# Software Design and Architecture

Theoretical Principles and Design Methods

## What is Design?

- Design is the process of creating a plan or blueprint to follow during actual construction
- Design is a problem-solving activity that is iterative in nature
- The outcome of design is the **design document** or **technical specification** (if emphasis on notation)

## "Wicked Problem"

- Software design is a "Wicked Problem"
  - Design phase can't be solved in isolation
    - Designer will likely need to interact with users for requirements, programmers for implementation
  - No stopping rule
    - How do we know when the solution is reached?
  - Solutions are not true or false
    - Large number of tradeoffs to consider, many acceptable solutions
  - Wicked problems are a symptom of another problem
    - Resolving one problem may result in a new problem elsewhere; software is not continuous

## Systems-Oriented Approach

- The central question: how to decompose a system into parts such that each part has lower complexity than the system as a whole, while the parts together solve the user's problem?
- In addition, the interactions between the components should not be too complicated
- Vast number of design methods exist

# Design Considerations

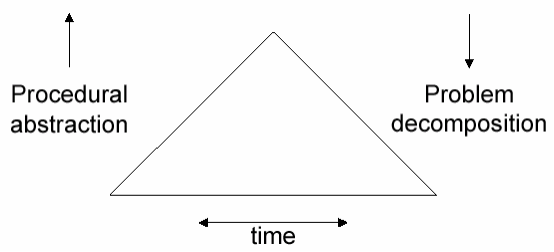
- "Module" used often – usually refers to a method or class
- In the decomposition we are interested in properties that make the system flexible, maintainable, reusable
  - Abstraction
  - Modularity
  - Information Hiding
  - Complexity
  - System Structure

# Abstraction

- Abstraction
  - Concentrate on the essential features and ignore, abstract from, details that are not relevant at the level we are currently working on
  - E.g. Sorting Module
    - Consider inputs, outputs, ignore details of the algorithms until later
  - Two general types of abstraction
    - Procedural Abstraction
    - Data Abstraction

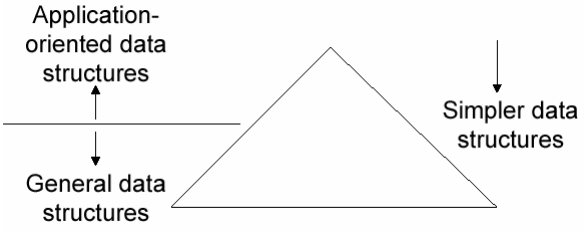
# Procedural Abstraction

- Fairly traditional notion
  - Decompose problem into sub-problems, which are each handled in turn, perhaps decomposing further into a hierarchy
  - Methods may comprise the sub-problems and sub-modules, often in time




# Data Abstraction

- From primitive to complex to abstract data types
  - E.g. Integers to Binary Tree to Data Store for Employee Records
- Find hierarchy in the data



# Modularity

- During design the system is decomposed into modules and the relationships among modules are indicated
- Two structural design criteria as to the "goodness" of a module
  - Cohesion : Glue for intra-module components
  - Coupling : Strength of inter-module connections

# Levels of Cohesion

1. Coincidental
   - Components grouped in a haphazard way
2. Logical
   - Tasks are logically related; e.g. all input routines. Routines do not invoke one another.
3. Temporal
   - Initialization routines; components independent but activated about the same time
4. Procedural
   - Components that execute in some order
5. Communicational
   - Components operate on the same external data
6. Sequential
   - Output of one component serves as input to the next component
7. Functional
   - All components contribute to one single function of the module
   - Often transforms data into some output format

# Coupling

- Measure of the strength of inter-module connections
- High coupling indicates strong dependence between modules
  - Should study modules as a pair
  - Change to one module may ripple to the next
- Loose coupling indicates independent modules
  - Generally we desire loose coupling, easier to comprehend and adapt

# Types of Coupling

1. Content
   - One module directly affects the workings of another
   - Occurs when a module changes another module's data
   - Generally should be avoided
2. Common
   - Two modules have shared data, e.g. global variables
3. External
   - Modules communicate through an external medium, like a file
4. Control
   - One module directs the execution of another by passing control information (e.g. via flags)
5. Stamp
   - Complete data structures or objects are passed from one module to another
6. Data
   - Only simple data is passed between modules

# Modern Coupling

- Modern programming languages allow private, protected, public access
- Coupling may be modified to indicate levels of visibility, whether coupling is commutative
- Simple Interfaces generally desired
  - Weak coupling and strong cohesion
  - Communication between programmers simpler
  - Correctness easier to derive
  - Less likely that changes will propagate to other modules
  - Reusability increased
  - Comprehensibility increased

# Information Hiding

- Each module has a secret that it hides from other modules
  - Secret might be inner-workings of an algorithm
  - Secret might be data structures
- By hiding the secret, changes do not permeate the module's boundary, thereby
  - Decreasing the coupling between that module and its environment
  - Increasing abstraction
  - Increasing cohesion (the secret binds the parts of a module)
- Design involves a series of decisions. For each such decision, questions are: who needs to know about these decisions? And who can be kept in the dark?

# Complexity

- Complexity refers to attributes of software that affect the effort needed to construct or change a piece of software
  - Internal attributes; need not execute the software to determine their values
- Many different metrics exist to measure complexity
- Two broad classes
  - Intra-Modular attributes
  - Inter-Modular attributes

# Intra-Modular Complexity

- Two types of intra-modular attributes
  - Size-Based Metrics
    - E.g. Lines of Code
      - Serious objections. Why?
  - Structure-Based Metrics
    - E.g. complexity of control or data structures

# Halstead's Software Science

- Size-based metric
- Uses number of operators and operands in a piece of software
  - $n_1$ is the number of unique operators
  - $n_2$ is the number of unique operands
  - $N_1$ is the total number of occurrences of operators
  - $N_2$ is the total number of occurrences of operands
- Halstead derives various entities
  - Size of Vocabulary: $n = n_1+n_2$
  - Program Length: $N = N_1+N_2$
  - Program Volume: $V = N\log_2 n$
  - Program Level: $L = V^*/V$
    - $V^*$ is the volume for the most compact representation for the algorithm
  - Programming Effort: $E = V/L$
  - Programming Time in Seconds: $T = E/18$
  - Numbers derived empirically, also based on speed human memory processes sensory input

# Software Science Example

```
1.  procedure sort(var x:array; n: integer)
2.          var i,j,save:integer;
3.          begin
4.                  for i:=2 to n do
5.                          for j:=1 to i do
6.                                  if x[i]<x[j] then
7.                                  begin  save:=x[i];
8.                                          x[i]:=x[j];
9.                                          x[j]:=save
10.                                 end
11.         end
```

# Software Science Example

| Operator | # |
|---|---|
| procedure | 1 |
| sort() | 1 |
| var | 2 |
| : | 3 |
| array | 1 |
| ; | 6 |
| integer | 2 |
| , | 2 |
| begin…end | 2 |
| for..do | 2 |
| if…then | 1 |
| := | 5 |
| < | 1 |
| [] | 6 |
| n1=14 | N1=35 |

| Operand | # |
|---|---|
| x | 7 |
| n | 2 |
| i | 6 |
| j | 5 |
| save | 3 |
| 2 | 1 |
| 1 | 1 |
| n2=7 | N2=25 |

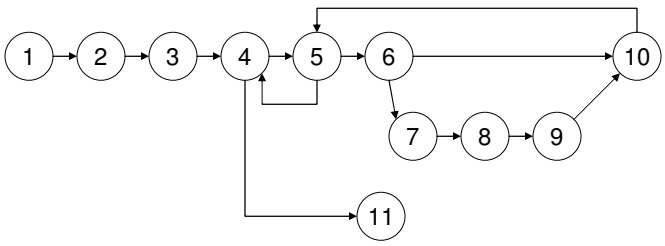
| | |
|---|---|
| Size of vocabulary: | 21 |
| Program length: | 60 |
| Program volume: | 264 |
| Program level: | 0.04 |
| Programming effort: | 6000 |
| Estimated time: | 333 seconds |

# Structure-Based Complexity

- McCabe's Cyclomatic Complexity
- Create a directed graph depicting the control flow of the program
  - CV = e – n + p + 1
    - CV = Cyclomatic Complexity
    - e = Edges
    - n = nodes
    - p = connected components

# Cyclomatic Example

For Sorting Code; numbers refer to line numbers




CV = 13 – 11 + 1 + 1 = 4

McCabe suggests an upper limit of 10

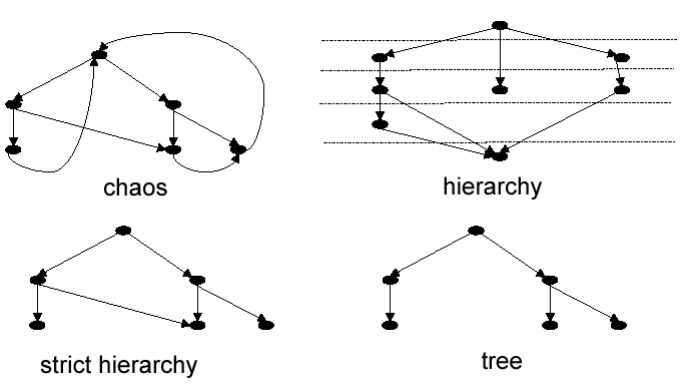
# Shortcomings of Complexity Metrics

- Not context-sensitive
  - Any program with five if-statements has the same cyclomatic complexity
  - Measure only a few facts; e.g. Halstead's method doesn't consider control flow complexity
- Others?
- Minix:
  - Of the 277 modules, 34 have a CV > 10
  - Highest has 58; handles ASCII escape sequences. A review of the module was deemed "justifiably complex"; attempts to reduce complexity by splitting into modules would increase difficulty to understand and artificially reduce the CV

# System Structure – Inter-Module Complexity

- The design may consist of modules and their relationships
- Can denote this in a graph; nodes are modules and edges are relationships between modules
- Types of inter-module relationships:
  - Module A contains Module B
  - Module A follows Module B
  - Module A delivers data to Module B
  - Module A uses Module B
- We are mostly interested in the last one, which manifests itself via a call graph
  - Possible shapes:
    - Chaotic
    - Directed Acyclic Graph (Hierarchy)
    - Layered Graph (Strict Hierarchy)
    - Tree

# Module Hierarchies



# Graph Metrics

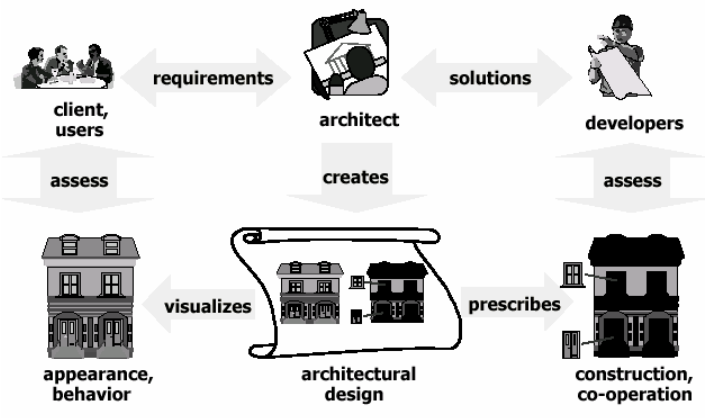
- Metrics use:
  - Size of the graph
  - Depth
  - Width (maximum number of nodes at some level)
- A tree-like call graph is considered the best design
  - Some metrics measure the deviation from a tree; the **tree impurity** of the graph
  - Compute number of edges that must be removed from the graph's minimum spanning tree
- Other metrics
  - Complexity(M) = fanin(M)*fanout(M)
  - Fanin/Fanout = local and global data flows

# Software Architecture

- Selecting an appropriate architecture can help reduce our complexity
- Definition of Architecture
  - The software architecture of a system is the structure or structures of the system that comprise software components, the externally visible properties of those components, and the relationships among them"

# Role of the Architect

- Global design




# Architectural Styles

- High-level abstractions of components and communication
  - Even higher than data types, algorithmic pseudocode
  - Also known as **design patterns** or **architectural patterns**
- Architectural styles become reusable for different problems
  - Collections of modules or classes that are often used in combination to provide a useful abstraction

## Some common architectural styles

- Main with Subroutines and Shared data
- Data abstraction
- Implicit invocation
- Pipes and filters
- Repository (blackboard)
- Layers of abstraction
- Model-view-controller

## Example Problem: Producing a KWIC-Index

- Book title “Introduction to Software Engineering”
- Reader might look under “I” or under “S” for Software Engineering, or maybe even “E” for Engineering
- Solution: KWIC-Index (Key Word In Context)
  - Given a title generate n shifts, where n is the number of words in the title
  - $w_1$ $w_2$ $w_3$
  - $w_2$ $w_3$ $w_1$
  - $w_3$ $w_1$ $w_2$
  - Sort shifts, output is a KWIC-index for one title
  - Repeat for a list of titles

# KWIC-Index

- Two titles, “Introduction to Software Engineering” and “Rapid Software Development”
- Introduction to Software Engineering
  - Engineering Introduction to Software
  - Introduction to Software Engineering
  - Software Engineering Introduction to
  - to Software Engineering Introduction
- Rapid Software Development
  - Development Rapid Software
  - Rapid Software Development
  - Software Development Rapid

# First Architecture: Main Program and Subroutines with Shared Data

- Following tasks must be accomplished
  - Read and store the input (list of titles)
  - Determine all shifts
  - Sort the shifts
  - Output the sorted shifts
- Allocate each task to different modules
- Each module shares the same internal data representation

# Module Details

- Input
  - Input stored in memory so that the lines are available for processing by later modules
  - Stored in a table called Store

| | |
|---|---|
| 0 | Introduction to Software Engineering |
| 1 | Rapid Software Development |

- Shift
  - Invoked after all lines are stored
  - Builds a table called Shifts that contains an array of shifts that contains, for each shift, the index in Store of the first character of that shift

| | |
|---|---|
| 0 | [0, 0], [1, 13], [2, 16], [3, 25] |
| 1 | [0, 0], [1, 7], [2, 16] |

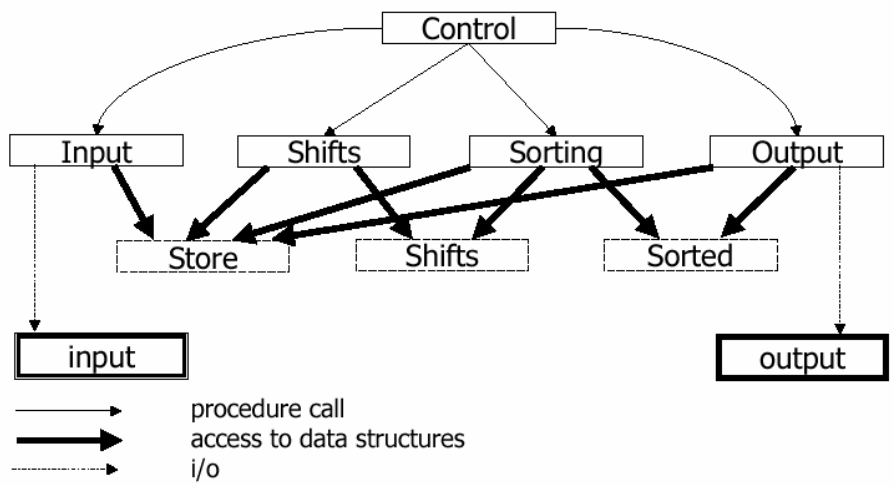
# Module Details

- Sort
  - Produces a new table, Sorted, with the same structure as Shifts but the ordering is such that the corresponding shifts are in lexicographic order

| | |
|---|---|
| 0 | [0, 25], [1, 0], [2, 16], [3, 13] |
| 1 | [0, 16], [1, 0], [2, 7] |

- Output
  - Produces a neat output of the sorted shifts
- Control
  - Calls the other modules in the appropriate order
  - Deals with error messages, memory organization, bookkeeping duties

# Module Diagram




# Main Program Style

- Approach: Hierarchy of functions resulting from functional decomposition; single thread of control
- Context: programming languages allowing nested procedures
- Properties:
  - Modules in a hierarchy (weak or strong); procedures grouped into modules according to coupling/cohesion principles
  - Modules store local data and shared access to global data
  - Control structure: single thread, centralized control
- Variants: distributed processing with RPC for process invocation

## Second Architecture: Abstract Data Types

- Previous decomposition required that each module had knowledge about the precise storage of data
  - Data representation must be selected early
  - What if wrong representation picked? E.g. fixed char array, perhaps too small, etc.
- One solution: Use abstract data types so that these data representation decisions are made locally within a module instead of globally
  - Implement Get/Set methods that input or return data in the desired format

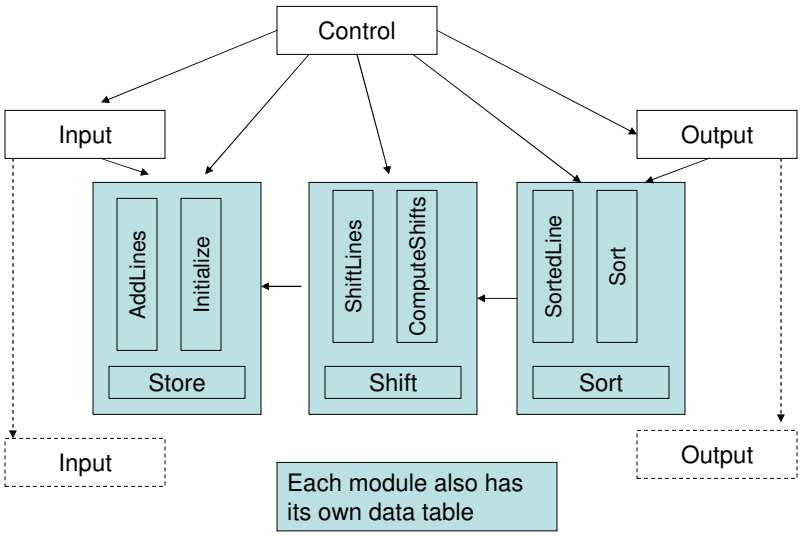
## Module Details : ADT

- Store
  - Module implements methods callable by users of the module
    - Initialize()
    - AddLine(String s) - Add a new line to storage
    - Lines() – Return number of lines
    - Line(int L) – Return line L
    - Words(int r) – Return number of words in line r
    - Word(int L, int i) – Return word i of line L
- Input
  - Initializes Store module, adds lines to the module from I/O

# Module Details : ADT

- Shift
  - Uses the Store module to compute shifts
  - Initialize()
  - ComputeShifts() – Computes all shifts
  - ShiftLines() – Returns total number of Lines
  - ShiftWords(int L) – Returns number of words (shifts) in shift line L
  - ShiftLine(int L, int i) – Returns entire title for shift line L ,shift i
  - ShiftWord(int L, int i,int j) – Returns word j of line L for shift i
- Sort
  - Uses Shift module to compute sort
  - Initialize()
  - Sort()
  - SortedLines() – Returns total number of lines
  - SortedWords(int L) – Returns number of words (shifts) in line L
  - SortedLine(int L, int i) – Returns entire title for line L ,shift i

# ADT Module Diagram (Simplified)



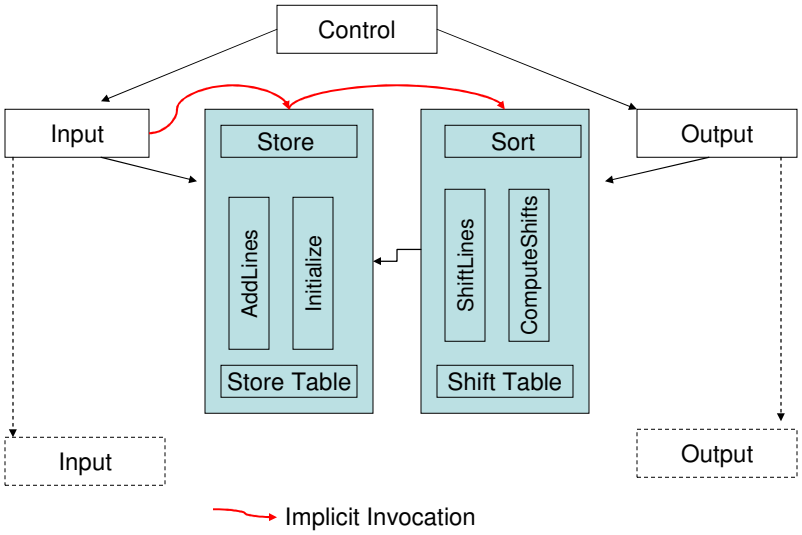
# ADT Style

- Approach: Identifies and protects related bodies of information. Suited when data representation is likely to change.
- Context: OO-methods guiding the design; OO-languages which provide the class-concept
- Properties:
  - Each component has its own local data (secret it hides)
  - Messages sent via procedure calls
  - Usually a single thread of control; control is decentralized

# Third Architecture: Implicit Invocation

- Event-based processing
- With Abstract data types, after the input was read, the Shifts module is invoked explicitly
- We may loosen the binding between modules by implicitly invoking modules
  - If something interesting happens, an **event** is raised
  - Any module interested in that event may react to it
  - Example: perhaps we would like to process data concurrently line by line
    - Raise an event when a line is ready to be processed by the next module

# Implicit Invocation Diagram



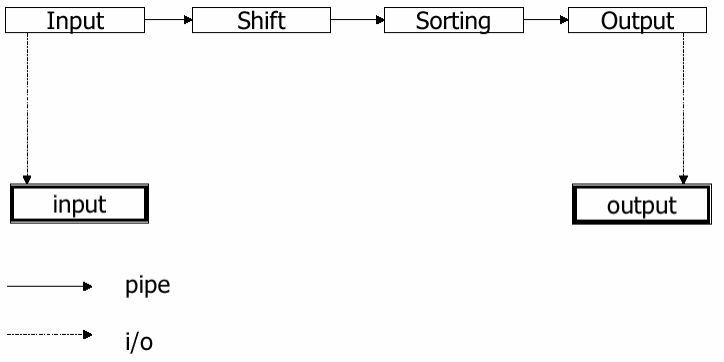

# Implicit Invocation Style

- Approach: Loosely coupled collection of components. Useful for applications which must be reconfigurable.
- Context: requires event handler, through OS or language-specific features.
- Properties:
  - Independent, reactive processes, invoked when an event is raised
  - Processes signal events and react to events, or directly invoked
  - Decentralized control. Components do not know who is going to react to a particular event

# Fourth Architecture: Pipes and Filters

- Major transformations in KWIC-index programs:
  - From lines to shifts
  - From shifts to sorted shifts
  - From sorted shifts to output
- Since each program in this scheme reads its input in the same order it is written by its predecessor, we can directly feed the output from one module to the input of the next
- Pipes and Filters model in UNIX
  - KWIC < input | Shift | Sort | Output > output
  - Data stream format of internal structure must be known from one program to the next
  - Enhancements are easy by adding another filter (e.g. filtering out stop words)

# Pipes and Filters Diagram



## Pipes and Filters Style

- Approach: independent, sequential transformations on ordered data. Usually incremental, ASCII pipes.
- Context: series of incremental transformations. OS-functions transfer data between processes. Error-handling difficult (who is doing what? Where did the error occur?)
- Properties:
  - Continuous data flow; components incrementally transform data
  - Filters for local processing
  - Data streams (usually plain ASCII)
  - Control structure: data flow between components; each component has its own thread of control
- Variants: From pure filters with little internal state to batch processes

## Evaluation of the Architectures

- All of the proposed architectures will work
- Architect should evaluate the architectures with respect to
  - Changes in data representation
  - Changes in algorithms
  - Changes in functionality
  - Degree to which modules can be implemented independently
  - Comprehensibility
  - Performance
  - Reuse

# Architecture Evaluation

| | Shared Data | ADT | Implicit Invocation | Pipes & Filters |
|---|---|---|---|---|
| Changes in data representation | - | + | + | - |
| Changes in algorithm | - | O | O | + |
| Changes in functionality | O | - | + | + |
| Independent Development | - | + | + | + |
| Comprehensibility | - | + | O | + |
| Performance | + | + | - | - |
| Reuse | - | + | + | + |

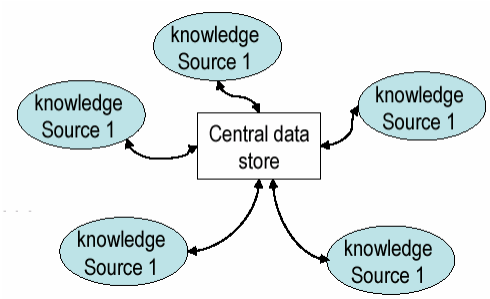
# Some Other Styles

- Repository
- Layered
- Client Server
- Model View Controller (MVC)

- We will cover other patterns in Chapter 8

# Repository Architecture

- Central data store
- Components to store, access, retrieve data in the data store

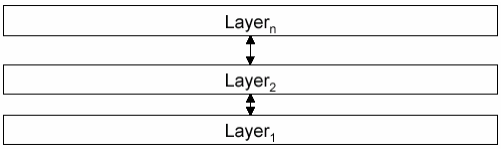



# Repository or Blackboard Style

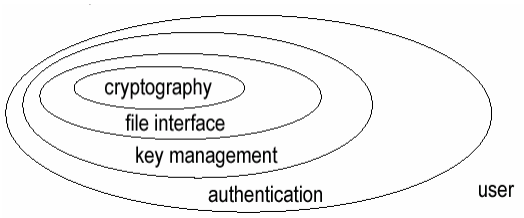
- Approach: manage richly structured information to be manipulated in many different ways. Data is long-lived.
- Context: shared data to be acted upon by multiple clients or processes
- Properties
  - Centralized body of information. Independent computational elements.
  - One shared memory component, many computational processes
  - Direct access or procedure call to the shared memory component
- Variants: traditional data base systems, compilers, blackboard systems

# Layered Architecture

- Build system in terms of hierarchical layers and interaction protocols




- E.g. TCP/IP Stack, Data Access




# Layered Style

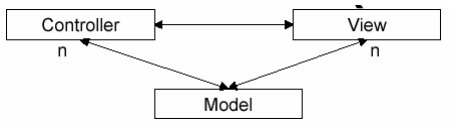
- Approach: distinct, hierarchical classes of services. "Concentric circles" of functionality
- Context: a large system that requires decomposition. E.g. OSI model, virtual machines
- Properties
  - Hierarchy of layers, often limited visibility, promotes layer reusability
  - Collections of procedures (module)
  - Limited procedure calls
  - Control structure: single or multiple threads
- Drawbacks
  - Performance
  - Not easy to construct system in layers
  - Hierarchical abstraction may not be evident from requirements

# Client-Server

- Popular form of distributed system architecture
  - Client requests an action or service
  - Server responds to the request
- Often server does not know number of potential clients or their identities
- Properties
  - Information exchanged in a need basis
  - Same data can be presented in different ways in different clients
- Drawbacks
  - Security
  - System management
  - “Sophisticated” application development
  - More resources to implement and support

# Model-View-Controller (MVC)

- Archetypical example of a design pattern
- Three components
  - Model : Encapsulates system data and operations on the data
  - View : Displays data obtained from the model to the user
  - Controller : Handles input actions (e.g. send requests to the model to update data, send requests to the view to update display)




- Separating user interface from computational elements considered a good design practice
  - Why?

# MVC Style

- Approach: Separation of UI from application is desirable due to expected UI adaptations
- Context: interactive applications with a flexible UI
- Properties:
  - UI (View&Controller) is decoupled from the application (Model component)
  - Collections of procedures (module)
  - Procedure calls
  - Generally single thread, but multiple threads possible

# Other Common Design Patterns

- Proxy Pattern
  - A client needs services from another component
  - Instead of hard-coding direct access, may have better efficiencies or security by additional control mechanisms separate from both the client and the component to access
  - The client communicates with a proxy representative rather than the component itself, which does necessary pre/post processing
- Command Processor Pattern
  - User interfaces must be flexible or provide functionality beyond direct user functions, e.g. Undo or Log facilities
  - A separate component, the command processor, takes care of all commands. The command processor schedules the execution of commands, stores them for undo, logs, etc. Execution delegated to another component.

# Summary

- Software architecture is concerned with the description of elements from which systems are built, the interaction among those elements, and patterns that guide their composition
- Complexity is still a novel and immature branch of Software Engineering
- Helps to have a repertoire of known architectures and use the one most appropriate
  - E.g. Shared Data, ADT, Implicit Invocation, Pipes/Filters, Layers, Repository, Client/Server, MVC
- Architecture is important
  - Starting point for design, captures early design decisions
  - Vehicle for stakeholder communication
  - Best known practices
  - Framework for software reuse