SOFTWARE ENGINEERING
Tutorial Sheet 1

Submitted By:

Name: Gursimar Kaur

Roll no: 1820036 (Uni.)

Class: D_3 CS E_3

Submitted To:

Prof. Jasdeep Kaur

Ques 1: Write the IEEE definition of software engineering?

Sol 1:

IEEE defines software engineering as:

1. The application of a systematic, disciplined, quantifiable approach to the development, operation and maintenance of software; that is, the application of engineering to software.
2. The study of approaches as in the above statement.

Ques 2: Compare and contrast: Structured programming and Un-Structured programming.

Sol 2:

TERMS	STRUCURED PROGRAMMING (MODULAR PROGRAMMING)	UN-STRUCTURED PROGRAMMING
1. Definition	Structured programming divides code into modules or functions.	In unstructured programming, the code is considered as one single block.
2. Readability	Structured based programs are easy to read.	They are hard to read.
3. Purpose	Structured programming makes code more efficient and easy to read.	It is just a solution to a problem and doesn't form a logical structure.
4. Complexity	It is easier because of formation of modules in the program.	Harder as compared to structured programming.
5. Modifications	Easy to do changes in structured programs.	Hard to do modifications in Unstructured programs.

6. Code duplication	Avoids code duplications through the use of modules or functions.	Unstructured programming may arise some code duplications in the programs.
7. Testing and Debugging	Testing and debugging to structured programs can be easily done.	Hard to do testing and debugging.
8. Applications	Can be used in medium and complex applications.	Can be used effectively only in small and sometimes in medium applications.
9. Languages developed	C, C+, C++, C#, Java, PERL, Ruby, PHP, ALGOL, Pascal, PL/I and Ada.	Early versions of BASIC, JOSS, FOCAL, MUMPS, TELCOMP, COBOL, machine-level code, early assembler systems, assembler debuggers and some scripting languages such as MS-DOS batch file language.

Ques 3: Explain how the use of software engineering principles helps to develop software products cost-effectively and timely. Elaborate your answer by using suitable examples.

Sol 3:

- KISS (Keep It Simple, Stupid):
  Software systems work best when they are kept simple. Avoiding unnecessary complexity will make your system more robust, easier to understand, easier to reason about, and easier to extend.

  Example: Whenever you add a new dependency to your project, or start using that fancy new framework, or create a new micro-service, you're introducing additional complexity to your system.
- DRY (Don't Repeat Yourself):
  It basically means that you should not write the same code/configuration in multiple places. Every piece of knowledge must have a single, unambiguous, authoritative representation within a system.

Example: Making use of Inheritance, Composition and Database Normalization helps in achieving DRY.

- YAGNI (You Aren't Gonna Need It):
  Like KISS principle, YAGNI also aims at avoiding complexity, especially the kind of complexity that arises from adding functionality that you think you may need in the future.
  Example: you may think that you need that functionality in the future. But a lot of times, you may not even need it due to the ever-changing requirements of our software world.
- SOLID Principles:
  SOLID is a list of 5 software engineering principles.
  - S - SRP (Single Responsibility Principle)
  - O - OCP (Open Closed Principle)
  - L - LSP (Liskov Substitution Principle)
  - I - ISP (Interface Segregation Principle)
  - D - DIP (Dependency Inversion Principle)

1. SRP (Single Responsibility Principle): The Single Responsibility Principle states that every function, class, module, or service should have a single clearly defined responsibility. In other words, A class/function/module should have one and only one reason to change.
   Example: In a house, the kitchen is for cooking and bedroom is for sleeping. Both have a single clearly defined responsibility.
2. OCP (Open/Closed Principle): The Open/Closed principle is a facilitator of the above idea. It advises that we should build our functions/classes/modules in such a way that they are open for extension, but closed for modification.
   - Open for Extension: We should be able to add new features to the classes/modules without breaking existing code. This can be achieved using inheritance and composition.
   - Closed for Modification: We should strive not to introduce breaking changes to the existing functionality, because that would force us to refactor a lot of existing code and write a whole bunch of tests to make sure that the changes work.
3. LSP (Liskov Substitution Principle): The Liskov Substitution Principle simply means that every child/derived class should be substitutable for their parent/base class without altering the correctness of the program.

4. ISP (Interface Segregation Principle): The Interface Segregation Principle states that a client should never be forced to depend on methods it does not use. By making your interfaces small and focused.
5. DIP (Dependency Inversion Principle): The Dependency Inversion Principle tries to avoid tight coupling between software modules. It states that High-level modules should not depend on low-level modules, but only on their abstractions.

Ques 4: What are the three basic types of constructs necessary to develop the program for any given problem? Give examples of these constructs from any high level language.

Sol 4:

All programming language utilise program constructs. In imperative languages they are used to control the order (flow) in which statements are executed (or not executed). There are a number of recognised basic programming constructs that can be classified as follows:

1. **Sequences (First Floor):** A sequence construct tells the CPU (processor) which statement is to be executed next. By default, in popular languages, this is the statement following the current statement or first statement in the program.

   Example:
   In PYTHON 3,
   a = int(input("Enter first number: "))
   b = int(input("Enter Second number: "))
   sum = a + b
   print("Sum is: ", sum)

   Here, every statement is executed one after the other this structure is named as 'sequence of statements'.

2. **Selection (Second Floor):** A selection statement provides for selection between alternatives, alternative as in available route options for instruction execution. A program can take certain route depending on a situation and selection statements help in choosing between the routes.

   Example:

In PYTHON 3,

```
a = int(input("Enter first number: "))
b = int(input("Enter Second number: "))
if a > b:
    print("First number is greater ")
else:
    print("Second number is greater ")
```

Here, the 2 control blocks are formed according to the inputs that is if a>b then first block will be executed and if not then second block will be executed.

3. **Repetition (Third Floor):** A repetition construct causes a group of one or more program statements to be invoked repeatedly until some end condition is met.

   Example:
   In PYTHON 3,

```
a = int(input("Enter any number: "))
for i in range(1, a):
    print(i)
```

   Here, every statement inside the looping or repetition statement will be executed more than once that is number of times the input will be given.

Ques 5: List major differences between the exploratory and modern software development practices.

Sol 5:

EXPLORATORY SOFTWARE DEVELOPMENT	MODERN SOFTWARE DEVELOPMENT
1. In this, software is developed on Trial and Error basis.	1. Software development is done by using Life Cycle Models through several well-defined stages.
2. Emphasis on Error Detection.	2. Emphasis on detection of errors as close to their point of introduction as possible.

3. Errors are detected only during testing.	3. Errors are detected in each phase of development.
4. Emphasis on coding during entire development cycle.	4. The entire development cycle is divided into distinct phases.
5. Review is done at the end of development phase.	5. Periodic reviews during each phase of development cycle.
6. No specific testing process.	6. Software testing has become systematic.
7. Less attention was being given to producing good quality and consistent documents.	7. Better visibility of design and code resulting in production of good quality and consistent documents
8. Inefficient planning.	8. Thorough planning in terms of estimation, scheduling and monitoring mechanisms.

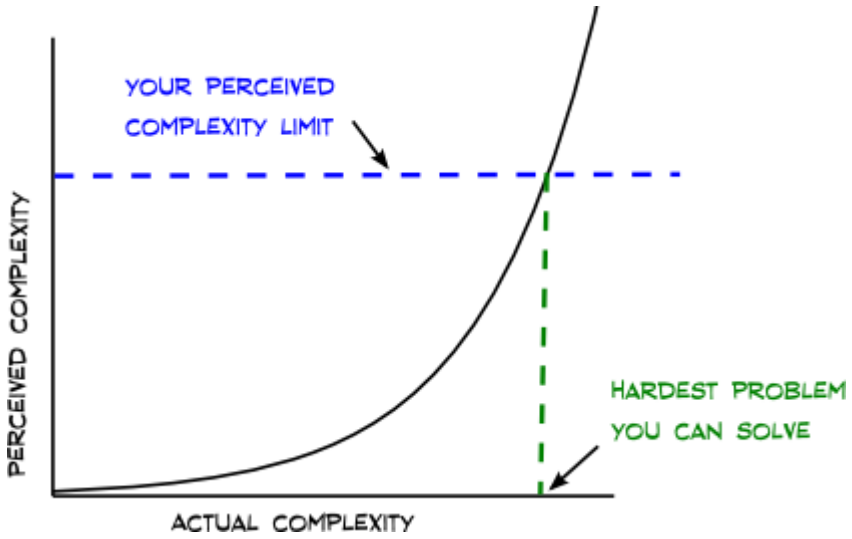
Ques 6: What is the difference between the actual complexity of solving a problem and its perceived complexity? What causes the difference between the two to arise?

Sol 6:

a)

ACTUAL COMPLEXITY	PERCEIVED COMPLEXITY
The actual complexity of an operation is determined by the step count for that operation, and the actual complexity of a sequence of operations is determined by the step count for that sequence.	It specifies how far an innovation is perceived as difficult to understand and use.

b)



A small increase in complexity might go unnoticed. But as complexity increases, your subjective perception of complexity increases even more. As you start to become stressed out, small increases in objective complexity produce big increases in perceived complexity. Eventually any further increase in complexity is fatal to creativity because it pushes you over your complexity limit.

Ques 7: What do you understand by control flow structure of a program? Why is it difficult to understand a program having a messy control flow structure? How can a good control flow structure for a program be designed?

Sol 7:

a) **Control Flow Structures:** In the programming language, all statements are executed sequentially from the top to the bottom. However, you can change the order (flow) of statements. Control flow structures allow you to alter the flow of the statements. Thus, you can develop complex programs that depict real world situations.

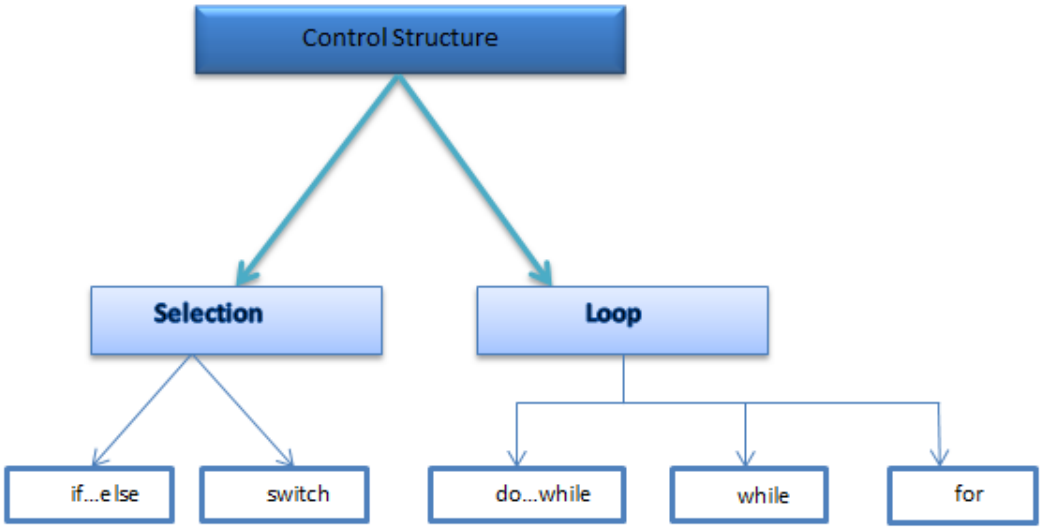


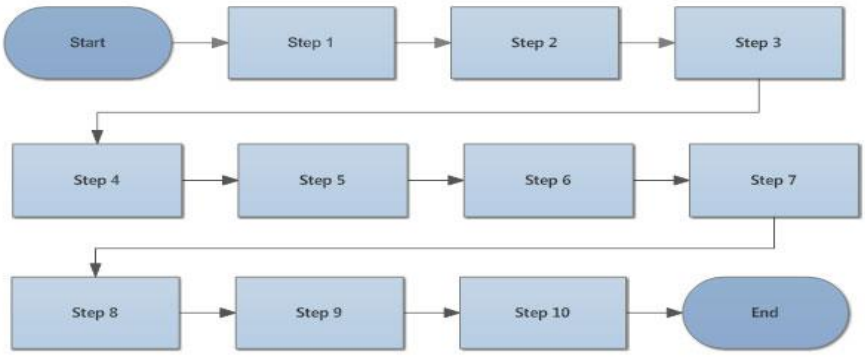
Figure 1: Control Flow Structures

b) It's difficult to understand a program with messy flow control because we'll not be able to:

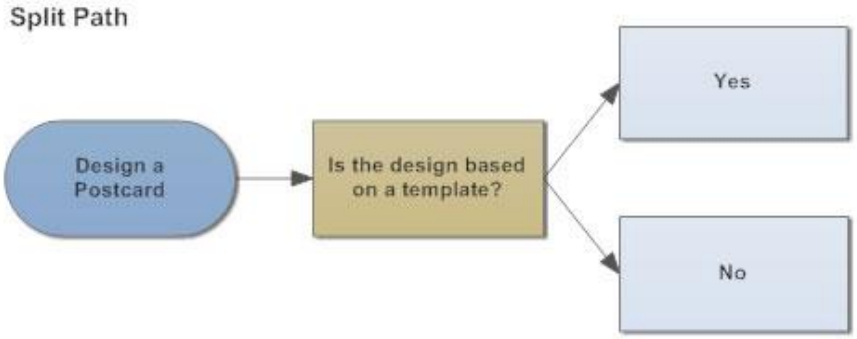
1. Figure out the division of statements.
2. Can't debug the program.
3. Messy flow control will not have easy modifications to the code.
4. New programmer's contributing to the program will not be able to get the variables, flow of structures, and blocks of code. Thus they will not be able to add-on to the code.

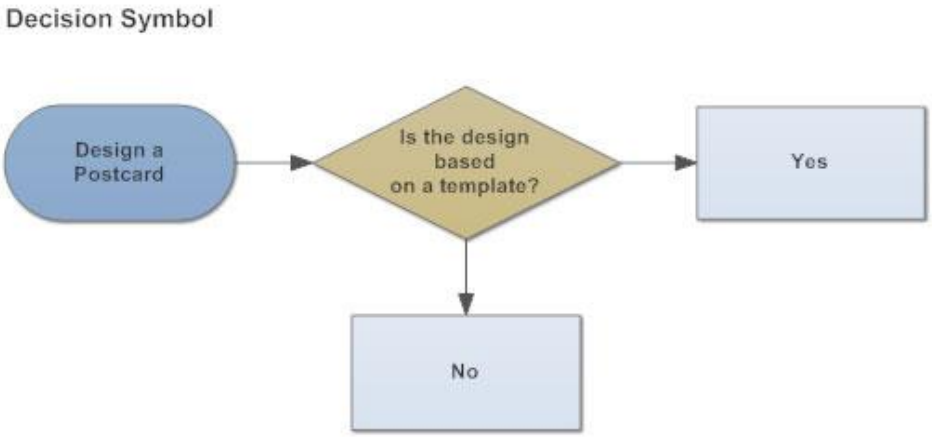
c) To have a good control flow of a program, the flow charting technique was developed. The idea was that before writing the program, represent the logic or the algorithm of the program in a flow chart and then translate the flow chart into code. Using this technique, one can represent and design a program's control structure. Some tips are as follows:

1. Use Consistent Design Elements
2. Keep Everything on One Page

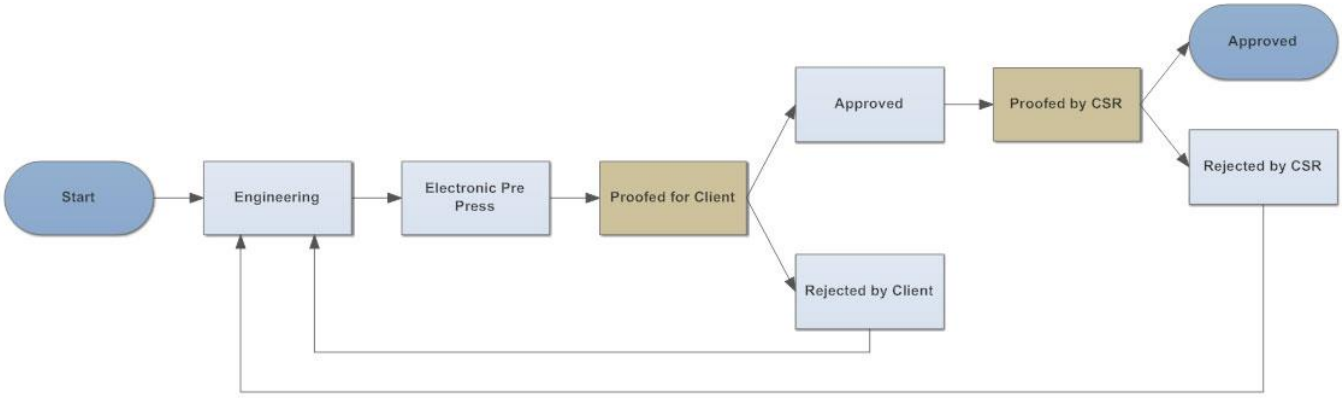


3. Flow Data from Left to Right
4. Use a Split Path Instead of a Traditional Decision Symbol





5. Place Return Lines Under the Flow Diagram



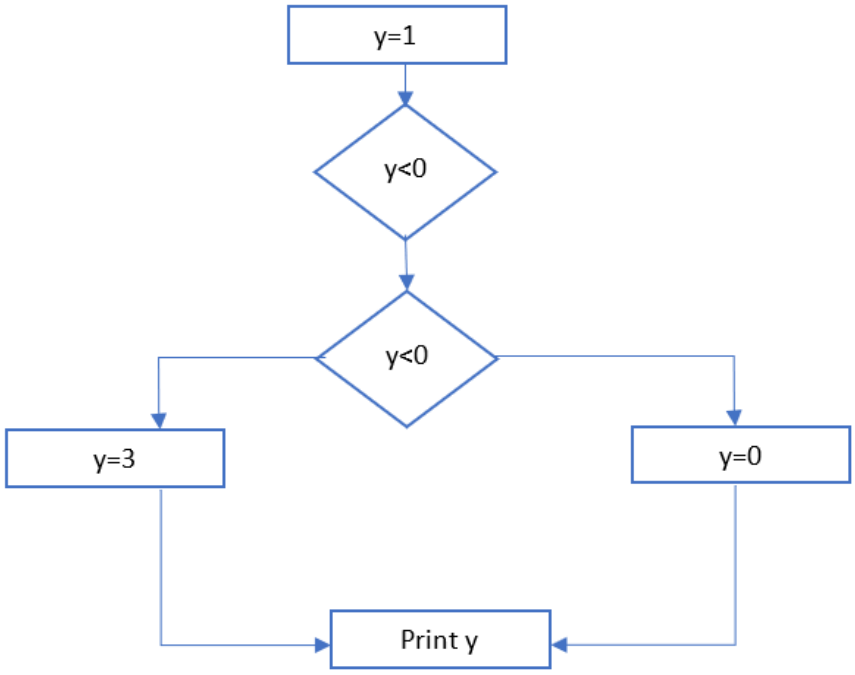
Ques 8: What does the control flow graph (CFG) of a program represent? Draw the CGF of the following program:

```
main(){
```

```
int y=1;
    if(y<0)
        if(y>0) y=3;sw
        else y=0;
    printf("%d\n", y);
}
```

Sol 8: A Control Flow Graph (CFG) is the graphical representation of control flow or computation during the execution of programs or applications.

CGF is as follows:



Ques 9: Discuss the possible reasons behind suppression of the data structure-oriented design methods by the data flow-oriented design methods.

Sol 9:

Possible reasons of data flow-oriented design methods to be more adopted over data structure-oriented design are:

- Major advantage of data flow-oriented design is its simplicity. The functioning can be easily explained through DFD.
- Data flow-oriented design is a generic technique which can be used to model working of any system and not just software systems as data structure designs.

Ques 10: What is computer systems engineering? How is it different from software engineering? Give examples of some types of product development projects for which systems engineering is appropriate.

Sol 10:

Computer (Systems) Engineering enables students to engage in the design of integrated hardware and software solutions for technical problems. They are involved in many aspects of computing, from the design of individual microprocessors, personal computers, and supercomputers, to circuit design.

Some differences between software and system engineering are:

- Software engineering highly focuses on implementing quality software while system engineers highly concern about the users and domains.
- Software engineering includes in computer science or computer based engineering background while system engineering may covers a broader education area includes Engineering, Mathematics and Computer science.).
- Software engineers focus solely on software components while system engineering deals with a substantial amount of physical component of computers.
- Software Engineering deals with designing and developing software of the highest quality, while Systems Engineering is the sub discipline of engineering, which deals with the overall management of engineering projects during their life cycle.
- Software engineering techniques such as use-case modelling and configuration management are being used in the systems engineering process.

Examples where system engineering is best suited are as follows:

- Computer chip design
- Spacecraft design
- Robotics
- Software integration

- Simulators
- Bridge building and many more.