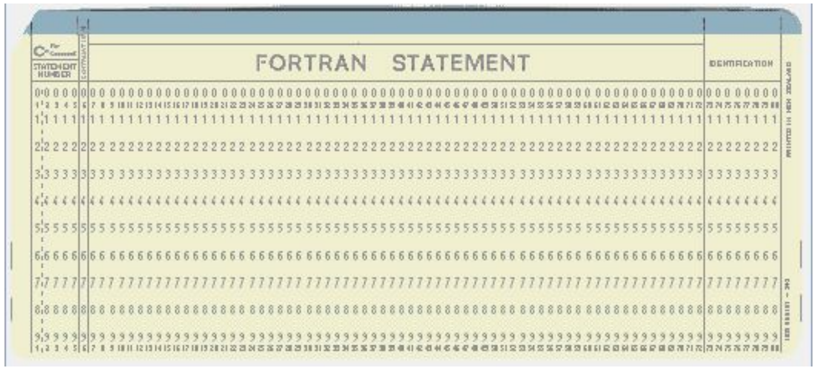
Software Architecture

CSCE 247 - Lecture 15 - 03/18/2019

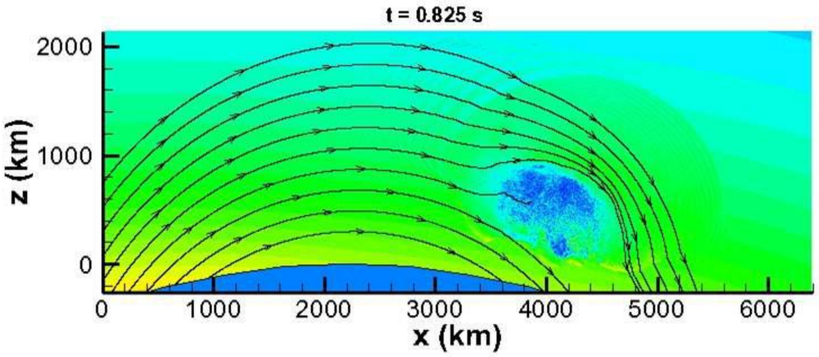
In the beginning…

Software Was Small

- Both physically…



- And in scope.



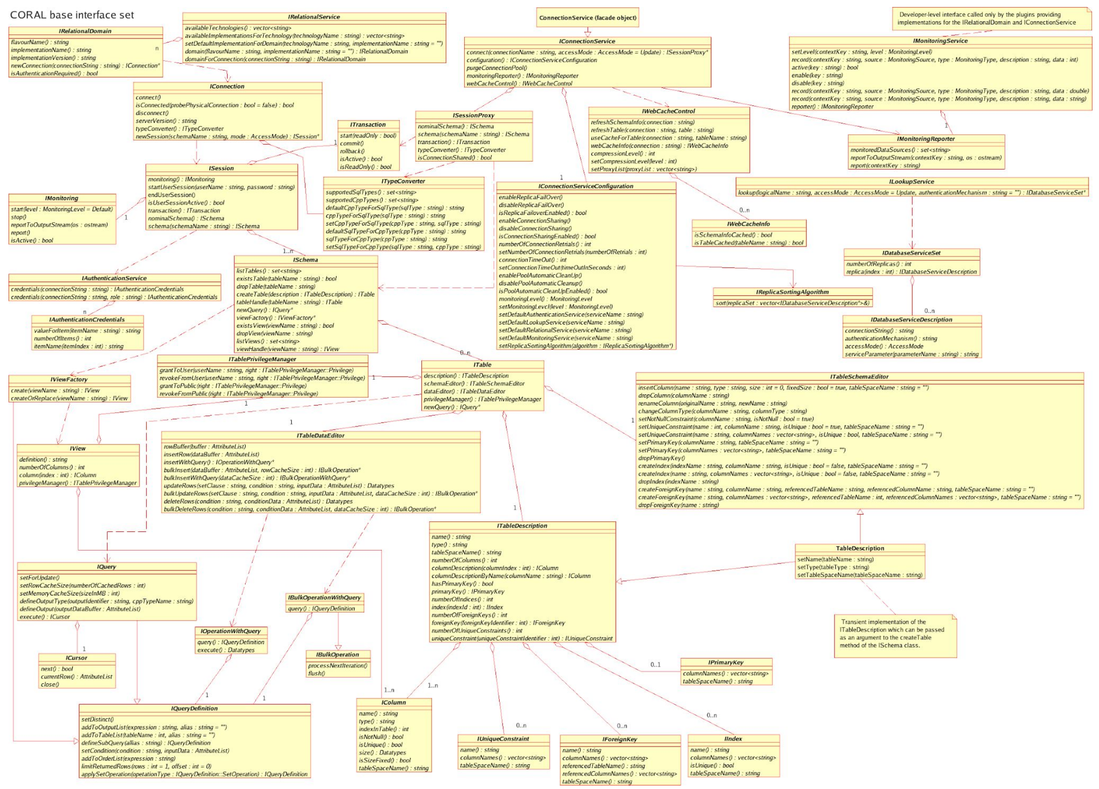
Software Starts to Grow Up

- Languages like C introduce file linking.
  - Enables organization of code and reuse of code.
- SIMULA-67, Smalltalk introduce objects.
  - Enables organization of code into focused units that work with other objects to perform larger tasks.
    - Sections of the code “activate” when needed.
    - We can group together related functionality, ignore unrelated functionality, and find what we need when making changes.
    - Code can be reused in future projects.

Flash forward to the present day...

Under the Hood

- Systems have millions of lines of code.
- Divided into hundreds of classes.



Growing Pains

- No person can understand an entire million-line codebase.
- Classes organize code, but how can you find the right clases when there are thousands?
- Results in chaos.
  - Only 16.1% of projects delivered on time and within budget, with all planned features complete as specified.
  - 31.1% of projects are cancelled before delivery.
  - Delivered projects may be slow, insecure, missing features, have duplicate code, go down often, etc.

The Concept: “Architect” Software

- The key to delivering robust software?
  - Designing an understandable, organized system.
  - **AKA: “taming the complexity”**
- **Architecting software** is the practice of partitioning a large system into smaller ones.
  - That can be created separately
  - That individually have business value
  - That can be straightforwardly integrated with one another and with existing systems.

Architectural Styles

What is Software Architecture?

“The **architecture** of a software-intensive system is the structure or structures of the system, which comprise software **elements**, the **externally-visible properties** of those elements, and the relationships among them.”

- Carnegie-Mellon Software Engineering Institute (SEI)

Architectural Design

- First stage of design.
- **Partitions** the requirements into self-contained subsystems.
    - Later, each subsystem will be decomposed into one or more classes.
- Plan how those subsystems cooperate and communicate.

Static Structures

- The **static structures** of a system define its internal design-time elements and their arrangement.
  - Software elements: modules, classes, packages.
  - Data elements: Database entries/tables, data files.
  - Hardware elements: Servers, CPUs, disks, networking environment
- The static arrangement of elements defines associations, relationships, or connectivity between these elements.

Static Structure Arrangement

- For software elements, static relationships define **hierarchy** (inheritance) or **dependency** (use of variables or methods).
- For data elements, static relationships define how data items are linked.
- For hardware elements, static relationships define physical interconnections between hardware elements.

Dynamic Structures

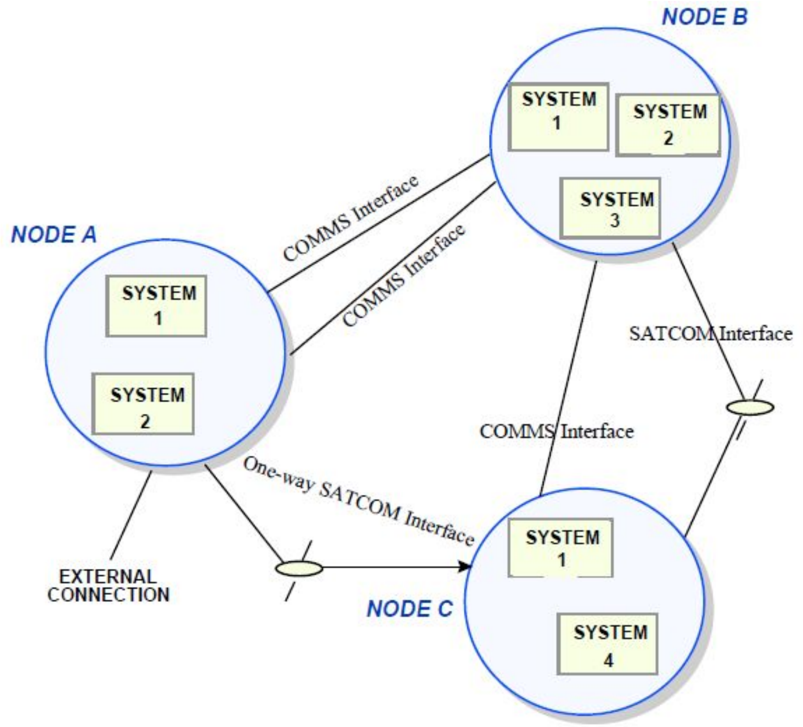
- The **dynamic structures** of a system define its runtime elements and their interactions.
- May depict flow of information between elements
  - A sends messages to B
- May depict flow of control in a particular scenario.
  - `A.action()` invokes `B.action()`
- May depict effect an action has on data.
  - Entry E is created, updated, and destroyed.

Architectural Elements

- An **architectural element** is a fundamental piece from which a system can be constructed.
- The scope of an element depends on the type of system.
  - A single method, a class, a set of related classes (a subsystem), an imported library can all be elements.
  - “Component”, “module”, and “unit” are often used interchangeably, but are overloaded terms.

Architectural Elements

- An element must possess key attributes:
    - A clearly defined set of **responsibilities**.
    - A clearly defined **boundary**.
    - A set of defined **interfaces**.
        - Define the services that the element provides to other elements.



Externally Visible Behavior

- The **externally visible behavior** of a system defines the functional interactions between the system and its environment.
    - Flow of information in and out of the system.
    - How the system responds to input.
    - The defined interfaces available to the outside world.
- Can be modeled in architecture as a black box (ignoring any internal information).
- Can also be modeled by including how internal state responds to external input.

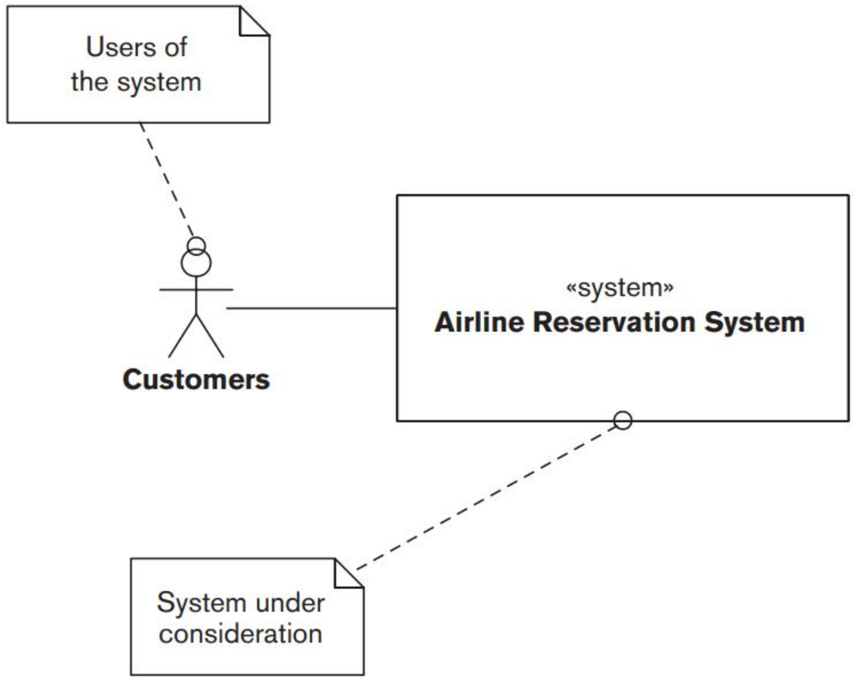
Quality Properties

- A **quality property** is an externally visible, nonfunctional property.
  - Performance, security, availability, safety, modifiability, testability, usability, etc.
    - How does the system perform under load?
    - How is information protected from unauthorized use?
    - How long will it be down on a crash?
    - How easy is it to manage, maintain, and enhance?
  - Tell us how an observer views the behavior of a system.

Airline Reservation System

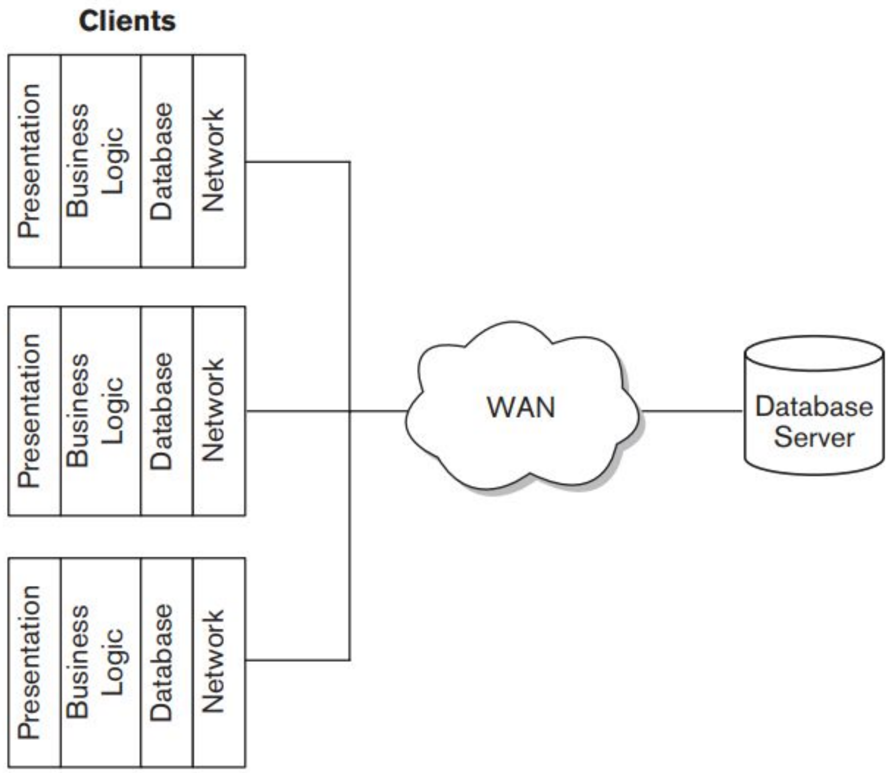
Airline Reservation System

- Allows seat booking, updating, cancellation, upgrading, transferring.
- **Externally visible behavior:** How it responds to submitted transactions.
- **Quality properties:** average response time, max throughput, availability, time required to repair issues.



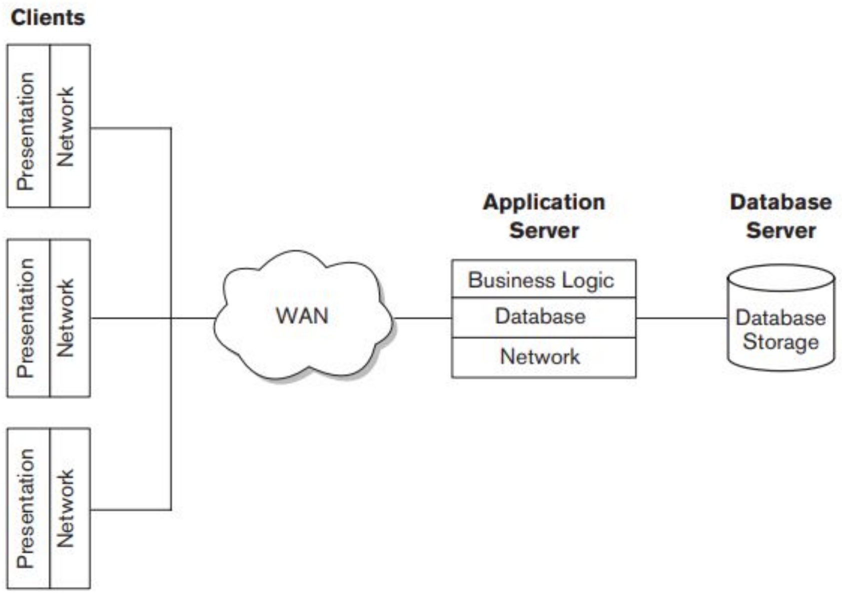
Option 1: Client/Server Architecture

- Clients communicate with a central server (with a database) over a network.
- **Static Structure:** Client programs, broken into layered elements, a server, and connections.
- **Dynamic Structure:** Request/response model.



Option 2: “Thin Client” (Client/Server) Architecture

- Clients communicate with a central server (with a database) over a network.
- **Static Structure:** Client programs only perform presentation. An application server performs logic computations.
- **Dynamic Structure:** Request/response model. Requests submitted to application server, then database server.



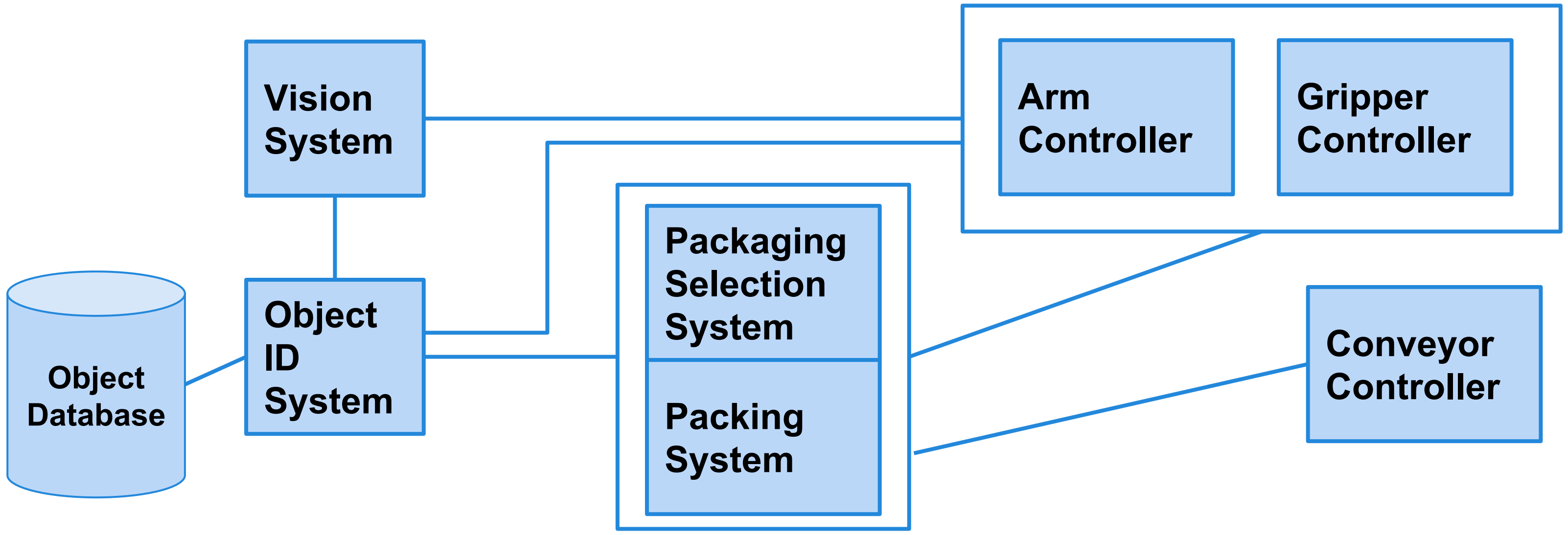
Which Would You Choose?

- Both display same externally behavior, but may differ in quality properties.
  - First approach is simpler.
  - Second might provide better options for scalability, or be more secure.
- Must select a candidate architecture that satisfies all requirements and meets the proposed quality properties.
- Extent that a model exhibits behaviors and quality properties must be studied further.

Static Structuring

Static Structuring

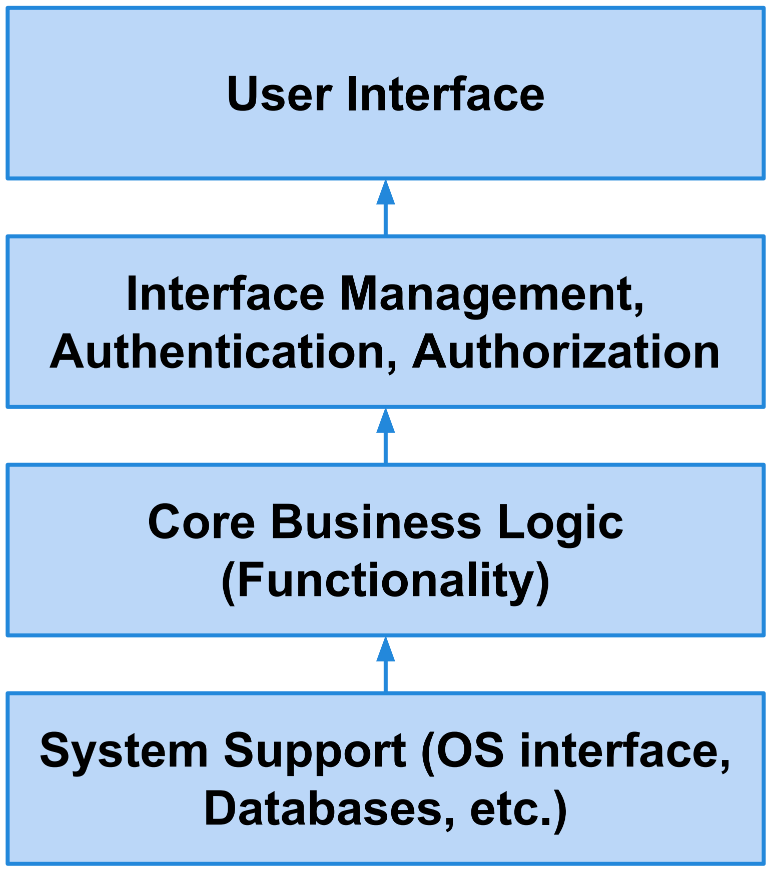
- How we decompose the system into interacting elements.
- Can be visualized as block diagrams presenting an overview of the system structure.



Basic Architectural Styles

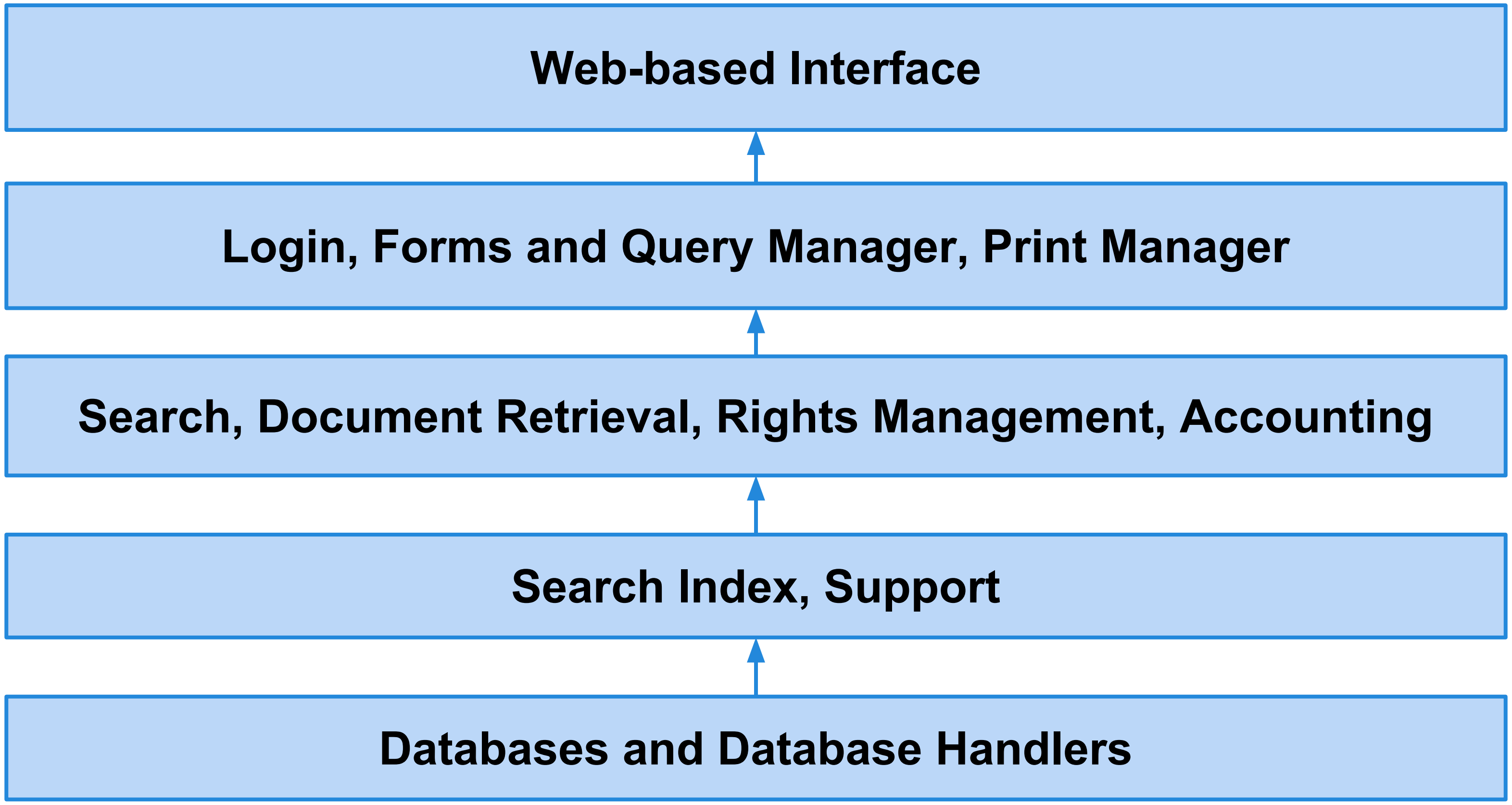
- Four common styles: layered, shared repository, client/server, pipe & filter
- The style used affects the performance, robustness, availability, maintainability, etc. of the system.
- Complex systems might not follow a single model - mix and match.

Layered Model


User Interface
Interface Management, Authentication, Authorization
Core Business Logic (Functionality)
System Support (OS interface, Databases, etc.)

- System functionality organized into layers, with each layer only dependent on the previous layer.
- Allows elements to change independently.
- Supports incremental development.

Copyright Management Example


Web-based Interface
Login, Forms and Query Manager, Print Manager
Search, Document Retrieval, Rights Management, Accounting
Search Index, Support
Databases and Database Handlers

Layered Model Characteristics

Advantages

- Allows replacement of entire layers as long as interface is maintained.
- When changes occur, only the adjacent layer is impacted.
- Redundant features (authentication) in each layer can enhance security and dependability.

Disadvantages

- Clean separation between layers is often difficult.
- Performance can be a problem because of multiple layers of processing between call and return.

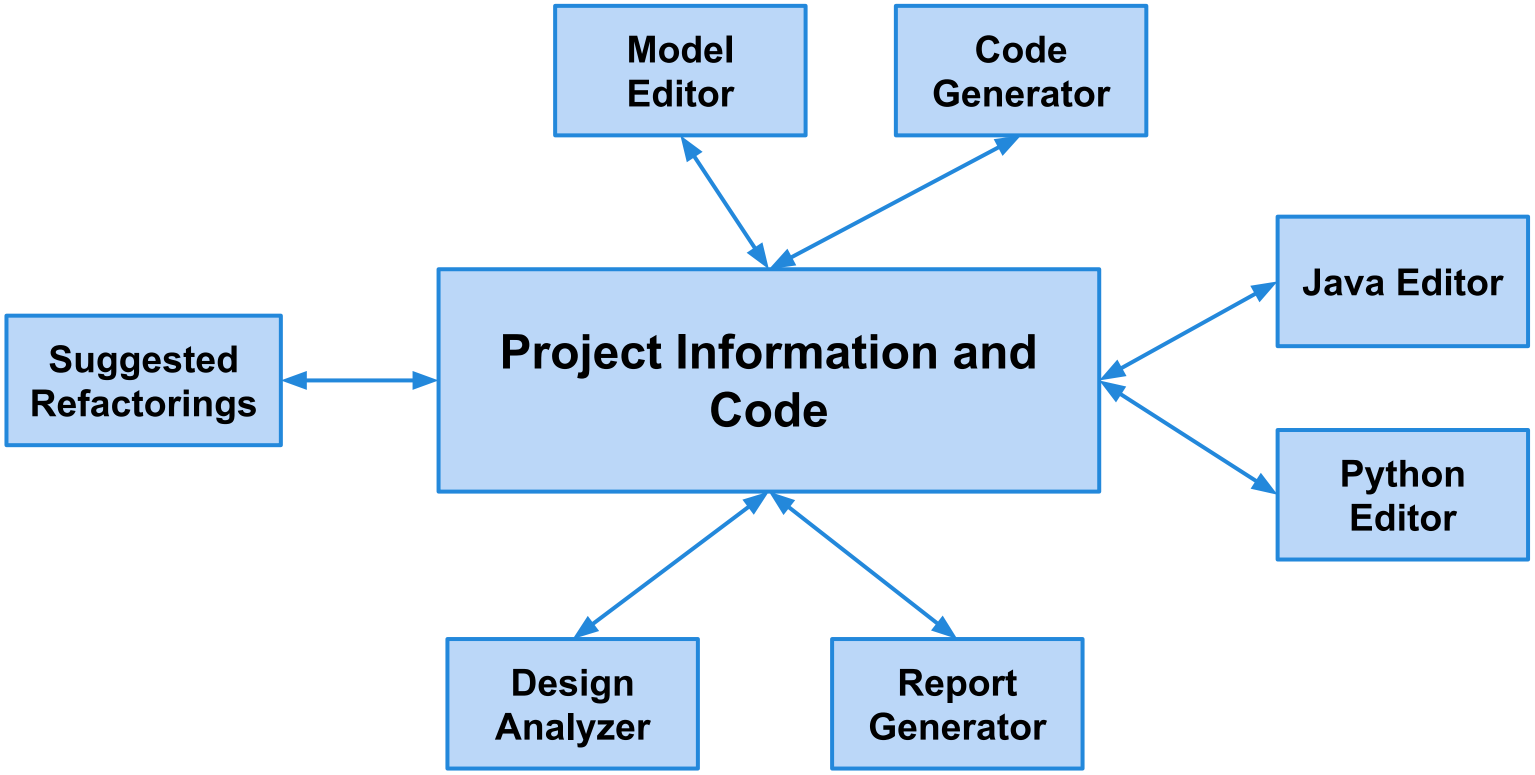
The Repository Model

Subsystems often exchange and work with the same data. This can be done in two ways:

- Each subsystem maintains its own database and passes data explicitly to other subsystems.
- **Shared data is held in a central repository and may be accessed by all subsystems.**

Repository model is structured around the latter.

IDE Example


Model Editor
Code Generator
Suggested Refactorings
Project Information and Code
Java Editor
Python Editor
Design Analyzer
Report Generator

Repository Model Characteristics

Advantages

- Efficient way to share large amounts of data.
- Components can be independent.
  - May be more secure.
- All data can be managed consistently (centralized backup, security, etc)

Disadvantages

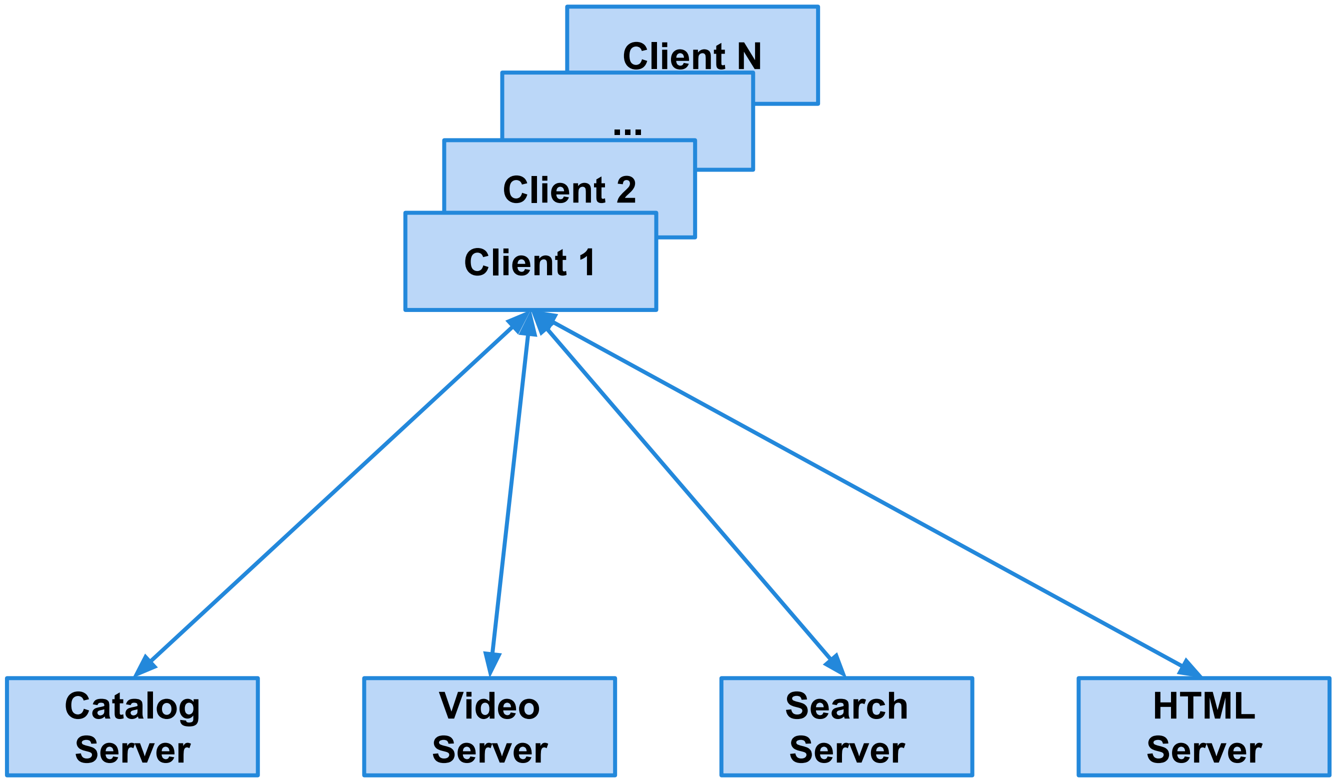
- Single point of failure.
- Subsystems must agree on a data model (inevitably a compromise).
- Data evolution is difficult and expensive.
- Communication may be inefficient.

Client-Server Model

Functionality organized into services, distributed across a range of components:

- A set of servers that offer services.
  - Print server, file server, code compilation server, etc..
- Set of clients that call on these services.
  - Through locally-installed front-end.
- Network that allows clients to access these services.
  - Distributed systems connected across the internet.

Film Library Example



Client-Server Model Characteristics

Advantages

- Distributed architecture.
    - Failure in one server does not impact others.
- Makes effective use of networked systems and their CPUs. May allow cheaper hardware.
- Easy to add new servers or upgrade existing servers.

Disadvantages

- Performance is unpredictable (depends on system and network).
- Each service is a point of failure.
- Data exchange may be inefficient (server -> client -> server).
- Management problems if servers owned by others.

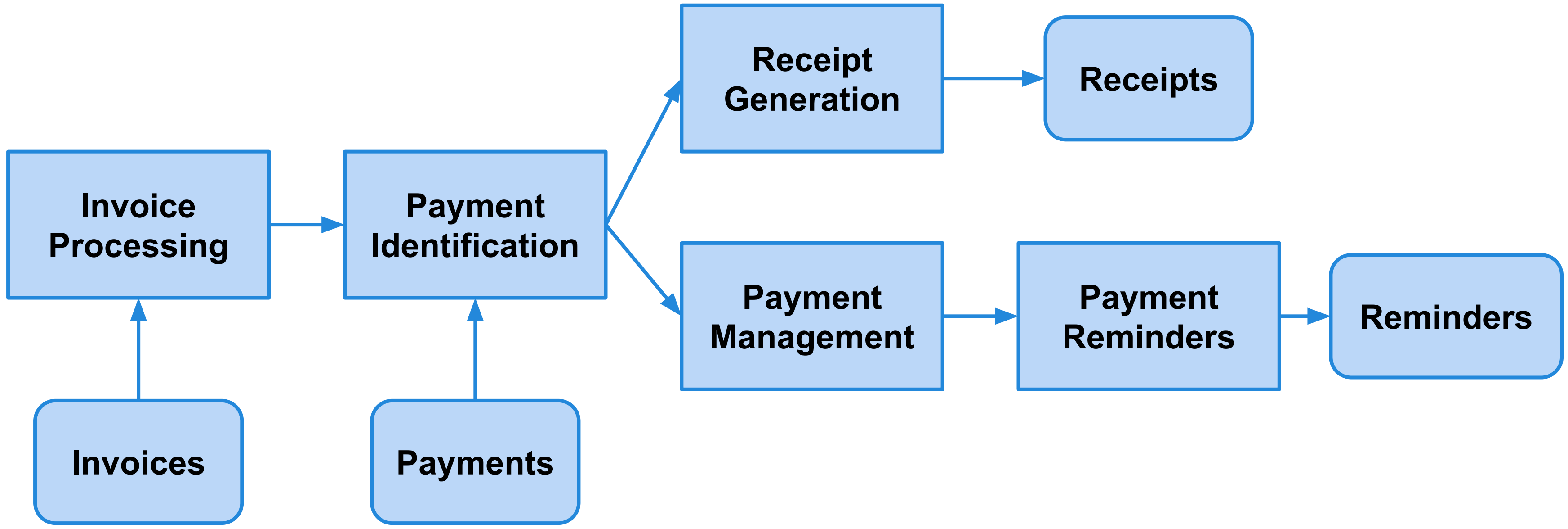
Pipe and Filter Model

Input is taken in by one component, processed, and the output serves as input to the next component.

- Each processing step transforms data.
- Transformations may execute sequentially or in parallel.
- Data can be processed as items or batches.
- Similar to Unix command line:

  -
    ```
    cat file.txt | cut -d, -f 2 | sort -n | uniq -c
    ```

Customer Invoicing Example



Pipe and Filter Characteristics

Advantages

- Easy to understand communication between components.
- Supports subsystem reuse.
- Can add features by adding new subsystems to the sequence.

Disadvantages

- Format for data communication must be agreed on. Each transformation needs to accept and output the right format.
- Increases system overhead.
- Can hurt reuse if code doesn’t accept right data structure.

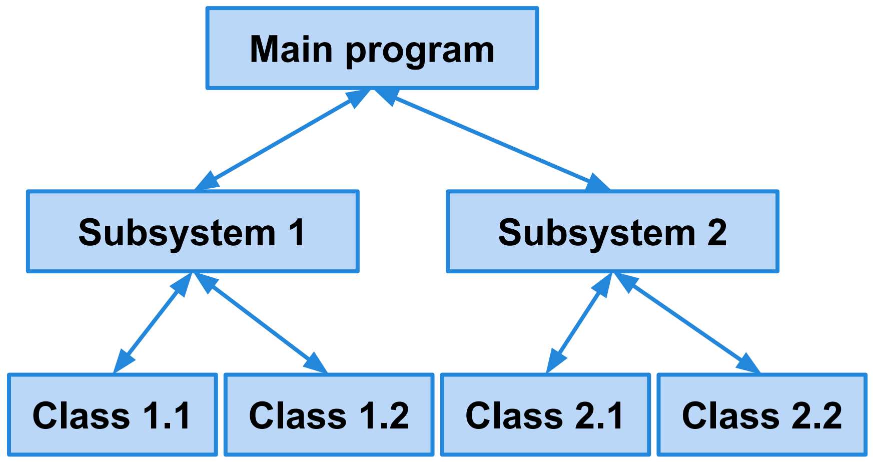
Dynamic Structuring

Control Models

- A model of the control relationships between the different parts of the system is established.
- During execution, how do the subsystems work together to respond to requests?
    - **Centralized Control:**
        - One subsystem has overall responsibility for control and stops/starts other subsystems.
    - **Event-Based Control:**
        - Each subsystem can respond to events generated by other subsystems or the environment.

Centralized Control: Call-Return

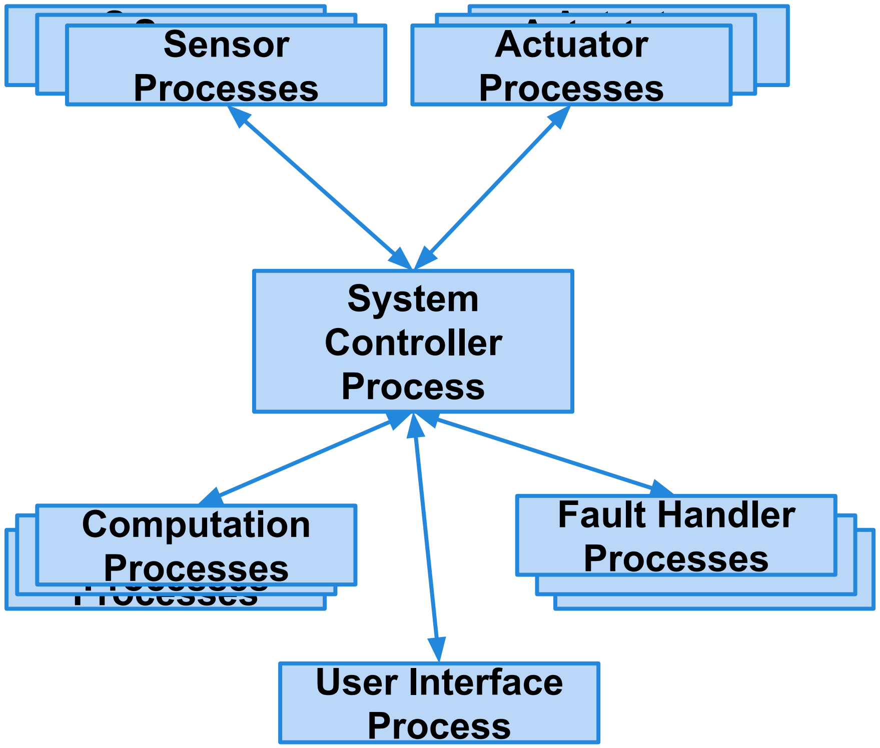
A central piece of code (Main) takes responsibility for managing the execution of other subsystems.



Call-Return Model

- Applicable to sequential systems.
- Top-down model where control starts at the top of a subroutine and moves downwards.

Centralized Control: Manager Model



Manager Model

- Applicable to concurrent systems.
- One process controls the stopping, starting, and coordination of other system processes.

Decentralized Control: Event-Driven Systems

Control is driven by externally-generated events where the timing of the event is out of control of subsystems that process the event.

- Broadcast Model
  - An event is broadcast to all subsystems.
  - Any subsystem that needs to respond to the event does do.
- Interrupt-Driven Model
  - Events processed by interrupt handler and passed to proper component for processing.

Broadcast Model

An event is broadcast to all subsystems, and any that can handle it respond.

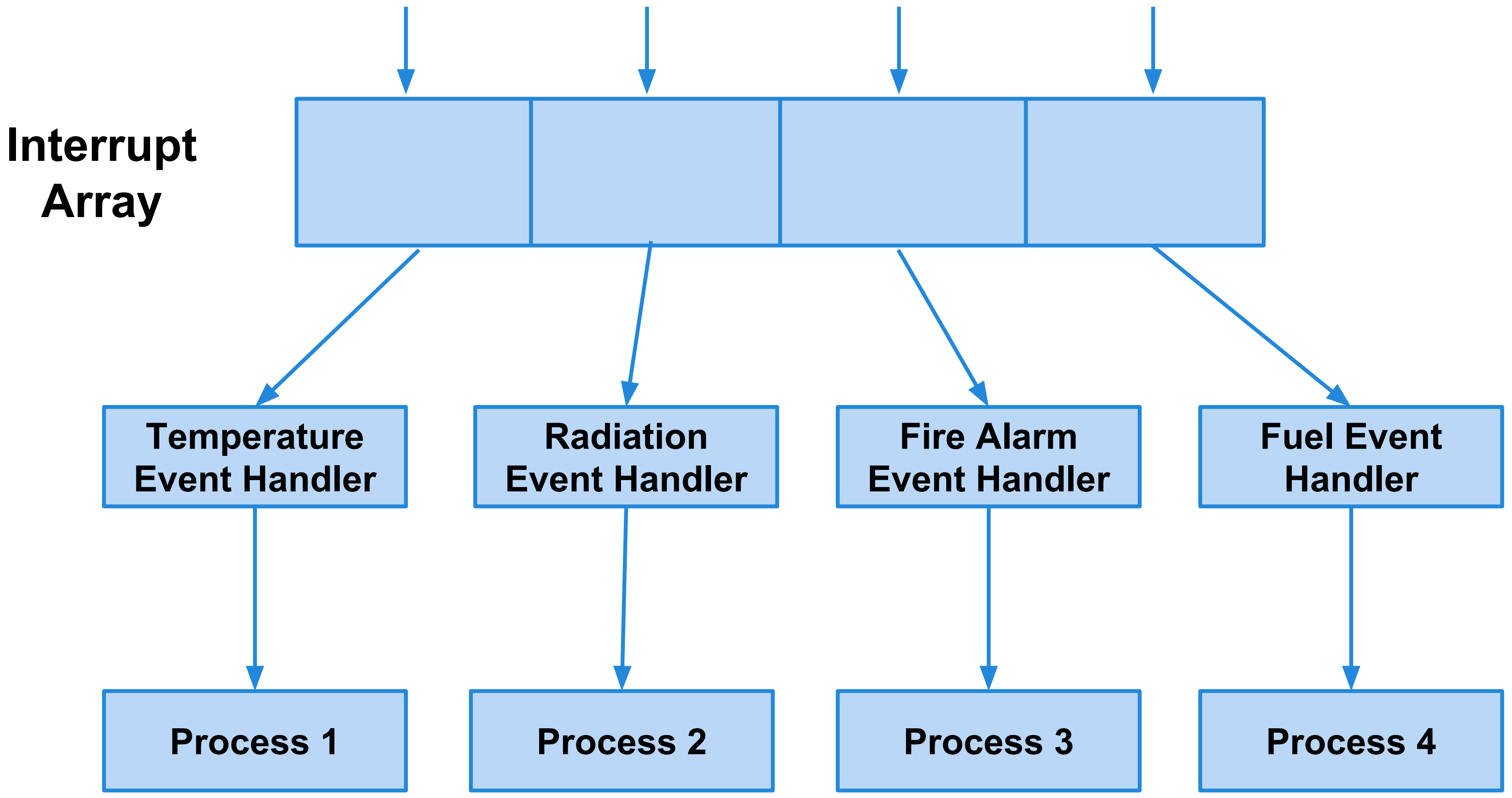
- Subsystems can register interest in specific events. When these occur, control is transferred to the registered subsystems.
- Effective for distributed systems. When one component fails, others can potentially respond.
    - However, subsystems don’t know when or if an event will be handled.

Interrupt-Driven Model

Events processed by interrupt handler and passed to proper component for processing.

- For each type of interrupt, define a handler that listens for the event and coordinates response.
- Each interrupt type associated with a memory location. Handlers watch that address.
- Used to ensure fast response to an event.
    - However, complex to program and hard to validate.

Nuclear Plant Interrupt Example



Example: The ASW

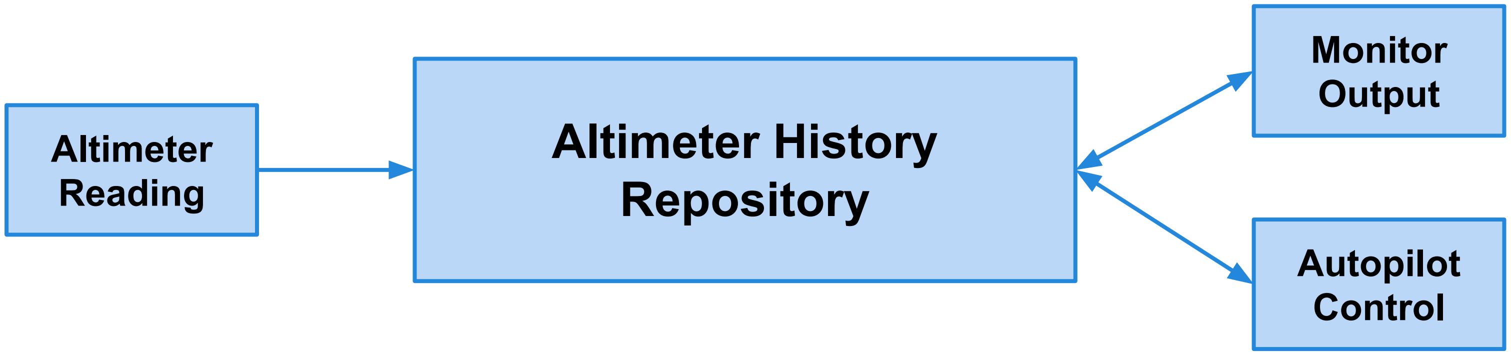
You are designing control software for an aircraft. In such software, multiple behaviors are based on altitude. The software interfaces with one of more altimeters, makes autopilot decisions based on this information, and outputs status information to a monitor that is viewed by the pilot. If altitude drops below certain thresholds, the system will send warnings to that monitor and, if autopilot is engaged, will attempt to correct the plane's orientation.

- **Perform static structuring. Try to use one or more of the models covered.**
- **Perform dynamic structuring. How should control be routed?**

ASW Solution

- Perform system structuring. Try to use one or more of the models covered.

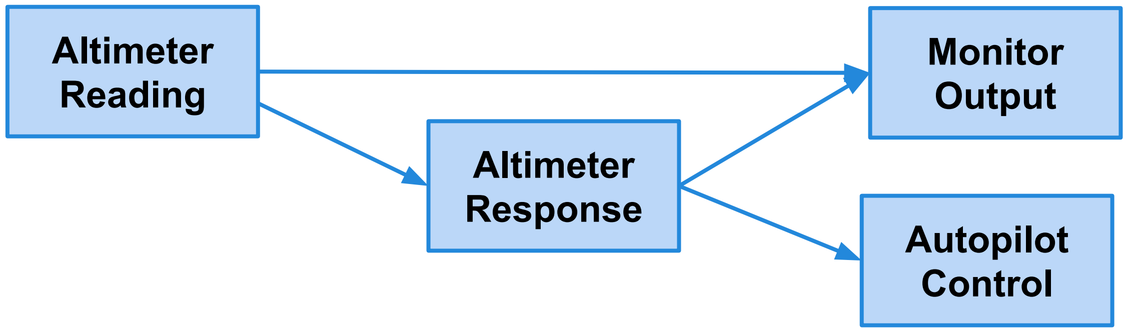
Option 1: Repository Model



ASW Solution

- **Perform system structuring. Try to use one or more of the models covered.**

Option 2: Pipe and Filter



ASW Solution

- **Perform control modeling. How should events be handled?**

Depends on how you answered the previous question, but a natural option would be an Interrupt-Driven Model.

Handlers for new altimeter readings, for error flags triggered by altimeter processing code.

Key Points

- The software architecture must consider static structure, dynamic structure, externally-visible behaviors, and quality properties.
- Architectural models can help organize a system.
    - Layered, repository, client-server, and pipe and filter models - also many others.
- Control models include centralized control and event-driven models.

Next Time

- Object-oriented design and class diagrams
- Reading
    - Sommerville, chapter 5, 7
    - Fowler UML, chapter 3
        - (or any resource on class diagrams)

- Homework: Assignment 3 is out!