Learning Resource

On

Software Engineering

Chapter-5

Software Design

Prepared By:

Kunal Anand, Asst. Professor

SCE, KIIT, DU, Bhubaneswar-24

Chapter Outcomes:

After completing this chapter successfully, the students will be able to:

- Distinguish between Analysis and Design
- Define the goal of Software Design.
- Explain different activities involved in design.
- List the properties of good software design.
- Discuss the significance of understandability and modularity of a software design.
- Define functional independence.
- Explain cohesion and coupling.
- List the types of cohesion and coupling.
- Explain the concept of module structure.
- Identify different design approaches.

Organization of the Chapter

- Introduction to Software Design
- Goodness of a Design
- Functional Independence
- Cohesion and Coupling
- Design Approaches
  - Function-oriented design
  - Object-oriented design
- Sample example on Software Design

Difference between Analysis and Design

- **Aim of Analysis**
  – To understand the problem with a view to **eliminate any deficiencies** such as incompleteness, inconsistencies, ambiguities etc, in the requirements.

- **Aim of Design**
  – To produce a model that will provide a **seamless transition** to the coding phase, i.e., once the requirements are analyzed and found to be satisfactory, **a design model is created which can be easily implemented.**

Software Design

- **Software Design** deals with the transformation of the client's requirements, as described in the approved SRS document, into a model that is suitable for implementation in a programming language.

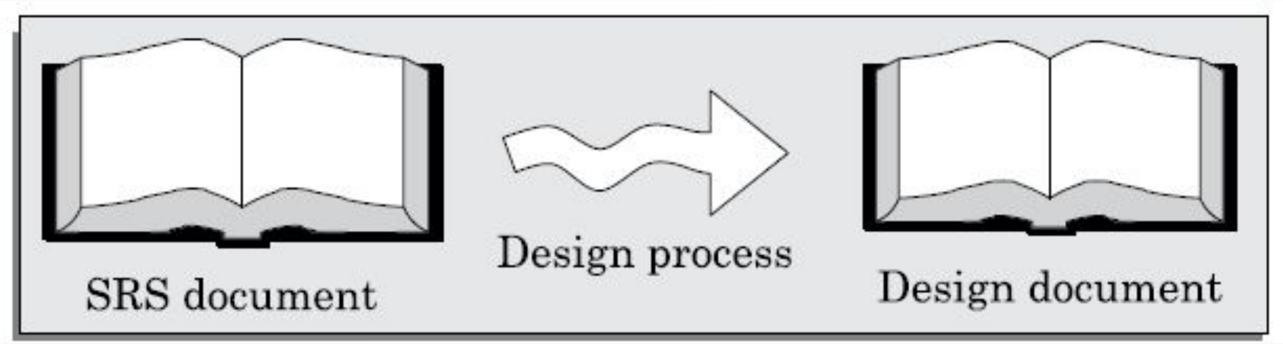


Fig. 5.1: The design process

- A good software design is seldom arrived by using a single step procedure but rather through several iterations in a series of steps.
- Design activities can be broadly classified into two important parts: **Preliminary design** (High-level design) and **Detailed design** (Low Level Design)

High Level and Detailed Design

- **High-level Design** identifies different **modules**, the **control relationships** among them, and the definition of the **interfaces** among these modules.
  - The outcome of high-level design is called the **"Software Architecture".** Ex: Tree-like structure, Jackson diagram.

- In **Detailed Design**, the **data structure** and the **algorithms** of the each module is designed.
  - The outcome of the detailed design stage is usually known as the **"Module-Specification (MSPEC)"** document.

Items designed during design phase

- Module Structure

- Control relationship among the modules
  - call relationship or invocation relationship

- Interface among different modules,
  - data items exchanged among different modules,

- Data structures of individual modules,

- Algorithms for individual modules.

Module Structure

- **Module Structure** of a software product is a graphical representation of the overall system i.e., how the complete system will look like.

- A **module** consists of several functions and associated data structures.

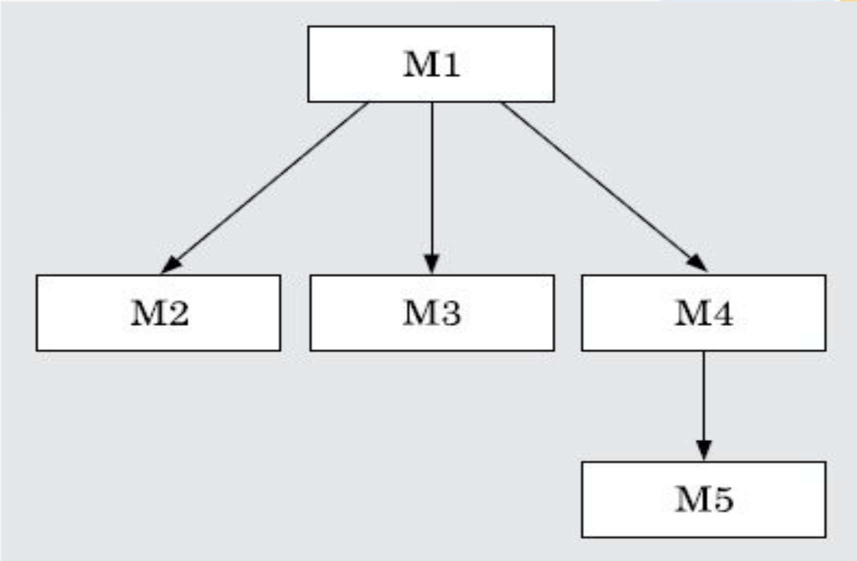


Fig. 5.2: Module structure

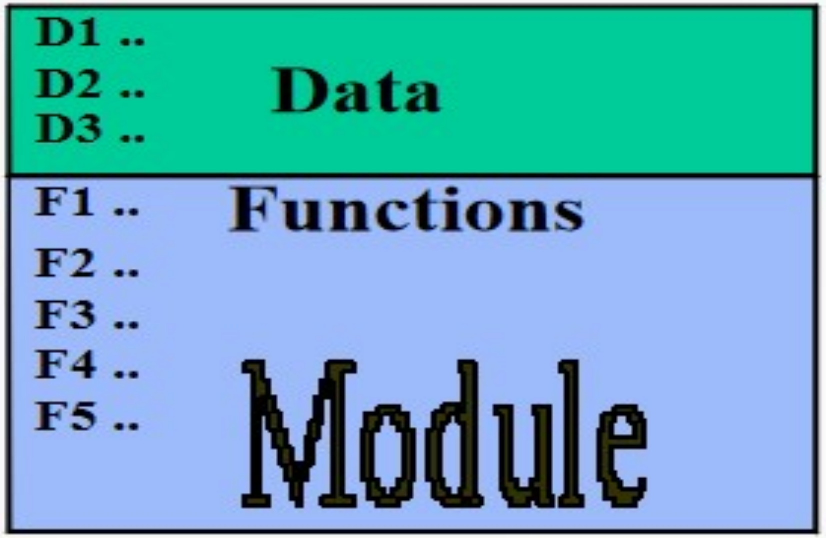


Fig. 5.3: Module details

What is Good Software Design?

- Should implement **all functionalities** of the system **correctly**.
- Should be **easily understandable**.
  - A design that is easy to understand is also easy to maintain and change.
- Should be **efficient**.
- Should be **amenable to change** i.e., easily maintainable.

Understandability and Modularity

- **Understandability**
  - Use **consistent and meaningful names** for various design components.
  - Design solution should consist of a **cleanly decomposed** set of modules (modularity),
  - Different modules should be **neatly arranged** in hierarchy in a neat tree-like diagram.
- **Modularity**
  - Modularity is a fundamental attribute of any good design.
  - Decomposition of a problem **cleanly into modules**
  - Modules are **almost independent** of each other
  - **Divide and conquer** principle.

Modularity

- If modules are independent, then they can be understood separately which reduces the complexity significantly.
- The following diagram represents a clean decomposition and non-clean decomposition.

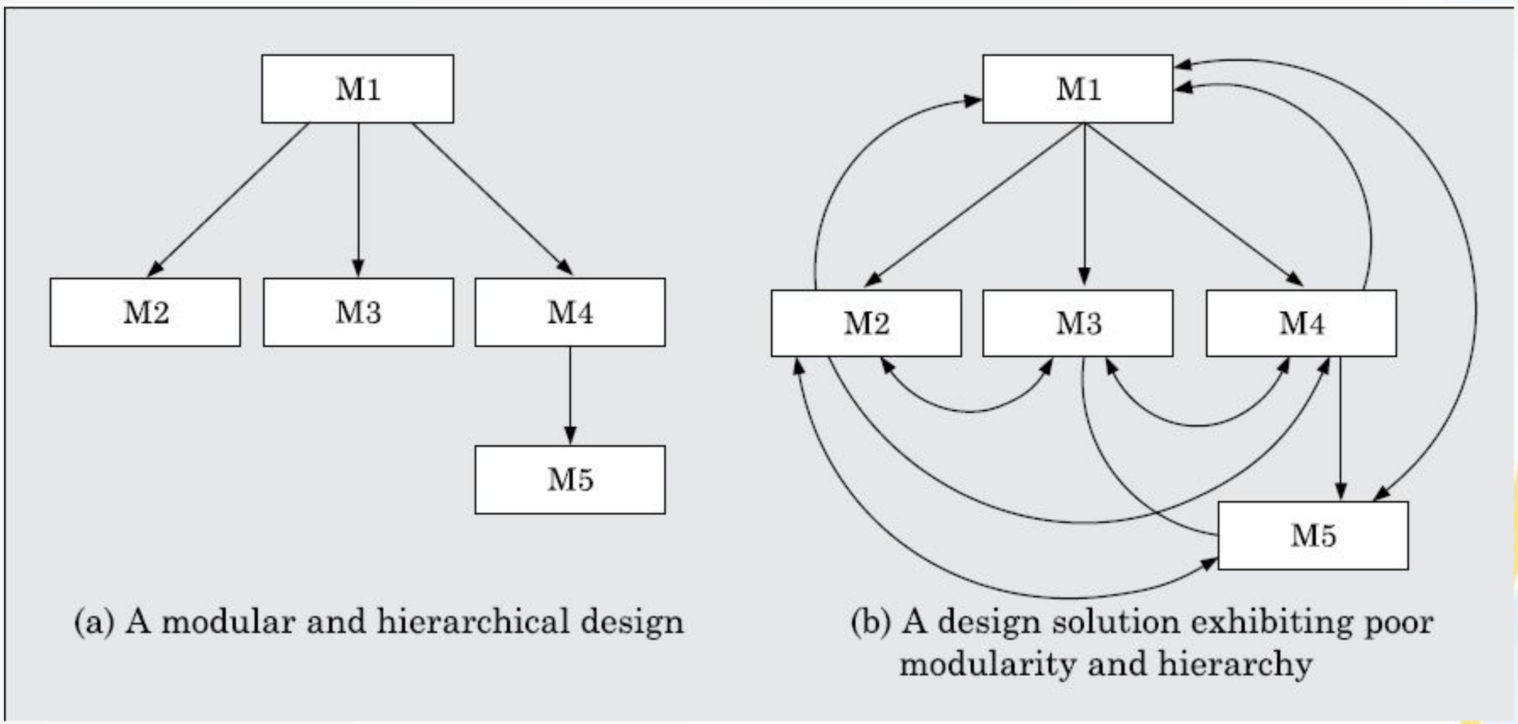


Fig. 5.4: Two design solution for same problem

Functional Independence

- Functional Independence of a module is the measure of its ability to perform its intended task without depending on any other module.
  - It reduces error propagation as the degree of interaction between modules is low.
  - Again, any error in one module does not directly affect other modules.
  - It facilitates the reuse of modules.
  - It provides better understandability and good design as the complexity of design is reduced.
  - Different modules easily understood in isolation.
- The interface of a functionally independent module with other modules is simple and minimal.

Cohesion and Coupling

- In technical terms, modules should display **"High Cohesion"** and **"Low Coupling".**
- A module having **high cohesion** and **low coupling** is functionally independent of other modules.
  - A **functionally independent module** has minimal interaction with other modules.
- **Cohesion** is a measure of **functional strength** of a module.
  - A cohesive module performs a single task or function.
- **Coupling** between two modules is a measure of the **degree of interdependence or interaction** between the two modules.

Type of Cohesion

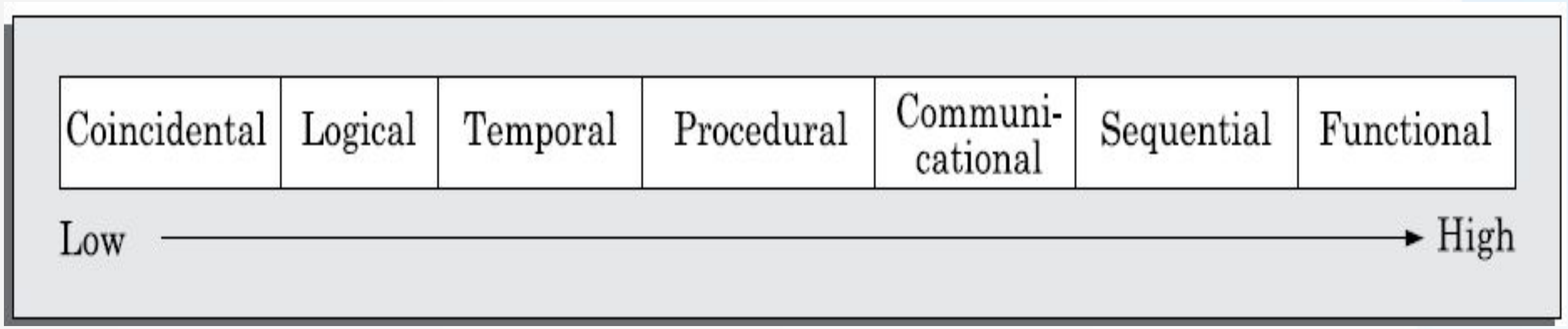


Fig. 5.5: Classification of Cohesion

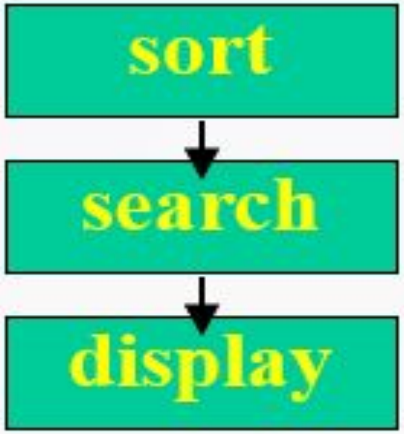
- **Coincidental:** In this cohesion, the module performs a set of tasks which relate to each other **very loosely**, if at all. The functions have been put in the module out of **pure coincidence** without any thought or design.
  - **For example**, in a Transaction Processing System (TPS), the **get-input**, **print-error**, and **summarize-members** functions are grouped into one module.

Types of Cohesion

- **Logical:** All elements of the module perform **similar operations**. **Example:** A set of print functions to generate an output report arranged into a single module.
- **Temporal:** The module contains tasks that must be executed in the **same time span**. **Example:** The set of functions responsible for initialization, start-up, shut-down of some process, etc.
- **Procedural:** The set of functions of the module are **part of a procedure** (algorithm). **Example:** An algorithm for decoding a message.
- **Communicational:** All functions of the module refers to or update the **same data structure**. **Example:** The set of functions defined on an array or a stack.

Types of Cohesion (contd..)

- **Sequential Cohesion:** Elements of a module form different parts of a **sequence i.e.,** output from one element of the sequence is input to the next.
  - **Ex:**



- **Functional Cohesion:** Different elements of a module cooperate to achieve a **single function**,
    - **e.g.,** managing an employee's pay-roll.
  - When a module displays functional cohesion, we can describe the function using a single sentence.
  –

Coupling

- **Coupling** indicates:
  - how closely two modules interact or how interdependent they are.
  - The degree of coupling between two modules depends on their interface complexity.

- There are no ways to precisely determine coupling between two modules.
- Classification of different types of coupling will help us to approximately estimate the degree of coupling between two modules.
  - Five types of coupling may exist between two modules.

Types of coupling

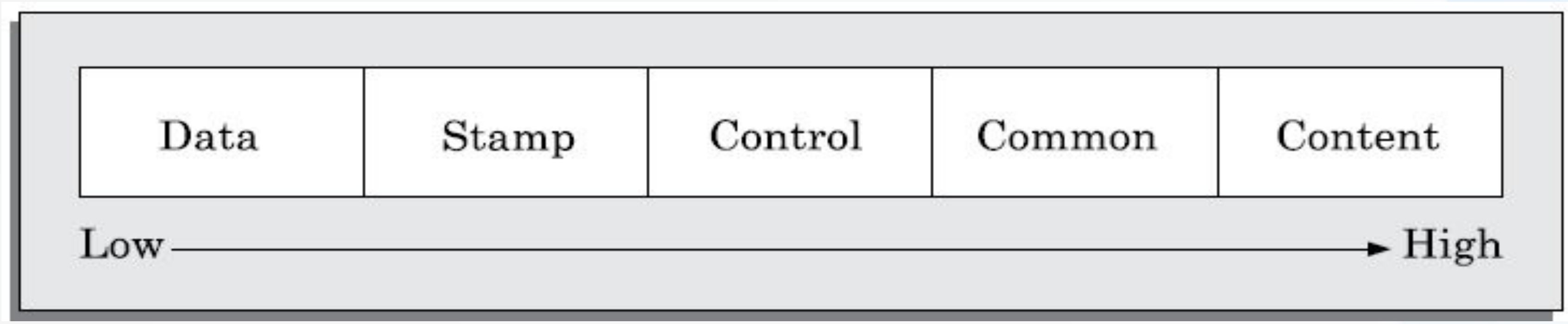


Fig. 5.6: Classification of Coupling

- **Data Coupling:** Two modules are data coupled, if they communicate via a **parameter**. The data item should be problem related and not used for control purpose.

  e.g., an elementary data item (an integer, a float, a character, etc.)

Types of Coupling (contd..)

- **Stamp Coupling:** Two modules are stamp coupled, if they communicate via a **composite data item** such as a record in PASCAL or a structure in C.

- **Control Coupling:** Data from one module is used to **direct order of instruction** execution in another.
  - **Ex:** a flag set in one module and tested in another module.

- **Common Coupling:** Two modules are common coupled, if they share some global data.

- **Content Coupling:** Content coupling exists between two modules **if they share code,**
    - e.g., branching from one module into another module.

Cohesion vs Coupling

Cohesion	Coupling
Cohesion is the indication of the relationship **within** module i.e., **Intra-module** concept.	**Coupling** is the indication of the relationships **between** modules i.e., **Inter-module** concept.
Cohesion shows the module's **relative functional strength**.	Coupling shows the **relative independence among the modules**.
While designing you should strive for **high cohesion** i.e. a cohesive component/ module focus on a single task.	While designing you should strive for **low coupling** i.e., Dependency between modules should be less.

Cohesion and Coupling

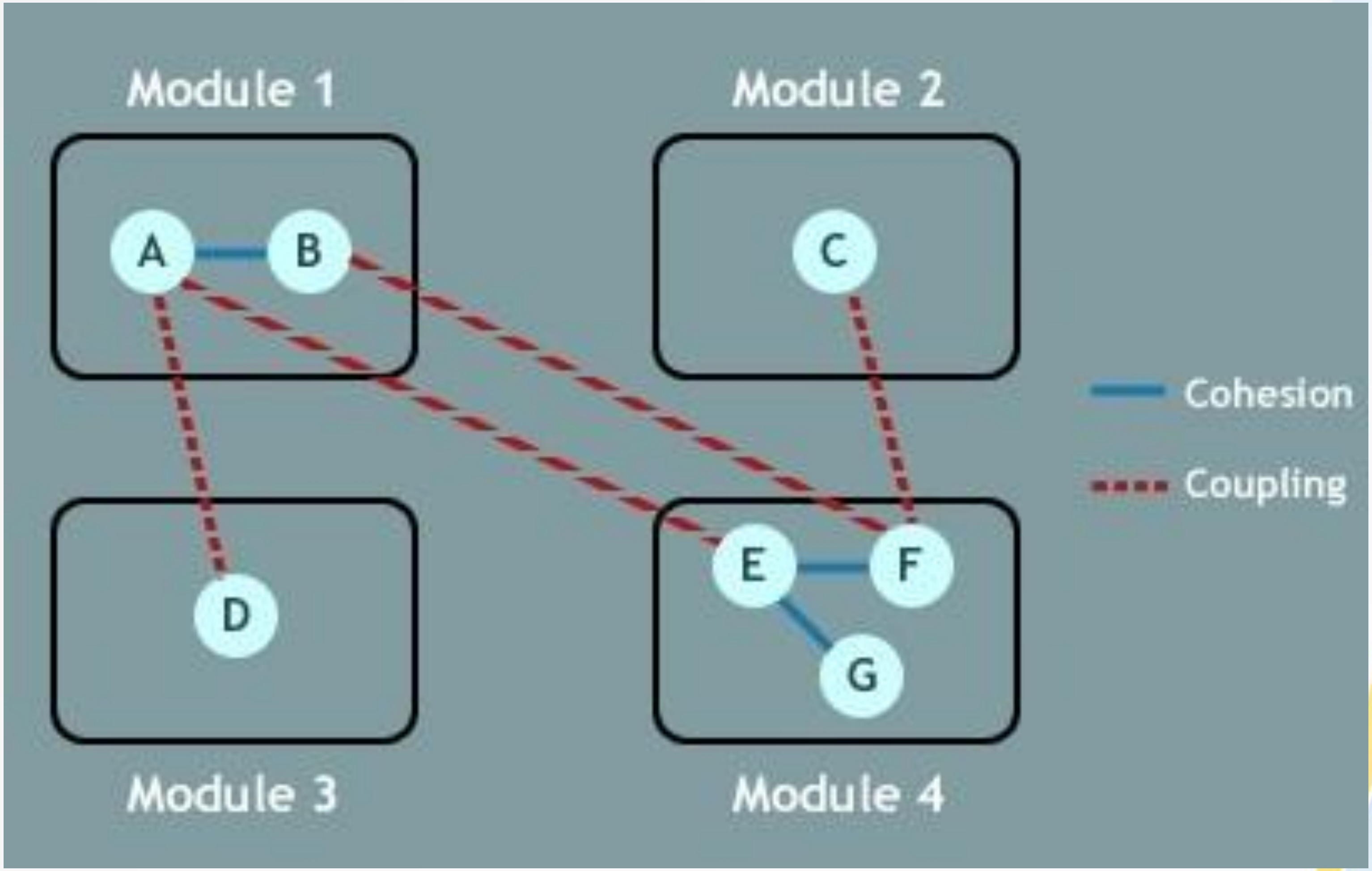


Fig. 5.7: Cohesion and Coupling: A larger picture

Layered Arrangements of Modules

- **Control hierarchy** represents organization of modules. It is also referred to as **Module Structure.** Most common notation is a tree-like diagram called **Structure Chart.**
- **Neat arrangement** essentially means **"Low fan-out"** and **"High fan-in".**
- **Characteristics of Module Structure**
  - **Depth:** number of levels of control.
  - **Width:** overall span of control i.e., the maximum number of modules among all the levels of the module structure.
  - **Fan-out:** a module **directly controls how many module** is measured through fan-out.
  - **Fan-in:** a module is **directly controlled by how many modules** can be measured by fan-in.

Points to be noted:

- **High fan-in** represents **code reuse** and is in general encouraged, whereas **Low fan-out** represents the functional strength of a module.

- A module that **controls another module** said to be **superordinate** to it. Conversely, a module **controlled by another module** said to be **subordinate** to it.

- A module A is said to be **visible** by another module B, if A **directly or indirectly** calls B.

Abstract

- Lower-level modules do input/output and other low-level functions.
- Upper-level modules do more managerial functions.
- The principle of abstraction requires:
  - lower-level modules **do not invoke** functions of higher-level modules.
  - Also known as **Layered Design**.
  - The **layering principle** requires modules at a layer can call only the modules immediately below it (Sequence).

contd..

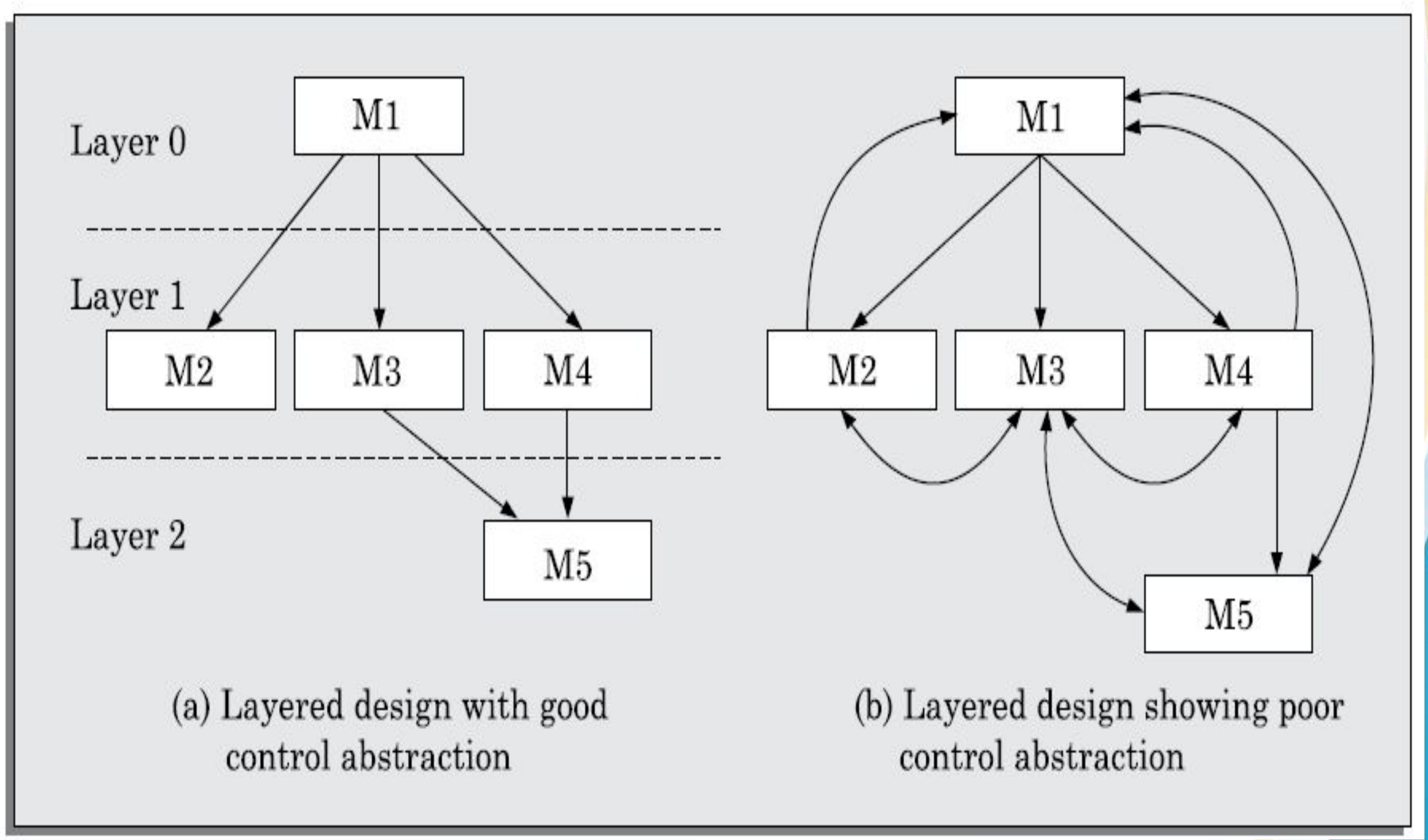


Fig. 5.8: Good and Poor control abstraction

Software Design Approaches

- Two fundamentally different software design approaches:
  - **Function-oriented design**
  - **Object-oriented design**

- These two design approaches are radically **different**.
  - However, they are **complementary** rather than competing techniques.
  - Each technique is applicable at different stages of the design process.

Function-Oriented Design

- A system is looked upon as something that performs a **set of functions.**
- Starting at this high-level view of the system:
  - each function is successively refined into more detailed functions.
  - Functions are mapped to a module structure.
- **Example:** In **Library Management System** software, the function **create-new-library- member**:
  - creates the record for a new member, (**Sub-function-1**)
  - assigns a unique membership number (**Sub-function-2**)
  - prints a bill towards the membership (**Sub-function-3**)

-

Function-Oriented Design

- Several function-oriented design approaches have been developed:
  - Structured design (Constantine and Yourdon, 1979)
  - Jackson's structured design (Jackson, 1975)
  - Warnier-Orr methodology
  - Wirth's step-wise refinement
  - Hatley and Pirbhai's Methodology

Object-Oriented Design

- System is viewed as a collection of **objects** (i.e., entities).
- System state is **decentralized** among the object and each object manages its own state information.
- Objects have their own internal data that defines their state.
- Similar objects constitute a class and each object is a member of some class.
- Classes may inherit features from a super class.
- Conceptually, objects communicate by message passing.
- **Ex: Library Automation Software**
  - each **library member** is a separate object with its **own data and functions.**
  - Functions defined for one object cannot directly refer to or change data of other objects.

Object-Oriented vs Function-Oriented Design

- Unlike FOD, the OOD uses **real-world entities** such as "employee", "picture", "machine", "radar system", etc. rather than the **functions** such as "sort", "display", "track", etc.,
- In OOD, the software is not developed by designing functions such as update-employee-record, get-employee-address, etc. but by designing objects such as: employees, departments, etc.
- In OOD:
  - state information is not shared in a centralized data.
  - but is distributed among the objects of the system.
- **Grady Booch** sums up this fundamental difference saying:
  - "Identify **verbs** if you are after **procedural** design and **nouns** if you are after **object-oriented** design."

Object-Oriented vs Function-Oriented Design

- Function-oriented techniques group functions together **if as a group, they constitute a higher-level function**.

- On the other hand, object-oriented techniques group functions together based **on the data they operate on**.

- To illustrate the differences between object-oriented and function-oriented design approaches,
  - let us consider an example:
    - An automated Fire Alarm System (FAS) for a large building.

Fire-Alarm System

- We need to develop a **computerized Fire Alarm System (CFAS)** for a large multi-storied building which has **80 floors and 1000 rooms** in the building.
- Different rooms of the building are fitted with **smoke detectors** and **fire alarms**. The CFAS would monitor the status of the smoke detectors.
- Whenever a fire condition is reported by any smoke detector:
  - the fire alarm system should:
    - **determine the location** from which the fire condition was reported
    - **sound the alarms** in the neighboring locations.
    - The fire alarm system should **flash an alarm message** on the computer console that is monitored by a fire fighting personnel round the clock.
  - After a fire condition has been successfully handled, the fire alarm system should let fire fighting personnel **reset the alarms**.

Function-Oriented Approach

- /* Global data (system state) accessible by various functions */
  BOOL detector_status[1000];
  int detector_locs[1000];
  BOOL alarm-status[1000]; /* alarm activated when status set */
  int alarm_locs[1000]; /* room number where alarm is located */
  int neighbor-alarms[1000][10];/*each detector has at most*/
  /* 10 neighboring alarm locations */

- The functions which operate on the system state:
  interrogate_detectors();
  get_detector_location();
  determine_neighbor();
  ring_alarm();
  report_fire_location();
  reset_alarm();

Object-Oriented Approach

- **class ‘DETECTOR’**
  - **attributes:** status, location, neighbors
  - **operations:** create, sense-status, get-location, find-neighbors

- **class ‘ALARM’**
  - **attributes:** location, status
  - **operations:** create, ring-alarm, get_location, reset-alarm

- In the object oriented program, appropriate number of instances of the class **DETECTOR and ALARM** should be created.