Unit-3 Software Design
(Lecture Notes)

Prepared by
Jay Nanavati, Assistant Professor, SEMCOM
Topics

• Software Design - Introduction
• Design Principles
• Module Level concepts
• Overview of Structured design
• Functional v/s Object-oriented approach *(Difference ONLY)*
• Design Specification
• Verification
Software Design - Introduction

• Design (as a verb)
  ✓ The **process** of creating a blueprint or a plan for the system.
  ✓ Goal – To produce a model of a system.

• Design (as a noun)
  ✓ A blueprint or a plan for the system.
  ✓ **Output** of the design *process*.
  The design activity begins after the SRS is prepared (& validated).
Software Design – Basic Concepts

• The design process has two levels
  1. Top-level
  2. Detailed design or Logic design

• Top-level Design
  - **Which** modules are needed for the system?
  - How these modules should be interconnected?
  - Controls testability, modifiability & efficiency.

• Detailed design
  - Represents the **internal** design of modules.
  - **How** the specifications of the modules can be satisfied?
  - Specifies logic & data structures.
Software Design – Basic Concepts

• Design *Methodology*
  - A set of techniques & guidelines for creating a design.

• Input /Output /Exit Criteria of Design Phase
  - Input: SRS
  - Output: System Design & Detailed Design
  - Exit Criteria: Design which is verified & approved for quality.

• Design can be function-oriented or object-oriented.
Design Principles

- Design Principles are guidelines that may be followed to create a good design of the system.
- A good system design should have following attributes:
  1. Problem Partitioning and Hierarchy
  2. Abstraction
  3. Modularity
  4. Top-down and Bottom-up Strategies
Design Principles (contd.)

• Problem Partitioning and Hierarchy
  ✓ A Divide-and-conquer principle.
  ✓ For software design, divide the problem into manageable small pieces that can be solved separately.
  ✓ The different pieces can not be completely independent of each other. (?)
  ✓ When to stop further partitioning?
  ✓ Problem partitioning leads to hierarchies in the design.
  ✓ Hierarchical structures are good in case of complex system.
Design Principles (contd.)

• Abstraction
  ✓ Abstraction means describing the external behavior of a component without specifying internal details.
  ✓ Describe what the component does without explaining how it does so.
  ✓ Abstraction is essential for partitioning. (why?)
  ✓ Abstraction is also useful for maintenance. (how?)
  ✓ Two types:
    (1) Functional abstraction
    (2) Data abstraction
Design Principles (contd.)

• Modularity
  ✓ A system is called modular if
    ➢ it consists of discreet components so that each component
      can be implemented separately and
    ➢ a change to one component has minimal impact on other
      components.
  ✓ Modularity helps in system debugging, repair &
    system building (i.e. assembly).
  ✓ For modularity, each component must support
    abstraction.
  ✓ Modularity = partitioning + abstraction.
Design Principles (contd.)

• Top-down and Bottom-up Strategies

Top-down Approach

✓ A top-down approach starts by identifying the major components of the system.

✓ The major components are then further decomposed into next, lower-level components.

✓ This decomposition repeated until desired level of detailing is achieved.

✓ Suitable where specifications are clearly known, e.g. waterfall model.
Design Principles (contd.)

• Top-down and Bottom-up Strategies

**Bottom-up Approach**

✓ A bottom-up approach starts with designing the most basic or primitive components.

✓ Then, lower-level components are combined together to form higher-level components.

✓ This step is repeated until the operations supported by that layer matches abstraction of that layer.

✓ Suitable for iterative enhancement.
Module-level Concepts

• In a system using functional abstraction, two criteria are: coupling and cohesion.

**Coupling**

- Coupling between module is the strength of interconnections between modules or a measure of interdependence among modules.
- Coupling is decided during system design. (why?)
- Highly coupled vs. loosely coupled modules.
- LOW COUPLING IS DESIRED.
- Complexity of interface and type of information flow between modules affect coupling.
Module-level Concepts

Cohesion

- Cohesion is a measure of how closely internal elements of a module are related to one another.
- HIGH COHESION IS DESIRED.
- Relationship between cohesion & coupling.
- Levels of cohesion
  1) Coincidental
  2) Logical
  3) Temporal
  4) Procedural
  5) Communicational
  6) Sequential
  7) Functional
Module-level Concepts

Cohesion (contd.)

- Coincidental Cohesion occurs when there is no meaningful relationship among the elements of a module.
- Logical Cohesion occurs when there is some logical relationship among the elements of a module and the elements perform same type of functionality.
- Temporal Cohesion occurs when the elements are related in time and are executed together.
- Procedural Cohesion occurs when elements of a module are basically procedural units like decision-making, loop or selection.
Cohesion (contd.)

- Communication Cohesion occurs when the elements of a module use same input or produce same output.
- Sequential Cohesion occurs when output of one element serves as input for some other element.
- Functional Cohesion occurs when elements of a module work collaboratively to accomplish common goal.

Finally, the goal should be to achieve **low coupling** & **high cohesion**.
# Functional vs. OO Approach

<table>
<thead>
<tr>
<th><strong>Functional Approach</strong></th>
<th><strong>Object-oriented Approach</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>1. A system is viewed as a transformation function transforming input to output.</td>
<td>A system is viewed as a set of objects providing some services.</td>
</tr>
<tr>
<td>2. The design consists of module definitions with each module supporting a functional abstraction</td>
<td>In object-oriented approach, the modules represent data abstraction</td>
</tr>
<tr>
<td>3. It is not easier to produce and understand design.</td>
<td>It is easier to produce and understand design.</td>
</tr>
<tr>
<td>4. Reuse of modules is not possible and hence development cost and time increases.</td>
<td>Reuse of objects is possible thereby reducing development cost &amp; time.</td>
</tr>
<tr>
<td>5. It does not allow building Object libraries.</td>
<td>It allows building Object libraries.</td>
</tr>
<tr>
<td>6. It is a top-down approach.</td>
<td>It is a bottom-up approach.</td>
</tr>
</tbody>
</table>
Design Specification

• Design specification is the process of documenting the design and attaching details to it.

• Why? (Designers are NOT going to use the design).

• Then, who will use them? (Of course, _______).

• What is the outcome? (*Design Document*)
Design Specification (contd.)

• What should a design document contain?
  ✓ Problem specification (Requirements specification)
  ✓ Major data structures (Data + Operations)
  ✓ Modules & their specifications (Major part)
  ✓ Design decisions (Explanations/Justifications, Why/Why not?)
Module Specifications

- The Detailed Design includes internal design and *detailed specification of each module*.
- Desirable properties of module specifications
  - ✓ complete
  - ✓ unambiguous
  - ✓ understandable
  - ✓ implementation-independent
  - ✓ operational specifications
Functional Module Specification

• A module is treated as a black box that takes some inputs and produces some outputs such that outputs have a specified relationship with the inputs.
• Input, Process, Output + Constraints
• Two methods for specifying functional modules:
  1) Hoare’s method
  2) A variation in Hoare’s method
Hoare’s method
- Based on pre-conditions and post-conditions.
- Pre-condition
- Post-condition
- Consider a module *sort*, to be written to sort a list $L$ of integers in ascending order.
- Pre-condition: non-null $L$
- Post-Condition: For all $i$, $1 \leq i < \text{size (} L\text{)}$
  \[ L[i] \leq L[i + 1] \]
A Variation in Hoare’s method

- The assertions for the output can be stated as a relation between the final state and the initial state.
- Pre-condition: non-null L
- Post-Condition: For all i,
  \[1 \leq i \leq \text{size (L')}\]
  \[L'(i) \leq L'(i+1]\]
  \[L' = \text{Permutation (L)}\]
Data Abstraction Specification

The Axiomatic Specification Technique

- **Axiom** - A saying that is widely accepted on its own merits.
- Axioms specify how different operations interact with one-another.
- The interactions completely describe the behavior of the operations.
- Axiomatic specifications precisely specify the datatype.
- Post-Condition: For all \( i \), \( 1 \leq i < \text{size} \ (L) \)
  \[
  L [i] \leq L [i + 1]
  \]
The Axiomatic Specification Technique (contd.)

- An Example: Writing specifications for a stack of integers.

- We define a stack that has four operations:
  
  Create: to create a new stack
  Push: to push an element on a stack
  Pop: to pop the top element from the stack
  Top: returns the element on the top of the stack.
The Axiomatic Specification Technique (contd.)

- The Axiomatic Specifications can be given as follows:

1. stack [integer]
   declare
2. create() → stack;
3. push(stack, integer) → stack;
4. pop(stack) → stack;
5. top(stack) → integer U undefined;
The Axiomatic Specification Technique (contd.)

var
6. s: stack; i : integer;
for all
7. top (create()) = undefined
8. top (push(s, i )) =i
9. pop (create()) =create();
10. pop( push (s, i))=s;
end
Data Abstraction Specification (contd.)

The Axiomatic Specification Technique (contd.)

- These specifications are specified using a specification language, which has two major components:
  1. Syntactic specifications
  2. Semantic specifications
- Axioms attach meaning to operations by specifying the relationship between operations. The following constructs are allowed for writing the axioms:
  Free variables
  If-then-else
  Recursion
  Boolean Expressions
PDL (Process Design Language)

- PDL is a systematic way to communicate the design precisely & completely.
- Two extremes:
  - Natural Language (Vocabulary)
  - Formal Language (Structured way)
- PDL falls in the centre.
PDL (contd.)

• An Example of a PDL of minmax
  minmax(input)
  ARRAY a
  DO UNTIL end of input
    READ an item into a
  ENDDO
  max, min := first item of a
  DO FOR each item in a
    IF max < item THEN set max to item
    IF min > item THEN set min to item
  ENDDO
END
• Logic is there but no detail of implementation.
• Then, how to implement PDL?
• PDL Constructs – similar to programming languages.

<table>
<thead>
<tr>
<th>IF condition THEN statements</th>
<th>DO condition statements</th>
<th>CASE variable OF value1: statements</th>
</tr>
</thead>
<tbody>
<tr>
<td>ELSE statements</td>
<td>ENDDO</td>
<td>...</td>
</tr>
<tr>
<td>ENDIF</td>
<td></td>
<td>valueN: statements</td>
</tr>
<tr>
<td></td>
<td></td>
<td>ENDCASE</td>
</tr>
</tbody>
</table>

• Anything not capitalized below can be replaced with natural language text or other PDL constructs
Logic/Algorithm Design

- The basic goal of detailed design
- Logic requires algorithm
- Steps to perform for developing an algorithm:
  1. Statement of problem
  2. Model development
  3. Design of algorithm
- Correctness of algorithm must be verified.
Design Verification

• Design verification ensures that the detailed design meets the specifications laid down in the system design.

• 3 verification methods:
  1. Design Walkthroughs
  2. Critical Design Review
  3. Consistency Checkers