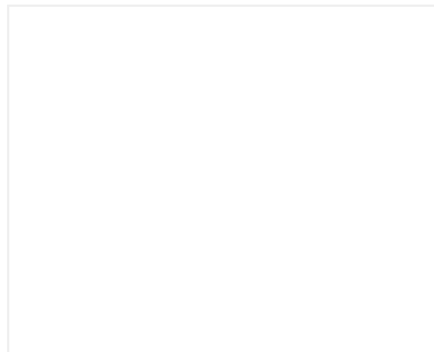
vulmshelp.com

INTRODUCION TO COMPUTING

CS101 FINAL TERM
HIGHLIGHTED HANDOUTS

ACCORDING TO NEW SYLLABUS 2020

vulmshelp.com



Chapter 6: Programming Languages

Topic 109 to 125

- ✓ Machine Language
- ✓ Assembly Language
- ✓ High level languages

EARLY GENERATIONS-1

Programs for modern computers consist of sequences of instructions that are encoded as numeric digits. Such an encoding system is known as a machine language. Unfortunately, writing programs in a machine language is a tedious task that often leads to errors that must be located and corrected (a process known as debugging) before the job is finished.

✓ Using Mnemonics and descriptive names

In the 1940s, researchers simplified the programming process by developing notational systems by which instructions could be represented in mnemonic rather than numeric form.

Machine codes	
1	Load
3	Store
4	Move
5	Add

For example, the instruction Move the contents of register 5 to register 6 4056 using the machine language introduced,

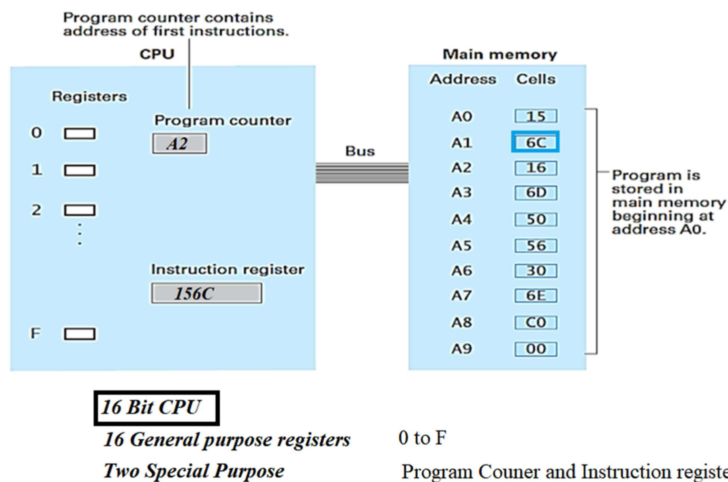
Whereas in a mnemonic system it might appear as MOV R5, R6

As a more extensive example, the machine language routine

Machine Language	Assembly Language
156C	**LD** R5, Price
166D	**LD** R6, ShippingCharge
5056	**ADDI** R0, R5 R6
306E	**ST** R0, TotalCost
C000	**HLT**

Which adds the contents of memory cells 6C and 6D and stores the result at location 6E

Note: 156C loads register 5 with bit pattern found in memory cell at address 6C



using mnemonics. (Here we have used LD, ADDI, ST, and HLT to represent load, add, store, and halt.

Moreover, we have used the descriptive names Price, ShippingCharge, and TotalCost to refer to the memory cells at locations 6C, 6D, and 6E, respectively. Such descriptive names are often called program variables or identifiers.) Note that the mnemonic form, although still lacking, does a better job of representing the meaning of the routine than does the numeric form.

TOPIC 110: EARLY GENERATIONS-2

- ✓ **Assemblers**

Once such a mnemonic system was established, programs called assemblers were developed to convert mnemonic expressions into machine language instructions. Thus, rather than being forced to develop a program directly in machine language, a human could develop a program in mnemonic form and then have it converted into machine language by means of an assembler.

MOV R5, R6 to 4056

- ✓ **Assembly language**

A mnemonic system for representing programs is collectively called an assembly language. At the time assembly languages were first developed, they represented a giant step forward in the search for better programming techniques. In fact, assembly languages were so revolutionary that they became known as second-generation languages, the first generation being the machine languages themselves.

- ✓ **Assembly language disadvantages**

Although assembly languages have many advantages over their machine language counterparts, they still fall short of providing the ultimate programming environment. After all, the primitives used in an assembly language are essentially the same as those found in the corresponding machine language. The difference is simply in the syntax used to represent them. Thus, a program written in an assembly language is inherently machine dependent – that is, the instructions within the program are expressed in terms of a particular machine's attributes. In turn, a program written in assembly language cannot be easily transported to another computer design because it must be rewritten to conform to the new computer's register configuration and instruction set.

Another disadvantage of an assembly language is that a programmer, although not required to code instructions in numeric form, is still forced to think in terms of the small, incremental steps of the machine's language. (Low level primitives) The situation is analogous to designing a house in terms of boards, nails, bricks, and so on. It is true that the actual construction of the house ultimately requires a description based on these elementary pieces, but the design process is easier if we think in terms of larger units such as rooms, windows, doors, and so on.

In short, the elementary primitives in which a product must ultimately be constructed are not necessarily the primitives that should be used during the product's design. The design process is better suited to the use of high-level primitives, each representing a concept associated with a major feature of the product. Once the design is complete, these primitives can be translated to lower-level concepts relating to the details of implementation.

- ✓ **Machine Independent**

Following this philosophy, computer scientists began developing programming languages that were more conducive to software development than were the low-level assembly languages. The result was the emergence of a third generation of programming languages that differed from previous generations in that their primitives were both higher level (in that they expressed instructions in larger increments) and machine independent (in that they did not rely on the characteristics of a particular machine). The best known early examples are FORTRAN (FORmula TRANslator), which was developed for scientific and engineering applications, and COBOL (COmmon Business-Oriented Language), which was developed by the U.S. Navy for business applications.

In general, the approach to third-generation programming languages was to identify a collection of high level primitives in which software could be developed. Each of these primitives was designed so that it could be implemented as a sequence of the low-level primitives available in machine languages. For example, the statement

TotalCost = Price + ShippingCharge

expresses a high-level activity without reference to how a particular machine should perform the task, yet it can be implemented by the sequence of machine instructions discussed earlier. Thus, our pseudocode structure

identifier = expression is a potential high-level primitive.

- **Translators**

Once this collection of high-level primitives had been identified, a program, called a translator, was written that translated programs expressed in these high-level primitives into machine-language programs. Such translator was similar to the second-generation assemblers, except that it often had to compile several machine instructions into short sequences to simulate the activity requested by a single high-level primitive.

- **Compilers**

Translators, which compile several machine instructions into short sequences to simulate the activity requested by a single high-level primitive. Such Translation programs were often called compilers.

- **Interpreters**

An alternative to translators, called interpreters, emerged as another means of implementing third generation languages. These programs were similar to translators except that they executed the instructions as they were translated instead of recording the translated version for future use. That is, rather than producing a machine-language copy of a program that would be executed later, an interpreter actually executed a program from its high-level form.

As a side issue, we should note that the task of promoting third-generation programming languages was not as easy as might be imagined. The thought of writing programs in a form similar to a natural language was so revolutionary that many in managerial positions fought the notion at first. Grace Hopper, who is recognized as the developer of the first compiler, often told the story of demonstrating a translator for a third-generation language in which German terms, rather than English, were used. The point was that the programming language was constructed around a small set of primitives that could be expressed in a variety of natural languages with only simple modifications to the translator. But she was surprised to find that many in the audience were shocked that, in the years surrounding World War II, she would be teaching a computer to "understand" German.

- **Natural languages and formal languages**

Today we know that understanding a natural language involves much more than responding to a few rigorously defined primitives. Indeed, natural languages (such as English, German, and Latin) evolved over time without formal grammatical analysis.

Formal languages (such as programming languages) are precisely defined by grammars.

TOPIC 111: MACHINE INDEPENDENCE

- **Goal of machine independence**

With the development of third-generation languages, the goal of machine independence was largely achieved. Since the statements in a third-generation language did not refer to the attributes of any particular machine, they could be compiled as easily for one machine as for another.

- **A program written in a third generation language could theoretically be used on any machine simply by applying the appropriate compiler.**

When a compiler is designed, particular characteristics of the underlying machine are sometimes reflected as conditions on the language being translated.

- **The different ways in which machines handle I/O operations have historically caused the "same" language to have different characteristics, or dialects, on different machines.**
- **Consequently, it is often necessary to make at least minor modifications to a program to move it from one machine to another.**

Compounding this problem of portability is the lack of agreement in some cases as to what constitutes the correct definition of a particular language.

- **Standardization**
  - **To aid in this regard, the American National Standards Institute and the International Organization for Standardization have adopted and published standards for many of the popular languages.**

- In other cases, informal standards have evolved because of the popularity of a certain dialect of a language and the desire of other compiler writers to produce compatible products.

✓ **Language Extensions**

- Compiler designers often provide features, sometimes called language extensions that are not part of the standard version of the language.
- If a programmer takes advantage of these features, the program produced will not be compatible with environments using a compiler from a different vendor, makes the code machine dependent

In the overall history of programming languages, the fact that third generation languages fell short of true machine independence is actually of little significance for two reasons. First, they were close enough to being machine independent that software could be transported from one machine to another with relative ease. Second, the goal of machine independence turned out to be only a seed for more demanding goals. Indeed, the realization that machines could respond to such high level statements as

Programming paradigms

TOPIC 112: IMPERATIVE PARADIGM

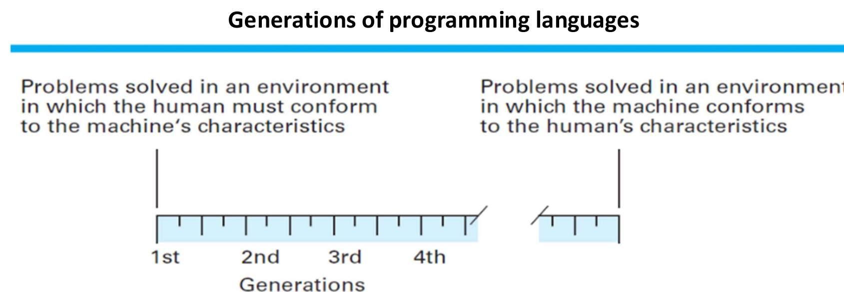
The generation approach to classifying programming languages is based on a linear scale on which a language's position is determined by the degree to which the user of the language is freed from the world of computer gibberish and allowed to think in terms associated with the problem being solved.

✓ **What are programming paradigms?**

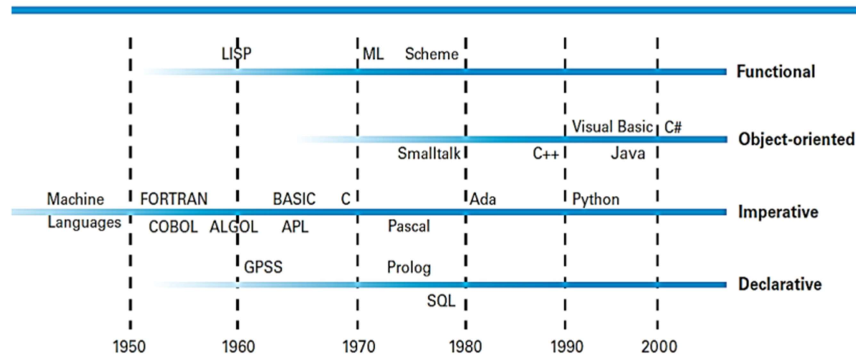
It is Fundamental style of computer programming. It serves as a pattern or model of a programming language.

In reality, the development of programming languages has not progressed in this manner but has developed along different paths as alternative approaches to the programming process (called programming paradigms) have surfaced and been pursued.

Generations of programming languages



Consequently, the historical development of programming languages is better represented by a multiple-track diagram, in which different paths resulting from different paradigms are shown to emerge and progress independently. In particular, the figure presents four paths representing the functional, object-oriented, imperative, and declarative paradigms, with various languages associated with each paradigm positioned in a manner that indicates their births relative to other languages. (It *does not imply* that one language necessarily evolved from a previous one.)



We should note that although the paradigms identified called *programming paradigms*, these alternatives have ramifications beyond the programming process. They represent fundamentally different approaches to building solutions to problems and therefore affect the entire software development process. In this sense, the term programming paradigm is a misnomer. A more realistic term would be software development paradigm.

✓ Imperative paradigm

The imperative paradigm, also known as the procedural paradigm, represents the traditional approach to the programming process. It is the paradigm on which Python and our pseudocode are based as well as the machine language. As the name suggests, the imperative paradigm defines the programming process to be the development of a sequence of commands that, when followed, manipulate data to produce the desired result. Thus, the imperative paradigm tells us to approach the programming process by finding an algorithm to solve the problem at hand and then expressing that algorithm as a sequence of commands.

- Develops a sequence of commands that when followed, manipulate data to produce the desired result
- Approaches a problem by trying to find an algorithm for solving it

TOPIC 113: DECLARATIVE PARADIGMS

✓ Declarative paradigm

In contrast to the imperative paradigm is the declarative paradigm, which asks a programmer to describe the problem to be solved (what) rather than an algorithm to be followed (How). More precisely, a declarative programming system applies a pre-established general-purpose problem-solving algorithm to solve problems presented to it. In such an environment the task of a programmer becomes that of developing a precise statement of the problem rather than of describing an algorithm for solving the problem.

- Emphasizes • "What is the problem?" • Rather than "What algorithm is required to solve the problem?
- Implemented a general problem-solving algorithm

✓ Obstacles

A major obstacle in developing programming systems based on the declarative paradigm is the need for an underlying problem-solving algorithm. Knowing the generic algorithm and then implementing it.

For this reason, early declarative programming languages tended to be special-purpose in nature, designed for use in particular applications or softwares. For example, the declarative approach has been used for many years to simulate a system (political, economic, environmental, and so on) in order to test hypotheses or to obtain predictions. In these settings, the underlying algorithm is essentially the process of simulating the passage of time by repeatedly re-computing values of parameters (gross domestic product, trade deficit, and so on) based on the previously computed values.

✓ Weather forecasteing

Thus, implementing a declarative language for such simulations requires that one first implement an algorithm that performs this repetitive function. Then the only task required of a programmer using the system is to describe the situation to be simulated. In this manner, a weather forecaster does not need to develop an algorithm for forecasting the weather but merely describes the current weather status, allowing the underlying simulation algorithm to produce weather predictions for the near future.

- ✓ **Logic programming**

A tremendous boost was given to the declarative paradigm with the discovery that the subject of formal logic within mathematics provides a simple problem- solving algorithm suitable for use in a general purpose declarative programming system. The result has been increased attention to the declarative paradigm and the emergence of logic programming.

TOPIC 114: FUNCTIONAL PARADIGM

Another programming paradigm is the functional paradigm. Under this paradigm a program is viewed as an entity that accepts inputs and produces outputs. Mathematicians refer to such entities as functions, which is the reason this approach is called the functional paradigm. Under this paradigm a is program constructed by connecting smaller predefined program units (predefined functions) so that each unit's outputs are used as another unit's inputs in such a way that the desired overall input-to-output relationship is obtained. In short, the programming process under the functional paradigm is that of building functions as nested complexes of simpler functions.

As an example, shows how a function for balancing your checkbook can be constructed from two simpler functions. One of these, called Find_sum, accepts values as its input and produces the sum of those values as its output. The other, called Find_diff, accepts two input values and computes their difference. The structure displayed in Figure 6.3 can be represented in the LISP programming language (a prominent functional programming language) by the expression

(Find_diff (Find_sum Old_balance Credits) (Find_sum Debits))

The nested structure of this expression (as indicated by parentheses) reflects the fact that the inputs to the function Find_diff are produced by two applications of Find_sum. The first application of Find_sum produces the result of adding all the Credits to the Old_balance. The second application of Find_sum computes the total of all Debits. Then, the function Find_diff uses these results to obtain the new checkbook balance.

To more fully understand the distinction between the functional and imperative paradigms, let us compare the functional program for balancing a checkbook to the following pseudocode program obtained by following the imperative paradigm:

Imperative program	Functional program
Total_credits = sum of all Credits Temp_balance = Old_balance + Total_credits Total_debits = sum of all Debits Balance = Temp_balance - Total_debits	Inputs: Old_balance Credits Debits Find_sum Find_sum Find_diff Output: New_balance

A function for checkbook balancing constructed from simpler functions

Note that this imperative program consists of multiple statements, each of which requests that a computation be performed and that the result be stored for later use.

In contrast, the functional program consists of a single statement in which the result of each computation is immediately channeled into the next. In a sense, the imperative program is analogous to a collection of factories, each converting its raw materials into products that are stored in warehouses. From these warehouses, the products are later shipped to other factories as they are needed. But the functional program is analogous to a collection of factories that are coordinated so that each produces only those products that are ordered by other factories and then immediately ships those products to their destinations without intermediate storage. This efficiency is one of the benefits proclaimed by proponents of the functional paradigm.

TOPIC 115: OBJECT ORIENTED PARADIGM

Object oriented paradigm

Another programming paradigm (and the most prominent one in today's software development) is the object-oriented paradigm, which is associated with the programming process called object-oriented programming (OOP). Following this paradigm, a software system is viewed as a collection of units, called objects, each of which is capable of performing the actions that are immediately related to itself as well as requesting actions of other objects. Together, these objects interact to solve the problem at hand.

- ✓ **Developing a GUI**

As an example of the object-oriented approach at work, consider the task of developing a graphical user interface. In an object-oriented environment, the icons that appear on the screen would be implemented as objects. Each of these objects would encompass a collection of functions (called methods in the object-oriented vernacular) describing how that object is to respond to the occurrence of various events, such as being selected by a click of the mouse button or being dragged across the screen by the mouse. Thus, the entire system would be constructed as a collection of objects, each of which knows how to respond to the events related to it.

- ✓ **Difference of List in imperative and OO-Paradigm**

To contrast the object-oriented paradigm with the imperative paradigm, consider a program involving a list of names. In the traditional imperative paradigm, this list would merely be a collection of data. Any program unit accessing the list would have to contain the algorithms for performing the required manipulations. In the object-oriented approach, however, the list would be constructed as an object that consisted of the list together with a collection of methods for manipulating the list.

You need your algorithm for searching, sorting etc in imperative, however, in OO-paradigm; all functionality will be available along with the list in terms of object

Its significance in today's software development arena dictates that we include the concept of a class in this introduction. To this end, recall that an object can consist of data (such as a list of names) together with a collection of methods for per- forming activities (such as inserting new names in the list). These features must be described by statements in the written program. This description of the object's properties is called a class. Once a class has been constructed, it can be applied anytime an object with those characteristics is needed. Thus, several objects can be based on (that is, built from) the same class. Just like identical twins, these objects would be distinct entities but would have the same characteristics because they are constructed from the same template (the same class). (An object that is based on a particular class is said to be an instance of that class.)

Composition of a Typical Imperative Program or Program Unit

TOPIC 116: VARIABLE AND DATA TYPES

High-level programming languages allow locations in main memory to be referenced by descriptive names rather than by numeric addresses. Such a name is known as a variable, in recognition of the fact that by changing the value stored at the location, the value associated with the name changes as the program

executes. These declarative statements also require that the programmer describe the type of data that will be stored at the memory location associated with the variable. Such a type is known as a data type.

- ✓ Data Types

1. Integer refers to numeric data consisting of whole numbers, probably stored using two's complement notation.
2. Float (sometimes called real) refers to numeric data that might contain values other than whole numbers, probably stored in floating-point notation. Operations performed on data of type float are similar to those performed on data of type integer.

```
int Height, Width;

int WeightLimit = 100;
```

3. Character refers to data consisting of symbols, probably stored using ASCII or Unicode. Operations performed on such data include comparisons such as determining whether one symbol occurs before another in alphabetical order, testing to see whether one string of symbols appears inside another, and concatenating one string of symbols at the end of another to form one long string. The statement

```
char Letter, Digit;
```

4. Boolean refers to data items that can take on only the values true or false. Operations on data of type Boolean include inquiries as to whether the current value is true or false. For example, if the variable LimitExceeded was declared to be of type Boolean, then a statement of the form if (LimitExceeded) then (...) else (...) would be reasonable.

The data types that are included as primitives in a programming language, such as int for integer and char for character, are called primitive data types. Data types such as: integer, float, character, and boolean are common primitives. Other data types that have not yet become widespread primitives include images, audio, video, and hypertext. However, types such as GIF, JPEG, and HTML might soon become as common as integer and float.

TOPIC 117: DATA STRUCTURE

In addition to data type, variables in a program are often associated with data structure, which is the conceptual shape or arrangement of data. For example:

✓ Name is a sequence of characters,

✓ Student marks in 5 subjects

One common data structure is the array, which is a block of elements of the same type such as a one dimensional list, a two-dimensional table with rows and columns, or tables with higher dimensions. To establish such an array in a program, many programming languages require that the declaration statement declaring the name of the array also specify the length of each dimension of the array. For example, displays the conceptual structure declared by the statement

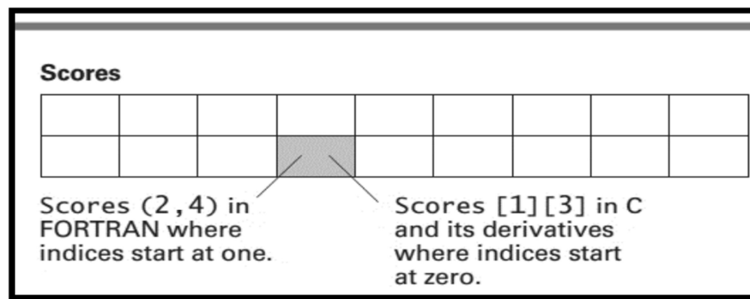
```
int Scores[2][9];
```

in the language C, which means "The variable Scores will be used in the following program unit to refer to a two-dimensional array of integers having two rows and nine columns." The same statement in FORTRAN would be written as

```
INTEGER Scores (2, 9)
```

Once an array has been declared, it can be referenced elsewhere in the program by its name, or an individual element can be identified by means of integer values called indices that specify the row, column, and so on, desired. However, the range of these indices varies from language to language. For example, in C (and its derivatives C++, Java, and C#) indices start at 0, meaning that the entry in the second row and

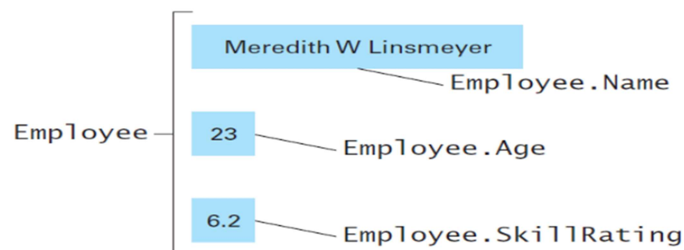
fourth column of the array called Scores (as declared above) would be referenced by Scores[1][3], and the entry in the first row and first column would be Scores[0][0]. In contrast, indices start at 1 in a FORTRAN program so the entry in the second row and fourth column would be referenced by Scores(2,4).



In contrast to an array in which all data items are the same type, an aggregate type (also called a structure, a record, or sometimes a heterogeneous array) is a block of data in which different elements can have different types. For instance, a block of data referring to an employee might consist of an entry called Name of type character, an entry called Age of type integer, and an entry called SkillRating of type float. Such an aggregate type would be declared in C by the statement

```
struct
{char Name[25];
int Age;
float SkillRating;
} Employee;
```

which says that the variable Employee is to refer to a structure (abbreviated struct) consisting of three components called Name (a string of 25 characters), Age, and SkillRating (Figure 95). Once such an aggregate has been declared, a programmer can use the structure name (Employee) to refer to the entire aggregate or can reference individual fields within the aggregate by means of the structure name followed by a period and the field name (such as Employee.Age).



TOPIC 118: ASSIGNMENT STATEMENT

Once the special terminology to be used in a program (such as the variables and constants) has been declared, a programmer can begin to describe the algorithms involved. This is done by means of imperative statements. The most basic imperative statement is the assignment statement, which requests that a value be assigned to a variable (or more precisely, stored in the memory area identified by the variable). Such a statement normally takes the syntactic form of a variable, followed by a symbol representing the assignment operation, and then by an expression indicating the value to be assigned. The semantics of such a statement is that the expression is to be evaluated and the result stored as the value of the variable. For example, the statement:

```
Z = X + Y;
```

in C, C++, C#, and Java requests that the sum of X and Y be assigned to the variable Z. The semicolon at the end of the line, which is used to separate statements in many imperative languages, is the only syntactic difference from an equivalent Python assignment statement. In some other languages (such as Ada) the equivalent statement would appear as:

Z := X + Y;

TOPIC 119: CONTROL STRUCTURES (IF-STATEMENT)

A control statement alters the execution sequence of the program. it determines whether other statements will be executed or not.

```
if (condition)
Statement A
else
Statement B
```

For example, if a student's gets more than or equal to 50 marks, the student passes the examination. To denote this using the if-statement, we will proceed as follow:

```
if (marks>=50)
You have passed the examination
Else
You have failed the examination
```

The result is the practice known as structured programming, which encompasses an organized design methodology combined with the appropriate use of the language's control statements.

TOPIC 120: CONTROL STRUCTURES (IF-STATEMENT EXAMPLES)

In this module, we will learn another example. Suppose a university wants to give a scholarship, if a student gets more than 3.0 CGPA in a given semester. We can write the program in the following way and you have also seen its implementation in the videos in the online compiler.

```
float CGPA=3.5;
If (CGPA>=3.0)
        cout<<"Give Scholarship";
else
        cout<<"Sorry you do not qualify for the scholarship";
```

TOPIC 121: CONTROL STRUCTURES (LOOPS)

There is another type of control structure known as loop. The loop control structure iterates a set of instructions based on the provided condition.

Syntax:
```
while (condition)
{loop body}
```

Consider the following example; we are interested to print the counting from 1 to 5. One way of doing this is as follows:

```
cout<<"1";
cout<<"2";
cout<<"3";
cout<<"4";
cout<<"5";
```

Doing this using loop is easier, you just need to give one statement that is printing counting and you need to tell that how many times, you want to do it. One can write the loop as follows:

```
int i=1;
while (i<=5)
```

```
{
cout<<i;
i=i+1;
}
```

This loop will execute 5 times as follows:

First the value of "i" would be 1. The condition would be checked and based on the condition as true (i<=5 i.e 1<=5), the loop will continue and will print the value of "i" which is 1. Then it will increment the value of "i". The new value of "i" would be 2. Then again condition would be checked, and the condition would be true as 2<=5. This will print "2" on the screen and will continue for "i" as 3, 4, and 5, and will print 3, 4, 5 on the screen as well. When the value of "i" would be 6, the loop will find the condition as false (as 6<=5). This would be the termination for the loop.

TOPIC 122: PROGRAMMING CONCURRENT ACTIVITIES

- ✓ **Concurrent Processing**

Simultaneous execution of multiple activations is called **parallel processing** or **concurrent processing**.

Tasks are broken down into subtasks that are then **assigned to separate processors to perform simultaneously**, instead of sequentially as they would have to be carried out by a single processor.

- ✓ **Scenario**

Suppose you have been asked to **produce animation for an action computer game**

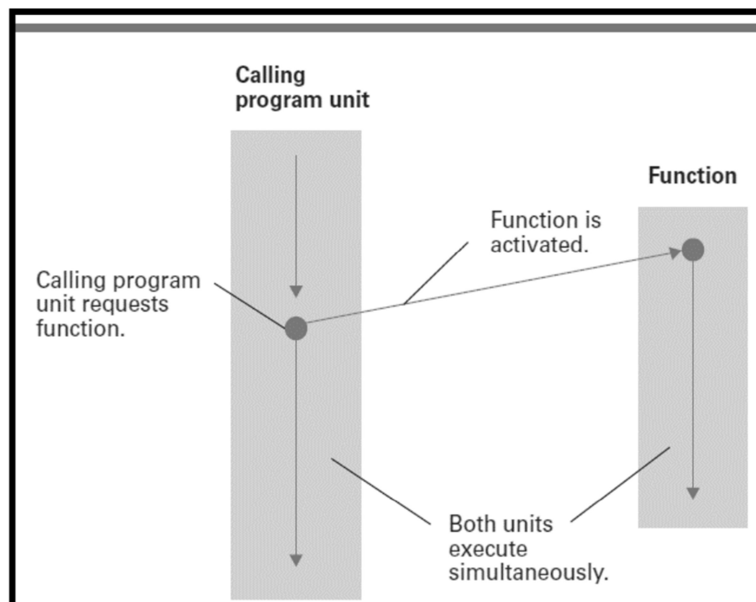
- ✓ **True Parallel Processing**
  - Possible when multiple CPU core process each activation
  - When one CPU, illusion can be created using multiprogramming systems.
- ✓ **Activation**

Different methodologies and naming conventions in different programming languages:

- Task in Ada
- Thread in Java

- ✓ **Procedure**

Creating main program which creates activations/threads/tasks



A more complex issue associated with parallel processing involves handling communication between threads. For instance, in our spaceship example, the threads representing the different spaceships might need to communicate their locations among themselves in order to coordinate their activities. In other cases, one thread might need to wait until another reaches a certain point in its computation, or one thread might need to stop another one until the first has accomplished a particular task.

Such communication needs have long been a topic of study among computer scientists, and many newer programming languages reflect various approaches to thread interaction problems. As an example, let us consider the communication problems encountered when two threads manipulate the same data. If each of two threads that are executing concurrently need to add the value three to a common item of data, a method is needed to ensure that one thread is allowed to complete its transaction before the other is allowed to perform its task. Otherwise they could both start their individual computations with the same initial value, which would mean that the final result would be incremented by only three rather than six. Data that can be accessed by only one thread at a time is said to have mutually exclusive access.

TOPIC 123: ARITHMETIC OPERATORS EXAMPLES

C-language has the following arithmetic operators:

Operator	Meaning
+	Addition
-	Subtraction
*	Multiplication
/	Division
%	Modulus

+, -, and * are the same as used in the mathematics. However, the "/" has a difference. If one of the operands is decimal number, then it results in the same way as in mathematics, for example:

5.0/2.0 would result into 2.5.

However, when both operands are integers, then it would truncate the decimal point and

5/2 would result into 2.

The remaining "1" can be acquired by using the modulus operator (%).

5%2 would give 1.

TOPIC 124: RELATIONAL OPERATORS EXAMPLES

C-language has the following relational operators:

Operator	Meaning
<	Less than
<=	Less than or equal to
>	Greater than
>=	Greater than or equal to
= =	Equal to
!=	Not Equal to

C++ Relational Operators are used to compare values of two variables. Here in example we used the operators in if statement.

Now if the result after comparison of two variables is True, then if statement returns value 1. And if the result after comparison of two variables is False, then if statement returns value 0.

```
#include<iostream>
using namespace std;
void main()
{
int a=10, b=20, c=10;
if(a>b)
cout<<"a is greater"<<endl;
if(a<b)
cout<<"a is smaller"<<endl;
if(a<=c)
cout<<"a is less than/equal to c"<<endl;
if(a>=c)
cout<<"a is greater than/equal to c"<<endl;
}
```

Output

```
a is smaller
a is less than/equal to c
a is greater than/equal to c
```

In C++ Relational operators, two operators that is = = (Is Equal to) and != (is Not Equal To), are used to check whether the two variables to be compared are equal or not.

Let us take one example which demonstrates these two operators.

```
#include<iostream>
using namespace std;
void main()
{
int num1 = 30;
int num2 = 40;
int num3 = 40;
if(num1!=num2)
cout<<"num1 Is Not Equal To num2"<<endl;
if(num2==num3)
cout<<"num2 Is Equal To num3"<<endl;
}
```

Output

```
num1 Is Not Equal To num2
num2 Is Equal To num3
```

TOPIC 125: LOGICAL OPERATORS EXAMPLES

1. C-language Logical Operators are used if we want to compare more than one condition.

2. Depending upon the requirement, proper logical operator is used.

3. Following table shows us the different C++ operators available

Operators	Operators	Operators
&&	AND Operator	Binary
||	OR Operator	Binary
!	NOT Operator	Unary

According to names of the Logical Operators, the condition satisfied in following situation and expected outputs are given

Operator	Output
AND	Output is 1 only when conditions on both sides of Operator become True

OR	Output is 0 only when conditions on both sides of Operator become False
NOT	It gives inverted Output

Let us look at all logical operators with example-

```
#include<iostream>
using namespace std;
int main()
{
int num1=30;
int num2=40;

if(num1>=40 || num2>=40)
        cout<<"OR If Block Gets Executed"<<endl;
if(num1>=20 && num2>=20)
        cout<<"AND If Block Gets Executed"<<endl;
if(!(num1>=40))
        cout<<"NOT If Block Gets Executed"<<endl;
return 0;
}
```

Output:

```
OR If Block Gets Executed
AND If Block Gets Executed
NOT If Block Gets Executed
```

Explanation of the Program

Or statement gives output = 1 when any of the two condition is satisfied.

if(num1>40 || num2>=40)

Here in above program , num2=40. So, one of the two conditions is satisfied. So, statement is executed.

For AND operator, output is 1 only when both conditions are satisfied.

if(num1>=20 && num2>=20)

Thus, in above program, both the conditions are True so if block gets executed.

Truth Table

Operator	1st Condition	2nd Condition	Output
AND	True	True	True
	True	False	False
	False	True	False
	False	False	False
OR	True	True	True
	True	False	True
	False	True	True
	False	False	False
NOT	True	-	False
	False	-	True

Chapter 7: Software Engineering

Topic 126 to 145

TOPIC 126: SOFTWARE ENGINEERING DISCIPLINE

In this chapter we explore the problems that are encountered during the development of large, complex software systems. The subject is called software engineering because software development is an engineering process. The goal of researchers in software engineering is to find principles that guide the software development process and lead to efficient, reliable software products.

Software engineering is the branch of computer science that seeks principles to guide the development of large, complex software systems. It is concerned with all the aspects of software production.

- ✓ **Scenario**

To understand the problems involved in software engineering think of constructing a Commercial building which has 5 Floors and it size is 50 feet by 90 feet.

- **Estimation**
  - **How can you estimate the cost in time, money, and other resources to complete the project?**
  - **How can you divide the project into manageable pieces?**
  - **How can you ensure that the pieces produced are compatible?**
  - **How can those working on the various pieces communicate? How can you measure progress?**
- **Wrong Estimation!**
  - **Could lead to cost overruns**
  - **Late delivery of products.**
  - **Dissatisfied customers**
- ✓ **Differences b/w Software Engineering and Engineering**
- **Traditional fields of engineering have long benefited from the ability to use "off-the-shelf" components as building blocks when constructing complex devices. Software engineering lags in this regard.**
- **Another distinction between *software engineering and other engineering disciplines* is the lack of quantitative techniques, called metrics, for measuring the properties of software. Methods for measuring the "complexity" of software are evasive.**
- **Similarly, if we talk about quality measures. Software does not wear out, so this method of measuring quality is not as applicable in software engineering.**
- ✓ **SE Progress**

Software engineering is currently progressing on two levels: Some researchers, sometimes called practitioners, work toward developing techniques for immediate application, whereas others, called theoreticians, search for underlying principles and theories on which more stable techniques can someday be constructed

- ✓ **Computer-aided SE**

Computer-aided software engineering(CASE), is continuing to streamline and otherwise simplify the software development process.

- ✓ **CASE tools**

CASE has led to the development of a variety of computerized systems, known as CASE tools, which include project planning systems (to assist in cost estimation, project scheduling, and personnel allocation), project management systems (to assist in monitoring the progress of the development project), documentation tools (to assist in writing and organizing documentation), prototyping and simulation systems (to assist in

the development of prototypes), interface design systems (to assist in the development of GUIs), and programming systems(to assist in writing and debugging programs).

- ✓ Integrated development environments (IDEs)

Sophisticated packages designed primarily for the software engineering environment are known as IDEs

Integrated development environments (IDEs) combine tools for developing software (editors, compilers, debugging tools, and so on) into a single, integrated package. Prime examples of such systems are those for developing applications for smartphones.

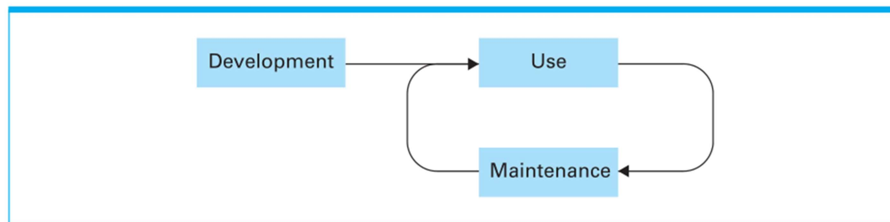
TOPIC 127: SOFTWARE LIFE CYCLE

The most fundamental concept in software engineering is the software life cycle.

- ✓ The Cycle as a Whole

The software life cycle represents the fact that once software is developed, it enters a cycle of being used and maintained—a cycle that continues for the rest of the software's life. Software moves into the maintenance stage because errors are discovered, changes in the software's application occur, or changes done in previous modification induce the errors. (MCQ Not repair process)

The software life cycle



- ✓ Maintenance v/s repair

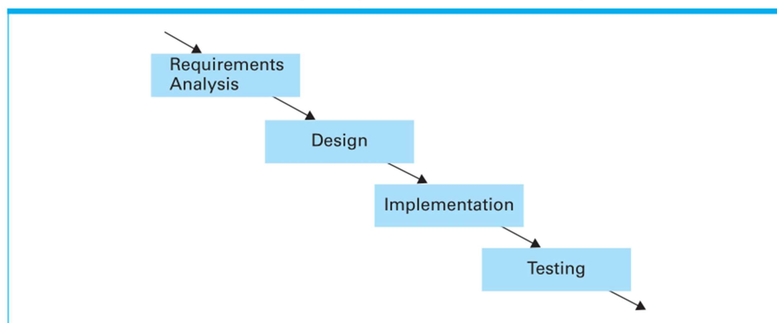
In the case of other products, the maintenance phase tends to be a repair process, whereas in the case of software, the maintenance phase tends to consist of correcting or updating.

The Traditional Development Phase

The major steps in the traditional software development life cycle are requirements analysis, design, implementation, and testing. (Not Maintenance)

Here design phase is followed by implementation. Second phase of Software development life cycle is design.

The traditional development phase of the software life cycle



Requirement Analysis

- Goals
- Inputs
- COTS
- Analysis
- SRS

TOPIC 128: REQUIREMENT ANALYSIS PHASE

- ✓ Goals of Requirement analysis

The software life cycle begins with requirements analysis—the goal of which is:

- To specify what services the proposed system will provide
- To identify any conditions (time constraints, security, and so on) on those services
- To define how the outside world will interact with the system.

✓ Inputs from Stakeholders

Future users as well as those with other ties, such as legal or financial interests

In fact, in cases where the ultimate user is an entity, such as a company or government agency, that intends to hire a software developer for the actual execution.

✓ Commercial off-the-shelf (COTS) software

- Software developed for the mass market, perhaps to be sold in retail stores or downloaded via the Internet.
- In this setting the user is a less precisely defined entity and requirements analysis may begin with a market study by the software developer. (Example MS Office)

✓ Requirement Analysis Process

In any case, the requirements analysis process consists of

- Compiling and analyzing the needs of the software user
- Negotiating with project stakeholders on trade-offs between wants, needs, cost, and feasibility
- Finally developing a set of requirements that identify the features and services that the finished software system must have

✓ Software Requirement Specification (SRS)

- These requirements are recorded in a document called a software requirements specification.
- This document is a written agreement between all parties concerned

SRS is intended to guide the software's development and provide a means of resolving disputes that may arise later in the development process. From the software developer's perspective, the software requirements specification should define a firm objective toward which the software's development can proceed.

TOPIC 129: DESIGN PHASE

✓ RA vs Design

- Requirements analysis provides a description of the proposed software product; design involves creating a plan for the construction of the proposed system.
- Requirements analysis is about identifying the problem to be solved, while design is about developing a solution to the problem.
- Requirements analysis decides what a system will do, and design identifies how the system will do it

✓ Design Outcome

- Internal structure of the software system is established.
- Detailed description of the software system's structure that can be converted into programs

✓ Office building example

- The design stage would consist of developing detailed structural plans for a building
- Making designs that can be converted into programs.
- Blueprints describing the proposed building at various levels of detail
- In Software design: notational system and many modeling and diagramming methodologies

TOPIC 130: IMPLEMENTATION PHASE

✓ Goal

- Implementation involves the actual writing of programs, creation of data files, and development of databases
- Building construction analogy

- ✓ **Software Analyst vs programmer**

It is at the implementation stage that distinct between the tasks of a software analyst (sometimes referred to as a system analyst) and a programmer.

System analyst is a person involved with the entire development process, perhaps with an emphasis on the requirements analysis and design steps. The programmer is a person involved primarily with the implementation step. In its narrowest interpretation, a programmer is charged with writing programs that implement the design produced by a software analyst.

TOPIC 131: TESTING PHASE

- ✓ **Testing in Traditional software development**
  Testing remained to be considered just to debug programs and confirming that the software is compatible to SRS.
- ✓ **Modern Testing**
- **Programs are not the only artifacts that are tested during the software development process. Indeed, the result of each intermediate step in the entire development process should be "tested" for accuracy.**
- **Testing should not be considered as separate step: Indeed it should be incorporated into the other steps, producing a three-step development process whose components might have names such as**
  1. **Requirements analysis and confirmation**
  2. **Design and validation**
  3. **Implementation and testing**
- ✓ **Need for better testing methodologies!**
  - **Unfortunately, even with modern quality assurance techniques, large software systems continue to contain errors, even after significant testing.**
  - **Many of these errors may go undetected for the life of the system**
  - **Others may cause major malfunctions**

TOPIC 132: SOFTWARE ENGINEERING METHODOLOGIES-1

SDLC is the acronym of Software Development Life Cycle. For the best software development model for the project, in System Analysis phase the developers decide a roadmap for project plan

- ✓ **Water Fall Model**
  - **An analogy to the fact that the development process was allowed to flow in only one direction is called waterfall model.**
  - **Entire preceding phase need to be completely done before the start of next phase.**
- ✓ **Incremental Model**
  - **Desired software system is constructed in increments--the first being a simplified version of the final product with limited functionality.**
  - **Once this version has been tested and perhaps evaluated by the future user, more features are added and tested in an incremental manner until the system is complete.**
  - **For example, if the system being developed is a patient records system for a hospital, the first increment may incorporate only the ability to view patient records from a small sample of the entire record system. Once that version is operational, additional features, such as the ability to add and update records, would be added in a stepwise manner.**
  - **Other Example: Student add, view, course addition**
- ✓ **Iterative Model**
  - **A model which represents the shift away from strict adherence to the waterfall model is the iterative model**
  - **It is similar to, and in fact sometimes equated with, the incremental model, although the two are distinct**
  - **Incremental model carries the notion of extending each preliminary version of a product into a larger version; the iterative model encompasses the concept of refining each version creating a working prototype first.**
    - o **Rational Unified Process**
    - ❖ **RUP is an example of iterative techniques, created by Rational Software Corporation, now owned by IBM.**
    - ❖ **Redefines the steps in the development phase of the software life cycle**

- RUP is widely applied now a days in software industry.
- Its Non-proprietary version is Unified Process that is available on a noncommercial basis.

TOPIC 133: SOFTWARE ENGINEERING METHODOLOGIES-2

- ✓ **Prototype**
  - Incomplete version of the proposed system, called prototypes, are built and evaluated in Iterative and Incremental models.
- ✓ **Paper – Prototype**
  - Paper Prototyping is a prototyping method in which paper models are used to simulate computer or web applications.

- ✓ **Evolutionary Prototyping**
  - In the case of the incremental model, initial prototypes evolve into the complete, final system. This process is called evolutionary prototyping.
- ✓ **Throwaway Prototyping**
  - In a more iterative situation, the prototypes may be discarded in favor of a fresh implementation of the final design. This approach is known as throwaway prototyping.
- ✓ **Rapid Prototyping**
  - Simple example of the proposed system is quickly constructed in the early stages of development.
  - Demonstration version
- ✓ **Open Source Development**
  - A less formal incarnation of incremental and iterative ideas that has been used for years by computer enthusiasts/hobbyists is known as open-source development
  - Purpose is to produce the free software.
  - A single author writes the initial version
  - Source code and documentation is shared via internet where others can contribute.
  - Example: Linux Operating system
- ✓ **Agile methods**
  - The most pronounced shift from the waterfall model is represented by the collection of methodologies known as agile methods. Each of which proposes early and quick implementation on an incremental basis
- Extreme Programming
  - One example of an agile method is extreme programming (XP).
  - Software is developed by a team of less than a dozen individuals working in a communal work space where they freely share ideas and assist each other in the development project.by means of repeated daily cycles and helping each other.
  - Can be evaluated by project stakeholders, at different stages

TOPIC 134: MODULAR IMPLEMENTATION

- ✓ **Modularity**
  - A way of producing manageable software
  - To modify software, one should understand the software, difficult in small programs and nearly impossible for large software.

- Module wise implementation – division of software into manageable units, generically called modules, each of which deals with only a part of the software's overall responsibility.

✓ What are modules?

Modules come in a variety of forms. We have already seen (Chapters 5 and 6), that in the context of the imperative paradigm, modules appear as procedures. In contrast, the object-oriented paradigm uses objects as the basic modular constituents

- Imperative programming paradigm – functions
- Object oriented paradigm – Objects

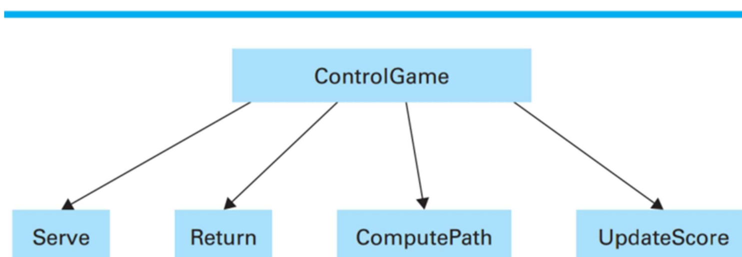
✓ Example – imperative Paradigm : to simulate a tennis game

✓ Serve () – Speed, Direction, players characteristics

✓ ComputePath () – hit net, where it bounce,

✓ Return () – will it be returned, next speed, direction

✓ UpdateScore ()



In above Figure, in which procedures are represented by rectangles and procedure dependencies (implemented by procedure calls) are represented by arrows. In particular, the chart indicates that the entire game is overseen by a procedure named ControlGame, and to perform its task, ControlGame calls on the services of the procedures Serve, Return, ComputePath, and UpdateScore.

TOPIC 135: COUPLING

✓ Modularity advantage

- Any modification will be applied to few of the modules
- Assumption: Changes in one module will not affect other modules.
- Goal: maximize the independence, minimize linkage between modules known as: intermodule coupling

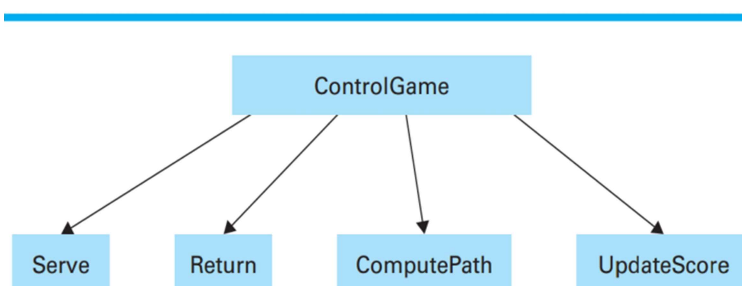
✓ Intermodule Coupling

Indeed, one metric to measure the complexity if a software is to measure the intermodule coupling

Intermodule coupling occurs in several forms

○ Control Coupling

Control coupling occurs when a module passes control of execution to another, as in a function call (e.g., as in a function call002E)



- **Data Coupling**

Refers to the sharing of data between modules. In data coupling if two modules interact with the same item of data, then modifications made to one module may affect the other.

Data coupling between procedures can occur in two forms.

- **Explicit passing**

Explicitly passing data from one procedure to another in the form of parameters. Such coupling is represented in a structure chart by an arrow between the procedures that is labeled to indicate the data being passed. The direction of the arrow indicates the direction in which the item is transferred.

Passing by parameters: ControlGame will tell the function Serve () which player's characteristics are to be simulated when it calls Serve and that the function Serve () will report the ball trajectory to ControlGame when Serve () has completed its task.

- **Global data**

Data items that are automatically available to all modules throughout the system as opposed to local data items that are accessible only within a particular module.

TOPIC 136: COHESION

Just as important as minimizing the coupling between modules is maximizing the internal binding within each module. The term cohesion refers to this internal binding or, in other words, the degree of relatedness of a module's internal parts. The term cohesion refers to this Compound cohesion, simple cohesion.

- ✓ **Goal**
  - **Low intermodule coupling**
  - **High intra module cohesion**
- ✓ **Logical Cohesion**
  - **A weak form of cohesion is known as logical cohesion**
  - **This is the cohesion within a module induced by the fact that its internal elements perform activities logically similar in nature. For example Communication module – obtaining data and reporting results or grouping ll mouse and keyboard handling functions.**
- ✓ **Functional Cohesion**
  - **A stronger form of cohesion is known as functional cohesion**
  - **All the parts of the module are focused on the performance of a single activity results.**
  - It focuses on exactly one goal or function.
  - **Can be increased by isolating subtasks in other modules and then using these modules as abstract tools**
  - **For example, a module that assigns seats to airline passenger**

TOPIC 137: INFORMATION HIDING

One of the cornerstones of good modular design is captured in the concept of information hiding.

- **Information hiding refers to the restriction of information to a specific portion of a software system.**
- ***Information* should be interpreted in a broad sense such as: data, the type of data structures used, encoding systems, the internal compositional structure of a module etc**
- ✓ **Why we need it**
  - **We use information hiding to reduce unnecessary dependencies or effects on other modules.**
  - **For example, a module does not restrict the use of its internal data from other modules, then that data may become corrupted by other modules**
- ✓ **Realization**

Information hiding has two incarnations

- **Design Goal**
- **Implementation Goal**
- ✓ **Design Goal**
  - **A module should be designed so that other modules do not need access to its internal information**

- Example: maximizing cohesion and minimizing coupling

✓ Implementation Goal

- A module should be implemented in a manner that reinforces its boundaries.
- Examples: use of local variables, applying encapsulation, and using well defined control structures

Finally we should note that information hiding is central to the theme of abstraction and the use of abstract tools. Indeed, the concept of an abstract tool is that of a "black box" whose interior features can be ignored by its user, allowing the user to concentrate on the larger application at hand.

TOPIC 138: COMPONENTS

✓ Why we need it?

- We should build off-the-shelf building blocks from which large software can be constructed.
- In imperative paradigm, Modular approach only promises hope in this regard.
- Object-oriented paradigm is helping as objects form complete, self-contained units that have clearly defined interfaces with their environments.
- Once an object, or more correctly a class, has been designed to fulfill a certain role, it can be used to fulfill that role in any program requiring that service.

✓ Prefabricated templates to realize the concept of components

The object-oriented programming languages C++, Java, and C# are accompanied by collections of prefabricated "templates" from which programmers can easily implement objects for performing certain roles. Using these templates you can build the software very quickly.

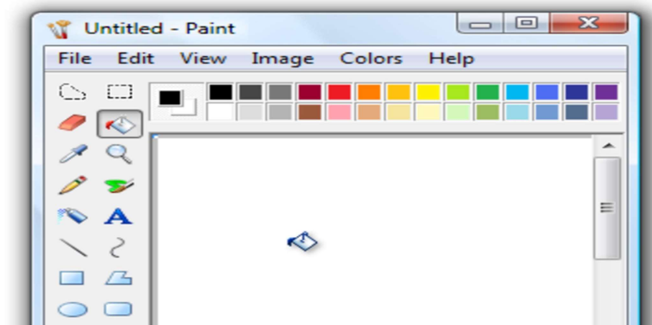
- In particular C++ has standard template library
- Java has Java Application Programming Interface (API)
- C# programmers have access to .NET Framework Class Library

✓ Components are not just objects!

An object is actually a special case of the more general concept of a component, which is, by definition, a reusable unit of software. In practice, most components are based on the object-oriented paradigm and take the form of a collection of one or more objects that function as a self-contained unit.

✓ Component Architecture

- Also known as (component-based software engineering) in which the traditional role of a programmer is replaced by a component assembler who constructs software systems from prefabricated components that, in many development environments, icons are displayed in a graphical interface
- The methodology of a component assembler is to select pertinent components from collections of predefined components and then connect them, with minimal customization, to obtain the desired functionality.



✓ More Examples

- Facebook when executed on a smartphone may use the components of the contact application to add all Facebook friends as contacts.
- The telephony application, may also access the contact components to lookup the caller of an incoming call

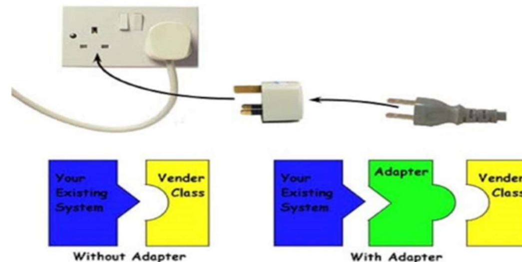
TOPIC-139. DESIGN PATTERNS

✓ **Why we need it?**

- **A design pattern is a pre-developed model for solving a recurring problem in software design**
- **You are not the first one working on the problem! The best possible model use by previous programmer can be consider as a design pattern (guidelines) which is already authenticated and test.**

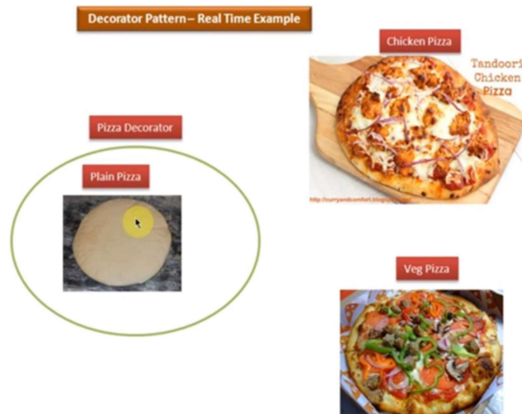
1. Adapter Pattern (Example of design pattern)

- **A prefabricated module may have all the functionality needed to solve the problem at hand but may not have an interface that is compatible with the current application.**
- **Adapter pattern provides a standard approach to "wrapping" that module inside another module**



2. Decorator Pattern

- **It provides a means of designing a system that performs different combinations of the same activities depending on the situation at the time.**
- **Such systems can lead to an explosion of options that, without careful design, can result in enormously complex software.**
- **Decorator pattern provides a standardized way of implementing such systems that leads to a manageable solution**



✓ **Goal of design pattern**

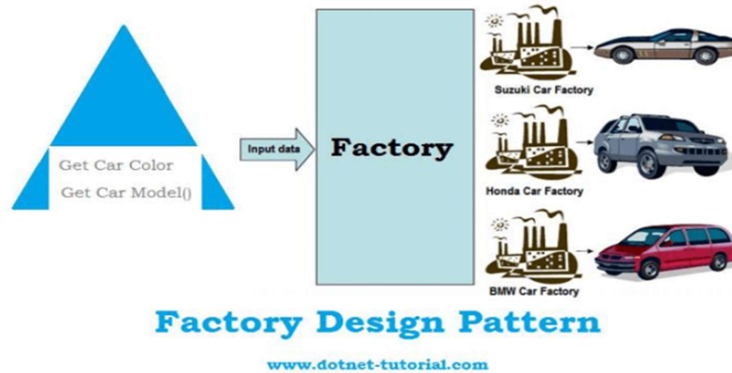
- **Identification of recurring problem, creation and cataloging of design patterns for solving them is an ongoing process in software engineering and making them available to other community.**
- **Goal is not to identify the solution; the goal is to identity the best solution flexible for future changes!**

TOPIC-140. DESIGN PATTERNS EXAMPLES

✓ **Factory Design Pattern**

The Factory Design Pattern is a commonly used design pattern where we need to create Loosely Coupled System

Factory Pattern is based on real time factory concept. As we know, a factory is used to manufacture something as per the requirement and if new items are added in the manufacturing process, the factory starts manufacturing those items as well. The factory has already created cars that have loosely coupled system, the engine, tire, lights. They just need to connect the things and present it to the user.

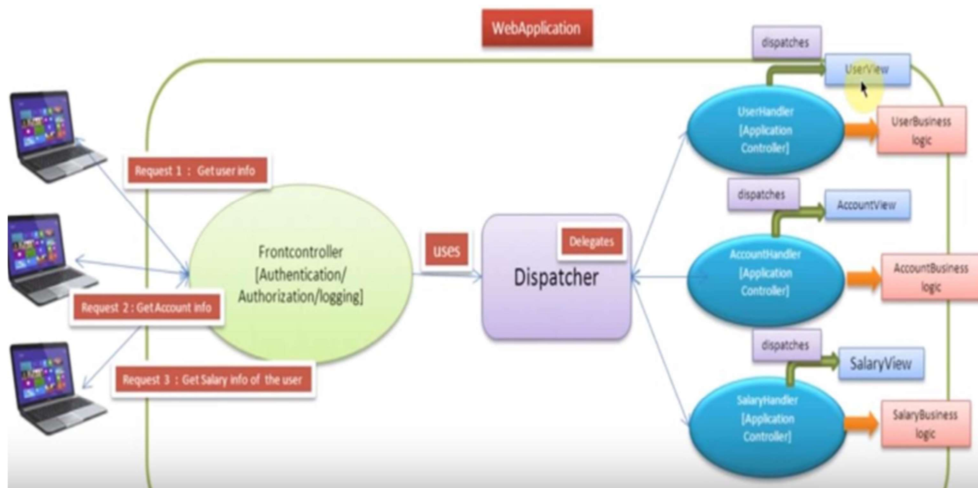


- ✓ **Front Controller Pattern**

- **Centralized request handling mechanism**

If these is a user who want user information, front controller will connect it to user handler module

When you design software the front controller pattern says that do not connect user directly to dis-handler. There should be front controller handler.



- ✓ **Shopping cart design pattern**

Usage when:

- **To buy more than one product.**
- **To buy more than one instance of a product.**
- **Want to return later to carry shopping.**
- **Want to return later for payment.**
- **Do not use when only one product to sale.**
- **Do not use when only one product can be sold.**
- **Do not use when your site is arranged in a way, so that it does not make sense for the user to buy more than one product at a time (for instance for Application Service Providers (ASPs) allowing a user to upgrade his service).**

Check out

This cart qualifies for Free Shipping

Cart Items	Quantity	Item Price	Item Total
Apple wireless Mighty Mouse Part Number: MB111LL/A Gift message: Add Estimated Ship: Within 24 hours	1 Remove	$69.00	$69.00
MacBook Pro, 17-inch, 2.4GHz Part Number: MA897LL/A 2GB 667 DDR2 - 2x1GB SO-DIMMs 160GB Serial ATA Drive @ 5400 rpm SuperDrive 8x (DVD±R DL/DVD±RW/CD-RW) Accessory Kit Backlit Keyboard/Mac OS - U.S. English MacBook Pro 17-inch Widescreen Display 2.4GHz Intel Core 2 Duo Gift message: Add Estimated Ship: Within 24 hours	1 Remove	$2,799.00	$2,799.00

Cart Subtotal: $2,868.00
Free Shipping: $0.00

Estimated Total: $2,868.00

Enter shipping ZIP to calculate tax: 1913

Update Subtotal

Continue shopping ▸ Save for later ▸ Sign up for 1-Click Check out

Saved Carts

TOPIC 141: SCOPE OF QUALITY ASSURANCE

✓ **Early years of Computing**

In the early years of computing, the problem of producing quality software focused on identifying and removing programming errors that occurred during implementation.

✓ **Contemporary scope**

Today, the scope of software quality control extends far beyond the debugging process, with branches including

- **Improvement of software engineering procedures**
- **Development of training programs that in many cases lead to certification**
- **Establishment of standards on which sound software engineering can be based**

✓ **Quality Standards**

In this regard, we have already noted the role of organizations such a

- **ISO, IEEE, and ACM establishing standards for assessing quality control within software development companies**
- **A specific example is ISO 9000 series of standards, which address numerous industrial activities such as design, production, installation, and servicing.**

✓ **Software Quality Assurance (SQA)**

- **Software development companies are establishing software quality assurance (SQA) groups, which are charged with overseeing and enforcing the quality control systems adopted by the organization**
- **Software contractors now require that the organizations meet such standards. They have then SQA groups.**
- **Example, in waterfall model, SQA approves SRS before design can start or approves design before implementation starts.**

TOPIC 142: SOFTWARE TESTING METHODOLOGIES

✓ **SQA and Testing**

SQA is a now recognized as a subject dealing entire development process, testing is concerned about the developed software/programs

✓ **Can we test all possibilities?**

- In simple programs, there may be billions of different paths that could potentially be traversed.
- Testing all paths in a complex program is nearly impossible.

What to do then?

✓ **Pareto Principle**

Economist and sociologist Vilfredo Pareto (1848–1923) invented that small part of Italy's population controlled most of Italy's wealth

- Pareto Principle in Software Engineering
  - Small number of modules within a large software system tends to be more problematic than the rest
  - Results can often be increased most rapidly by applying efforts in a concentrated area

✓ **Basis Path Testing**

Basis Path Testing is to develop a set of test data that insures that each instruction in the software is executed at least once.

✓ **Glass-box testing**

Glass-box testing, meaning that the software tester is aware of the interior structure of the software and uses this knowledge

✓ **Black-box Testing**

- Black-box testing refers to tests that do not rely on knowledge of the software's interior composition.
- Black-box testing is a method of software testing that examines the functionality of an application without peering into its internal structures or workings.
- Black-box testing is performed from the user's point of view. It is concerned with whether the software performs correctly in terms of accuracy and timeliness

- Boundary value analysis (Example of black-box testing)

- It consists of Identifying ranges of data, called equivalence classes, over which the software should perform in a similar manner and then testing the software on data close to the edge of those ranges.
- For example, if the software is supposed to accept input values within a specified range, then the software would be tested at the lowest and highest values in that range

- Beta Testing (Black-box testing methodology)

- In which a Preliminary version of the software is given to a segment of the intended audience with the goal of learning how the software performs in real-life situations before the final version of the product is released to the market

✓ **Alpha Testing**

- Similar testing performed at the developer's site is called alpha testing. (To identify bugs).

TOPIC 143: SOFTWARE DOCUMENTATION

✓ **Why we need this?**

A software system is of little use unless people can learn to use and maintain it.

Software documentation serves three purposes, leading to three categories of documentation:

- User documentation
- System documentation
- Technical documentation

✓ **User Documentation**

- The purpose of user documentation is to explain the features of the software and describe how to use them.
- It is intended to be read by the user of the software and is therefore expressed in the terminology of the application documentation
- User documentation is recognized as an important marketing tool
- Good user documentation along with good GUI increases sales
- Many software developers hire technical writers to produce this part of their product

✓ **System Documentation**

- The purpose of the System Documentation is to describe the software's internal composition so that the software can be maintained later in its life cycle.
- A major component of system documentation is the source version of all the programs in the system.
    - Commenting
    - Indentation
    - Naming Conventions
- Another component of system documentation is a record of the design documents including the software requirements specification and records showing how these specifications were obtained during design.

✓ **Technical Documentation**

The purpose of technical documentation is to describe how a software system should be installed and serviced

- Adjusting operating parameters
- Installing updates
- Reporting problems back to the software's developer

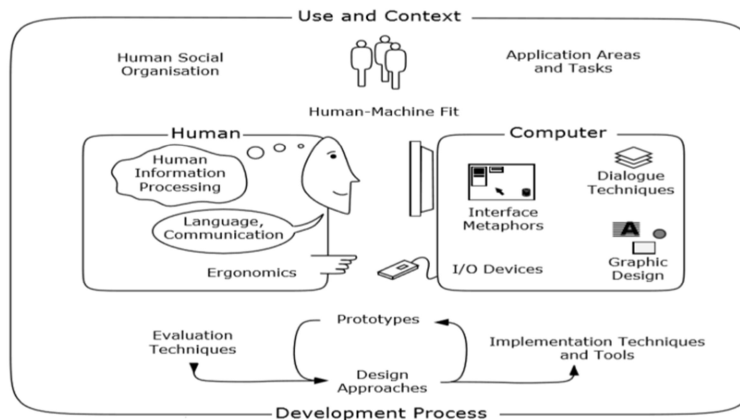
TOPIC 144: HUMAN MACHINE INTERFACE

One of the tasks during requirements analysis is to define how the proposed software system will interact with its environment. In this section we consider topics associated with this interaction when it involves communicating with humans. Humans should be allowed to use a software system as an abstract tool. This tool should be easy to apply and designed to minimize (ideally eliminate) communication errors between the system and its human users. This means that the system's interface should be designed for the convenience of humans rather than merely the expediency of the software system.

✓ **Why it is important?**

- A good system's interface is likely to make a stronger impression on a user than any other system characteristic
- User is not interested to learn the inside of the software.

  Thus, the design of a system's interface can ultimately be the determining factor in the success or failure of a software engineering project.



✓ **Ergonomics and Cognetics**

Research in human-machine interface design draws heavily from the areas of engineering called ergonomics

✓ Ergonomics deals with designing systems that harmonize with the physical abilities of humans

✓ Cognetics deals with designing systems that harmonize with the mental abilities of humans

Of the two, ergonomics is the better understood, largely because humans have been interacting physically with machines for centuries. Examples are found in ancient tools, weaponry, and transportation systems.

Another human characteristic that concerns researchers in human-machine interface design is the narrowness of a human's attention, which tends to become more focused as the level of concentration increases.

Still another human characteristic that must be anticipated during interface design is the mind's limited capacity to deal with multiple facts simultaneously. In an article in Psychological Review in 1956, George A. Miller reported research indicating that the human mind is capable of dealing with only about seven details at once.

The GOMS (rhymes with "Toms") model, originally introduced in 1954, is representative of the search for metrics in the field of human-machine interface design. The model's underlying methodology is to analyze tasks in terms of user goals (such as delete a word from a text), operators (such as click the mouse button), methods (such as double-click the mouse button and press the delete key), and selection rules (such as choose between two methods of accomplishing the same goal). This, in fact, is the origin of the acronym GOMS—goals, operators, methods, and selection rules.

In short, GOMS is a methodology that allows the actions of a human using an interface to be analyzed as sequences of elementary steps (press a key, move the mouse, make a decision). The performance of each elementary step is assigned a precise time period, and thus, by adding the times assigned to the steps in a task, GOMS provides a means of comparing different proposed interfaces in terms of the time each would require when performing similar tasks

TOPIC-145. SOFTWARE OWNERSHIP AND LIABILITY

Once software is made! One needs the ownership and should get profit from the investment.

- ✓ Intellectual Property Law
- Legal Efforts to provide you ownership of the developed software.
- based on the well-established principles of:
  - Copyright
  - patent law
- ✓ Purpose of Copyright or patent
- Allow the developer of a product to release that product to intended parties while protecting his or her ownership rights

- ✓ Requirement
- The developer will assert his or her ownership by including a copyright statement in all produced works; including requirement specifications, design documents, source code, test plans, and in some visible place within the final product.
- A copyright notice clearly identifies ownership, the personnel authorized to use the work, and other restrictions.

- ✓ Software License

The rights of the developer are formally expressed in legal terms in a software license

A Software License is a legal agreement between the owner and user of a software product that grants the user certain permissions to use the product without transferring ownership rights to the intellectual property.

- ✓ Patent
- Patent laws were established to allow an inventor to benefit commercially from an invention.
- Expensive and time-consuming to acquire
- Given the right to inventor for a limited period of time, which is typically 20 years

- ✓ Consequences of breaking law
- In 2004, a little-known company, NPT Inc., successfully won a case against Research In Motion (RIM—the makers of the BlackBerry smartphones) for breaking the patent law few key technologies embedded in RIM's email systems
- The judgment included an injunction to suspend email services to all BlackBerry users in the United States!
- RIM eventually reached an agreement to pay NPT a total of $612.5 million, thereby averting a shutdown.

vulmshelp.com