# DESIGN CONCEPTS

A set of fundamental software design concepts has evolved over the history of software engineering, here is a brief overview of important software design concepts:

## 1- **Abstraction** (hide unimportant data)

- abstraction is the simplification of a problem by focusing on only one aspect of the problem while omitting all other aspects.
- Take an example - printing a document from your computer. You just open the document, click on the "print" command, and in a short while, the printed document is ready. You are not really bothered about how the computer stores the document, nor about how it is transferred to the printer.
- When we consider a modular solution to any problem, many levels of abstraction can be posed.
  - At the highest level of abstraction, a solution is stated in broad terms using the language of the problem environment.
  - At lower levels of abstraction, a more procedural orientation is taken and the solution is stated in a manner that can be directly implemented.
- Use of abstraction <u>permits one to work with concepts and terms that are familiar in the problem environment without having to transform them into an unfamiliar structure</u>.
- Each step in the software process is a refinement in the level of abstraction of the software solution. During system engineering, the software is allocated as an element of a computer-based system. During software requirements analysis, the software solution is stated in terms "that are familiar in the problem environment." As we move through the design

process, the level of abstraction is reduced. Finally, the lowest level of abstraction is reached when source code is generated.

- As we move through different levels of abstraction, we work to create ***procedural abstractions, data abstractions****, and* **Control abstraction**.

A- A procedural abstraction is a named sequence of instructions that has a specific and limited function.

An example of a procedural abstraction would be the word open for a door. Open implies a long sequence of procedural steps (e.g., walk to the door, reach out and grasp the knob, turn the knob and pull the door, step away from the moving door, etc.).

B- A data abstraction is a named collection of data that describes a data object.

In the context of the procedural abstraction open, we can define a data abstraction called a door. Like any data object, the data abstraction for the door would encompass a set of attributes that describe the door (e.g., door type, swing direction, opening mechanism, weight, and dimensions). It follows that the procedural abstraction open would make use of the information contained in the attributes of the data abstraction door.

Many modern programming languages provide mechanisms for creating abstract data types. A data type is already an abstraction. For example, the data type (*string*) is, to us, a collection of characters, but in memory, the value of your variable will be a bunch of bits.

C- Control abstraction is the third form of abstraction used in software design. Like procedural and data abstraction, control abstraction implies a program control mechanism without specifying internal details. An example of a control abstraction is the synchronization semaphore used to coordinate activities in an operating system.

## 2- **Software Architecture** (design a structure of something)

Software architecture refers to “the overall structure of the software and the ways in which that structure provides conceptual integrity for a system”.
A number of different architectural description languages (ADLs) have been developed to represent these models. Although many different ADLs have been proposed, the majority provide mechanisms for describing system components and the manner in which they are connected to one another.

## 3- **Pattern** (a repeated form)

A pattern simply means a repeated form or design in which the same shape is repeated several times to form a pattern. The pattern in the design process means the repetition of a solution to a common recurring problem within a certain context.

## 4- **Modularity (**subdivide the system**)**

Modularity is software divided into separately named and addressable components, often called modules, that are integrated to satisfy problem requirements.

modularity is the single attribute of software that allows a program to be manageable. Monolithic software (i.e., a large program composed of a single module) cannot be easily understood by a reader. The number of control paths, number of variables, and overall complexity would make understanding close to impossible.

Let $C(x)$ be a function that defines the perceived complexity of a problem x, and $E(x)$ be a function that defines the effort (in time) required to solve a problem x. For two problems, p1 and p2, if

$C(p1) > C(p2)$

it follows that

$E(p1) > E(p2)$

As a general case, this result is obvious. It does take more time to solve a difficult problem.

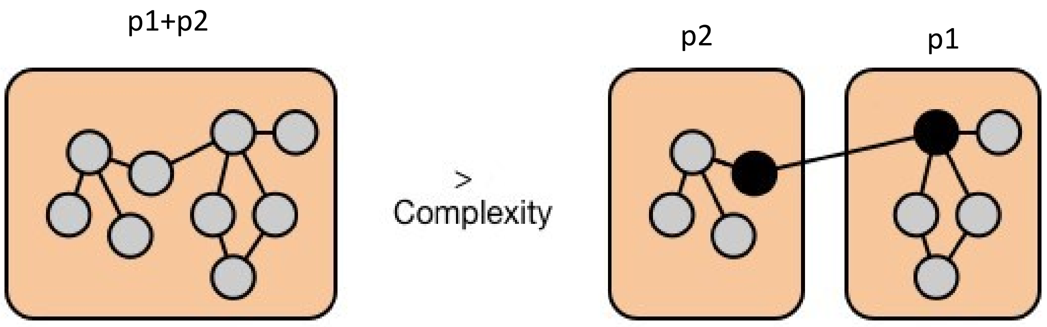
Another interesting characteristic has been uncovered through experimentation

in human problem-solving. That is,

$C(p1 + p2) > C(p1) + C(p2)$

implies that the perceived complexity of a problem that combines p1 and p2 is greater than the perceived complexity when each problem is considered separately, it follows that

$E(p1 + p2) > E(p1) + E(p2)$



It is possible to conclude from above expression that, if we subdivide software, the effort required to develop it will become small. Referring to Figure 2.1, the (cost) to develop an individual software module does decrease as the total number of modules increases. Given the same set of requirements, more modules mean smaller individual size. However, as the number of modules grows, the (cost) associated with integrating the modules also grows. These characteristics lead to a total cost or effort curve shown in the figure. There is a number, M, of modules that would result in minimum development cost, but we do not have the necessary sophistication to predict M with assurance.

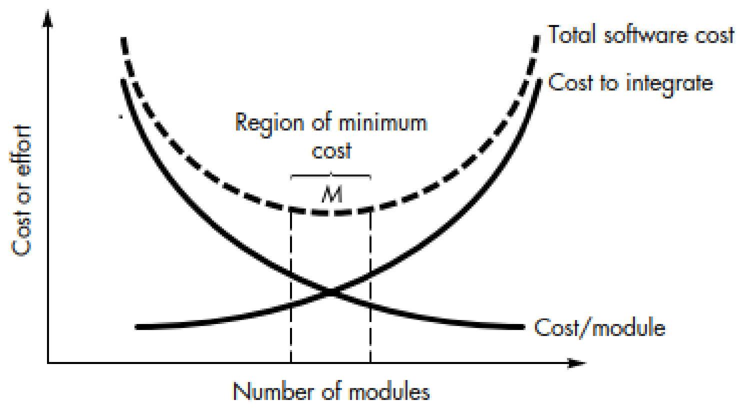


Figure 2.1: Modularity and software cost

The curves shown in Figure do provide useful guidance when modularity is considered. We should modularize, but care should be taken to stay in the range of M. Under modularity or over modularity should be avoided.

## 5- Information Hiding

The principle of information hiding suggests that modules be characterized by design decisions that each hide from all others. In other words, modules should be specified and designed so that information (procedure and data) contained within a module is inaccessible to other modules that have no need for such information.

## 6- **Refinement** (filtering)

Refinement simply means to refine something to remove any unwanted details if present and increase the quality. The refinement concept of software design is actually a process of developing or presenting the software or system in a detailed manner which means elaborating a system or software. Refinement is very necessary to find out any error if present and then to reduce it.

Abstraction and refinement are complementary concepts. Abstraction enables a designer to specify procedures and data and yet hide low-level details. Refinement helps the designer to reveal low-level details as the design progresses. Both concepts aid the designer in creating a complete design model as the design evolves.

## 7- **Refactoring** (reconstructing something)

Refactoring simply means reconstructing something in such a way that it does not affect the behavior of any other features. Refactoring in software design means reconstructing the design to reduce complexity and simplify it without affecting the behavior or its functions.