# System Design

Acknowledge: Atlee and Pfleeger (Software Engineering: Theory and Practice)



## Design: HOW to implement a system

- **Goals:**
  - **Satisfy the requirements**
  - **Satisfy the customer**
  - **Reduce development costs**
  - **Provide reliability**
  - **Support maintainability**
  - **Plan for future modifications**

## Design Issues

- Architecture
- User Interface
- Data Types
- Operations
- Data Representations
- Algorithms



## System Design

- Choose high-level strategy for solving problem and building solution
- Decide how to organize the system into subsystems
- Identify concurrency / tasks
- Allocate subsystems to HW and SW components

## Strategic vs. Local Design Decisions

- **Defn**: A high-level or ***strategic*** design decision is one that influences the form of (a large part) of the final code
- Strategic decisions have the most impact on the final system
- So they should be made carefully
- **Question**: Can you think of an example of a strategic decision?



## System Design

- Defn: The high-level strategy for solving an [information flow] problem and building a solution
  - Includes decisions about organization of functionality.
  - Allocation of functions to hardware, software and people.
  - Other major conceptual or policy decisions that are made prior to technical design.
- Assumes and builds upon thorough requirements and analysis.

## Taxonomy of System-Design Decisions

- **<u>Devise a system architecture</u>**
- Choose a data management approach
- Choose an implementation of external control



## System Architecture

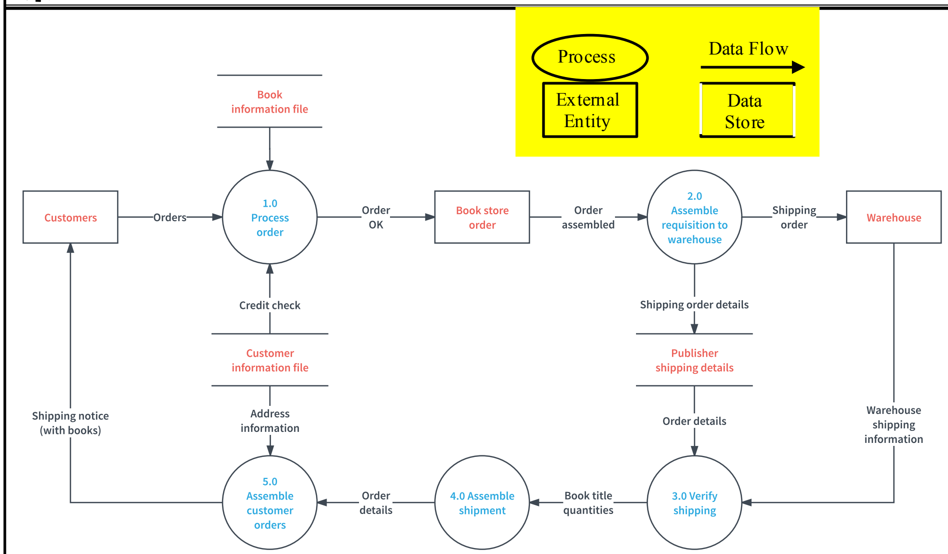
- A collection of ***subsystems*** and interactions among subsystems.
- Should comprise a small number (<20) of subsystems
- A subsystem is a package of classes, associations, operations, events and constraints that are interrelated and that have a reasonably well-defined interface with other subsystems,
- Example subsystems:
  - Database management systems (RDBMS)
  - Interface (GUI) package

# System Topology (also known as SW Architecture)

- Describe information flow
  - Can use DFD (data flow diagram) to model flow
- Some common topologies
  - Pipes-and-Filter
  - Star topology
  - Client-Server
  - Peer-to-Peer
  - Publish-Subscribe
  - Repositories
  - Layering



# Data Flow Diagram



## Terminology

- **Idioms:**
  - paradigm/language-specific programming techniques.
- **Design Patterns:**
  - reusable (problem, design strategy) pair with context for application, consequences for use.
- **Architectural Patterns:**
  - High-level strategies for system design
  - Involves large-scale components and their relationships



## Architectural Styles

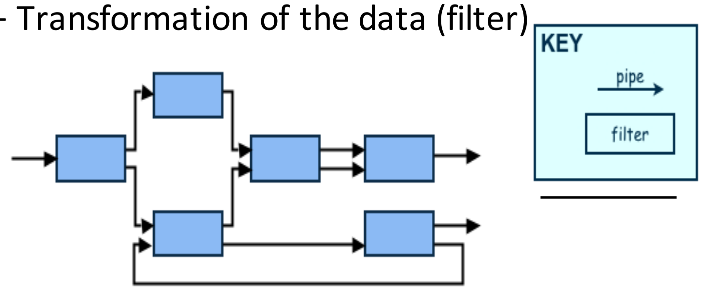
## Architectural Styles and Strategies

- Pipes-and-Filter
- Client-Server
- Peer-to-Peer
- Publish-Subscribe
- Layering



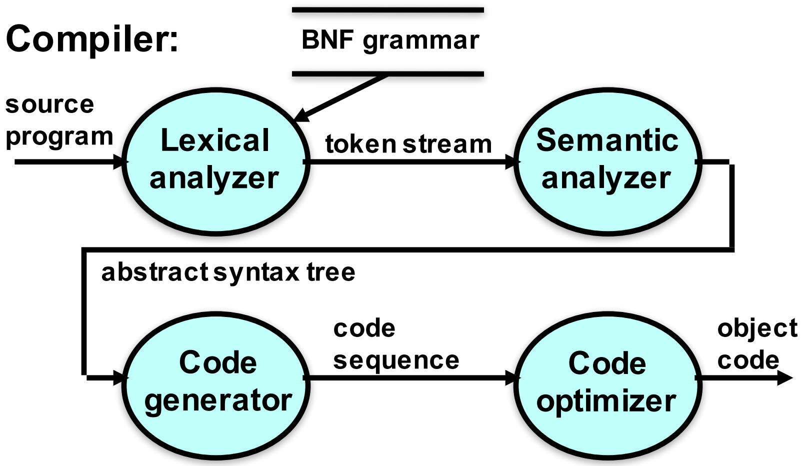
## Architectural Styles and Strategies
### Pipes-and-Filter

- The system has
  - Streams of data (pipe) for input and output
  - Transformation of the data (filter)



## Ex: Pipeline Topology (Architecture)

**Compiler:**





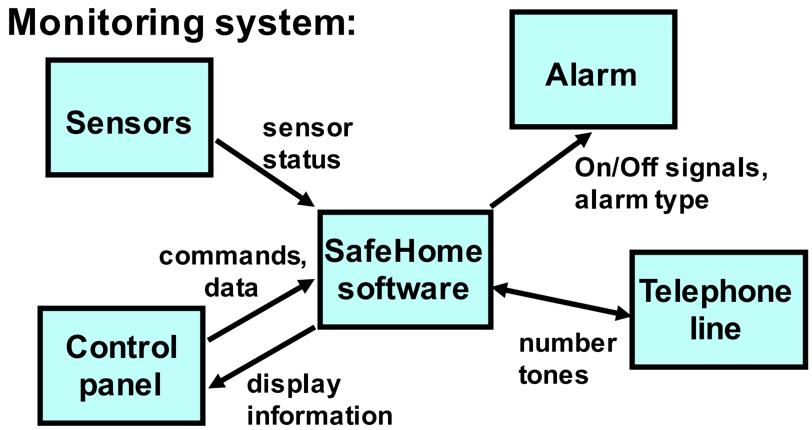

## Architectural Styles and Strategies

### Pipes-and-Filter (continued)

- Several important properties
  - The designer can understand the entire system's effect on input and output as the composition of the filters
  - The filters can be reused easily on other systems
  - System evolution is simple
  - Allow concurrent execution of filters
- Drawbacks
  - Encourages batch processing
  - Not good for handling interactive application
  - Duplication in filters functions

Ex: compilers; scripting applications; series of data transformations

## Ex: Star Topology (Architecture)



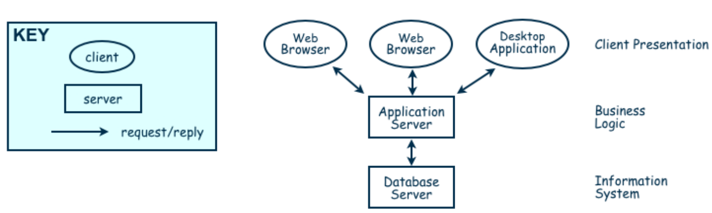

Ex: embedded systems; ECUs; cyber-physical systems (CPS)



## Architectural Styles and Strategies

Client-Server

- Two types of components:
  - Server components offer services
  - Clients access them using a request/reply protocol
- Client may send the server an executable function, called a callback
  - The server subsequently calls under specific circumstances




Ex: web-based applications; email; ftp

## Architectural Styles and Strategies

### Peer-to-Peer (P2P)

- Each component acts as its own process and acts as both a client and a server to other peer components.
- Any component can initiate a request to any other peer component.
- Characteristics
  - Scale up well
  - Increased system capabilities
  - Highly tolerant of failures

Examples: Napster, Skype, BitTorrent



## Architectural Styles and Strategies

### Publish-Subscribe

- Components interact by broadcasting and reacting to events
  - Component expresses interest in an event by subscribing to it
  - When another component announces (publishes) that event has taken place, subscribing components are notified
  - Implicit invocation is a common form of publish-subscribe architecture
    - Registering: subscribing component associates one of its procedures with each event of interest (called the procedure)
- Characteristics
  - Strong support for evolution and customization
  - Easy to reuse components in other event-driven systems
  - Need shared repository for components to share persistent data
  - Difficult to test

Ex: News feeds; social media updates; online notifications

# Architectural Styles and Strategies
Repositories

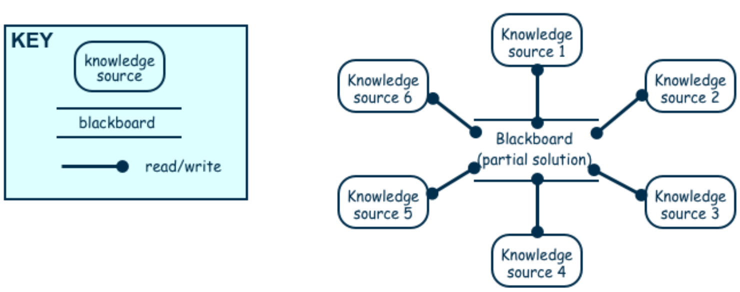
- Two components
  - A central data store
  - A collection of components that operate on it to store, retrieve, and update information
- The challenge is deciding how the components will interact
  - A traditional database: transactions trigger process execution
  - A blackboard: the central store controls the triggering process
  - Knowledge sources: information about the current state of the system's execution that triggers the execution of individual data accessors

Ex: SW repository, EMRs



# Architectural Styles and Strategies
Repositories (continued)

- Major advantage: openness
  - Data representation is made available to various programmers (vendors) so they can build tools to access the repository
  - But also a disadvantage: the data format must be acceptable to all components



# Architectural Design Principles for Layered Systems

- Decompose into subsystems *layers* and *partitions*.
- Separate application logic from user interface
- Simplify the interfaces through which parts of the system will connect to other systems.
- In systems that use large databases:
  - Distinguish between *operational (transactional)* and *inquiry* systems.
  - Exploit features of DBMS



# Architectural Styles and Strategies Layering

- Layers are hierarchical
  - Each layer provides service to the one outside it and acts as a client to the layer inside it
  - Layer bridging: allowing a layer to access the services of layers below its lower neighbor
- The design includes protocols
  - Explain how each pair of layers will interact
- Advantages
  - High levels of abstraction
  - Relatively easy to add and modify a layer
- Disadvantages
  - Not always easy to structure system layers
  - System performance may suffer from the extra coordination among layers

# Layered Subsystems

- Set of “virtual” worlds
- Each layer is defined in terms of the layer(s) below it
  - Knowledge is one-way: Layer knows about layer(s) below it
- Objects within layer can be independent
- Lower layer (server) supplies services for objects (clients) in upper layer(s)



# Example: Layered architecture

**Interactive Graphics Application**

**Windows Operations**

**Screen Operations**

**Pixel Operations**

**Device I/O Operations**

## Closed Architectures

- Each layer is built only in terms of the immediate lower layer
- Reduces dependencies between layers
- Facilitates change



## Open Architectures

- Layer can use any lower layer
- Reduces the need to redefine operations at each level
- More efficient /compact code
- System is less robust/harder to change

## Properties of Layered Architectures

- **Top and bottom layers specified by the problem statement**
  - **Top layer is the desired system**
  - **Bottom layer is defined by available resources (e.g. HW, OS, libraries)**
- **Easier to port to other HW/SW platforms**

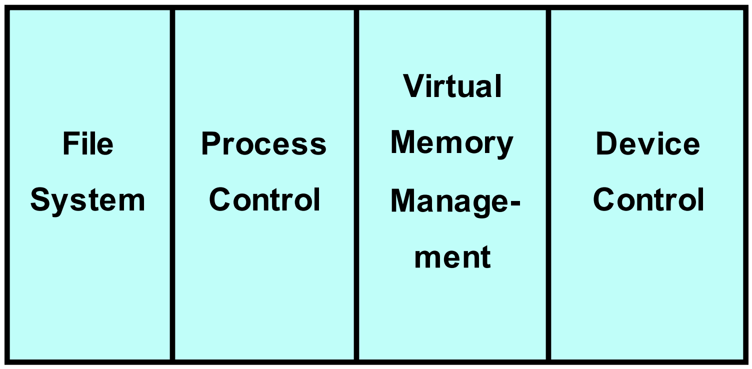


## Partitioned Architectures

- Divide system into weakly-coupled subsystems
- Each provides specific services
- Vertical decomposition of problem

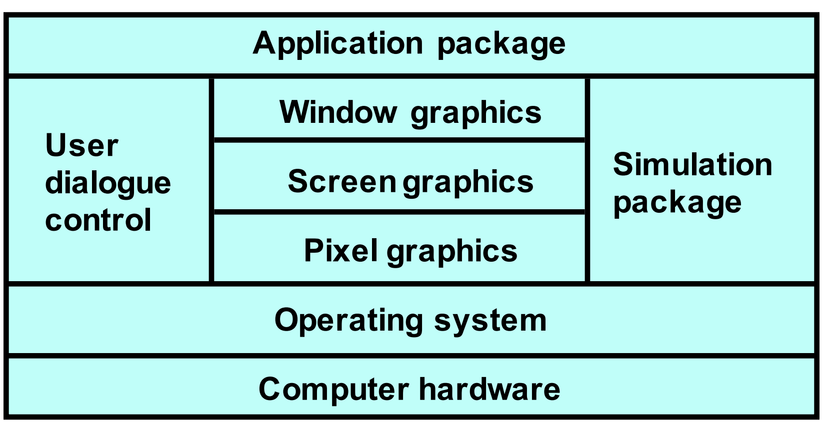
## Ex: Partitioned Architecture

**Operating System**






## Typical Application Architecture



## Architectural Styles and Strategies

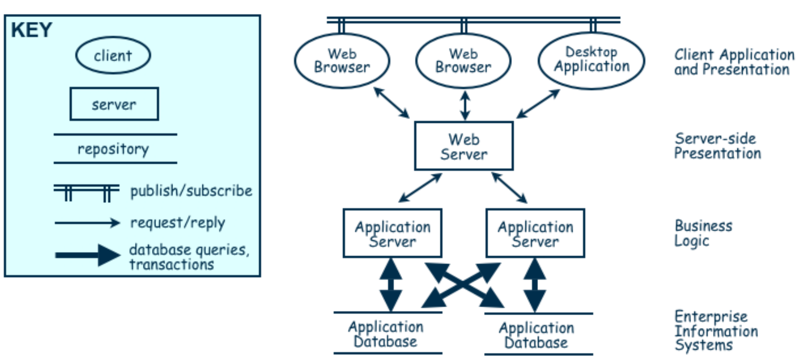
### Combining Architectural Styles

- Actual software architectures rarely based on purely one style
- Architectural styles can be combined in several ways
  - Use different styles at different layers (e.g., overall client-server architecture with server component decomposed into layers)
  - Use mixture of styles to model different components or types of interaction (e.g., client components interact with one another using publish-subscribe communications
- If architecture is expressed as collection of models, documentation must be created to show relation between models



## Architectural Styles and Strategies

### Combination of Publish-Subscribe, Client-Server, and Repository Architecture Styles




CSE 870: Advanced Software Engineering (System Design): Cheng

## Taxonomy of System-Design Decisions

- Devise a system architecture
- **Choose a data management approach**
- Choose an implementation of external control



## Choosing a Data Management Approach

- Databases:
  - Advantages:
    - Efficient management
    - multi-user support.
    - Roll-back support
  - Disadvantages:
    - Performance overhead
    - Awkward (or more complex) programming interface
    - Hard to fix corruption

## Choosing a Data Management Approach (continued)

- “Flat” files
  - Advantages:
    - Easy and efficient to construct and use
    - More readily repairable
  - Disadvantages:
    - No rollback
    - No *direct* complex structure support
    - Complex structure requires a ***grammar*** for file format



## Flat File Storage and Retrieval

- Useful to define two components (or classes)
  - ***Reader*** reads file and instantiates internal object structure
  - ***Writer*** traverses internal data structure and writes out presentation
- Both can (should) use formal grammar
  - Tools support: Yacc, Lex.

## Taxonomy of System-Design Decisions

- Devise a system architecture
- Choose a data management approach
- **Choose an implementation of external control**



## Implementation of External Control

***Four general styles for implementing software control***

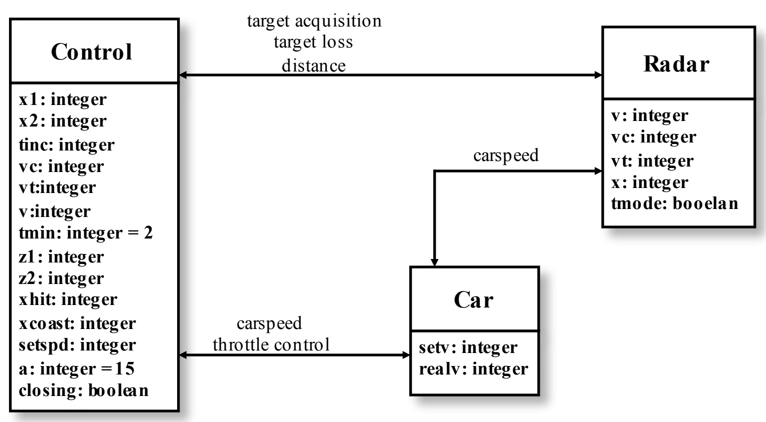
- **Procedure-driven:**
  - Control = location in the source code.
  - Requests block until request returns
- **Event-Driven: Control resides in dispatcher**
  - Uses callback functions registered for events
  - Dispatcher services events by invoking callbacks

# Implementation of External Control

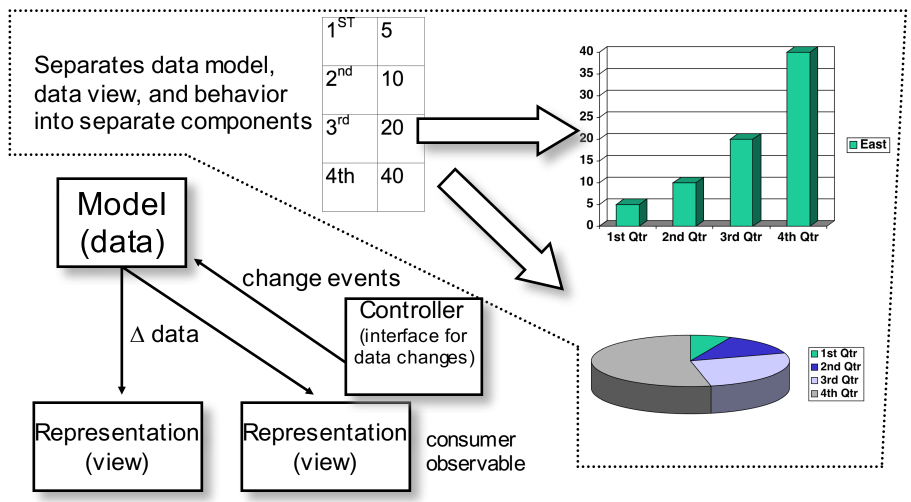
- **Concurrent**
  - Control resides in multiple, concurrent objects
  - Objects communicate by passing messages
    - across busses, networks, or memory.
- **Transactional**
  - Control resides in servers and saved state
  - Many server-side E-systems are like this



# Sample Concurrent System



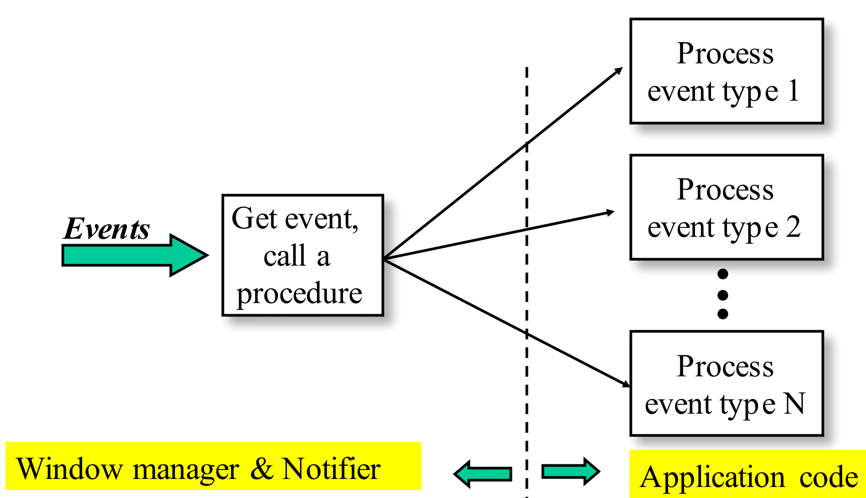
# MVC (Model/View/Controller)



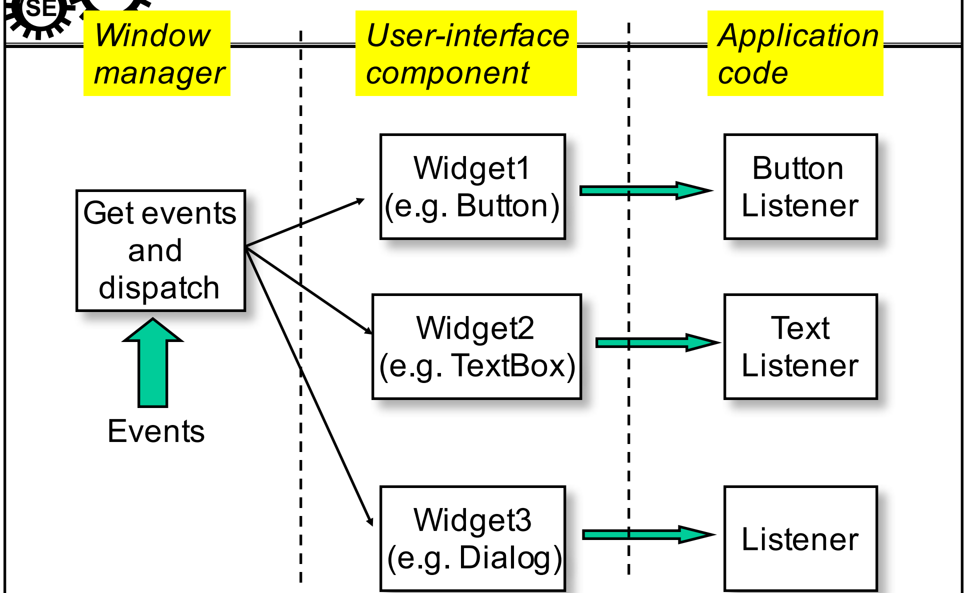



# Dispatcher Model

(event driven)

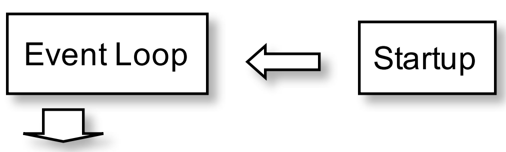


## Event-driven architecture in UI toolkits



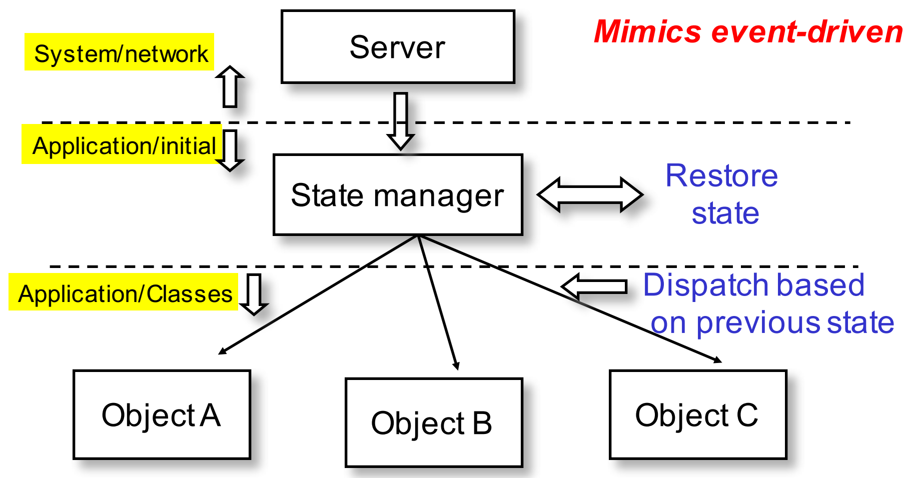



## Typical Dispatcher Code




```
while (!quit) {
   WaitEvent(timeout, id);
    switch (id) {
       case ID1:  Procedure1(); break;
       case ID2:  Procedure2(); break;
       ….
     }
}
```

# Transactional Model






# General Design Concerns

- Modularity
- Abstraction
- Cohesion
- Coupling
- Information Hiding
- Abstract Data Types
- Identifying Concurrency
- Global Resources
- Boundary Conditions
- Tradeoffs

# Modularity

- Organize modules according to resources/objects/data types
- Provide cleanly defined interfaces
  - operations, methods, procedures, ...
- Hide implementation details
- Simplify program understanding
- Simplify program maintenance



# Abstraction

- Control abstraction
  - structured control statements
  - exception handling
  - concurrency constructs
- Procedural abstraction
  - procedures and functions
- Data abstraction
  - user defined types

## Abstraction (cont.)

- Abstract data types
  - encapsulation of data
- Abstract objects
  - subtyping
  - generalization/inheritance



## Cohesion

- Contents of a module should be *cohesive*
  - Somehow related
- Improves maintainability
  - Easier to understand
  - Reduces complexity of design
  - Supports reuse

## (Weak) Types of cohesiveness

- Coincidentally cohesive
  - contiguous lines of code not exceeding a maximum size
- Logically cohesive
  - all output routines
- Temporally cohesive
  - all initialization routines

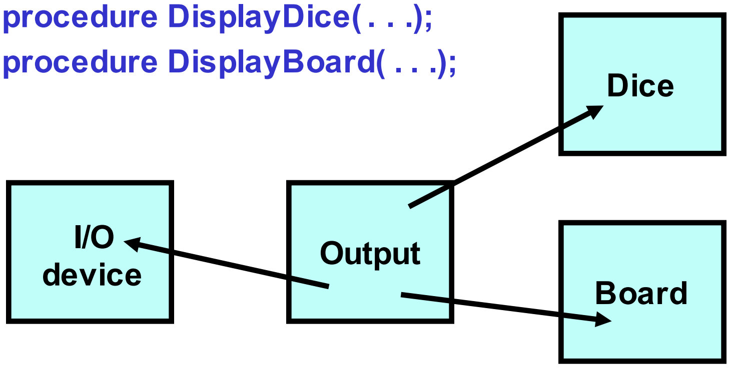


## (Better) Types of cohesiveness

- Procedurally cohesive
  - routines called in sequence
- Communicationally cohesive
  - work on same chunk of data
- Functionally cohesive
  - work on same data abstraction at a consistent level of abstraction

## Example: Poor Cohesion

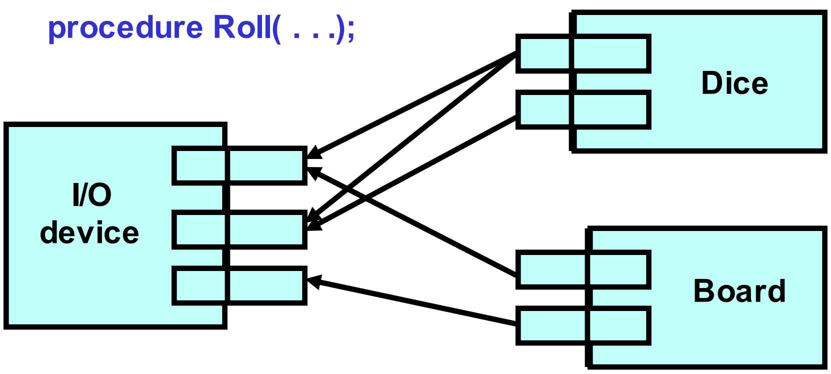
package Output is
  procedure DisplayDice( . . .);
  procedure DisplayBoard( . . .);






## Example: Good Cohesion

package Dice is
  procedure Display ( . . .);
  procedure Roll( . . .);



## Coupling

- *Connections* between modules
- Bad coupling
  - Global variables
  - Flag parameters
  - Direct manipulation of data structures by multiple classes



## Coupling (cont.)

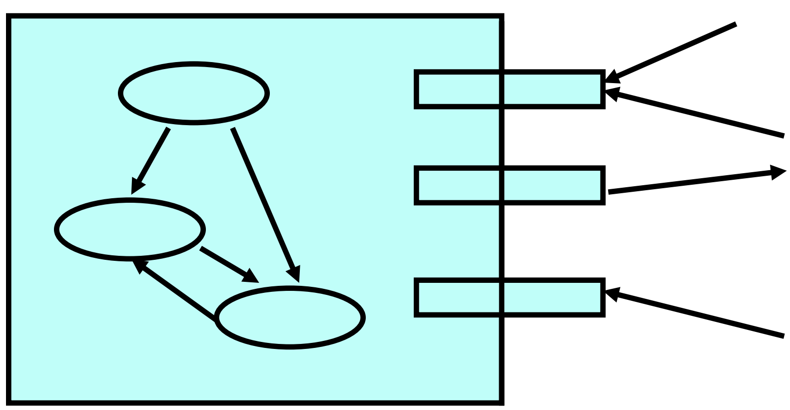
- ***Good coupling***
  - Procedure calls
  - Short argument lists
  - Objects as parameters
- Good coupling improves maintainability
  - Easier to localize errors, modify implementations of an objects, ...

# Information Hiding

- Hide decisions likely to change
  - Data representations, algorithmic details, system dependencies
- Black box
  - Input is known
  - Output is predictable
  - Mechanism is unknown
- Improves maintainability



# Information Hiding



CSE 870: Advanced Software Engineering (System Design): Cheng

## Abstract data types

- Modules (Classes, packages)
  - Encapsulate data structures and their operations
  - Good cohesion
    - implement a single abstraction
  - Good coupling
    - pass abstract objects as parameters
  - Black boxes
    - hide data representations and algorithms



## Identifying Concurrency

- Inherent concurrency
  - May involve synchronization
  - Multiple objects receive events at the same time with out interacting
  - Example:
    - User may issue commands through control panel at same time that the sensor is sending status information to the SafeHome system

## Determining Concurrent Tasks

- ***Thread of control***
  - Path through state diagram with only one active object at any time
- Threads of control are implemented as *tasks*
  - Interdependent objects
  - Examine state diagram to identify objects that can be implemented in a task



## Global Resources

- Identify global resources and determine access patterns
- Examples
  - physical units (processors, tape drives)
  - available space (disk, screen, buttons)
  - logical names (object IDs, filenames)
  - access to shared data (database, file)

## Boundary Conditions

- Initialization
  - Constants, parameters, global variables, tasks, guardians, class hierarchy
- Termination
  - Release external resources, notify other tasks
- Failure
  - Clean up and log failure info



## Identify Trade-off Priorities

- Establish priorities for choosing between incompatible goals
- Implement minimal functionality initially and embellish as appropriate
- Isolate decision points for later evaluation
- Trade efficiency for simplicity, reliability, . . .