Software Design Strategies

Software Design Strategies

Software design is a process to conceptualize the software requirements into software implementation. Software design takes the user requirements as challenges and tries to find optimum solution. While the software is being conceptualized, a plan is chalked out to find the best possible design for implementing the Intended solution.

There are multiple variants of software design.

Structured Design

- Structured design is a conceptualization of problem into several well-organized elements of solution. It is basically concerned with the solution design. Benefit of structured design is, it gives better understanding of how the problem is being solved. Structured design also makes it simpler for designer to concentrate on the problem more accurately.
- Structured design is mostly based on 'divide and conquer' strategy where a problem is broken into several small problems and each small problem is individually solved until the whole problem is solved.
- The small pieces of problem are solved by means of solution modules. Structured design emphasis that these modules be well organized in order to achieve precise solution.

Structured Design

These modules are arranged in hierarchy. They communicate with each other. A good structured design always follows some rules for communication among multiple modules, namely -

- **Cohesion** - grouping of all functionally related elements.
- **Coupling** - communication between different modules.

A good structured design has ***high*** cohesion and ***low*** coupling arrangements.

Function Oriented Design

- In function-oriented design, the system is comprised of many smaller sub-systems known as functions. These functions are capable of performing significant task in the system. The system is considered as top view of all functions.
- Function oriented design inherits some properties of structured design where divide and conquer methodology is used.
- This design mechanism divides the whole system into smaller functions, which provides means of abstraction by concealing the information and their operation. These functional modules can share information among themselves by means of information passing and using information available globally.
- Another characteristic of functions is that when a program calls a function, the function changes the state of the program, which sometimes is not acceptable by other modules. Function oriented design works well where the system state does not matter and program/functions work on input rather than on a state.

Function Oriented Design

Design Process

- The entire system is logically broken down into smaller units known as functions on the basis of their operation in the system.
- The whole system is seen as how data flows in the system by means of data flow diagram.
- DFD depicts how functions change the data and state of entire system.
- Each function is then described at large.

Object Oriented Design

Object oriented design works around the entities and their characteristics instead of functions involved in the software system. This design strategy focuses on entities and its characteristics. The whole concept of software solution revolves around the engaged entities.

The important concepts of Object Oriented Design:

- **Objects -** All entities involved in the solution design are known as objects. For example, person, banks, company and customers are treated as objects. Every entity has some attributes associated to it and has some methods to perform on the attributes.
- **Classes -** A class is a generalized description of an object. An object is an instance of a class. Class defines all the attributes, which an object can have and methods, which defines the functionality of the object.

Object Oriented Design

In the solution design, attributes are stored as variables and functionalities are defined by means of methods or procedures. Some features of Object Oriented Designs are-

- **Encapsulation** - In OOD, the attributes (data variables) and methods (operation on the data) are bundled together is called encapsulation. Encapsulation not only bundles important information of an object together, but also restricts access of the data and methods from the outside world. This is called information hiding.
- **Inheritance** - OOD allows similar classes to stack up in hierarchical manner where the lower or sub-classes can import, implement and re-use allowed variables and methods from their immediate super classes. This property of OOD is known as inheritance. This makes it easier to define specific class and to create generalized classes from specific ones.
- **Polymorphism** - OOD languages provide a mechanism where methods performing similar tasks but vary in arguments, can be assigned same name. This is called polymorphism, which allows a single interface performing tasks for different types. Depending upon how the function is invoked, respective portion of the code gets executed.

Object Oriented Design

Design Process

Software design process can be perceived as series of well-defined steps. Though it varies according to design approach (function oriented or object oriented), yet It may have the following steps involved:

- A solution design is created from requirement or previous used system and/or system sequence diagram.
- Objects are identified and grouped into classes on behalf of similarity in attribute characteristics.
- Class hierarchy and relation among them are defined.
- Application framework is defined.

Software Design Approaches

A **system** is composed of more than one sub-systems and it contains a number of components. Further, these sub-systems and components may have their one set of sub-system and components and creates hierarchical structure in the system.

There are two generic approaches for software designing:

Top down Design

- Top-down design takes the whole software system as one entity and then decomposes it to achieve more than one sub-system or component based on some characteristics. Each subsystem or component is then treated as a system and decomposed further. This process keeps on running until the lowest level of system in the top-down hierarchy is achieved.
- Top-down design starts with a generalized model of system and keeps on defining the more specific part of it. When all components are composed the whole system comes into existence.
- Top-down design is more suitable when the software solution needs to be designed from scratch and specific details are unknown.

Software Design Approaches

Bottom Up Design

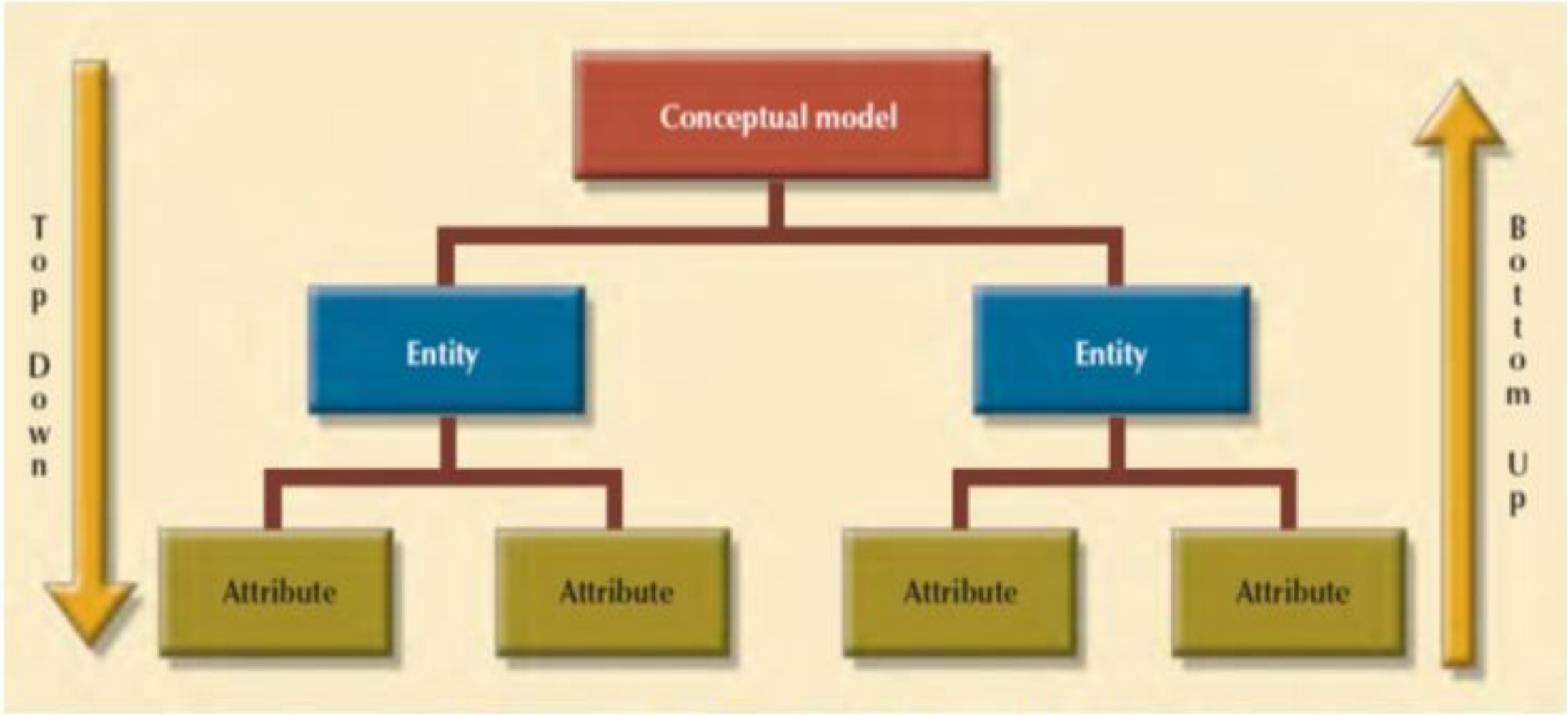
- The bottom up design model starts with most specific and basic components. It proceeds with composing higher level of components by using basic or lower level components. It keeps creating higher level components until the desired system is not evolved as one single component. With each higher level, the amount of abstraction is increased.
- Bottom-up strategy is more suitable when a system needs to be created from some existing system, where the basic primitives can be used in the newer system.
- Both, top-down and bottom-up approaches are not practical individually. Instead, a good combination of both is used.

Software Design Approaches

Bottom Up Design

- The bottom up design model starts with most specific and basic components. It proceeds with composing higher level of components by using basic or lower level components. It keeps creating higher level components until the desired system is not evolved as one single component. With each higher level, the amount of abstraction is increased.
- Bottom-up strategy is more suitable when a system needs to be created from some existing system, where the basic primitives can be used in the newer system.
- Both, top-down and bottom-up approaches are not practical individually. Instead, a good combination of both is used.

Top down Design VS. Bottom Up Design



Top-down vs. bottom-up design sequencing

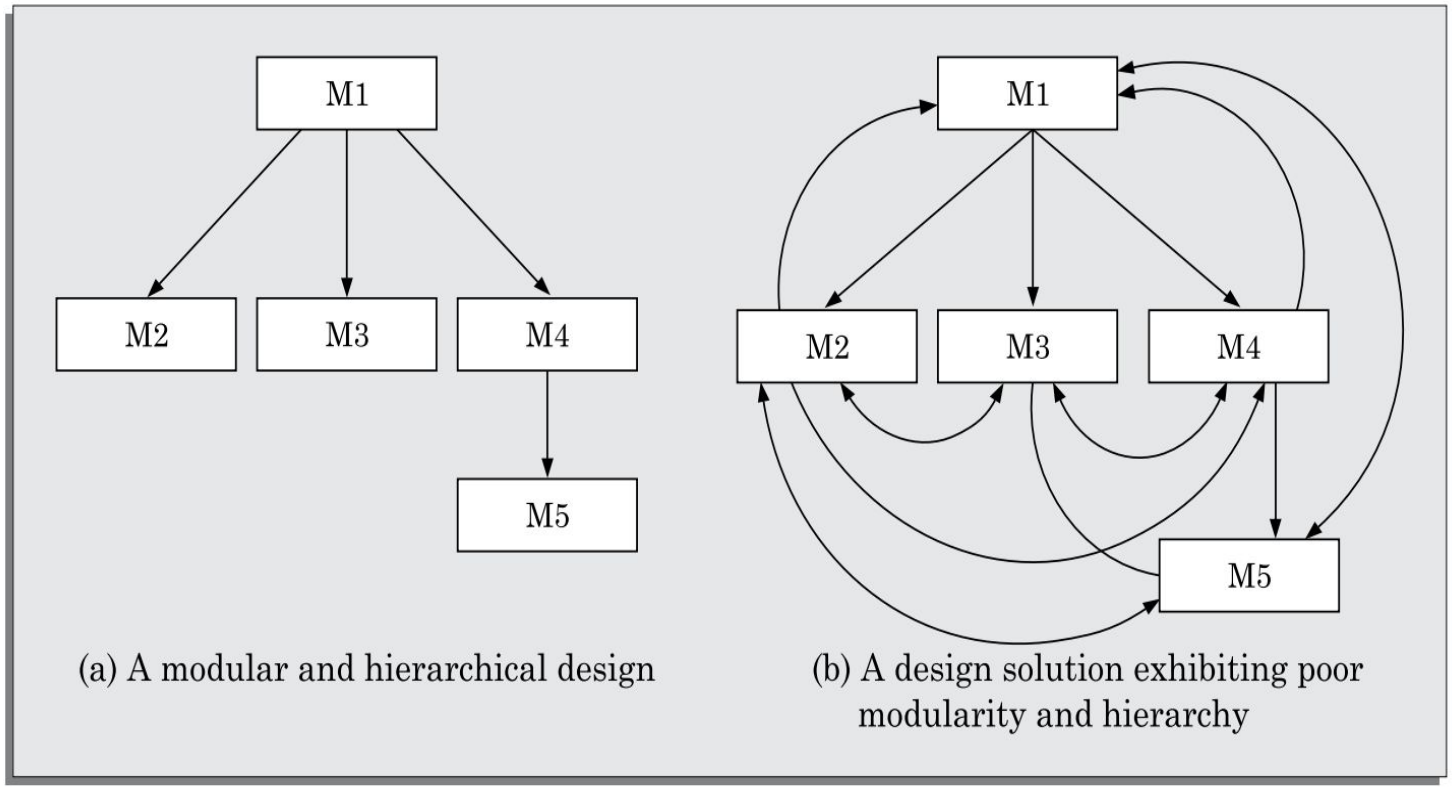
Analysis *vs.* Design

- Analysis and design activities differ in goal and scope.
- The analysis results are generic and does not consider implementation or the issues associated with specific platforms.
- The analysis model is usually documented using some graphical formalism.
- In case of the function-oriented approach, the analysis model would be documented using *data flow diagrams* (DFDs), whereas the design would be documented using structure chart.
- For object-oriented approach, both the design model and the analysis model will be documented using *unified modelling language* (UML).
- The analysis model would normally be very difficult to implement using a programming language.
- The design model is obtained from the analysis model through transformations over a series of steps.

Comparison of Modularity

- How can we compare the modularity of two alternate design solutions?
- From an inspection of the module structure, it is at least possible to intuitively form an idea as to which design is more modular.
- For example, consider two alternate design solutions to a problem that are represented in Figure given next, in which the modules *M1*, *M2*, etc. have been drawn as rectangles. The invocation of a module by another module has been shown as an arrow.
- It can easily be seen that the design solution of the Figure-(a) would be easier to understand since the interactions among the different modules is low. But, can we quantitatively measure the modularity of a design solution? Unless we are able to quantitatively measure the modularity of a design solution, it will be hard to say which design solution is more modular than another.
- Unfortunately, there are no quantitative metrics available yet to directly measure the modularity of a design. However, we can quantitatively characterize the modularity of a design solution based on the cohesion and coupling existing in the design.

Comparison of Modularity



(a) A modular and hierarchical design

(b) A design solution exhibiting poor modularity and hierarchy

A software design with high cohesion and low coupling among modules is the effective problem decomposition. Such a design would lead to increased productivity during program development by bringing down the perceived problem complexity.

Layered design

- A layered design is one in which when the call relations among different modules are represented graphically, it would result in a tree-like diagram with clear layering.
- In a layered design solution, the modules are arranged in a hierarchy of layers. A module can only invoke functions of the modules in the layer immediately below it.
- The higher layer modules can be considered to be similar to managers that invoke (order) the lower layer modules to get certain tasks done.
- A layered design can be considered to be implementing *control abstraction*, since a module at a lower layer is unaware of (about how to call) the higher layer modules.
- When a failure is detected while executing a module, it is obvious that the modules below it can possibly be the source of the error. This greatly simplifies debugging since one would need to concentrate only on a few modules to detect the error.

LAYERED ARRANGEMENT OF MODULES

- The *control hierarchy* represents the organisation of program components in terms of their call relationships.
- The control hierarchy of a design is determined by the order in which different modules call each other.
- Many different types of notations have been used to represent the control hierarchy. The most common notation is a treelike diagram known as a *structure chart.*
- In a layered design solution, the modules are arranged into several layers based on their call relationships.
- A module is allowed to call only the modules that are at a lower layer. That is, a module should not call a module that is either at a higher layer or even in the same layer.

LAYERED ARRANGEMENT OF MODULES

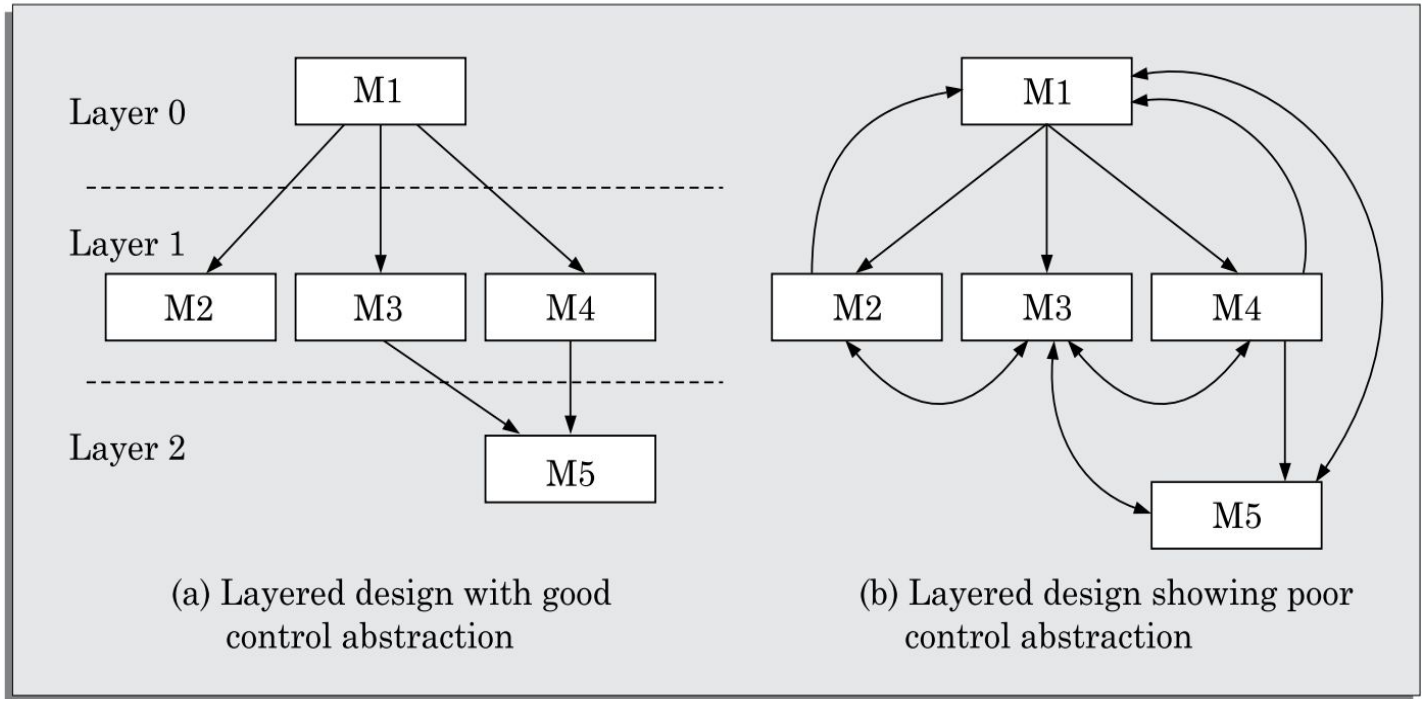


Figure (a) shows a layered design, whereas Figure (b) shows a design that is not layered.
The design solution shown in Figure (b), is actually not layered since all the modules can be considered to be in the same layer.

LAYERED ARRANGEMENT OF MODULES

- In a layered design, the top-most module in the hierarchy can be considered as a manager that only invokes the services of the lower level module to discharge its responsibility.
- The modules at the intermediate layers offer services to their higher layer by invoking the services of the lower layer modules and also by doing some work themselves to a limited extent.
- The modules at the lowest layer are the worker modules. These do not invoke services of any module and entirely carry out their responsibilities by themselves.

LAYERED ARRANGEMENT OF MODULES

Terminologies associated with a layered design:

- **Superordinate and subordinate modules:** In a control hierarchy, a module that controls another module is said to be *superordinate* to it. Conversely, a module controlled by another module is said to be *subordinate* to the controller.
- **Visibility:** A module B is said to be visible to another module A, if A directly calls B. Thus, only the immediately lower layer modules are said to be visible to a module.
- **Control abstraction:** In a layered design, a module should only invoke the functions of the modules that are in the layer immediately below it. In other words, the modules at the higher layers, should not be visible (that is, abstracted out) to the modules at the lower layers. This is referred to as *control abstraction.*
- **Depth and width:** Depth and width of a control hierarchy provide an indication of the number of layers and the overall span of control respectively. For the design of Figure 5.6(a), the depth is 3 and width is also 3.

LAYERED ARRANGEMENT OF MODULES

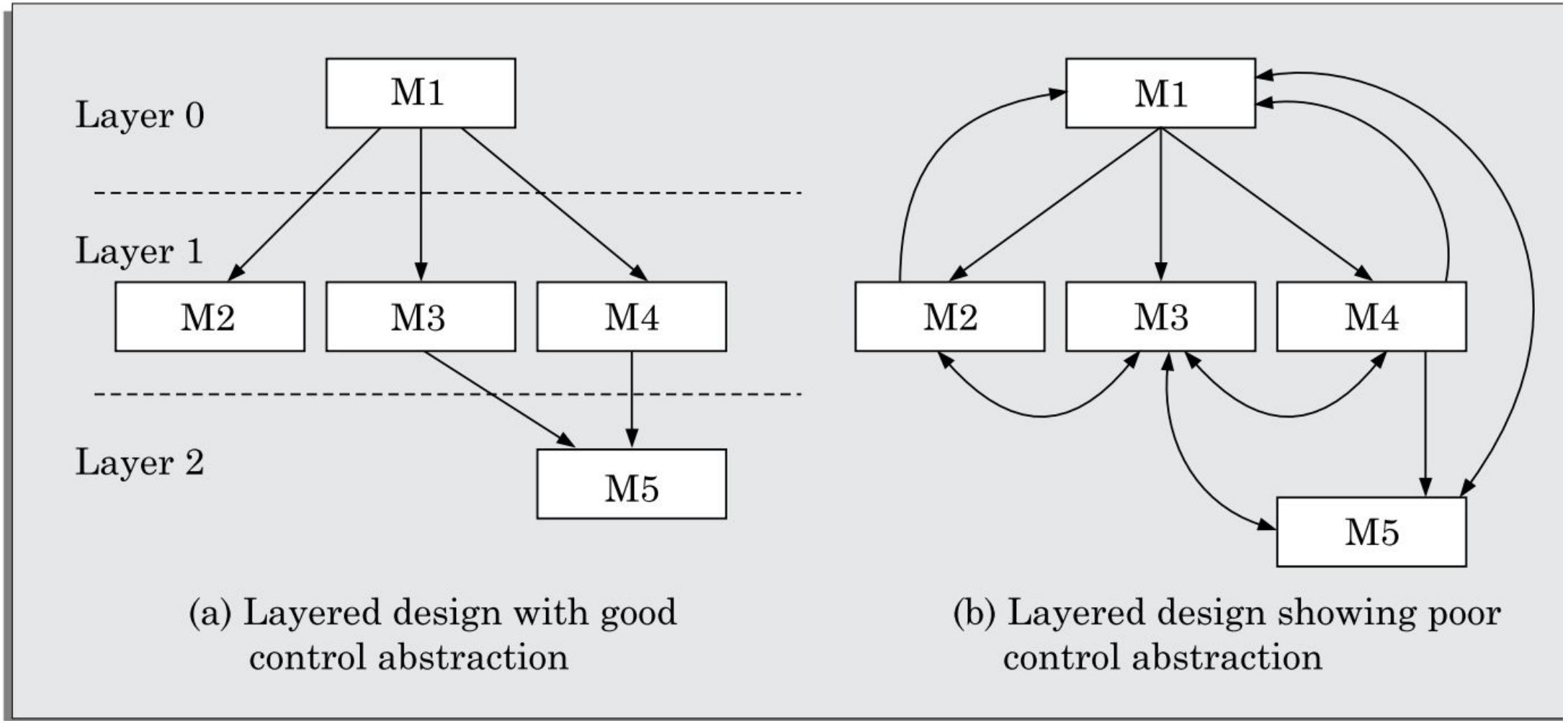
Terminologies associated with a layered design:

- **Fan-out:** Fan-out is a measure of the number of modules that are directly controlled by a given module. In Figure (a), the fan-out of the module M1 is 3. A design in which the modules have very high fan-out numbers is not a good design. The reason for this is that a very high fan-out is an indication that the module lacks cohesion. A module having a large fan-out (greater than 7) is likely to implement several different functions and not just a single cohesive function.

- **Fan-in:** Fan-in indicates the number of modules that directly invoke a given module. High fan-in represents code reuse and is in general, desirable in a good design. In Figure (a), the fan-in of the module M1 is 0, that of M2 is 1, and that of M5 is 2.

LAYERED ARRANGEMENT OF MODULES

Terminologies associated with a layered design:



(a) Layered design with good control abstraction

(b) Layered design showing poor control abstraction

Functional Independence

By the terms Functional independence, we mean that a module performs a single task and needs very little interaction with other modules.

Functional independence is a key to any good design primarily due to the following advantages it offers:

Error isolation: Whenever an error exists in a module, functional independence reduces the chances of the error propagating to the other modules. The reason behind this is that if a module is functionally independent, its interaction with other modules is low. Therefore, an error existing in the module is very unlikely to affect the functioning of other modules.

Further, once a failure is detected, error isolation makes it very easy to locate the error. On the other hand, when a module is not functionally independent, once a failure is detected in a functionality provided by the module, the error can be potentially in any of the large number of modules and propagated to the functioning of the module.

Scope of reuse: Reuse of a module for the development of other applications becomes easier. The reasons for this is as follows. A functionally independent module performs some well-defined and precise task and the interfaces of the module with other modules are very few and simple. A functionally independent module can therefore be easily taken out and reused in a different program. On the other hand, if a module interacts with several other modules or the functions of a module perform very different tasks, then it would be difficult to reuse it.