**SOFTWARE DESIGN (UNIT 3)**

**WHAT IS SOFTWARE DESIGN**

Software design is the process of implementing software solutions to one or more set of problems. One of the important parts of software design is the software requirements analysis (SRA). It is a part of the software development process that lists specifications used in software engineering. If the software is "semi-automated" or user centered, software design may involve user experience design yielding a story board to help determine those specifications. If the software is completely automated (meaning no user or user interface), a software design may be as simple as a flow chart or text describing a planned sequence of events. There are also semi-standard methods like Unified Modeling Language and Fundamental modeling concepts. In either case, some documentation of the plan is usually the product of the design. Furthermore, a software design may be platform-independent or platform-specific, depending on the availability of the technology used for the design.

Software design can be considered as creating a solution to a problem in hand with available capabilities. The main difference between Software analysis and design is that the output of a software analysis consist of smaller problems to solve. Also, the analysis should not be very different even if it is designed by different team members or groups. The design focuses on the capabilities, and there can be multiple designs for the same problem depending on the environment that solution will be hosted. They can be operations systems, webpages, mobile or even the new cloud computing paradigm. Sometimes the design depends on the environment that it was developed, whether if it is created from with reliable frameworks or implemented with suitable design patterns).

When designing software, two important factors to consider are its security and usability.

**Design concepts**

The design concepts provide the software designer with a foundation from which more sophisticated methods can be applied. A set of fundamental design concepts has evolved. They are:

**1.** **Abstraction** - Abstraction is the process or result of generalization by reducing the information content of a concept or an observable phenomenon, typically in order to retain only information which is relevant for a particular purpose.

**2. Refinement** - It is the process of elaboration. A hierarchy is developed by decomposing a macroscopic statement of function in a step-wise fashion until programming language statements are reached. In each step, one or several instructions of a given program are decomposed into more detailed instructions. Abstraction and Refinement are complementary concepts.

3. **Modularity** - Software architecture is divided into components called modules.

Software Architecture - It refers to the overall structure of the software and the ways in which that structure provides conceptual integrity for a system. A good software architecture will yield a good return on investment with respect to the desired outcome of the project, e.g. in terms of performance, quality, schedule and cost.

4. **Control Hierarchy** - A program structure that represents the organization of a program component and implies a hierarchy of control.

5. **Structural Partitioning** - The program structure can be divided both horizontally and vertically. Horizontal partitions define separate branches of modular hierarchy for each major program function. Vertical partitioning suggests that control and work should be distributed top down in the program structure.

6. **Data Structure** - It is a representation of the logical relationship among individual elements of data.

7. **Software Procedure** - It focuses on the processing of each modules individually

8. **Information Hiding** - Modules should be specified and designed so that information contained within a module is inaccessible to other modules that have no need for such information.

**Design considerations**

There are many aspects to consider in the design of a piece of software. The importance of each should reflect the goals the software is trying to achieve. Some of these aspects are:

1. **Compatibility** - The software is able to operate with other products that are designed for interoperability with another product. For example, a piece of software may be backward-compatible with an older version of itself.
2. **Extensibility** - New capabilities can be added to the software without major changes to the underlying architecture.
3. **Fault-tolerance** - The software is resistant to and able to recover from component failure.
4. **Maintainability** - A measure of how easily bug fixes or functional modifications can be accomplished. High maintainability can be the product of modularity and extensibility.
5. **Modularity** - the resulting software comprises well defined, independent components. That leads to better maintainability. The components could be then implemented and tested in isolation before being integrated to form a desired software system. This allows division of work in a software development project.
6. **Reliability** - The software is able to perform a required function under stated conditions for a specified period of time.
7. **Reusability** - the software is able to add further features and modification with slight or no modification.
8. **Robustnes**s - The software is able to operate under stress or tolerate unpredictable or invalid input. For example, it can be designed with a resilience to low memory conditions.
9. **Security** - The software is able to withstand hostile acts and influences.

10. **Usability** - The software user interface must be usable for its target user/audience. Default values for the parameters must be chosen so that they are a good choice for the majority of the users.

**ABSTRACTION:** 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. 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.

Each step in the software process is a refinement in the level of abstraction of the software solution. During system engineering, 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 and data abstractions. 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 knob, turn knob and pull door, step away from moving door, etc.).

**DEFINE: Abstractions may be formed by reducing the information content of a concept or an observable phenomenon, typically to retain only information which is relevant for a particular purpose. For example, abstracting a leather soccer ball to the more general idea of a ball retains only the information on general ball attributes and behavior, eliminating the other characteristics of that particular balance.**

Abstract

1: disassociated from any specific instance 2: difficult to understand.

Example of abstraction:

1. Unix file descriptor
2. Can read, write, and (maybe) seek
3. File, IO device, pipe

Kinds of abstraction :

* Procedural abstraction

1. Naming a sequence of instruction
2. Parameterizing a procedure

* Data abstraction

1. Naming a collection of data
2. Data type defined by a set of procedures

* Control abstraction
* Performance abstraction

Modularity: Modularity refers to breaking down software into different parts. These parts have different names depending on your programming paridigm (for example, we talk about modules in imperative programming and objects in object oriented programming). By breaking the project down into pieces, it's (1) easier to both FIX (you can isolate problems easier) and (2) allows you to REUSE the pieces.

or

Is the degree to which a system's components may be separated and recombined. The meaning of the word, however, can vary somewhat by context:

**In software design**, modularity refers to a logical partitioning of the "software design" that allows complex software to be manageable for the purpose of implementation and maintenance. The logic of partitioning may be based on related functions, implementation considerations, data links, or other criteria.

Software architecture embodies modularity; that is, software is divided into separately named and addressable components, often called *modules that* are integrated to satisfy problem requirements.

It has been stated that "modularity is the single attribute of software that allows a program to be intellectually manageable". Monolithic software (i.e., a large program

composed of a single module) cannot be easily grasped by a reader. The number of control paths, span of reference, 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, *p1* and *p2*, if

*C(p1) > C(p2)* (9.1a)

it follows that

*E(p1) > E(p2)* (9.1b)

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(p1 + p2) > C(p1) + C(p2)* (9.2)

Expression (9-2) 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. Considering Expression (19-2) and the condition implied by Expressions

(9.1), it follows that

*E(p1 + p2) > E(p1) + E(p2)* (9.3)

This leads to a "divide and conquer" conclusion—it's easier to solve a complex problem when you break it into manageable pieces. The result expressed in Expression (9.3) has important implications with regard to modularity and software. It is, in fact, an argument for modularity.

It is possible to conclude from Expression (9-3) that, if we subdivide software indefinitely, the effort required to develop it will become negligibly small!

Unfortunately, other forces come into play, causing this conclusion to be (sadly) invalid. Referring to Figure.2,



**FIGURE 2:** Modularity and software cost

The effort (cost) to develop an individual software module does decrease as the total number of modules increases. Given the same set of requirements, more modules means smaller individual size. However, as the number of modules grows, the effort (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.

The curves shown in Figure 2 do provide useful guidance when modularity is considered. We should modularize, but care should be taken to stay in the vicinity of *M.* Undermodularity or overmodularity should be avoided.

**software architecture**

The term software architecture intuitively denotes the high level structures of a software system. It can be defined as the set of structures needed to reason about the software system, which comprise the software elements, the relations between them, and the properties of both elements and relations.

The term software architecture also denotes the set of practices used to select, define or design a software architecture.

Finally, the term often denotes the documentation of a system's "software architecture". Documenting software architecture facilitates communication between stakeholders, captures early decisions about the high-level design, and allows reuse of design components between projects..

**A set of properties should be specified as part of an architectural design:**

**Structural properties.** This aspect of the architectural design representation defines the components of a system (e.g., modules, objects) and the manner in which those components are packaged and interact with one another. For example, objects are packaged to encapsulate both data and the processing that manipulates the data and interact via the invocation of methods.

**Extra-functional properties.** The architectural design description should address how the design architecture achieves requirements for performance, capacity, reliability, security, adaptability, and other system characteristics.

**Families of related systems.** The architectural design should draw upon repeatable patterns that are commonly encountered in the design of families of similar systems. In essence, the design should have the ability to reuse architectural building blocks.

Given the specification of these properties, the architectural design can be represented using one or more of a number of different models.

***1-Structural models*** represent architecture as an organized collection of program components.

***2-Framework models*** increase the level of design abstraction by attempting to identify repeatable architectural design frameworks (patterns) that are encountered in similar types of applications.

***3-Dynamic models*** address the behavioral aspects of the program architecture, indicating how the structure or system configuration may change as a function of external events.

***4-Process models*** focus on the design of the business or technical process that the system must accommodate.

***5-Functional models*** can be used to represent the functional hierarchy of 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.

**Software architecture characteristics:**

Software architecture exhibits the following characteristics.

**Multitude of stakeholders:** software systems have to cater to a variety of stakeholders such as business managers, owners, users and operators. These stakeholders all have their own concerns with respect to the system. Balancing these concerns and demonstrating how they are addressed is part of designing the system.[2] This implies that architecture involves dealing with a broad variety of concerns and stakeholders, and has a multidisciplinary nature.

**Separation of concerns**: the established way for architects to reduce complexity is by separating the concerns that drive the design. Architecture documentation shows that all stakeholder concerns are addressed by modeling and describing the architecture from separate points of view associated with the various stakeholder concerns. These separate descriptions are called architectural views.

**Quality-driven**: classic software design approaches (e.g. Jackson Structured Programming) were driven by required functionality and the flow of data through the system, but the current insight is that the architecture of a software system is more closely related to its quality attributes such as fault-tolerance, backward compatibility, extensibility, reliability, maintainability, availability, security, usability, and other such –ilities. Stakeholder concerns often translate into requirements on these quality attributes, which are variously called non-functional requirements, extra-functional requirements, system quality requirements or constraints.

**Recurring styles**: like building architecture, the software architecture discipline has developed standard ways to address recurring concerns. These “standard ways” are called by various names at various levels of abstraction. Common terms for recurring solutions are architectural style, strategy or tactic, reference architecture and architectural pattern.

**Conceptual integrity:** a term introduced by Fred Brooks in The Mythical Man-Month to denote the idea that the architecture of a software system represents an overall vision of what it should do and how it should do it. This vision should be separated from its implementation. The architect assumes the role of “keeper of the vision”, making sure that additions to the system are in line with the architecture, hence preserving conceptual integrity.

**Effective Modular Design**:

Modularity has become an accepted approach in all engineering disciplines. A modular design reduces complexity, facilitates change (a critical aspect of software maintainability), and results in easier implementation by encouraging parallel development of different parts of a system.

**Functional Independence**

The concept of *functional independence* is a direct outgrowth of modularity and the concepts of abstraction and information hiding.

Functional independence is achieved by developing modules with "single-minded" function and an "aversion" to excessive interaction with other modules. Stated another way, we want to design software so that each module addresses a specific subfunction of requirements and has a simple interface when viewed from other parts of the program structure. It is fair to ask why independence is important. Software with effective modularity, that is, independent modules, is easier to develop because function may be compartmentalized and interfaces are simplified (consider the ramifications when development is conducted by a team). Independent modules are easier to maintain (and test) because secondary effects caused by design or code modification are limited, error propagation is reduced, and reusable modules are possible. To summarize, functional independence is a key to good design, and design is the key to software quality.

Independence is measured using two qualitative criteria: cohesion and coupling. *Cohesion* is a measure of the relative functional strength of a module. *Coupling* is a measure of the relative interdependence among modules.

1. **Cohesion**

Cohesion is a natural extension of the information hiding concept. A cohesive module performs a single task within a software procedure, requiring little interaction with procedures being performed in other parts of a program. Stated simply, a cohesive module should (ideally) do just one thing.

Cohesion may be represented as a "spectrum." We always strive for high cohesion, although the mid-range of the spectrum is often acceptable. Low-end cohesiveness is much "worse" than middle range, which is nearly as "good" as high-end cohesion. In practice, a designer need not be concerned with categorizing cohesion in a specific module. Rather, the overall concept should be understood and low levels of cohesion should be avoided when modules are designed.

At the low (undesirable) end of the spectrum, we encounter a module that performs a set of tasks that relate to each other loosely, if at all. Such modules are termed ***coincidentally cohesive****.* A module that performs tasks that are related logically is ***logically cohesive****.* e.g., one module may read all kinds of input (from tape ,disk and telecommunications port). When a module contains tasks that are related by the fact that all must be executed with the same span of time, the module exhibits ***temporal cohesion.***

As an example of low cohesion, consider a module that performs error processing for an engineering analysis package. The module is called when computed data exceed prespecified bounds. It performs the following tasks:

(1) computes supplementary data based on original computed data,

(2) produces an error report (with graphical content) on the user's workstation,

(3) performs follow-up calculations requested by the user

(4) updates a database, and

(5) enables menu selection for subsequent processing. Although the preceding tasks are loosely related, each is an independent functional entity that might best be performed as a separate module. Combining the functions into a single module can serve only to increase the likelihood of error propagation when a modification is made to one of its processing tasks.

Moderate levels of cohesion are relatively close to one another in the degree of module independence. When processing elements of a module are related and must be executed in a specific order, ***procedural cohesion*** exists. When all processing elements concentrate on one area of a data structure, ***communicational cohesion*** is present.

High cohesion is characterized by a module that performs one distinct procedural task.

**Note:**

It is unnecessary to determine the precise level of cohesion. Rather it is important to strive for high cohesion and recognize low cohesion so that software design can be modified to achieve greater functional independence.

**2. Coupling**

Coupling is a measure of interconnection among modules in a software structure. Coupling depends on the interface complexity between modules, the point at which entry or reference is made to a module, and what data pass across the interface.

In software design, we strive for lowest possible coupling. Simple connectivity among modules results in software that is easier to understand and less prone to a "ripple effect", caused when errors occur at one location and propagate through a system.

Figure 1 provides examples of different types of module coupling.



**Figure 1**: Types of coupling

Modules ***a*** and ***d*** are subordinate to different modules. Each is unrelated and therefore no direct coupling occurs. Module ***c*** is subordinate to module ***a*** and is accessed via a conventional argument list, through which data are passed. As long as a simple argument list is present (i.e., simple data are passed; a one-to-one correspondence of items exists), low coupling (called ***data coupling***) is exhibited in this portion of structure. A variation of data coupling, called ***stamp coupling****,* is found when aportion of a data structure (rather than simple arguments) is passed via a module interface. This occurs between modules ***b*** and ***a***.

At moderate levels, coupling is characterized by passage of control between modules. ***Control coupling*** is very common in most software designs and is shown in Figure 1 where a “control flag” (a variable that controls decisions in a subordinate or

superordinate module) is passed between modules ***d*** and ***e***.

High coupling occurs when a number of modules reference a global data area ***Common coupling****,* as this mode is called, is shown in Figure 1. Modules ***c****,* ***g****,* and ***k*** each access a data item in a global data area (e.g., a disk file or a globally accessible memory area). Module ***c*** initializes the item. Later module ***g*** recomputes and updates the item. Let's assume that an error occurs and ***g*** updates the item incorrectly. Much later in processing module, ***k*** reads the item, attempts to process it, and fails, causing the software to abort. The apparent cause of abort is module ***k***; the actual cause, module ***g****.* Diagnosing problems in structures with considerable common coupling is time consuming and difficult. However, this does not mean that the use of global data is necessarily "bad." It does mean that a software designer must be aware of potential consequences of common coupling and take special care to guard against them.

The highest degree of coupling, ***content coupling****,* occurs when one module makes use of data or control information maintained within the boundary of another module.

Secondarily, content coupling occurs when branches are made into the middle of a module. This mode of coupling can and should be avoided.

**Notes:**

***1-Attempt to minimize structures with high fan-out; strive for fan-in as depth increases.***

The structure shown inside the cloud in Figure 2 does not make effective use of factoring. All modules are “pancaked” below a single controlmodule. In general, a more reasonable distribution of control is shown in the upper structure. The structure takes an oval shape, indicating a number of layers of control and highly utilitarian modules at lower level



Figure 2: Program structures

2**- *Evaluate the "first iteration" of the program structure to reduce coupling and improve cohesion.***

Once the program structure has been developed, modules may be exploded or imploded with an eye toward improving module independence.

An *exploded module* becomes two or more modules in the final program structure. An *imploded module* is the result of combining the processing implied by two or more modules.

An exploded module often results when common processing exists in two or more modules and can be redefined as a separate cohesive module. When high coupling is expected, modules can sometimes be imploded to reduce passage of control, reference to global data, and interface complexity.

**structure chart :**

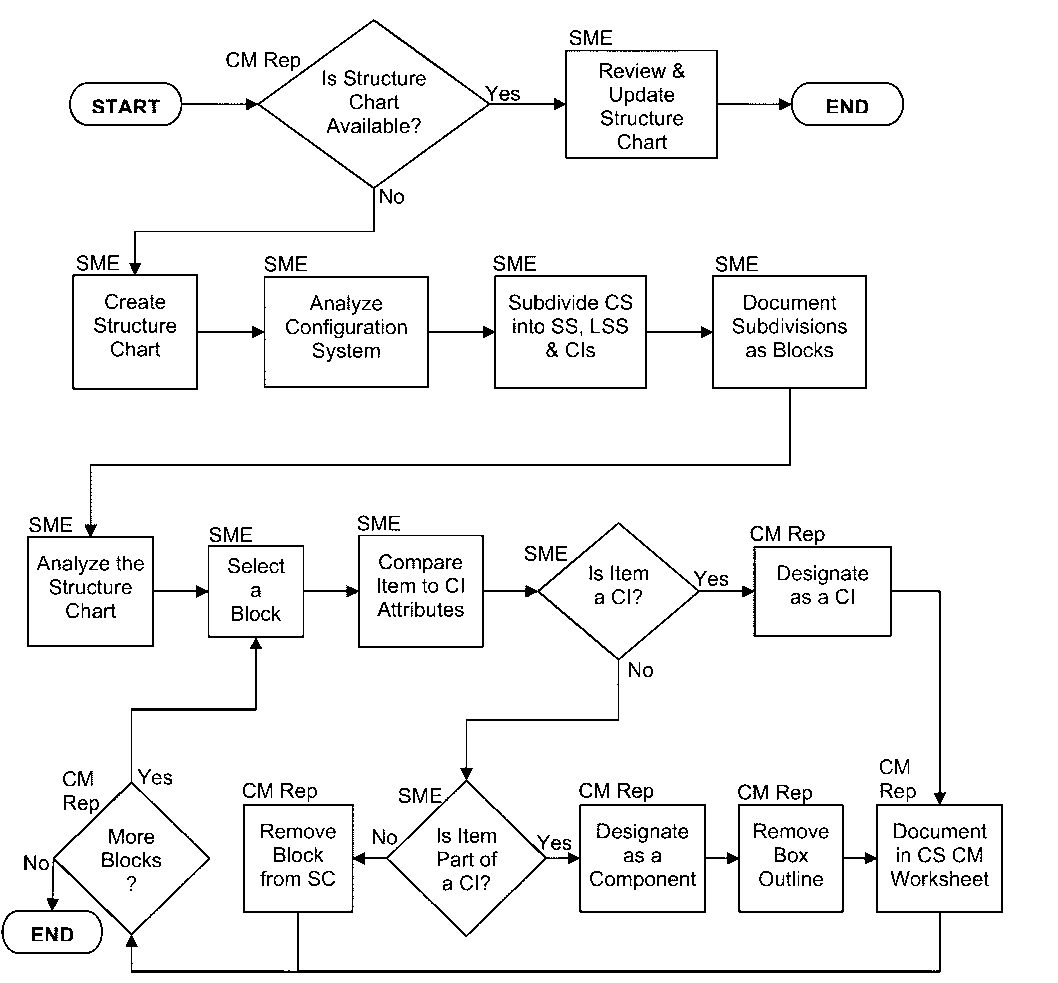
A structure chart is a top-down modular design tool, constructed of squares representing the different modules in the system, and lines that connect them. The lines represent the connection and or ownership between activities and subactivities as they are used in organization charts.

In structured analysis structure charts, according to Wolber (2009), "are used to specify the high-level design, or architecture, of a computer program. As a design tool, they aid the programmer in dividing and conquering a large software problem, that is, recursively breaking a problem down into parts that are small enough to be understood by a human brain. The process is called top-down design, or functional decomposition. Programmers use a structure chart to build a program in a manner similar to how an architect uses a blueprint to build a house. In the design stage, the chart is drawn and used as a way for the client and the various software designers to communicate. During the actual building of the program (implementation), the chart is continually referred to as "the master-plan".

**A structure chart depicts:**

* the size and complexity of the system, and
* number of readily identifiable functions and modules within each function and
* whether each identifiable function is a manageable entity or should be broken down into smaller components.

A structure chart is also used to diagram associated elements that comprise a run stream or thread. It is often developed as a hierarchical diagram, but other representations are allowable. The representation must describe the breakdown of the configuration system into subsystems and the lowest manageable level. An accurate and complete structure chart is the key to the determination of the configuration items, and a visual representation of the configuration system and the internal interfaces among its CIs. During the configuration control process, the structure chart is used to identify CIs and their associated artifacts that a proposed change may impact.



A process flow diagram describing the construction of a structure chart by a so called Subject Matter Experts (SME)

**Pseudocode:** is an informal high-level description of the operating principle of a computer program or other algorithm.

It uses the structural conventions of a programming language, but is intended for human reading rather than machine reading. Pseudocode typically omits details that are not essential for human understanding of the algorithm, such as variable declarations, system-specific code and some subroutines. The programming language is augmented with natural language description details, where convenient, or with compact mathematical notation. The purpose of using pseudocode is that it is easier for people to understand than conventional programming language code, and that it is an efficient and environment-independent description of the key principles of an algorithm. It is commonly used in textbooks and scientific publications that are documenting various algorithms, and also in planning of computer program development, for sketching out the structure of the program before the actual coding takes place.

No standard for pseudocode syntax exists, as a program in pseudocode is not an executable program. Pseudocode resembles, but should not be confused with skeleton programs, including dummy code, which can be compiled without errors. Flowcharts and Unified Modeling Language (UML) charts can be thought of as a graphical alternative to pseudocode, but are more spacious on paper.

**SYNTAX OF PSEUDOCODE:**

As the name suggests, pseudocode generally does not actually obey the syntax rules of any particular language; there is no systematic standard form, although any particular writer will generally borrow style and syntax; for example, control structures from some conventional programming language. Popular syntax sources include Pascal, BASIC, C, C++, Java, Lisp, and ALGOL. Variable declarations are typically omitted. Function calls and blocks of code, such as code contained within a loop, are often replaced by a one-line natural language sentence.

Depending on the writer, pseudocode may therefore vary widely in style, from a near-exact imitation of a real programming language at one extreme, to a description approaching formatted prose at the other.

This is an example of pseudocode (for the mathematical game fizz buzz):

Fortran style pseudo code

program fizzbuzz

do i = 1 to 100

set print\_number to true

if i is divisible by 3

print "Fizz"

set print\_number to false

if i is divisible by 5

print "Buzz"

set print\_number to false

if print\_number, print i

print a newline

end do

Pascal style pseudo code

procedure fizzbuzz

for i := 1 to 100 do

set print\_number to true;

if i is divisible by 3 then

print "Fizz";

set print\_number to false;

if i is divisible by 5 then

print "Buzz";

set print\_number to false;

if print\_number, print i;

print a newline;

end

C style pseudo code:

void function fizzbuzz

for (i = 1; i<=100; i++) {

set print\_number to true;

if i is divisible by 3

print "Fizz";

set print\_number to false;

if i is divisible by 5

print "Buzz";

set print\_number to false;

if print\_number, print i;

print a newline;

}

**Flowcharts:**

Flowcharts are used in designing and documenting complex processes or programs. Like other types of diagrams, they help visualize what is going on and thereby help the viewer to understand a process, and perhaps also find flaws, bottlenecks, and other less-obvious features within it. There are many different types of flowcharts, and each type has its own repertoire of boxes and notational conventions. The two most common types of boxes in a flowchart are:

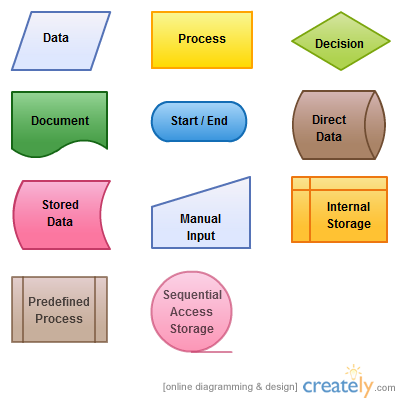
a processing step, usually called activity, and denoted as a rectangular box

a decision, usually denoted as a diamond.

A flowchart is described as "cross-functional" when the page is divided into different swimlanes describing the control of different organizational units. A symbol appearing in a particular "lane" is within the control of that organizational unit. This technique allows the author to locate the responsibility for performing an action or making a decision correctly, showing the responsibility of each organizational unit for different parts of a single process.

Flowcharts depict certain aspects of processes and they are usually complemented by other types of diagram. For instance, Kaoru Ishikawa defined the flowchart as one of the seven basic tools of quality control, next to the histogram, Pareto chart, check sheet, control chart, cause-and-effect diagram, and the scatter diagram. Similarly, in UML, a standard concept-modeling notation used in software development, the activity diagram, which is a type of flowchart, is just one of many different diagram types.

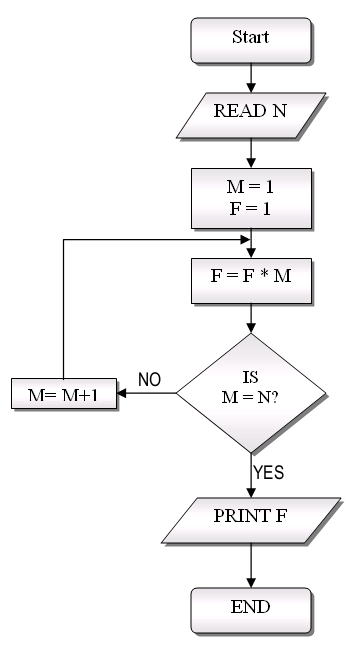
**FLOWCHART SYMBOLS:**



Before drawing flowcharts you need to understand the different symbols used in Flowcharts. Most people are aware of basic symbols like processes and decision blocks. But there are many more symbols to make your flowchart more meaningful. Below image shows all the standard flowchart symbols.

Most people don’t know about some rarely used flowchart symbols like Sequential access storage, Direct data, Manual Input etc. Check the flowchart symbols page for a detailed explanation of different symbols. Although these are the standard symbols available in most flowchart software, some people do use different shapes for different meanings. The most common example of this is the using circles to denote start and end. The examples in this flowchart guide will stick with the standard symbols.

**A simple flowchart example for computing factorial N** (N!)



**Measurement Method :**

We formally specify a scenario Σ as Σ= (SE, ∠SE, SO, MEO, MET), where SE represents all the environmental (Input/Output) events participating in the scenario, ∠SE is an order imposed on the events in time, SO is the set of domain concepts participating in the scenario, MEO is a mapping from SE to the pairs of objects that exchange events, and MET is a mapping from SE to the time axis. The environmental Input/Output events in a scenario are observable. The ordering ∠SE always produces a *legal sequence* of events, where legal means that the first event is an environmental Input event which is unconstrained, the last event is an environmental output event, and the partial order between the events satisfy the ordering ∠SE. The term legal sequence has been first introduced in [2].

*Cohesion measurement in a use-case*. We consider a use-case to be a set of scenarios. Scenarios that are operating on common legal subsequences of exchanged messages are considered similar. Similar scenarios contribute to increase the cohesion level in a use-case, while scenarios accessing disjoint legal subsequences of messages reduce the cohesion of a use-case, hinting at the possibility to split the use-case into two or smaller and more cohesive use cases.

Let Q be the set of the similar pairs of scenarios belonging to one use-case, and P be the set of all pairs of scenarios belonging to the same use-case. We define the Cohesion Level in the Use-Case measure reflecting our view on cohesion as follows:

CL\_UC = |Q|/|P|

The following steps are defined for our measurement method:

Step 1. Map each scenario Σ to a timed sequence of events ℑ.

Step 2. Identify the set ℜ of legal subsequences for each sequence ℑ.

Step 3. At this step, the set Q is identified. For each pair of scenarios (order is not important to avoid duplication of the results) find the intersection of the corresponding sets of legal sequences. If the intersection is nonempty, add the pair to the set Q.

Step 4. Calculate |P|.

Step 5. Calculate CL\_UC.

The unit of cohesion in the CL\_UC measure is “similarity” abstracted as a pair of similar scenarios whose goals are related by a given legal subsequence. The scale type for CL\_UC is absolute since the only allowable transformations are identities, and there exist an absolute zero indicating a lack of the quality attribute (cohesion) in the use-case.

The range of the values for the measure CL\_UC is [0..1], where 1 indicates the highest cohesion (all scenarios in the use-case are intersected by each other and thus all scenarios are similar), and 0 indicates the lack of cohesion in the use-case thus requiring re-analysis of the functional requirements related to this use-case.

*Cohesion measurement in a use-case model*. We state that a crosscutting concern is a subgoal corresponding to a legal subsequence common to at least two scenarios belonging to different use cases. Let two different use-cases U1 and U2 be presented by the sets of scenarios Σ1 and Σ2 respectively. Intuitively, the existence of similar scenarios in different use-cases indicates a low level of separation of concerns, i.e., low level of partitioning quality. The non-empty intersection of the sets of legal subsequences characterizing Σ1 and Σ2 not only indicate the presence of candidate crosscutting concerns (aspects), but also identifies U1 and U2 as candidates for crosscutting functional requirements. Thus, our cohesion measurement mechanism contributes to the “Analysis and crosscuttings realization” activity of our AOSD framework (see Figure 1)

The Cohesion Level in the Use-Case Model measure is defined on the set of all scenarios belonging to all use-cases in the use-case model:

CL\_UCM = 1-|QM|/|PM|

where QM is the set of the pairs of similar scenarios, each pair containing scenarios belonging to different use-cases, and PM is the set of all pairs of scenarios (same condition apply).

The Steps 1 and 2 of the CL\_UCM measurement method are identical to the corresponding steps in CL\_UC. The rest of the steps are described below:

Step 3. At this step, the set Q is identified. For each pair of scenarios (order is not important to avoid duplication of the results) find the intersection of the corresponding sets of legal subsequences. If the intersection is nonempty, add the pair to the set Q. The non-empty intersections not only indicate the presence of candidate crosscutting concerns, but also identify them.

Step 4. Calculate |PM|.

Step 5. Calculate CL\_UCM.

The unit of cohesion in the CL\_UCM measure is a crosscutting concern abstracted as a pair of scenarios whose goals are related by the given crosscutting concern. The scale type of CL\_UCM is absolute since the only allowable transformations are identities, and there exist an absolute zero indicating a lack of the quality attribute (cohesion) in the use-case model.

The range of the values for the measure CL\_UCM is [0..1], where 1 indicates the highest level of cohesion (there is no intersection between the scenarios from different use-cases), and 0 indicates the lack of cohesion (all pairs of scenarios are crosscutted). Higher CL\_UCM values indicate that possible crosscuttings are to be identified.

***Formal Properties of Cohesion***

The CL\_UC and CL\_UCM measures are theoretically validated against the set of axioms proposed in [1]:

1. ***Cohesion is non-negative*.**

*Yes*. Discussion: the ranges of values for both CL measures are [0..1], therefore negative values are not allowed.

***2. Cohesion is independent of size***

*Yes*. Discussion: CLAIM targets the usage model of the system without accounting for the size aspects.

3. ***Cohesion can be null***

*Yes*. Discussion: the 0 value for the cohesion indicates lack of it in the use-case model.

4. ***Cohesion of a collection of elements or properties is independent of the internal structure of the collection of its components.***

5. ***Cohesion forms a weak order.***

*Yes.* Discussion: we can always compare and order the use cases in terms of their CL\_UC data, and the use-case models in terms of their CL\_UCM values.

**4.2 Coupling**

We have adopted the MOOD’s Coupling Factor measure [4] to quantify the existing level of coupling in the domain model due to associations between the conceptual classes. *Coupling Factor (CF)* is a measurement of the level of coupling in the (partial) domain model and is defined as follows:

CF = ΣTCi=1 **[**ΣTCj=1 is\_client(Ci, Cj)**]** / (TC2 – TC)

where

1, iff Cc=> Cs ^ Cc ≠ Cs

0, otherwise

and Cc => Cs represents the relationship between a client class Cc and a supplier class Cs . The range of the values for the CF is [0..1], where 0 indicates lack of coupling, and 1 is corresponding to the highest possible level of coupling. As a higher value of CF would indicate higher level of coupling between the concepts in the (partial) domain, this value may be considered as an implication of crosscutting requirement(s) to be realized.

The unit of measurement in our version of the CF measure is an abstraction of the coupling unit, namely, an association between two concepts in the domain model expressed as an ordered pair of concepts (Ci,Cj).

The scale type of the CF measure is absolute since the only allowable transformations are identities, and there exist a hypothetical absolute zero indicating a lack of the quality attribute (coupling) in the domain model.

We have validated theoretically the proposed coupling measure against the axioms for coupling proposed in [1], as discussed below.

***Formal Properties of Coupling***

1***. Coupling is non-negative.***

*Yes*. Discission: the range of CF values is [0..1], therefore negative values are not allowed.

2. ***Coupling can be null***.

*Yes*. Discussion: theoretically, the domain model can exhibit 0 coupling corresponding to the lack of it in the model.

3. ***Adding an intercomponent relationship does not decrease coupling.***

*Yes.* Discussion: adding one more association would increase the value of CF.

4. ***Merging two components does not increase coupling.***

*Yes*. Discussion: the number of coupled pairs of concepts will remain the same or decrease (if duplicated), the number of classes might increase, therefore the CF value would eventually decrease.

5. ***Merging two unconnected components does not change coupling.***

*No.* Discussion: the number of coupled pairs of concepts will remain the same, the number of classes might increase, therefore the CF value would in general change.

6. ***Coupling forms a weak order.***

*Yes.* Discussion: domain models can be ordered in terms of their coupling.

The following section

**Function-oriented design**

A function-oriented design strategy relies on decomposing the system into a set of

interacting functions with a centralised system state shared by these functions

(Figure 15.1). Functions may also maintain local state information but only for the

duration of their execution..

Function-oriented design conceals the details of an algorithm in a function but system state information is not hidden. This can cause problems because a

function can change the state in a way which other functions do not expect.

Changes to a function and the way in which it uses the system state may cause

unanticipated changes in the behaviour of other functions.

A functional approach to design is therefore most likely to be successful

when the amount of system state information is minimised and information sharing

is explicit. Systems whose responses depend on a single stimulus or input and

which are not affected by input histories are naturally functionally-oriented. Many

transaction-processing systems and business data-processing systems fall into this

class. In essence, they are concerned with record processing where the processing of

one record is not dependent on any previous processing.

An example of such a transaction processing system is the software which

controls automatic teller machines (ATMs) which are now installed outside many

banks. The service provided to a user is independent of previous services provided so

can be thought of as a single transaction.



Function-oriented design

* Practiced informally since programming began
* Thousands of systems have been developed   
  using this approach
* Supported directly by most programming   
  languages
* Most design methods are functional in their   
  approach
* CASE tools are available for design support

**ATM software design:**

**loop**

**loop**

Print\_input\_message (” Welcome - Please enter your card”) ;

**exit when** Card\_input ;

**end loop** ;

Account\_number := Read\_card ;

Get\_account\_details (PIN, Account\_balance, Cash\_available) ;

**if** Validate\_card (PIN) **then**

**loop**

Print\_operation\_select\_message ;

**case** Get\_button **is**

**when** Cash\_only =>

Dispense\_cash (Cash\_available, Amount\_dispensed) ;

**when** Print\_balance =>

Print\_customer\_balance (Account\_balance) ;

**when** Statement =>

Order\_statement (Account\_number) ;

**when** Check\_book =>

Order\_checkbook (Account\_number) ;

**end case** ;

Eject\_card ;

Print (“Please take your card or press CONTINUE”) ;

**exit when** Card\_removed ;

**end loop** ;

Update\_account\_information (Account\_number, Amount\_dispensed) ;

**else**

Retain\_card ;

**end if** ;

**end loop** ;

**Object-oriented design:**

An object contains encapsulated data and procedures grouped together to represent an entity. The 'object interface', how the object can be interacted with, is also defined. An object-oriented program is described by the interaction of these objects. Object-oriented design is the discipline of defining the objects and their interactions to solve a problem that was identified and documented during object-oriented analysis.

What follows is a description of the class-based subset of object-oriented design, which does not include object prototype-based approaches where objects are not typically obtained by instancing classes but by cloning other (prototype) objects.

Both the sequential and concurrent designs of the OIRS are functional because the

principal decomposition strategy identifies functions such as Execute command,

Update index, etc. By contrast, an object-oriented design focuses on the entities in

the system with the functions part of these entities. There is not enough space here

to develop a complete object-oriented design but I show the entity decomposition in

this section. Notice that this is quite different from the functional system

decomposition.

**Designing concepts:**

* Defining objects, creating class diagram from conceptual diagram: Usually map entity to class.
* Identifying attributes.
* Use design patterns (if applicable): A design pattern is not a finished design, it is a description of a solution to a common problem, in a context.[1] The main advantage of using a design pattern is that it can be reused in multiple applications. It can also be thought of as a template for how to solve a problem that can be used in many different situations and/or applications. Object-oriented design patterns typically show relationships and interactions between classes or objects, without specifying the final application classes or objects that are involved.
* Define application framework (if applicable): Application framework is a term usually used to refer to a set of libraries or classes that are used to implement the standard structure of an application for a specific operating system. By bundling a large amount of reusable code into a framework, much time is saved for the developer, since he/she is saved the task of rewriting large amounts of standard code for each new application that is developed.
* Identify persistent objects/data (if applicable): Identify objects that have to last longer than a single runtime of the application. If a relational database is used, design the object relation mapping.
* Identify and define remote objects (if applicable).

**Input (sources) for object-oriented design:**

The input for object-oriented design is provided by the output of object-oriented analysis. Realize that an output artifact does not need to be completely developed to serve as input of object-oriented design; analysis and design may occur in parallel, and in practice the results of one activity can feed the other in a short feedback cycle through an iterative process. Both analysis and design can be performed incrementally, and the artifacts can be continuously grown instead of completely developed in one shot.

**Some typical input artifacts for object-oriented design are**:

**Conceptual model:** Conceptual model is the result of object-oriented analysis, it captures concepts in the problem domain. The conceptual model is explicitly chosen to be independent of implementation details, such as concurrency or data storage.

**Use case**: Use case is a description of sequences of events that, taken together, lead to a system doing something useful. Each use case provides one or more scenarios that convey how the system should interact with the users called actors to achieve a specific business goal or function. Use case actors may be end users or other systems. In many circumstances use cases are further elaborated into use case diagrams. Use case diagrams are used to identify the actor (users or other systems) and the processes they perform.

**System Sequence Diagram:** System Sequence diagram (SSD) is a picture that shows, for a particular scenario of a use case, the events that external actors generate, their order, and possible inter-system events.

**User interface documentations (if applicable):** Document that shows and describes the look and feel of the end product's user interface. It is not mandatory to have this, but it helps to visualize the end-product and therefore helps the designer.

**Relational data model (if applicable):** A data model is an abstract model that describes how data is represented and used. If an object database is not used, the relational data model should usually be created before the design, since the strategy chosen for object-relational mapping is an output of the OO design process. However, it is possible to develop the relational data model and the object-oriented design artifacts in parallel, and the growth of an artifact can stimulate the refinement of other artifacts.

**Output (deliverables) of object-oriented design:**

**Sequence Diagrams**: Extend the System Sequence Diagram to add specific objects that handle the system events.

A sequence diagram shows, as parallel vertical lines, different processes or objects that live simultaneously, and, as horizontal arrows, the messages exchanged between them, in the order in which they occur.

**Class diagram:** A class diagram is a type of static structure UML diagram that describes the structure of a system by showing the system's classes, their attributes, and the relationships between the classes. The messages and classes identified through the development of the sequence diagrams can serve as input to the automatic generation of the global class diagram of the system.

**Top-down and bottom-up design:**

Top-down and bottom-up are both strategies of information processing and knowledge ordering, used in a variety of fields including software, humanistic and scientific theories (see systemics), and management and organization. In practice, they can be seen as a style of thinking and teaching.

**A top-down approach** (also known as stepwise design or deductive reasoning, and in many cases used as a synonym of analysis or decomposition) is essentially the breaking down of a system to gain insight into its compositional sub-systems. In a top-down approach an overview of the system is formulated, specifying but not detailing any first-level subsystems. Each subsystem is then refined in yet greater detail, sometimes in many additional subsystem levels, until the entire specification is reduced to base elements. A top-down model is often specified with the assistance of "black boxes", these make it easier to manipulate. However, black boxes may fail to elucidate elementary mechanisms or be detailed enough to realistically validate the model. Top down approach starts with the big picture. It breaks down from there into smaller segments.

**A bottom-up approach** (also known as inductive reasoning, and in many cases used as a synonym of synthesis) is the piecing together of systems to give rise to grander systems, thus making the original systems sub-systems of the emergent system. Bottom-up processing is a type of information processing based on incoming data from the environment to form a perception. Information enters the eyes in one direction (input), and is then turned into an image by the brain that can be interpreted and recognized as a perception (output). In a bottom-up approach the individual base elements of the system are first specified in great detail. These elements are then linked together to form larger subsystems, which then in turn are linked, sometimes in many levels, until a complete top-level system is formed. This strategy often resembles a "seed" model, whereby the User interface design or user interface engineering is the design of computers, appliances, machines, mobile communication devices, software applications, and websites with the focus on the user's experience and interaction. The goal of user interface design is to make the user's interaction as simple and efficient as possible, in terms of accomplishing user goals**—what is often called user-centered design. Good user interface design facilitates finishing the task at hand without drawing unnecessary attention to itself. Graphic design may be utilized to support its usability. The design process must balance technical functionality and visual elements (e.g., mental model) to create a system that is not only operational but also usable and adaptable to changing user needs.**

**Interface design is involved in a wide range of projects from computer systems, to cars, to commercial planes; all of these projects involve much of the same basic human interactions yet also require some unique skills and knowledge. As a result, designers tend to specialize in certain types of projects and have skills centered around their expertise, whether that be software design, user research, web design, or industrial design.**"organic strategies" may result in a tangle of elements and subsystems, developed in isolation and subject to local optimization as opposed to meeting a global purpose.

**User Interface Design:**

**User Interface Design:User interface design or user interface engineering is the design of computers, appliances, machines, mobile communication devices, software applications, and websites with the focus on the user's experience and interaction. The goal of user interface design is to make the user's interaction as simple and efficient as possible, in terms of accomplishing user goals—what is often called user-centered design. Good user interface design facilitates finishing the task at hand without drawing unnecessary attention to itself. Graphic design may be utilized to support its usability. The design process must balance technical functionality and visual elements (e.g., mental model) to create a system that is not only operational but also usable and adaptable to changing user needs.**

**Interface design is involved in a wide range of projects from computer systems, to cars, to commercial planes; all of these projects involve much of the same basic human interactions yet also require some unique skills and knowledge. As a result, designers tend to specialize in certain types of projects and have skills centered around their expertise, whether that be software design, user research, web design, or industrial design.**

**Human–computer interaction:involves the study, planning, and design of the interaction between people (users) and computers. It is often regarded as the intersection of computer science, behavioral sciences, design and several other fields of study. The term was popularized by Card, Moran, and Newell in their seminal 1983 book, "The Psychology of Human-Computer Interaction", although the authors first used the term in 1980, and the first known use was in 1975. The term connotes that, unlike other tools with only limited uses (such as a hammer, useful for driving nails, but not much else), a computer has many affordances for use and this takes place in an open-ended dialog between the user and the computer.**

**Because human–computer interaction studies a human and a machine in conjunction, it draws from supporting knowledge on both the machine and the human side. On the machine side, techniques in computer graphics, operating systems, programming languages, and development environments are relevant. On the human side, communication theory, graphic and industrial design disciplines, linguistics, social sciences, cognitive psychology, and human factors such as computer user satisfaction are relevant. Engineering and design methods are also relevant. Due to the multidisciplinary nature of HCI, people with different backgrounds contribute to its success. HCI is also sometimes referred to as man–machine interaction (MMI) or computer–human interaction (CHI).**

**Attention to human-machine interaction is important because poorly designed human-machine interfaces can lead to many unexpected problems. A classic example of this is the Three Mile Island accident, a nuclear meltdown accident, where investigations concluded that the design of the human–machine interface was at least partially responsible for the disaster.Similarly, accidents in aviation have resulted from manufacturers' decisions to use non-standard flight instrument and/or throttle quadrant layouts: even though the new designs were proposed to be superior in regards to basic human–machine interaction, pilots had already ingrained the "standard" layout and thus the conceptually good idea actually had undesirable results.**

**Human-computer interface :**

**What do these things have in common?**

1. **A Computer Mouse**
2. **A Touch Screen**
3. **A program on your Mac or Windows machine that includes a trashcan, icons of disk drives, and folders**
4. **Pull-down menus**

**Give up? These are all examples of advances in human-computer interface design which were designed to make it easier to accomplish things with a computer. Do you remember the first days of desktop computers? I do. I remember when I had to remember long strings of commands in order to do the simplest things like copy or format a disk or move to a new directory. I don't miss those days! Thank you, human-computer interface designers.**

**Human-Computer Interface Design seeks to discover the most efficient way to design understandable electronic messages . Research in this area is voluminous; a complete branch of computer science is devoted to this topic, with recommendations for the proper design of menus, icons, forms, as well as data display and entry screens. This browser you are using now is a result of interface design - the buttons and menus have been designed to make it easy for you to access the web. Some recommendations from this research are discussed below.**

**Note: Many of these recommendations concern the design of computer interfaces like Windows or the Mac Finder or how to make programs easier to use. Some of these recommendations are not so relevant to web design. Still, it is an important area of research, and some of the recommendations relate to any kind of communication between user and computer. Wherever possible, I have included examples more directly related to web design.**

**Interface design:**

**Interface design deals with the process of developing a method for two (or more) modules in a system to connect and communicate. These modules can apply to hardware, software or the interface between a user and a machine. An example of a user interface could include a GUI, a control panel for a nuclear power plant,or even the cockpit of an aircraft.**

**In systems engineering, all the inputs and outputs of a system, subsystem, and its components are listed in an interface control document often as part of the requirements of the engineering project.**

**The development of a user interface is a unique field. More information can be found on the subject here: User interface design.**

**Interface standard:**

**In telecommunications, an interface standard is a standard that describes one or more functional characteristics (such as code conversion, line assignments, or protocol compliance) or physical characteristics (such as electrical, mechanical, or optical characteristics) necessary to allow the exchange of information between two or more (usually different) systems or pieces of equipment. Communications protocols are an example.**

**An interface standard may include operational characteristics and acceptable levels of performance.**

**In the military community, interface standards permit command and control functions to be performed using communication and computer systems.**