

SOFTWARE IMPLEMENTATION

http://www.tutorialspoint.com/software_engineering/software_implementation.htm

Copyright ©
tutorialspoint.com

In **this** chapter, **we will** study about programming methods, documentation and challenges in software implementation.

Structured Programming

In the **process** of coding, the lines **of code** keep multiplying, thus, size of the software increases. Gradually, it becomes next **to** impossible to remember the **flow** of program. **If one forgets how** software and its underlying programs, files, **procedures** are constructed it then becomes very difficult to share, debug **and** modify the program. The solution to **this is** structured programming. It encourages the developer to use subroutines and loops instead of **using** simple jumps in the code, thereby bringing clarity in the code and improving its efficiency. Structured programming **also** helps programmer to reduce coding time and organize code properly.

Structured programming states how the program shall be coded. Structured programming uses three main concepts:

- **Top-down analysis** - A software is always made to perform some rational work. This rational work is known **as** problem in the **software** parlance. Thus **it is** very important that we understand **how to** solve the problem. Under top-down analysis, the problem is broken **down** into small pieces where each one has some significance. Each problem **is** individually solved and steps are clearly stated about how **to solve** the problem.
- **Modular Programming** - While programming, the code is broken down into smaller group of instructions. These groups are known as modules, subprograms **or** subroutines. Modular programming based on the understanding of top-**down** analysis. It discourages jumps using 'goto' statements in the program, which often makes the program flow **non**-traceable. Jumps are prohibited and modular format is encouraged in structured programming.
- **Structured Coding** - In reference with top-down analysis, structured coding sub-**divides** the modules into further smaller units **of code** in the order of their execution. Structured programming **uses** control structure, which controls the **flow of** the program, whereas structured coding uses control structure to organize its instructions in definable patterns.

Functional Programming

Functional programming is style **of** programming language, which uses the concepts **of** mathematical functions. A function in mathematics should always produce the same result on receiving the same argument. In procedural languages, the **flow of** the program **runs** through procedures, **i.e.** the control of program **is** transferred to the called procedure. While **control flow is** transferring **from** one procedure to another, the program

changes **its** state.

In procedural programming, **it is** possible for a procedure to produce different results when it is called **with** the same argument, as the program itself can be in different state while calling it. **This is** a property **as well as a** drawback of procedural programming, in which the sequence or timing of the procedure execution becomes important.

Functional programming provides means **of** computation **as** mathematical functions, which produces results irrespective **of** program state. **This** makes **it** possible to predict the behavior **of** the program.

Functional programming uses the following **concepts**:

- **First class and High-order functions** - **These** functions have capability to accept another function **as** argument **or** they return other functions as results.

- **Pure functions** - **These functions do** not include destructive updates, that **is**, they do not affect any I/O **or** memory and **if** they are **not** in **use**, they can easily be removed **without** hampering the rest **of** the program.

Discussion is a PROGEO naming technique where a function calls itself and recursive

creating loops in functional programming.

- **Strict evaluation** - **It** is a method of evaluating the expression passed to a function as an argument. Functional programming **has** two types **of** evaluation **methods**, strict *eager* **or non-strict lazy**. **Strict** evaluation **always** evaluates the expression before invoking the function. **Non-strict** evaluation does not evaluate the expression unless it **is needed**.

- **λ -calculus** - Most functional programming languages use λ -calculus **as** their type systems. **λ -expressions** are executed by evaluating them **as** they occur.

Common Lisp, Scala, Haskell, Erlang and **F#** are some examples **of** functional programming languages.

Programming style

Programming style is set of coding rules followed by all the programmers **to** write the code. When multiple programmers work on the same software project, they frequently need to work with the program code written by some other developer. This becomes tedious **or** at times impossible, **if all** developers do **not follow** some standard programming style to code the program.

An appropriate programming style includes using function and variable names relevant to the intended task, using well-placed indentation, commenting code **for** the convenience of reader and overall presentation of code. This makes the program code readable and understandable by all, which in turn makes debugging and **error** solving easier. **Also**, proper coding style helps ease the documentation and updation.

Coding Guidelines

Practice of coding style varies **with** organizations, **operating** systems and language of coding itself.

The following coding elements may be defined under coding guidelines **of** an organization:

Naming conventions - This section defines how to name functions, variables, constants and global variables.

- **Indenting** - This is the space left at the beginning of line, usually **2-8** whitespace or single tab.
- **Whitespace** - It is generally omitted at the end of line.
- **Operators** - Defines the rules **of writing** mathematical, assignment and logical operators. For example, **assignment** operator '=' should have space before and after it, **as in "x = 2"**.
 - **Control Structures** - The rules **of writing** **if-then-else**, **case-switch**, **while-until** and **for** control **flow** statements solely and in nested fashion.
 - **Line length and wrapping** - Defines how many characters should be there in one line, mostly a line is **80** characters **long**. Wrapping defines how a line should be wrapped, **if** is too long.
 - **Functions** - This defines **how** functions **should** be declared and invoked, **with** and without parameters.
- **Variables** - This mentions how variables **of** different **data** types are declared and defined.
 - **Comments** - This **is** one of the important coding components, **as** the comments included in the code describe what the code actually does and all other associated descriptions. This section also helps creating help documentations **for** other developers.

Software Documentation

Software documentation is an important part **of software** process. **A** well written document **provides** a great tool and means **of** information **repository** necessary to know about software **process**.

Software documentation also **provides** information about **how** to use the product.

Requirement documentation - This documentation **works** as key tool **for software** designer, developer **and** the test team to carry out their respective tasks. **This** document contains all the functional, non-functional and behavioral description of the intended software.

Source of this document can be previously stored data about the software, already running software **at** the client's end, client's interview, **questionnaires** and research. Generally it **is** stored in the form of spreadsheet or **word** processing document with the high-end software management team.

This documentation **works as** foundation **for** the software to be developed and is majorly used in verification and validation **phases**. **Most** test-cases are built directly from requirement documentation.

- **Software Design documentation** - These documentations contain all the necessary

information, which are needed to build the software. **It** contains: High-level software architecture, Software design details, Data **flow** diagrams, Database design

These documents **work as** repository for developers to implement the software. **Though** these documents do not give any details on how **to** code the program, they give all necessary information that is required **for** coding and implementation.

- **Technical documentation** - These documentations are maintained by the developers and actual coders. These documents, **as a whole**, represent information **about** the code. While writing the code, the programmers also mention objective **of** the code, who wrote it, where **will** it be required, what it does and how it does, what other **resources** the code uses, etc.

The technical documentation increases the understanding between various programmers working on the same **code**. **It** enhances re-use capability of the code. It makes debugging **easy** and traceable.

There are various automated **tools** available **and** some comes **with** the programming language itself. For example java comes Java **Doc** tool to generate technical documentation **of** code.

- **User documentation** - This documentation **is** different from **all** the above explained. All previous documentations are maintained to provide information about the software and its development process. But user documentation explains how the software product **should work** and how **it** should be used to get the desired results.

These documentations may include, software installation procedures, how-**to** guides, user- guides, uninstallation method and special references to get more information like license updation etc.

Software Implementation Challenges

There are some challenges faced by the development team while implementing the software. **Some of** them are mentioned below:

- **Code-reuse** - Programming interfaces **of** present-day languages are very sophisticated and are equipped huge library functions. **Still**, to bring the cost **down** of end **product**, the organization management prefers to re-use the code, which was created earlier for some other software. There are huge **issues** faced by programmers for compatibility checks and deciding how much code to re-**use**.

- **Version Management** - Every time a new software **is** issued to the customer, developers have to maintain version and configuration related documentation. **This** documentation needs to be highly accurate and available on time.

- **Target-Host** - The **software** program, which is being developed in the organization, needs **to** be designed for **host** machines at the customers end. **But** at times, **it is impossible** to design a

software that works on the targot machinoc