System Design

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 prior to technical design.

- Assumes and builds upon thorough requirements and analysis.
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

CSE 870: Advanced Software Engineering
System Topology (also known as SW Architecture)

- Describe information flow
  - Can use DFD to model flow

- Some common topologies
  - Pipeline (batch)
  - Star topology
  - Client-server; P2P
  - Layered

Ex: Pipeline Topology (Architecture)

Compiler:

- Source program
- Lexical analyzer
- Token stream
- Semantic analyzer
- Abstract syntax tree
- Code generator
- Code sequence
- Code optimizer
- Object code
Ex: Pipeline Topology (DFD notation example)

Compiler:

- Lexical analyzer
  - source program
  - token stream
  - abstract syntax tree
- Semantic analyzer
- Code generator
- Code optimizer
- Platform

Ex: Star Topology (Architecture: Components/Connectors)

Monitoring system:

- Sensors
  - sensor status
- Control panel
  - commands, data
- SafeHome software
  - display information
  - number tones
- Alarm
  - On/Off signals, alarm type
- Telephone line
  - number tones
Client-Server Architecture

- 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

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)
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

Example: Layered architecture

<table>
<thead>
<tr>
<th>Interactive Graphics Application</th>
</tr>
</thead>
<tbody>
<tr>
<td>Windows Operations</td>
</tr>
<tr>
<td>Screen Operations</td>
</tr>
<tr>
<td>Pixel Operations</td>
</tr>
<tr>
<td>Device I/O Operations</td>
</tr>
</tbody>
</table>
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
    - Pixel graphics
    - Operating system
    - Computer hardware
  - Simulation package
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

```
Control
x1: integer
x2: integer
tinc: integer
vc: integer
vt: integer
v: integer
tmin: integer = 2
z1: integer
z2: integer
xhit: integer
xcoast: integer
setspd: integer
a: integer = 15
closing: boolean

Radar
v: integer
vc: integer
vt: integer
x: integer
tmode: boolean
setv: integer
realv: integer

Car
carspeed
set: integer
real: integer

target acquisition
target loss
distance
carspeed
throttle control
```
**MVC (Model/View/Controller)**

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

- **Model (data)**: Represents data.
- **Representation (view)**: Observable for changes.
- **Controller**: Interface for data changes.

**Dispatcher Model** (event driven)

- **Events**: Get event, call a procedure.
- **Process event type 1**
- **Process event type 2**
- **Process event type N**

**Window manager & Notifier**

**Application code**
Event-driven architecture in UI toolkits

Window manager

Get events and dispatch

Events

User-interface component

Widget1 (e.g. Button)

Widget2 (e.g. TextBox)

Widget3 (e.g. Dialog)

Application code

Button Listener

Text Listener

Listener

Typical Dispatcher Code

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

Server

Mimics event-driven

State manager

Object A
Object B
Object C

System/network
Application/initial
Application/Classes

Dispatch based on previous state
Restore state

General Design Concerns

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

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

```
I/O device

Output

Dice

Board
```

Example: Good Cohesion

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

```
I/O device

Board

Dice
```
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 the 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, . . .