interface definitions must consider syntax and semantics.

Concern: Internal Structure

- How do you construct your system?
  - How much code do you create yourself?
  - What is provided by external systems?
  - What middleware do you use?
  - What libraries do you import?
- Internal structure defined by:
  - The elements
  - What they do (how do they map to requirements?)
  - How they interact
- Choice impacts quality properties.

Functional View Elements

- Functional Elements
  - A well-defined part of the system that has responsibilities and a defined interface.
  - Software subsystem, cluster of classes, a package, a data store, a complete system.
- Interfaces
  - A well-defined mechanism by which the functions of an element can be accessed by other elements.
  - Defined by inputs, outputs, and semantics of each operation.

Functional View Elements

- Connectors
  - Define the interactions between elements.
  - Can be simple or complex.
    - Simple: A calls B.
    - More complex: message-passing (the message could be a type of element).
- External Entities
  - Other systems, software, hardware devices, etc. that interact with your system.
  - Also often have responsibilities and a defined interface.

Functional View Elements

- Does not define how code is packaged and executed in processes or threads.
  - Deployment and Concurrency Views
- Does not depict underlying infrastructure.
  - I.e., server or networking infrastructure.
    - Deployment View
  - Might show things like message queues that are interelement connectors.
    - But you would omit the message broker that provides the queues - not relevant to functionality.

Functional Design Characteristics

- Coherence
  - Does the architecture have a logical structure with elements working together to form a whole?
- Cohesion
  - To what extent are the functions provided by an element strongly related to each other?
- Consistency
  - Are mechanisms and design decisions applied consistently throughout the architecture?

Functional Design Characteristics

- Coupling
  - How strong are the element interrelationships?
  - Do changes in one element affect others?
- Extensibility
  - Will the architecture be easy to extend to allow the system to perform new functions in the future?
- Functional Flexibility
  - How amenable is the system to supporting functional changes?
- Generality
  - Are the mechanisms and decisions in the architecture as general as is practicable?

Functional Design Characteristics

- Interdependency (Volume of Element Interactions)
  - What proportion of processing involves interactions between elements versus within elements?
- Separation of Concerns
  - To what extent is each internal element responsible for a distinct part of the system's operation?
  - To what extent is common processing performed in only one place?
- Simplicity
  - Are the design solutions used within the system the simplest ones that would be suitable?

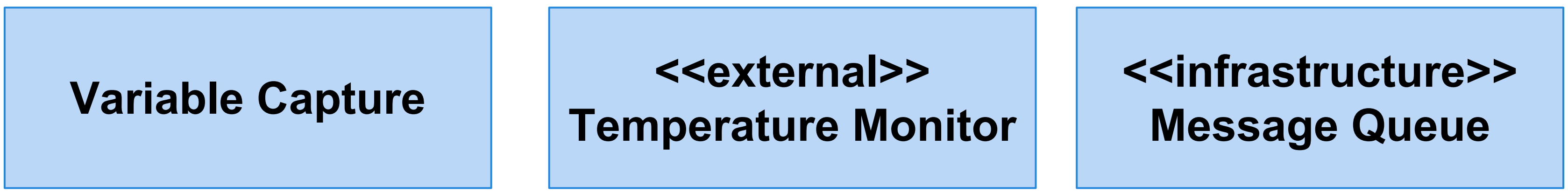
Functional Design Characteristics

- Achieving characteristics affects quality.
  - High cohesion, low coupling improve modifiability.
  - Separation of concerns, simplicity improve security.
  - Consistency improves performance, scalability.
- However, “good” design can also negatively impact qualities.
  - Low coupling may degrade performance by increasing the number of communication steps.
- Establish architectural principles to prioritize favored design characteristics.

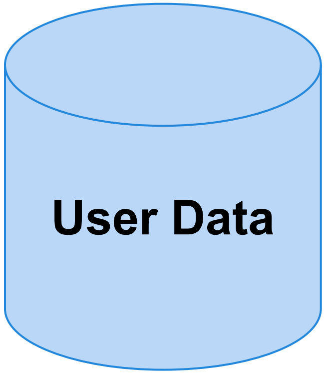
Documenting the Functional View

Functional View Notation

- UML Component Diagram
- Elements

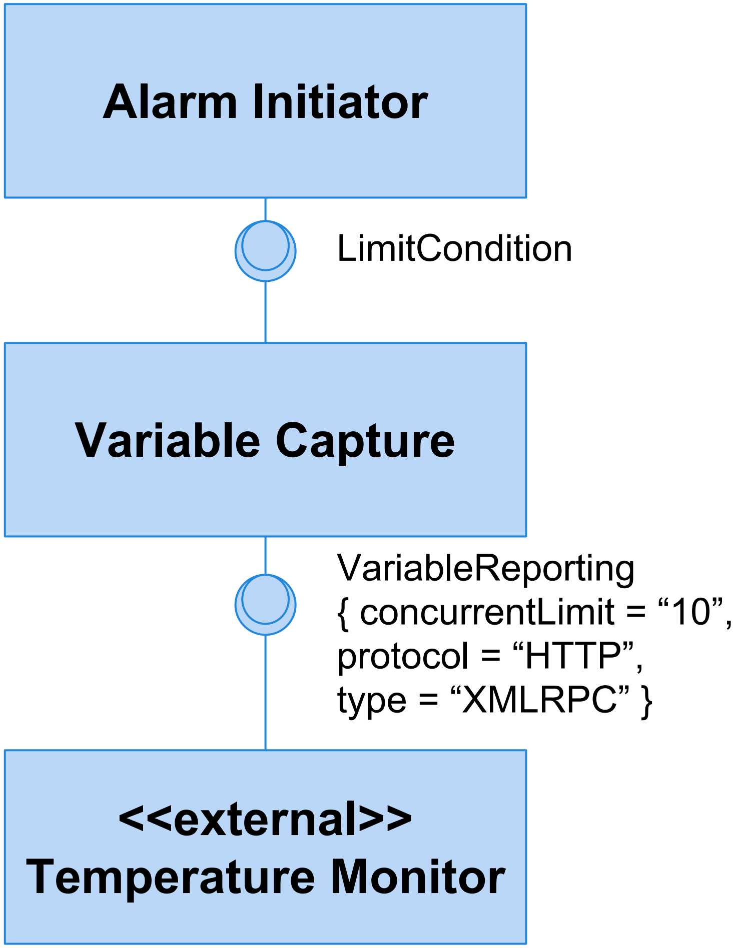


- Data Stores



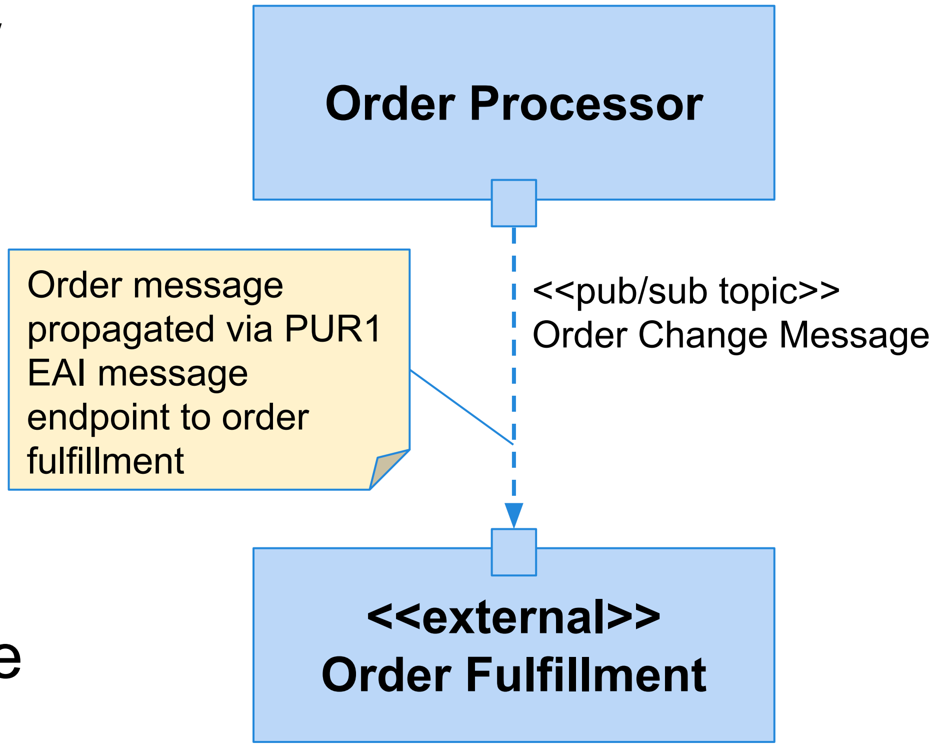
Functional View Notation

- Interfaces
  - Attached to an element (half-circle facing away from its element)
  - Named and tagged with attribute-value pairs that characterize the interface.

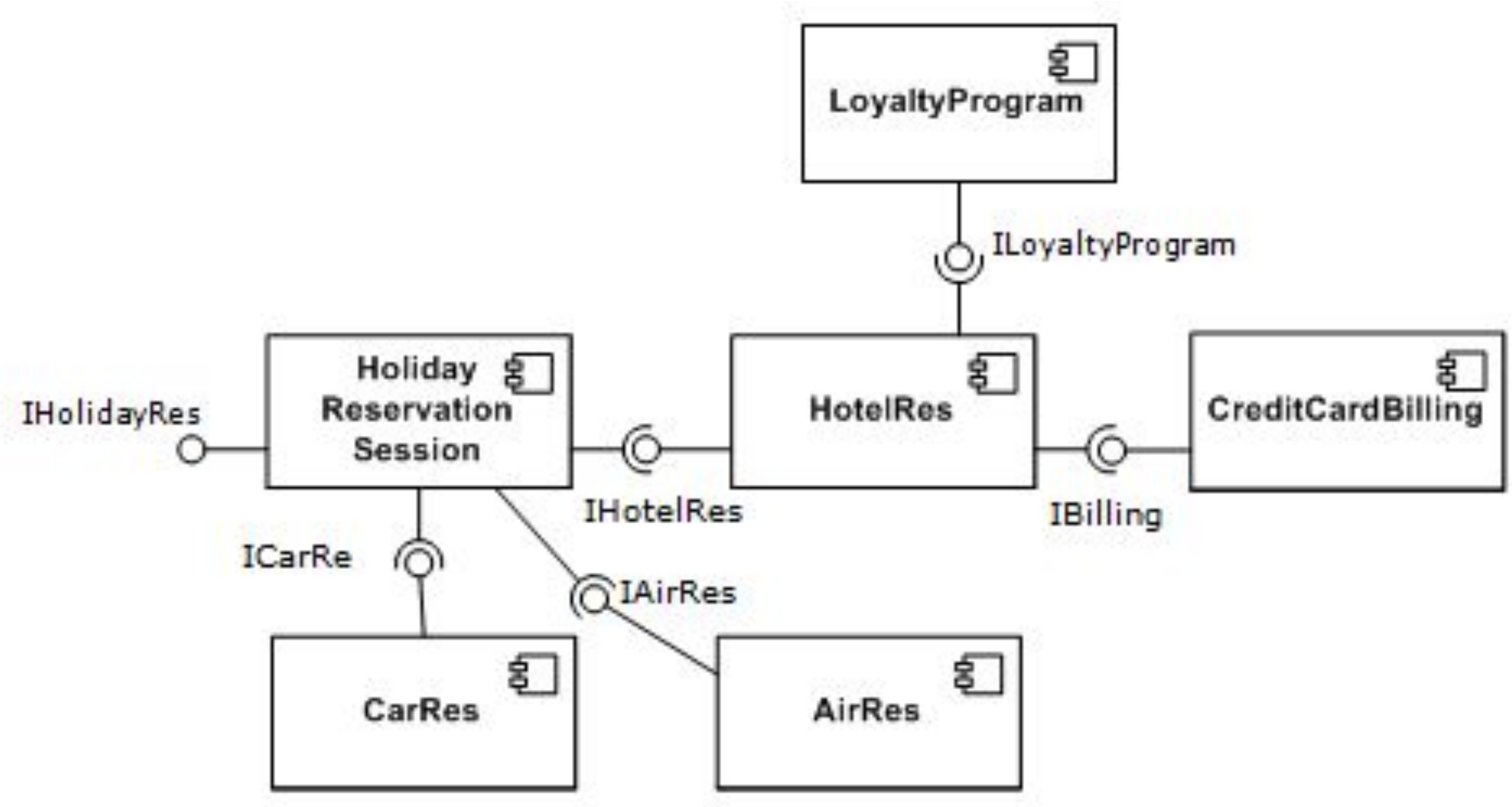


Functional View Notation

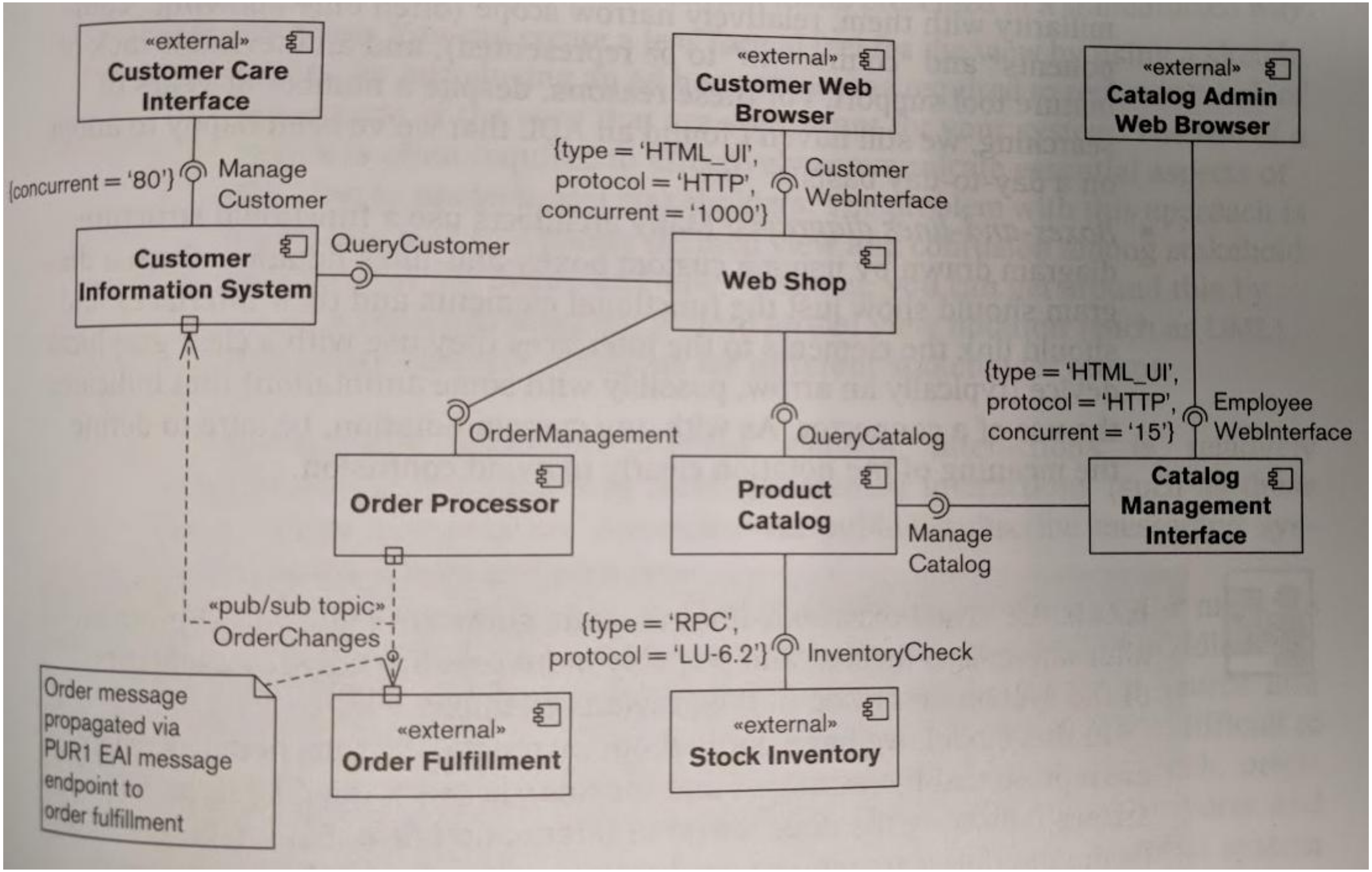
- Information Flow
  - Elements may exchange data outside of direct connectors.
  - Show flow of information (i.e., messages)
  - Boxes at ends are message ports (sending/receiving, based on direction)



Example - Vacation Reservations

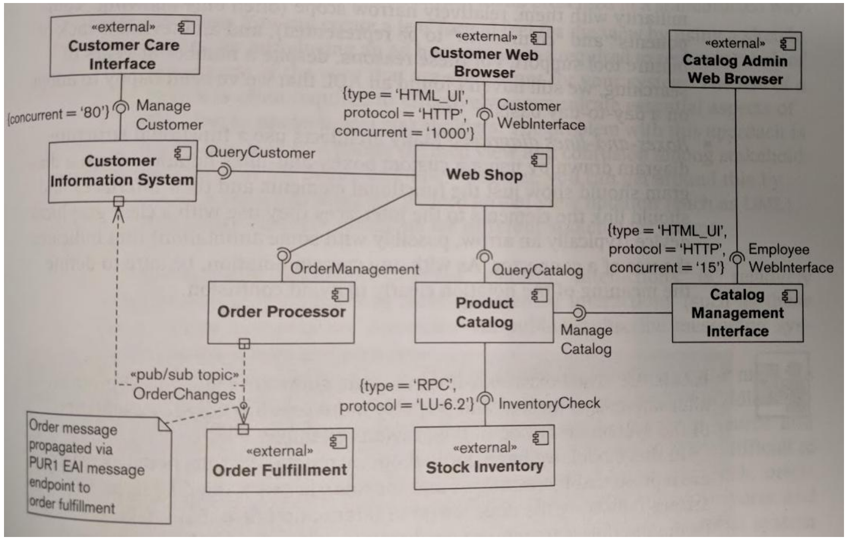


Example - Web Store



Example - Web Store

- What this tells us
  - Up to 1000 customers, 80 care reps, 15 admins may access the system at once.
  - Interaction between Product Catalog and Stock Inventory takes place using a specific protocol.



- Unadorned communication takes place via a standard remote procedure call.

Example - Web Store

- What this model does NOT tell us
  - Element responsibilities are not clear.
  - Details of interfaces are not clear.
  - Details of how components interact are not clear.
- No one diagram will fill in all details.
  - This is an overview, detail all elements, interfaces, and connectors using text descriptions.
  - Scenario models can fill in additional details.

Elements

Element Definition

- Szyperski Definition:
  - An element has the following characteristic properties:
    - A software element is a unit of composition with contractually specified interfaces and explicit context dependencies only.
    - A software element can be deployed independently and is subject to composition by third party.

Element Definition

- Szyperski definition implies that:
  - For an element to be deployed independently, a clear distinction from its environment and other elements is required.
  - An element must have clearly specified interfaces.
  - The implementation must be encapsulated in the element and is not directly reachable from the environment.

Element Definition

- D’Souza and Wills definition:
  - An element is a reusable part of software, which is independently developed, and can be brought together with other elements to build larger units.
  - It may be adapted but may not be modified.
  - An element can be compiled code without a source.
- To describe an element completely, the element should consist of:
  - A set of interfaces provided to or required from the environment.
  - Executable code which can be coupled to the code of other elements through these interfaces.

Identifying Elements

1. Work through functional requirements, deriving key system-level responsibilities.
2. Identify the functional elements that will perform the responsibilities.
3. Assess against desired design criteria.
4. Iterate and refine until sound.
5. If an element is pre-defined (libraries or existing systems), understand their responsibilities and how they affect the architecture.

Refining the Element Set

- Generalization
  - Identify common responsibilities across elements, introduce new elements encapsulating those.
  - Allows reuse of elements across systems.
- Decomposition
  - Break complex elements into smaller subelements.
  - Often needed in large systems to produce manageable subsystem-level elements.

Refining the Element Set

- Amalgamation
  - Replace small elements with a larger element that includes all functions of smaller ones.
  - Group similar standalone elements into one to increase cohesion.
- Replication
  - Replicate either an element or a piece of processing.
  - Data validation might be repeated across multiple external interfaces.
  - Can bring performance benefits, but makes consistency difficult.

Assigning Responsibility

- Once elements are created, they must be assigned clear responsibilities.

Web Shop	• Present customers with an HTML-based user interface they can access with a Web browser. • Manage all state related to the customer interface session. • Interact with other parts of the system to allow customers to view the catalog and stock levels, buy goods, and view their customer information.
Customer Information System	• Manage all persistent information about customers of the system. • Provide a query-only interface that can be used to retrieve information held on a particular customer that should be visible to that customer. • Provide an information management programmatic interface that can be used to create customer information management applications. • Provide an event-driven message-handling interface to accept details of orders placed by customers and the state changes of those orders

Interfaces and Connectors

Interface Definition

- An interface of an element can be defined as a specification of its access point, offering no direct implementation for any of its operations.
  - The implementation can be replaced without replacing the interface.
  - New interfaces can be added without changing the existing implementation.

Interface Definition

- An interface defines a contract specifying how elements interact:
  - Set of participating elements
  - Role of each element, specified through its contractual obligations (i.e., data type).
  - Invariants to be maintained by elements.
    - Pre and post-conditions on calls to interface.
  - Specification of the methods that instantiate the contract.

Element Replacement

- Substituting element Y for element X is said to be safe if all systems that work with X will also work with Y.
- From a syntactic viewpoint, an element can safely be replaced if:
    - The new element implements at least the same interfaces as the older elements, or
    - The interface of the new element is a subtype of the interface of the old element.

Importance of Interfaces

- Architectural thinking depends on interfaces!
  - Partitioning
  - Structuring
  - Testability
  - Reuse
  - Portability
  - Scalability
  - All depend on the interfaces that you design or that are made available.

Designing the Interfaces

- The definition of an interface must include:
  - The operations that the interface offers
  - Inputs, outputs, pre-conditions, and post-conditions of each operation.
  - Nature of the interface
    - (messaging, procedure call, web service, etc.)
    - **Computational Interfaces:** Clients invoke defined functions.
    - **Data-oriented Interfaces:** Clients communicate through unidirectional data transfer.

Design by Contract

- Define an interface by establishing promises with the user of an element.
  - Pre-conditions: What the client must promise to the element in order to expect correct behavior.
    - (Binary Search: the input array must be sorted)
  - Post-conditions: What the element promises will happen on return.
    - (Quick Sort: the array will be sorted numerically)
  - Invariants: Conditions that will be met during execution of the operation.

Computational Interfaces

- Elements publish a set of operations that can be invoked. Clients call these operations to have the element perform them.
- Can be directly defined in a program.
    - Simple, but ties you to use of that language.
- Can be defined through interface definition languages (IDLs).
    - .NET IDL, CORBA IDL, Web Services Description Language (WSDL)
    - Programming language independent

Computational Interfaces

Responses are *synchronous* or *asynchronous*.

- Synchronous
  - Typical approach.
  - Acts like a function call in a normal program.
- Asynchronous
  - Type 1: Client provides an interface for callback when complete
  - Type 2: Client receives a token object (sometimes called a future) that will eventually hold output.

Interface Consistency

- We say a two elements have consistent interfaces if:
  - Interface names match.
  - Provided and required function lists match.
  - Function parameter lists match.

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

```
Provided 1: getSubQ(Index first, Index last)
returns FIFOQueue;
```

```
Provided 2: getSubQ(Natural first, Natural last,
Boolean remove) returns Queue;
```

Behavioral Consistency

- Interfaces of interacting elements may match, but behaviors may not.
  - Example: subtraction
    - subtract(Integer x, Integer y) returns Integer;
    - Do we know what the subtract operation does?
  - Example: QueueClient and QueueServer elements
    - QueueClient
      - pre-condition q.size >= 1;
      - post-condition q'.size = q.size;
    - QueueServer
      - pre-condition q.size > 0;
      - post-condition q'.size = q.size - 1;
    - **Pre-conditions are consistent, Post-conditions are not.**

Interfaces and Parameters

- Structure depends on *distribution model*.
  - Function calls within a program are very cheap.
  - Calls between processes on the same machine are 100x - 1000x more expensive.
  - Calls between machines are (at least) 100,000x more expensive and are **much more likely to fail**.
- Parameter passing via interfaces
  - Base types (bool, int, float, char, etc.): passed by value
  - Data structures: can be passed by reference if synchronized; must be passed by value if asynchronous.
  - References to other elements; get back a reference and make calls to it to get data.

Remote Procedure Calls

- Can use the resources of multiple servers to solve a client's goal.
  - **Synchronous timing:** Client blocks during call, so familiar computational model (function call)
  - **Load Balancing:** If interfaces are *stateless*, then it is possible to throttle scale RPCs through load-balancing across multiple servers
  - **Speed:** RPCs are faster than messaging for call and response operations.

Remote Procedure Calls

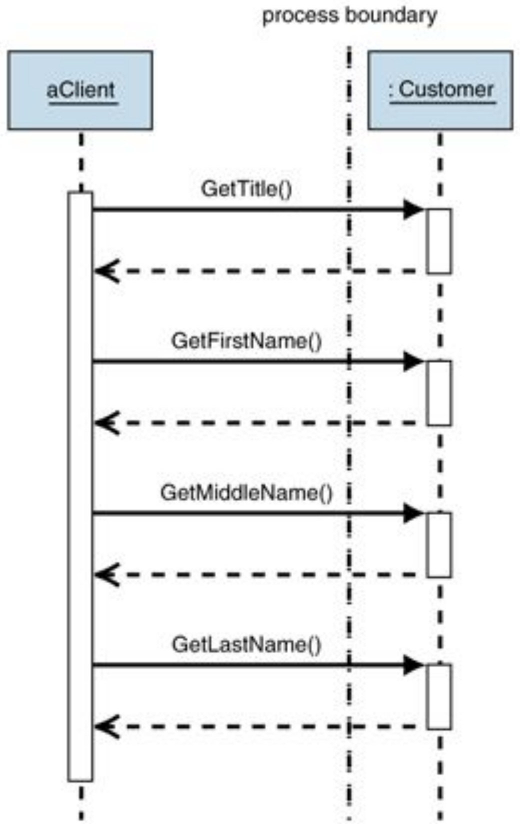
- **Unreliable Communication & Idempotence:** RPCs are difficult to make 100% reliable. Need to ensure that operations are idempotent!
  - Idempotent - messages/data is retransmitted if there is a failure.
- **Thread management:** Servers can handle many concurrent clients.

Data Transfer Objects

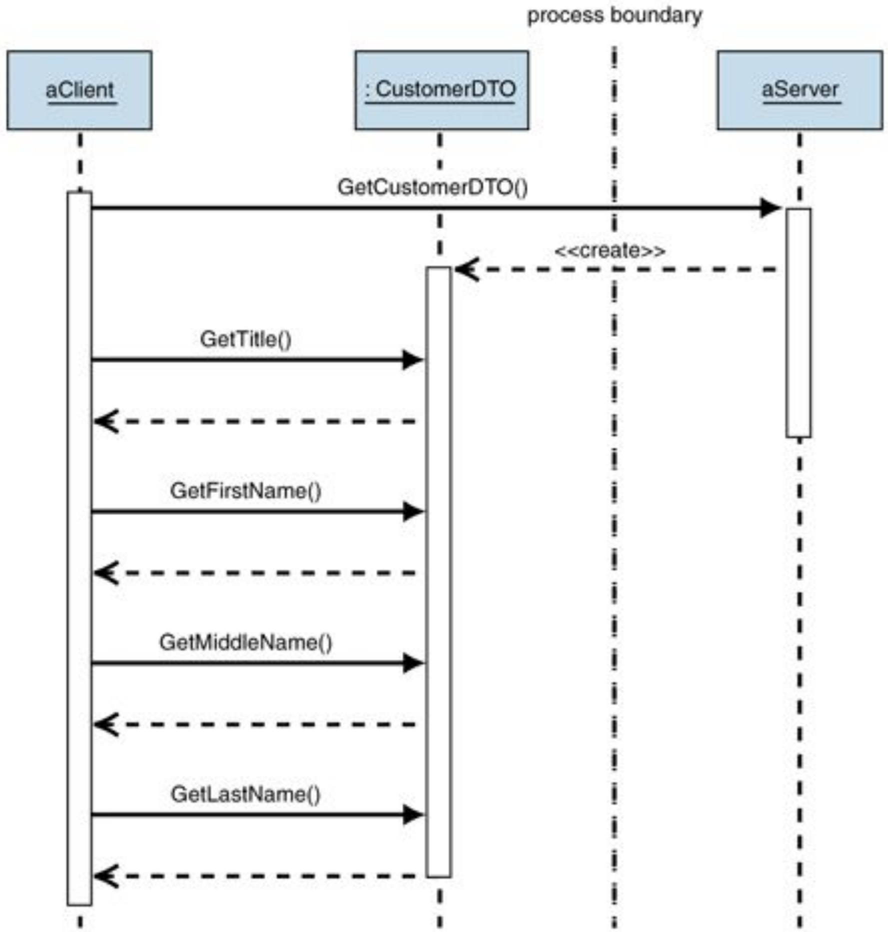
- Remote calls (i.e., through web services) are expensive and failure-prone.
  - Majority of the cost is related to round-trip time between client and server.
- DTOs carry data between processes.
  - Aggregate data that would be transferred over several calls, and handle it in a single call.
- Offer storage, retrieval, serialization, and deserialization of data, but no other functionality.

Data Transfer Objects

Without DTO



With DTO:



Data-Oriented Interfaces

- Elements communicate through unidirectional data transfer.
  - Data defines the means of communication.
  - Messages or documents are transferred, initiating a process.
- Common for elements that perform event-based actions, rather than on-command actions.
  - Pipe and filter, real-time architecture, message queueing systems.

Data Interface Considerations

- Message (packet) vs Stream:
  - Are items batched and sent as individual messages (packets), or are items processed immediately (sent through an open stream to the element)?
- Are messages queued by the element?
  - If a job is in-process, are new requests queued or ignored?
- Is data transferred using lossy or guaranteed delivery?
  - TCP vs UDP
  - UDP allows faster transfer, but lacks error checking.
  - TCP guarantees correct data, but transfers may fail.

Data Interface Advantages

- Can transfer packets of data frequently, immediately, reliably, and asynchronously using customizable formats.
- **Variable timing**
  - Unlike RPCs, sender and receiver can work at their own pace.
- **Throttling**
  - Because receiver buffers requests, it can control rate at which they are consumed so to avoid overload.
- **Reliable Communication**
  - “Store and forward” communication ensures delivery.

Data Interface Advantages

- **Disconnected Operation:**
  - Can run client applications disconnected with server and then synchronize when connection is available
- **Mediation**
  - Messaging system acts as a mediator between all programs that send and receive messages. If an application becomes disconnected, it must only reconnect with messaging system, not other apps.
- **Thread management**
  - Threads do not block waiting for remote server.

Data Interface Challenges

- **Complex Programming Model**
  - Messaging requires developers work with an event-driven programming model; applications must have callbacks for events from remote applications
- **Sequence Issues**
  - Message channels guarantee delivery, but not when message will be delivered. This can lead to messages being delivered out of sequence
- **Synchronous scenarios**
  - Many times we want application to behave synchronously. Data interfaces tend to be asynchronous.

Data Interface Challenges

- **Performance**
  - Messaging systems add overhead to communications for each message.
  - Structuring messages correctly is important to performance.
- **Vendor Lock-in**
  - Many messaging systems rely on proprietary protocols.
  - Even specifications such as JMS do not control the physical implementation of the solution, so different messaging systems may not connect to one another, leading to yet another integration problem.

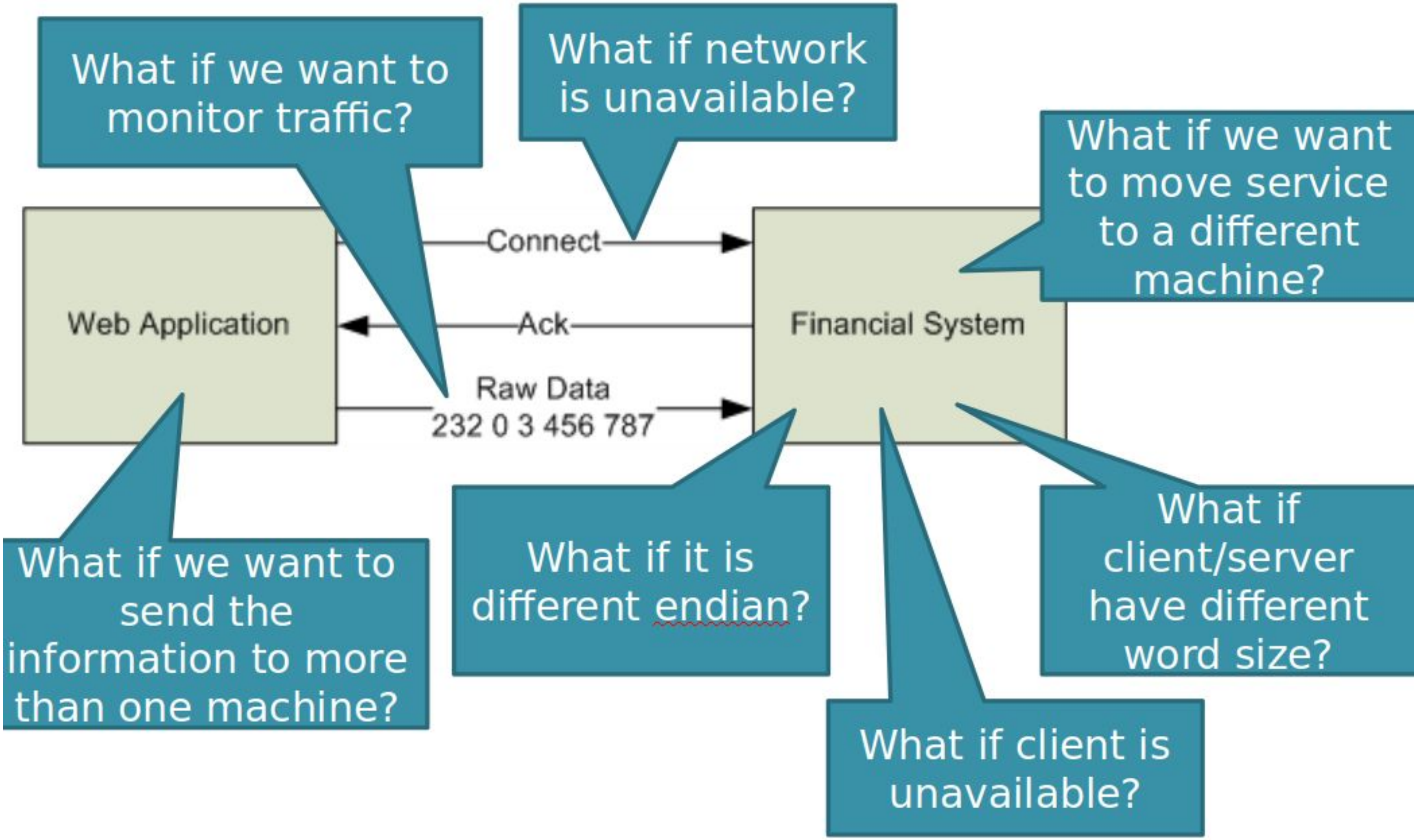
Connectors

- Elements must communicate in order to achieve system goals.
- Connectors link elements and the interfaces of elements they depend on.
  - How we implement a data or procedure-based interface (RPC, messaging, file transfer, etc)
- Must consider:
  - Synchronous or asynchronous communication
  - Resiliency of the connector
  - Concurrent users
  - Acceptable latency of connections

Connectors on One Machine

- Tend to be simple
  - Procedure call
  - Data interfaces:
    - Message queueing through a mutex-protected queue object.
    - “Last update” through mutex-protected shared memory.
  - What about when we move to multiple machines?

Moving to Remote Access

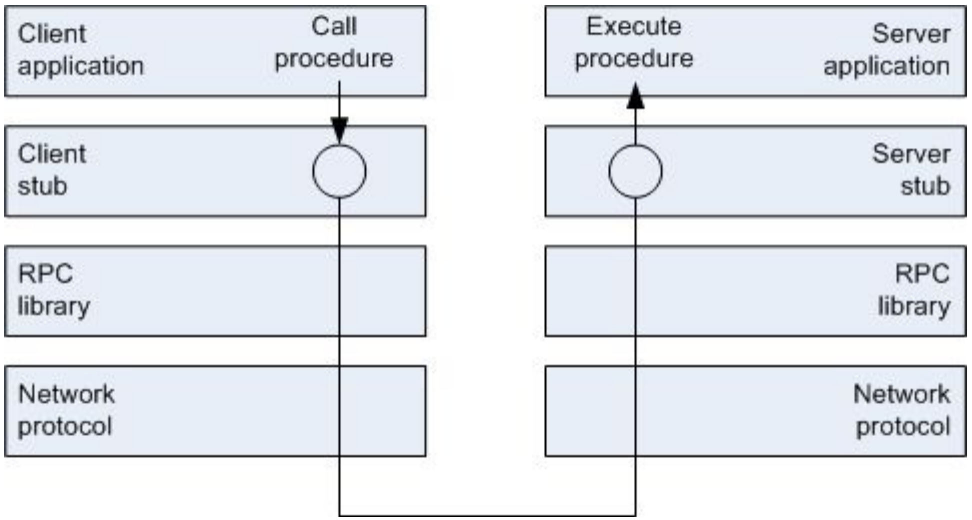


Object Request Brokers

- Middleware that allows program calls to be made from one computer to another.
  - Allows objects from one process to be used in another process as though they were part of the same process.
- Transform in-process data structures into a byte sequence, and transmit it over a network to another process (serialization).

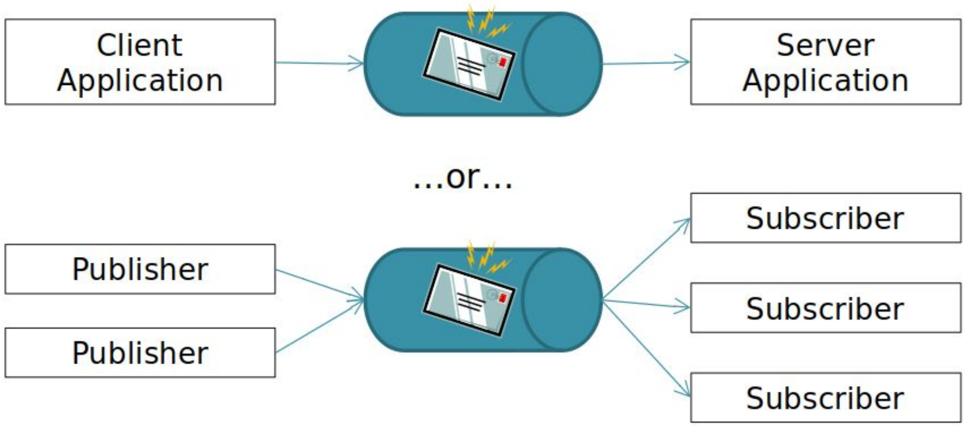
Object Request Brokers

- Client creates a stub object and call a method.
- Client-side ORB serializes data and transfers to server-side ORB.
- Server-side ORB executes operation and returns result.



- Same semantics whether or not client and server are on same computer.

Message-Oriented Middleware



- One-way message is queued and later processed (asynchronous)
  - Decouples sender and receiver
  - Message queues ensure that messages are not lost
- Message relate to transaction to be executed
  - Example: SMTP (e-mail)

More Subtle Than It Looks...

- Possible to implement message-oriented middleware through remote procedure calls, and vice-versa.
- Possible to implement asynchronous communication over RPCs.
  - Pass in a “callback interface”
  - Once task is completed, result is returned to the client through callback.

Interfaces and Distribution

For distributed interfaces, should synchronous or asynchronous interaction be preferred?

- Often asynchronous:
  - Higher availability, higher throughput, higher performance
  - Implement through messaging, RPCs with “callbacks”

Interfaces and Distribution

When communicating with a remote component or service, the chance of failure goes up dramatically. Why? How can we address it?

- Idempotence (resend messages)
- Statelessness (ensure we do not corrupt state when something goes wrong)
- Component and service isolation
  - Can service continue if something happens?
  - Cache data, preload data, defer processing

Functional View Pitfalls

Refining the Functional View

- Check the functional traceability.
    - Ensure all functional requirements are met by the proposed functional structure.
    - Table relating requirements to elements.
- Walk through common scenarios.
    - Use the functional view to illustrate how the system behaves in a scenario.
    - Explain how the elements would interact to implement that scenario.
    - Will point out weaknesses (i.e., too much interaction) or missing elements.

Refining the Functional View

- Analyze the interactions.
  - Analyze the chosen structure based on the number of interelement interactions taken during processing.
  - Reducing interactions results in better structure, efficiency, reliability.
  - Revised system must still be appropriately partitioned, without undesirable redundancy.
- Analyze for flexibility.
  - Walk through “what-if” changes to see if the proposed structure can change with minimal impact.
  - Often conflicts with interaction analysis. Must trade efficiency for flexibility where it makes sense.

Pitfall: Poorly Defined Interfaces

- Without good interface definition, development teams will make implementation mistakes.
  - Leads to build errors, obviously incorrect behavior, subtle unreliability.
- To reduce risk:
  - Define interfaces and connectors clearly and early.
  - Review frequently to ensure clear understanding.
  - Do not consider element definition complete until interfaces have been designed.
  - Make sure interface definitions include operations, their semantics, and examples.

Pitfall: Poorly Understood Responsibilities

- If you don’t define all responsibilities of the elements, confusion can remain over exactly what each element is meant to do.
  - Can lead to missing or duplicated functionality.
- To reduce risk:
  - Ensure responsibilities are formally defined early.
  - Do not allow development to drift into element design without responsibilities being formally defined
  - Make sure all implementers understand where their boundaries are (and why they are there).
  - Make sure all requirements have been mapped to elements that implement them.

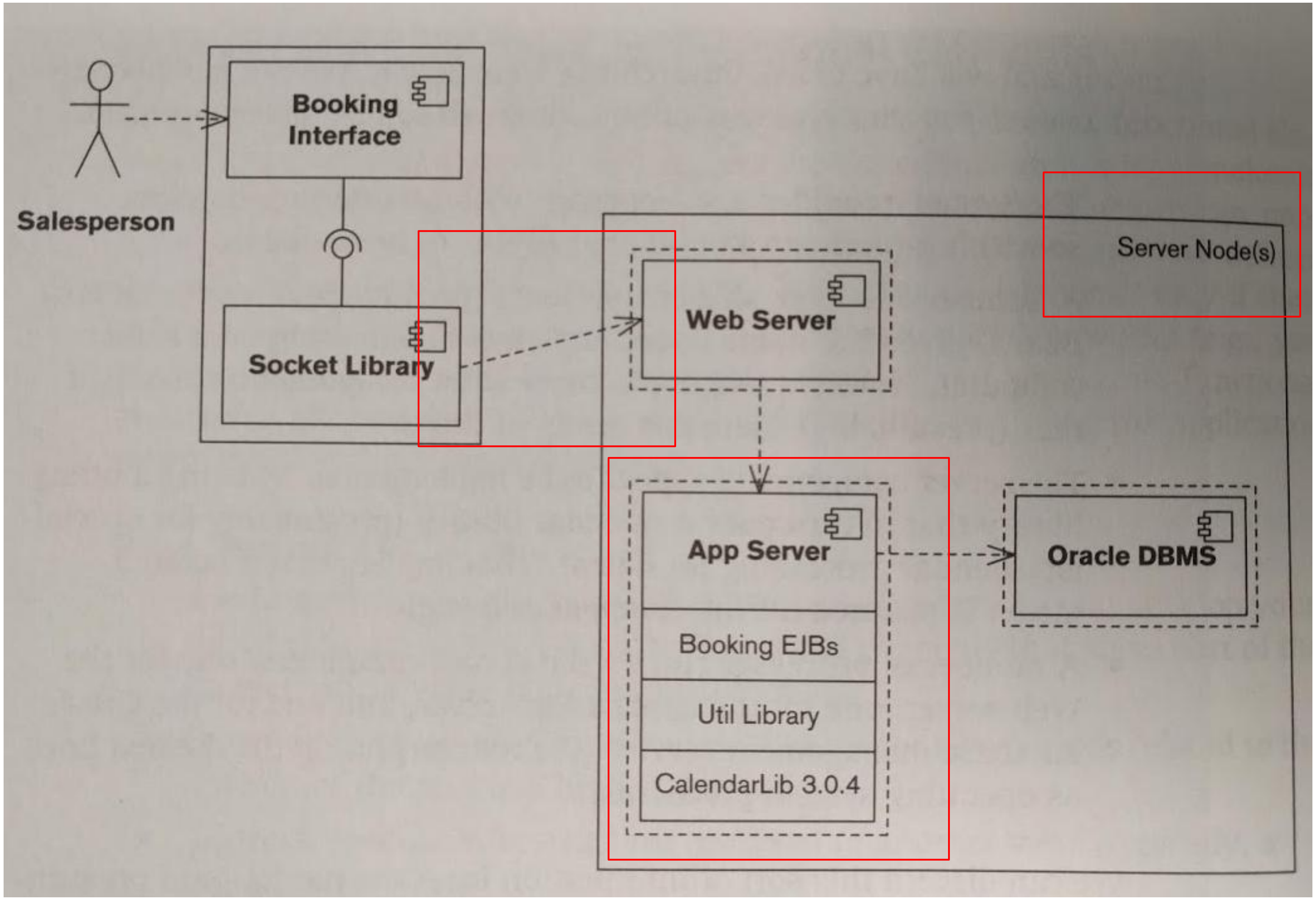
Pitfall: Infrastructure Modeled as Functional Elements

- Include infrastructure elements only if important to understanding functional view.
  - Include a messaging gateway that performs some functional processing, but not the application server you are using.
- To reduce risk:
  - Avoid modeling infrastructure elements as you develop your initial model. Focus on functional elements that solve part of the problem.
  - Question any elements that do not have names related to the problem domain.
  - Address infrastructure concerns in deployment view.

Pitfall: Overloaded View

- Often tempting to add deployment or concurrency information to this view.
  - Do not allow functional view to become overloaded. Will be harder to understand and follow.
- To reduce risk:
  - Remove everything except for items related to the functional elements, interfaces, and connectors.
  - Create other views to describe the other aspects of your architecture.
  - Develop the other views in parallel and cross-reference between views to illustrate other aspects of the architecture.

Example: Overloaded View

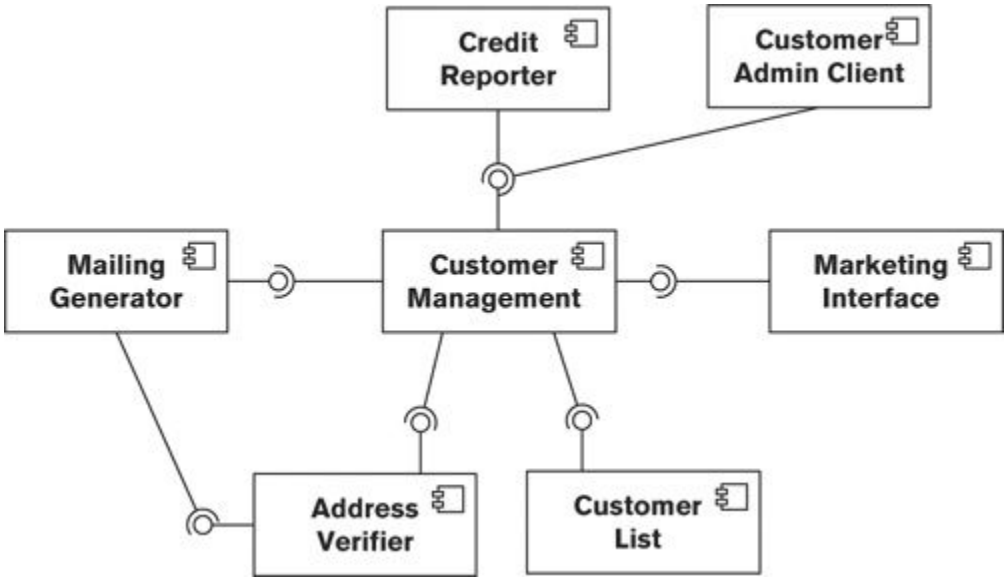


Pitfall: Wrong Level of Detail

- If too detailed, or define too many layers of elements, you are constraining design.
  - Can lead to mistakes on your behalf.
- Too little detail risks misinterpretation.
- To reduce risk:
  - Avoid defining more than 2-3 levels of elements, with 8-10 elements at the top level.
  - Avoid too many details about the internal structure of functional elements in main view.
    - If system is very large, model it as a group of systems.

Pitfall: “God Elements”

- A single huge “God Element” sits at the center of a design, with many small elements attached.



Pitfall: “God Elements”

- “God Elements” contain too much functionality and have too many dependencies.
  - Often, “God Element” is the entire program and the small elements are just data storage.
  - Often result of too much consolidation after interaction analysis.
  - Results in difficult maintenance.
  - “God Element” dominates quality properties.
- To avoid, aim for even ditribution of responsibilities. If >50% of functionality is in <25% of elements, may have “God Elements”

Pitfall: Too Many Dependencies

- Avoid having too many small elements that depend on each other.
  - Will make the system harder to change, will worsen performance.
- To reduce risk:
  - Compress related elements together.
  - In general, an element should be aware of the existence of only a couple of other elements in order to perform its functions.
    - If any elements need to services from more than 50% of the other elements, revising the structure.

Checklist (Food for Thought)

- Do you have fewer than 15-20 top-level elements?
- Do all elements have a name, responsibilities, and clear interfaces?
- Do all element interactions take place via well-defined interfaces and connectors?
- Do your elements exhibit an appropriate level of cohesion?
- Do your elements exhibit an appropriate level of coupling?

Checklist (Food for Thought)

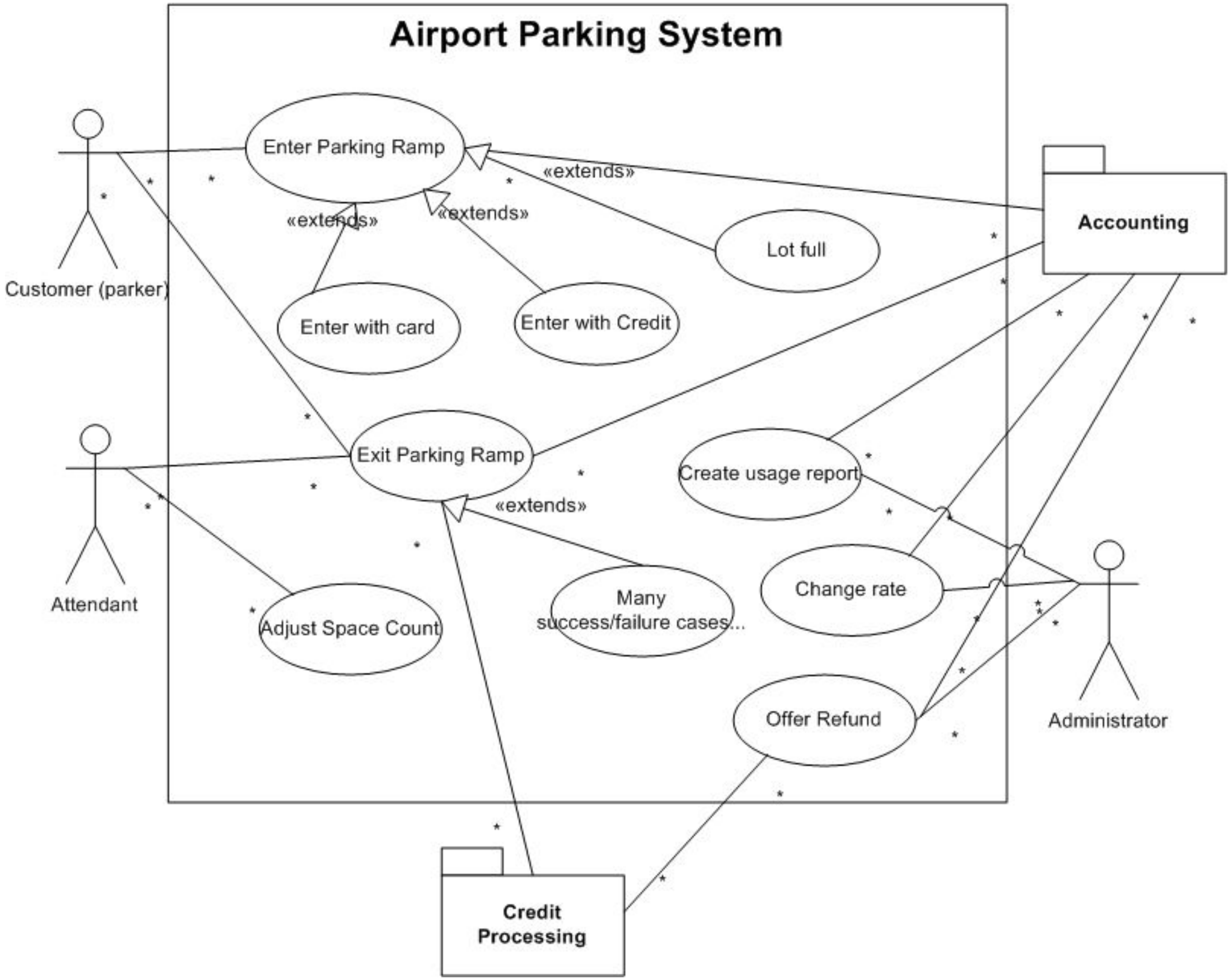
- Have you identified the important usage scenarios and used these to validate the functional structure?
- Have you checked the coverage of requirements by your architecture?
- Have you defined and documented architectural design principles, and does your architecture comply with these principles?

Checklist (Food for Thought)

- Have you considered how the architecture is likely to cope with future change?
- Does the presentation of the view take into account the concerns and capabilities of all interested stakeholder groups?
- Will the view act as an effective communication vehicle for all groups?

Activity: Airport Parking

Use Cases



Airport Parking System
Enter Parking Ramp
«extends»
«extends»
«extends»
Lot full
Accounting
Customer (parker)
Enter with card
Enter with Credit
Exit Parking Ramp
Create usage report
«extends»
Attendant
Change rate
Many
success/failure cases...
Adjust Space Count
Administrator
Offer Refund
Credit
Processing

General Principles

- Encapsulate components that are likely to change
  - Hardware
  - Policies (pricing, lot capacity, etc.)
- Define services that individually and collectively have value
  - High Coherence
  - Low Coupling

Food for Thought

What do you need to track?

- On entry?
- On exit?
    - Where do you store completed transactions? In the system? Sent through interface to accounting system?
- For pricing?
- When performing manual overrides
    - Who can perform them?
    - How do you log them?

What do you need to control?

- Physical gates for entry / exit
- Entry kiosks
    - Credit card reader
    - Parking card dispenser
- Exit kiosks
    - Automated: credit card / parking card reader
        - Optional: cash input
    - Attendant kiosks
        - Point of sale device: in or out of system?
        - Allow manual override of charges?

Food for Thought

How do you want to report?

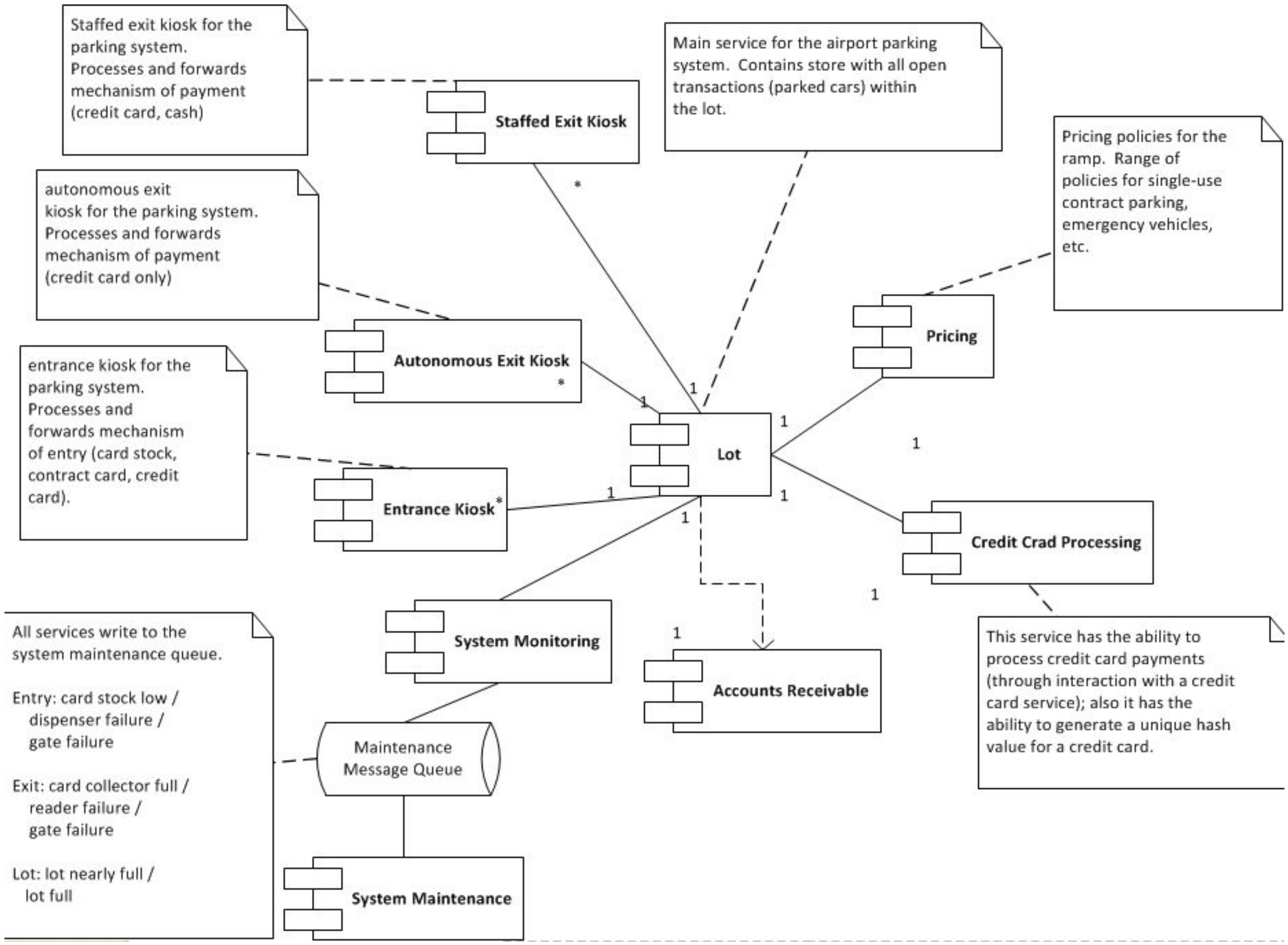
- Revenue?
  - Partitioned by pricing type?
  - Current? Over time?
- Card stock levels per entry kiosk?
- Mechanical failures?
- Ramp usage? Utilization over time?

The Activity

From the requirements, use cases, and other provided information:

- Derive elements. For each, briefly note the responsibilities and purpose of that element.
- Draw a UML Context Diagram depicting the system.
- You do not need to design interfaces, but think about how you would implement important ones.

Suggested Solution



Key Points

- The **functional view** of a system defines the architectural elements that deliver the functions of the system being described.
- Documents the system's functional structure:
  - Key functional elements and their responsibilities.
  - The interfaces they expose (internal/external).
  - The interactions between them.
- This view demonstrates how the system will perform the functions required of it.

Next Time

- Architectural Style: REST
  - Source: Roy Fielding. “Architectural Styles and the Design of Network-based Software Architectures”
- Homework:
  - Assignment 1 - due 10/02
  - Project Part 2 - due 10/11
  - Assignment 2 - due 10/25