**Design Principles**

**What is Design**

Design is a conceptual model of any process, product or object. We convert the design into implementation and build the product.

Examples:

* Blueprints of a building architecture is design and implementation is the actual building.
* Plan for celebrating your birthday is design and celebration of your birthday is implementation.

**Software Design**

If we consider the software as a product or object, the plan we will follow to build it will be the design.

Software development has a life cycle or process. We have to design it. Again, we need a design for the software or the product we will build. Thus, the software design has two parts-

* **Process design**

- The software development life cycle which plans the activities such as selection of methodology, project execution plan, selection of tools, documentation standards etc.

* **Product design.**

- It describes the architecture, subsystems, modules, packages and libraries, components and their interconnections and low-level entities like data structures, algorithms even table structures etc.

Design represents a high-level specification of the implementation that we are going to build.

**What is Design Principle?**

As a programmer, every individual can have their own style of writing programs. This may be applicable for writing simple programs for an individual’s own use. This is programming in small.

On the other hand, programs written for industrial requirements are generally very complex and teams of programmers are involved in writing these programs known as programming in large.

Thus, in programing paradigm, two types of programming seen.

* Programing in small
* Programming in large

Here we are going to discuss about programming in large.

Most real world software systems are large scale and complex, are of millions of lines of code, divided into several classes, and distribute among many people, expanded by many revisions.  In such cases, individual style of writing program is not applicable. Developers have to think of software systems at a higher level than just coding, as it is not enough to make the code just work correctly. We have to write the code in a way that makes it easy to develop and maintain further, hence we have to follow a standard to specify some main principles and there comes design principles.

**Why We Use Them**

Mostly for programming in large, developers should follow the design principles to achieve the following qualities in the software developed.

* **Readability:** Programming in large is a team effort. To make the code understandable to all members in the team following coding standards is important. In helps not only the developers, helps everyone, for example future maintenance if required.
* **Extensibility:** In the software world, requests for new requirements or changes is very rapid. It comes at any time. Hence, the code should be easy to extend without breaking existing business rules. Code must be flexible enough to adapt changes easily. For example, customer wants a change two days before the production go live
* **Maintainability**: Software should be easy to maintain by the development team and the production support team.
* **Reusability**: Productivity is one key requirement of software industry. Hence, software components should be designed in such a waythat they can be effectively reused. Divide a program into reusable pieces: functions, modules, libraries.

**Core Design Principles:**

There are some well-established design principles applicable in structures, procedural and object oriented programming. List of core design principles are

 · KISS

 · DRY

 · WET

 · YAGNI

 · SOLID

**KISS Design Principle**

 • Keep

 • It

 • Simple

 • Stupid

Keep it simple, do not apply circus.

According to this principle, simplicity should be a key goal in design. In programming, unnecessary complexity should be avoided. Simple systems work best than a complex one. A solution that follows the  KISS  principle might look dull or even “stupid” but simple and understandable.

Nowadays programming languages, frameworks, and APIs have powerful means to create sophisticated solutions for even simple kinds of problems. Developers might feel tempted to write “clever” solutions that use all these complex features and this may lead to complexities later.

KISS principle states we should not use the fancy features from the programming languages only because the language lets us use them. We should use such features only when there are perceptible benefits to the problem we are solving.

**Advantages**

Simple solutions has increased readability, understandability, and changeability. Hence are easier to maintain. Specifically when developer is not the maintainer.

Following are the benefits of KISS principle for developers.

 – Will be able to solve more problems, faster.

 – Will be able to produce code to solve complex problems in fewer lines of code

 – Will be able to produce higher quality code

 – Will be able to build larger systems, easier to maintain

 – Code base will be more flexible, easier to extend, modify or refactor when new requirements arrive

 – Will be able to work in large development groups and large projects since all the code is stupid simple

**How to Achieve the KISS principle**

* **Keep simple**: Try to write the simplest code.
* **Find alternatives**: Think of alternative solutions to your problem, choose the best one, and transform that into your code.
* **Decompose code**: Wherever there is lengthy code, divide that into multiple methods.
* **Single block Single task**: Try to write small blocks of code, which do a single task.

Let us see some examples.

**Example 1:**

Let us consider the problem statement as “Write a program to print the name of the Day of week if the day of the week in terms of number is given”.

We can write the code for the given requirement in two different forms using C programming language

Solution 1:

In this approach, we will use the switch case construct.

#include

int main()

{

       int day;

       printf("Enter a number ");

       scanf("%d",&day);

       if (day<1 || day>7)

          printf("\nAn invalid value for day. Give a value between 1 - 7\n\n");

       else

       {

         switch(day)

         {

              case 1:

                     printf("It is Monday\n\n");

                     break;

              case 2:

                     printf("It is Tuesday\n\n");

                     break;

              case 3:

                     printf("It is Wednesday\n\n");

                     break;

              case 4:

                     printf("It is Thursday\n\n");

                     break;

              case 5:

                     printf("It is Friday\n\n");

                     break;

              case 6:

                     printf("It is Saturday\n\n");

                     break;

              case 7:

                     printf("\n It is Sunday");

                     break;

              default:

                      printf("\n Have a nice day");

         }

       }

}

Solution 2:

In this approach , we will use array instead of the switch case construct.

#include

int main()

{

     int day;

     char \*daysOfWeek[7]={"Monday","Tuesday","Wednesday","Thursday","Friday","Saturday","Sunday"};

     printf("Enter a number ");

     scanf("%d",&day);

     if (day<1 || day>7)

        printf("\nAn invalid value for day. Give a value between 1 - 7\n\n");

     else

      printf("This is %s\n",daysOfWeek[day-1]);

}

If we compare the above two solutions, the second solution, seems to be compact in terms of lines of code. In case of the first solution approach, Instead of days of week, if we have to consider Months in an year or some other scenario like this , the cases in switch case construct will be more and if we have add some more conditions later to support change requirement, we may have to add some more lines of code. That means Contrary to that, in case of the second solution approach, there will not be any change in the number of lines of statements, as simply we have to increase the size of the array and add the additional elements in the array. Hence, in this case the second solution approach is implementation of KISS.

**DRY Design Principle**

* **Do not**
* **Repeat**
* **Yourself**

According to this principle,

* Every piece of knowledge must have a single, unambiguous, authoritative representation within a system.
* Have Single Source of Truth.
* Where “duplication” is required use code generation. E.g. Table in Database from one place, maybe ER diagram
* Practice: Forget that ctrl-c ctrl-v exists when coding
* Use Refactoring to remove duplications in code.
* Once And Only Once programming Vs Clone And Modify Programming

Duplication of code is the root cause of most of the maintenance problems, poor factoring, and logical contradictions. It is also a waste, waste of effort and time .

According to DRY, in an application, we should write a piece of logic once only .

**How to Achieve DRY Principle**

· **Divide and Reuse**: Divide your system into pieces. Ask yourself a question “am I going to do this again?” If yes, make it a reusable unit or method. Use this method by calling it wherever required. Thus, you will divide your code into smaller units and reuse.

·**Keep Short:** Keep methods short, divide logic and try to use the existing piece of code in your method.

Advantages of DRY Principle

**Advantages of DRY Principle**

* **Maintainability**: Ensures easy maintenance. There is no repeat of code, hence change in single place works for multiple instances.
* **Readability**: Code is more structured hence increases the readability.
* **Reusability**: DRY promotes reuse of code. We write a single block of code or method to replace two or more instances of repeating code and reuse the same block or method for multiple instances.
* **Cost Effective**: DRY insists on minimized repetition and maximized reuse, which promotes, less time to develop, debug and maintain. Thus saves human effort and time.
* **Testability**: If there are more paths or functions for a single behavior in code, more number of test cases are required. If code is without repetition, we have to test only one for single behavior.

**Disadvantages of DRY**

Implementation of DRY without proper care may make the code complex and make difficult to read and understand. One good example of the DRY principle is the helper class in enterprise libraries, in which every piece of code is unique in the libraries and helper classes.

**YAGNI Design Principle**

• You  
• Aren’t/Ain’t  
• Gonna  
• Need  
• It

As per this principle

• Do the simplest thing that will work

• Do not implement a feature/functionality until it is deemed necessary. E.g. caching  
• Extreme Code tweaking for Performance Tuning.

YAGNI is a principle of extreme programming (XP) .This principle promotes to ensure the following quote by XP co-founder Ron Jeffries “Always implement things when you actually need them, never when you just foresee that you may need them.”  
As per this principle, we should not implement anything, which is not required now. This minimizes the amount of unnecessary work, which is a great way to improve developer productivity and product simplicity.

**Advantages of YAGNI**

• **Facilitates Agile Implementation**: This principle becomes important in projects being executing the agile methodology and projects requiring tight schedule. In Agile projects, team will have a release plan and a product backlog having list of user stories. Team has to complete each User Story in a specific sprint or iteration. Under the current sprint plan, team member will working on a task or functionality having an estimated effort that the member has signed up for, in support of one of the iteration's User Stories.

**SOLID Design Principle**

Robert C. Martin in the early 2000s defined five design principles for Object Oriented Programming. Later, Michael Feathers introduced SOLID principles acronym for these five principles. These principles are essential for building robust working softwares.

The five principles of the acronym SOLID are as follows.

• Single Responsibility Principle – Do One Thing At all levels (infrastructure, application, module, class, method, and attribute, variable). The objective of this principle is to reduce coupling and increase cohesion.

• Open Closed Principle - Open for Extension closed for modification means we should be able to modify the behaviour of any entity by extending it but never by modifying its source code. We should write a class or module in such a way that  
it could be used as it is or can be extended if needed, but it can never be modified.

• Liskov Substitution Principle - Objects in a program should be replaceable with instances of their subtypes without altering the correctness of that program.

In case of OOP, it means, the child class should be able to substitute parent classes easily without disturbing the application.

• **Interface Segregation (Loose Coupling) Principle**- Every client should use an interface instead of actual implementation. Many client interfaces are better than one general-purpose interface. ISP says, we should avoid fat interfaces. A fat interface, an interface with additional useless methods lead to unintentional coupling between classes.

• **Dependency Inversion Principle** - High-level modules should not depend on low-level modules, rather both should depend on abstraction. Abstraction should not depend on details; rather detail should depend on abstraction. This principle  
is primarily concerned with reducing dependencies amongst the code modules and it helps in understanding how we can integrate the system together correctly.

**Advantages of SOLID**

• Helps design systems that are robust  
• Helps design systems that are easy to debug  
• Helps design systems that are flexible  
• Helps design systems that exhibit ease of maintenance  
• Helps design system that do not cost much to maintain  
• Helps design systems that are easily, cheaply and quickly extensible  
• Helps design systems that are scalable

As per good programming practices, it is better to minimize coupling and increase cohesion in codes. The five SOLID principles help us in ensuring the same in our code.

**Programming Practices**

In different design principles discussed above, we have seen that there are some common benefits of following them like readability, maintainability, scalability etc. A programmer can achieve them through implementation of some basic programming practices like, Top down approach, bottom up approach, structured programming, information hiding, cohesion, coupling etc. Hence, it is important to understand these basic programming practices. In addition, it is important to know how to implement them to develop a robust program.

**Top Down and Bottom Up Programming:**

One of the key principle of program design is code decomposition means, decompose or divide the code into subsections or modules to implement different concerns or functionality of the program as a module or function. We can manage the complexity by decomposing a big system into smaller units or modules, which interact or collaborate. We can decompose in two ways and thus the programming is divided into two categories , top down and bottom up programming

**Top down Programming:** In this approach, design takes the whole software system as one entity and then decomposes it to achieve more than one sub-system or component based on some characteristics. Each sub-system treated as a system, is decomposed further. This process keeps on running until we achieve the lowest level of system in the top-down hierarchy. We will start with a generalized model of the system and keep on defining the more specific part of it.

The advantages of top down approach are fixed style, clarity, and productivity, ease of testing and maintenance and ease of redesign. Structured programming follows top down approach.

By implementing code decomposition, we can improve productivity. This is because we are able to decompose independent concerns with single entry and single exit criteria. If we can decompose different functionality independently, we can compose them together to form bigger modules or bigger constructions such as, bigger functions or bigger objects. Team members can focus on independent component, once they completes a component, and if they follow well-established principles of programming, we can reuse them. We can reuse the existing components, developed components repeatedly, and  
that improves higher productivity.

Another advantage of code decomposition is ease of testing. It is easy to test independent modules and then take concerns or handle concerns one by one.

Modularization helps in easy maintenance and redesign. If something is wrong with a module, we can simply replace that if there are no side effects on the overall functioning of the system. Similarly, we can add or remove an additional module easily.  
One disadvantage of top down decomposition is the risk of implementing data structures as the modules are dependent on each other and they have to share one or more functions and procedures. In this way, the functions and procedures are globally visible. In addition to modules, the top-down programming uses sequences and the nested  
levels of commands.

**Bottom up Programming**: In this approach, we will start with the most specific and basic components and proceed composing the higher level of components until the desired system is not built as single component. Here, the program modules are more general and reusable than top-down programming. Thus, it is easier to add new features  
in a bottom-up approach than in a top-down programming approach.

Bottom-up programming requires a way of passing complicated arguments between functions. Hence, it is easier to construct functions. It takes the form of constructing abstract data types in languages such as C++ or Java, which can be used to implement an entire class of applications and not only the one that is to be written.

**Structured Programming**

A computer program comprises of some basic structures like sequences, decisions, and loops. Structured programming generally makes use of top-down design. Therefore, during top down decomposition, we have to design a program as a top down hierarchy of modules. The heart of structured programming is the idea of using only the single  
entry and single exit blocks. For every programming constructs such as individual sequences, decisions and loops and so on so forth, we need to implement single entry single exit principle.

Structured programming follows two basic principles by Dijkstra  
1.1)use of single entry and single exit blocks  
1.2)avoid GOTO statement.  
Dijkstra stated to avoid GOTO as it results in unstructured programming. Use of GOTO statement in program makes the program very hard to understand and hard to debug. Using GOTO, we can jump from any instruction to any instruction, we may not be able to structure the program in terms of well-defined blocks, and we will be unable to connect the blocks from smaller levels to higher levels in a top down fashion to have a clear top down structure.

Example of Single Entry Single Exit Structure

Let us consider the following statements

* **A primitive statement**

               S;  
               Ex:   int x=20;

* **Sequence of Statements**

            S1;  
            S2;  
            S3;  
            ………  
            ……..  
  Ex:  
      int number;  
      printf(“Enter a number “);  
      scanf(“%d”,number);  
      int sqr=number\*number;  
      printf(“The square of the number is %d”,sqr);

* **Conditional Statement**

          if (condition) then S1 else s2  
          Ex:  
          if (x>0)  
             printf(“%d is greater than 0”,x);  
         else  
             printf(“%d is not greater than 0”,x);

* **Loop or repetition**

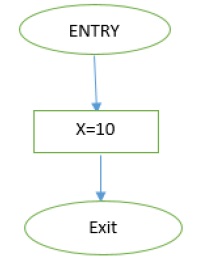
           while (condition) do S

           Ex:  
            while (x>10)  
          {  
             printf(“%d “,x);  
             x--;  
          }  
All these statements are single entry single exit, even the primitive statements like assignments and sequences.

The following flow charts will represent it clearly

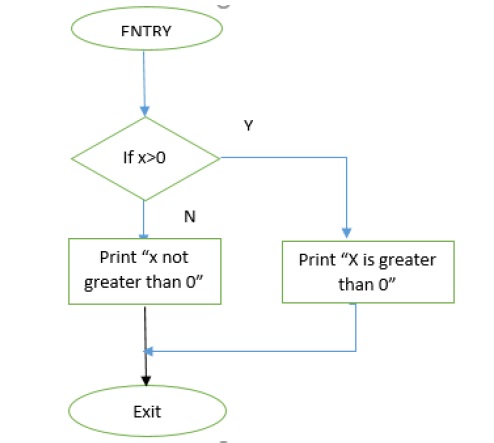
**A primitive statement**

**x=10;**

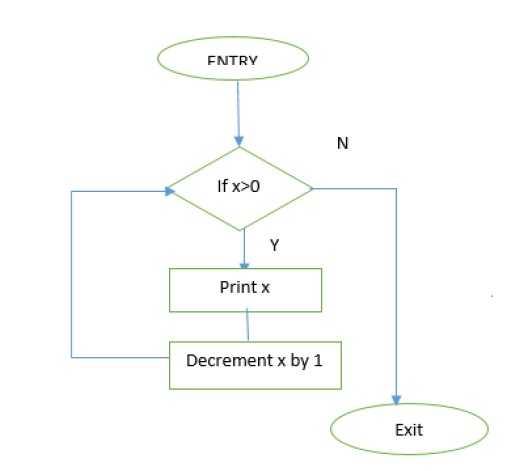


There is one entry and one exit for the block having one statement..

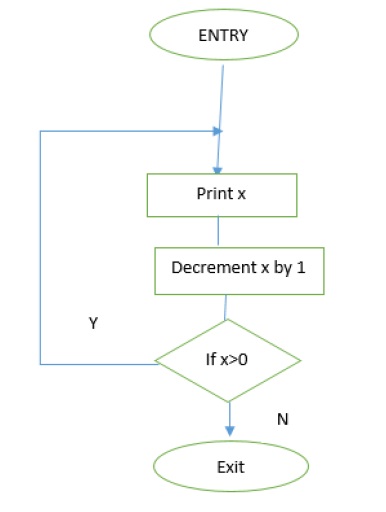
**A Conditional Statement (if – then – else )**



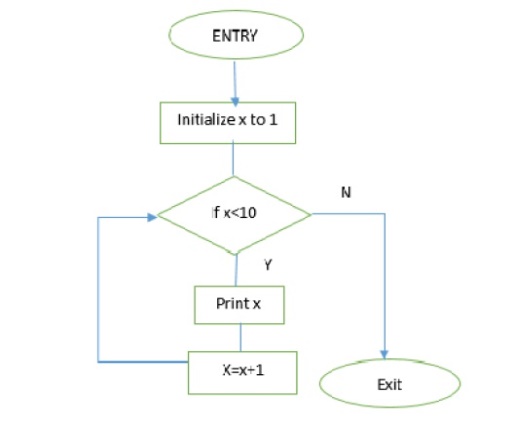
**A repetitive statement – while – do loop**



**A repetitive statement – repeat until or do – while**



**A repetitive statement – for loop of C programming language**  
A loop to print 1 to 10.



As seen in the above flow charts, we can construct a statement with single entry and single exit. Finally, we can club together all these statements to build bigger blocks. We can also build functions with function bodies without using ‘Goto’ statement and rely on the basic structured program principle.

**Disadvantages of GOTO:**

* Use of GOTO produce unstructured programs.
* Use of GOTO facilitates multiple entries into a block as and when you require. We can jump wherever we want in the program and hence, multiple exits are possible.
* Programs become difficult to understand and debug.

We should use GOTO only for low level or machine level programming. For example, in machine language, we might use a ‘Goto’ or equivalent ‘jump’ to jump around to a specific label or to a specific memory address.

**Example with GOTO:**  
Now let us see how use of GOTO can produce unstructured programs.

Let us use C programming language.  
int main()

{  
int age =0;  
printf(“Enter your age”);  
scanf(“%d”,age);  
if (age>=65) goto label1;  
if (age>=20) goto label2;  
if (age>=13) goto label3;  
if (age>=5) goto label4;  
printf(“You are minor”);  
goto end;  
label1: printf(“You are Senior citizen”);  
goto end;  
label2: printf(“You are an Adult”);  
goto end;  
label3:printf(“You are a teen”);  
goto end;  
label4:printf(“You are a child”);  
goto end;  
end: printf(“bye”);  
}  
This is not a structured code. We can rewrite the same code in a more structured way using if - then-else single entry single exit construct.  
int main()  
{  
int age =0;  
printf(“Enter your age”);  
scanf(“%d”,age);  
if (age>=65)

     printf(“You are Senior citizen”);  
else  
    if (age>=20)  
          printf(“You are an Adult”);  
   else  
       if (age>=13)  
           printf(“You are a teen”);  
       else  
            if (age>=5)  
                 printf(“You are a child”);  
           else  
                 printf(“You are minor”);  
      printf(“Bye”);  
}

**Multiple Exit cases:**

In some situations, programmers need to use multiple exit forms to have better clarity in the code. However, multiple entry forms are heavily discouraged.

**The situations where multiple exit forms are encouraged are:**

* Where single exit forms may be cumbersome. Basically to improve clarity or avoid use of temporary variables.
* In case of exceptions
* Return from a function to improve clarity.
* Labeled break and continue, in case of for loop