CSE 435
Nov 9, 2015

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.
Software Architect

How is a “software architect” different than a “software developer”?

A: Design the structure of the system and document it.
Top Down or Bottom Up?
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.
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

<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>
Mobile Robot Subsumption Architecture

"Subsumption Architecture Abstract Diagram" by KodoKB - Own work. Licensed under CC0 via Commons - https://commons.wikimedia.org/wiki/File:Subsumption_Architecture_Abstract_Diagram.png#/media/File:Subsumption_Architecture_Abstract_Diagram.png
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
  - Simulation package
  - Computer hardware
Software Architectures

Some common topologies

- Pipes and Filter
- Peer-to-peer
- Publish - Subscribe
- Repositories
Pipes-and-Filter

- The system has
  - Streams of data (pipe) for input and output
  - Transformation of the data (filter)
Ex: Pipeline Topology (Architecture)

Compiler:

BNF grammar

source program

Lexical analyzer

token stream

Semantic analyzer

abstract syntax tree

Code generator

code sequence

Code optimizer

object code
Pipes and Filter

• 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
  – batch processing
  – Not good for handling interactive applications
  – Duplication in filters functions

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

Atlee and Pfleeger
Examples of Client-Server?
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
  - Scales up well
  - Increased system capabilities
  - Highly tolerant of failures

- Examples: Napster and Freenet
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
Repositories

• Two components
  ▪ A central data store
  ▪ A collection of components that operate on it to store, retrieve, and update information

• Decide 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
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

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)
What architecture(s) have you considered for your project?
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 (handles mutual exclusion)
    ○ Roll-back support
  ▪ Disadvantages:
    ○ Performance overhead
    ○ Awkward (or more complex) programming interface
    ○ Hard to fix corruption

CSE 435: Software Engineering
“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 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
  ▪ Example: standard computational and/or scientific applications

● Event-Driven: Control resides in dispatcher
  ▪ Uses callback functions registered for events
  ▪ Dispatcher services events by invoking callbacks
  ▪ Example: graphical user interfaces; windowing systems
Implementation of External Control

● Concurrent
  ▪ Control resides in multiple, concurrent objects
  ▪ Objects communicate by passing messages
    ○ across busses, networks, or memory.
  ▪ Example: onboard electronics in vehicles

● Transactional
  ▪ Control resides in servers and saved state
  ▪ Example: Many server-side E-systems are like this
Dispatcher Model
(event driven)

Get event, call a procedure

Process event type 1
Process event type 2
Process event type N

Events

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

CSE 435: Software Engineering
Typical Dispatcher Code

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

Server

State manager

Object A

Object B

Object C

System/network

Application/initial

Application/Classes

Mimics event-driven

Restores state

Dispatch based on previous state