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?
**Taxonomy of System-Design Decisions**

- Devise a system architecture
- 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

**Architectural Design Principles**

- 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

**Choosing a Data Management Approach**

**Choosing a Data Management Approach (continued)**

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

- "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.

Java Data Marshalling

- Provides a means of “serializing” a set of objects
- Requires classes to implement the “Serializable” interface.
- Stream can be written/read to a file
- Stream can be written/read to a network socket.

Serialization Example

```java
private void writeObject(ObjectOutputStream out) throws IOException,
    ClassNotFoundException;

FileOutputStream osStream = new FileOutputStream(“tmp”);
ObjectOutputStream outStream = new ObjectOutputStream(osStream);
outStream.writeObject(“Today”);
outStream.writeObject(new Date());
outStream.flush();
osStream.close();
```

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
- **Concurrent:**
  - Control resides in multiple, concurrent objects
  - Objects communicate by passing messages
  - across buses, 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 modern UI toolkits

Typical Dispatcher Code

Transactional Model

**Sample Concurrent System**

Control
- x1: integer
- x2: integer
- t in c: integer
- v c: integer
- v t: integer
- v: integer
- t m in: integer = 2
- z 1: integer
- z 2: integer
- x h i t: integer
- x c o a s t: integer
- s e t s p d: integer
- a: integer = 15
- c lo s in g: boolean
- Radar

**MVC (Model/View/Controller)**

Separates data model, data view, and behavior into separate components

Model (data)
- change events
- data
- consumer observable
- Representation (view)
- Representation (view)
- Application code

**Dispatcher Model (event driven)**

Events
- Get event
- call a procedure
- Process event type 1
- Process event type 2
- Window manager
- Notifier
- Application code
- Process event type N

**Event-driven architecture in modern UI toolkits**

- Get events and dispatch
- Window manager
- Events
- Application code
- User-interface component
- Application code
- Button Listener
- Text Listener
- Widget1 (e.g. Button)
- Widget2 (e.g. TextBox)
- Widget3 (e.g. Dialog)

**Typical Dispatcher Code**

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

**Transactional Model**

Mimics event-driven

- Server
- State manager
- Restore state
- Dispatch based on previous state
- Object A
- Object B
- Object C
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

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

- File System
- Process Control
- Virtual Memory Management
- Device Control

Typical Application Architecture

Application package
- User dialogue control
  - Window graphics
  - Screen graphics
- Simulation package
  - Pixel graphics
- Operating system
- Computer hardware

System Topology

- Describe information flow
  - Can use DFD to model flow
- Some common topologies
  - Pipeline (batch)
  - Star topology

Ex: Pipeline Topology

Compiler:

- source program
- Lexical analyzer
- token stream
- Semantic analyzer
- abstract syntax tree
- Code generator
- code sequence
- Code optimizer
- object code

Ex: Star Topology

Monitoring system:

- Sensors
- Control panel
- SafeHome software
- Alarm
- Telephone line

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
- 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

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

```
package I/O device

procedure Output

package Board
```
**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

**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 without 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, . . .