Software Design:

Software design is a process to transform user requirements into some suitable form, which helps the programmer in software coding and implementation. For assessing user requirements, an SRS (Software Requirement Specification) document is created whereas for coding and implementation, there is a need of more specific and detailed requirements in software terms. The output of this process can directly be used into implementation in programming languages. It tries to specify how to fulfill the requirements mentioned in SRS.

Design and Quality:

The design must implement all of the explicit requirements contained in the analysis model, and it must accommodate all of the implicit requirements desired by the customer. The design must be a readable, understandable guide for those who generate code and for those who test and subsequently support the software. The design should provide a complete picture of the software, addressing the data, functional, and behavioral domains from an implementation perspective.

Algorithm Design Model:

Represents the algorithm at a level of detail that can be reviewed for quality options:

- graphical (e.g. flowchart, box diagram)
- pseudocode (e.g., PDL)
- programming language

Software Design Levels:

Software design yields three levels of results:

1. **Architectural Design -** is the highest abstract version of the system. It identifies the software as a system with many components interacting with each other.
2. **High-level Design -** breaks the 'single entity-multiple component' concept of architectural design into less-abstracted view of sub-systems and modules and depicts their interaction with each other. It focuses on how the system along with all of its components can be implemented in forms of modules.
3. **Detailed Design -** deals with the implementation part of what is seen as a system and its sub-systems in the previous two designs. It is more detailed towards modules and their implementations. It defines logical structure of each module and their interfaces to communicate with other modules.

Fundamental Design Concepts:

The software engineer must recognize the difference between getting a program to work, and getting it *right".* Fundamental software design concepts provide the necessary framework for "getting it right".

- **Abstraction** data, procedure, control
- **Architecture** the overall structure of the software
- **Modularity** compartmentalization of data and function
- **Hiding** controlled interfaces
- **Refinement** elaboration of detail for all abstractions
- **Refactoring** a reorganization technique that simplifies the design
- **Functional independence** single-minded function and low coupling

Abstraction*:*

When we consider a modular solution to any problem, many *levels of abstraction* can be post. At the highest level of abstraction, a solution is stated in broad terms using the language of the problem environment. At levels of abstraction, a more procedural orientation is taken. Problem-oriented terminology is coupled with implementation- oriented terminology in an effort to state a solution. Finally, at the lowest level of abstraction, the solution is stated in a manner that can be directly implemented.

Software architecture:

Indicate to two important characteristics of a computer program:

- (1) the *hierarchical structure of procedural components (modules)* and
- (2) *the structure of data*.

Software architecture is derived through a partitioning process that relates elements of a software solution to parts of a real-world problem implicitly defined during requirements analysis.

Modularization:

Is a technique to divide a software system into multiple discrete and independent modules, which are expected to be capable of carrying out task(s) independently. These modules may work as basic constructs for the entire software. Designers tend to design modules such that they can be executed and/or compiled separately and independently. "*Modularity is the single attribute of software that allows a program to be intellectually manageable*".

Advantages of Modularization

- Smaller components are easier to maintain
- Program can be divided based on functional aspects
- Desired level of abstraction can be brought in the program
- Components with high cohesion can be re-used again
- Concurrent execution can be made possible

Monolithic software is a large program comprised of a single module. It cannot be easily understand by a reader. The number of control paths, number of variables, and overall complexity would make understanding close to impossible.

To illustrate this point, consider the following argument based on observations of human problem solving.

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, p_1 and p_2, if

$$C(p_1) > C(p_2)$$

It follows that

$$E(p_1) > E(p_2)$$

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

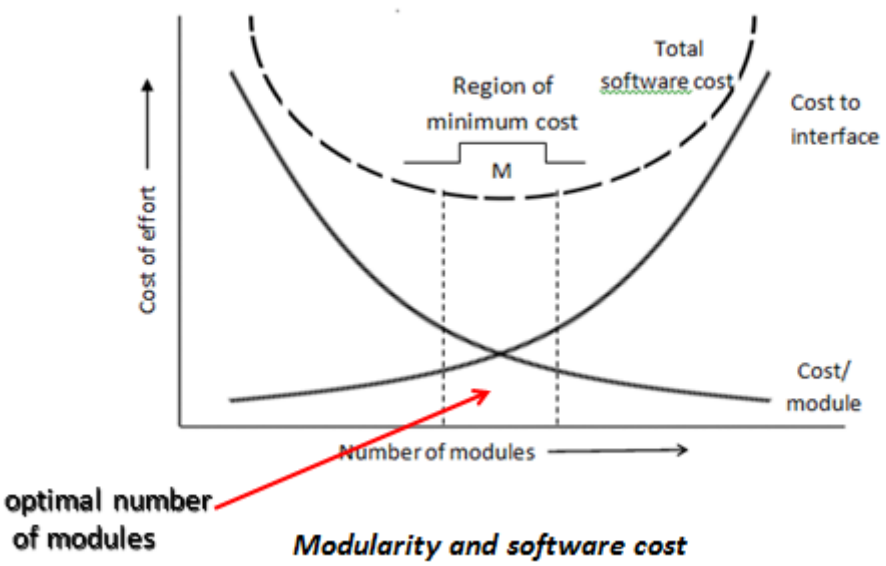
Another interesting characteristic has been uncovered through experimentation in human problem solving. That is,

$$C(p_1 + p_2) > C(p_1) + C(p_2)$$

This Equation implies that the perceived complexity of a problem that combines p_1 and p_2 is greater than when the perceived complexity when each problem is considered separately and it follows that

$$E(p_1 + p_2) > E(p_1) + E(p_2)$$

What is the "right" number of modules for a specific software design?



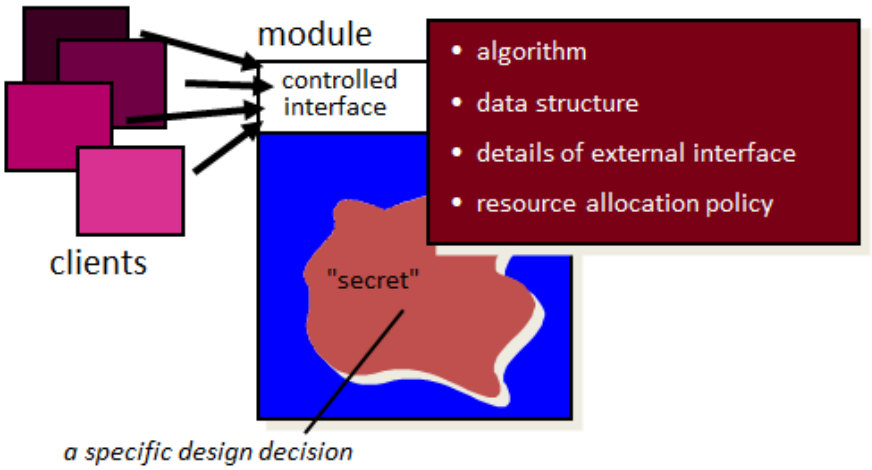
Modularity and software cost

Information Hiding:

The concept of modularity leads every software designer to a fundamental question: How do we decompose a software solution to obtain the best set of modules? The principle of *Information Hiding* suggests that modules be "characterized by design decisions that (each) hides from all others". In other words, modules should be specified and designer so that information (procedure and data) contained within a module are inaccessible to other modules that have no need for such information.

The use of information hiding as a design criterion for modular systems provides the greatest benefits when modifications are required during testing and later during software maintenance. Because most data and procedures are hidden from other parts of the software,

inadvertent errors introduces during modification are less likely to propagate to other locations within the software.



Why Information Hiding?

- Reduces the likelihood of "side effects"
- Limits the global impact of local design decisions
- Emphasizes communication through controlled interfaces
- Discourages the use of global data
- Leads to encapsulation—an attribute of high quality design
- Results in higher quality software

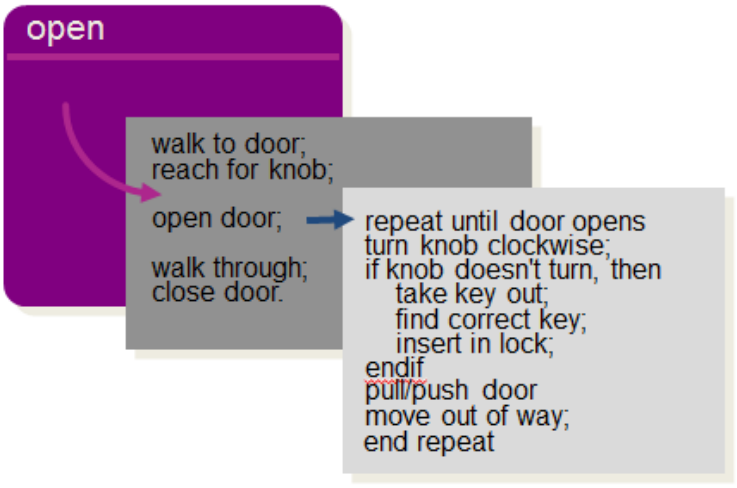
Refinement:

Stepwise refinement is an early top-down design strategy proposed by Niklaus Wirth. The architecture of a program is developed by successively refining levels of procedural detail. A hierarchy is developed by decomposing a macroscopic statement of function (a procedural abstraction) in a stepwise fashion until programming language statements are reached.

In each step (of the refinement), one or several instructions of the given program are decomposed into more detailed instructions. This successive decomposition or refinement of specifications terminates when all instructions are expressed in terms of any underlying computer or

programming language as tasks are refined. So the data may have to be refined, decomposed, or structured, and it is natural to refine the program and the data specifications in parallel.

Every refinement step implies some design decisions. It is important that the programmer be aware of the underlying criteria (for design decisions) and of the existent of alternative solutions.



Refactoring:

Refactoring is the process of changing a software system in such a way that it does not alter the external behavior of the code [design] yet improves its internal structure.

When software is refactored, the existing design is examined for

- redundancy
- unused design elements
- inefficient or unnecessary algorithms
- poorly constructed or inappropriate data structures
- or any other design failure that can be corrected to yield a better design.