Lecture 1: Software Engineering Principles

objectives: - learn about software engineering principles and Program life cycle.
- Discover what an algorithm is and explore problem-solving techniques
- become a ware of structured design and object-orinted design programming methodologies

Software life Cycle

software: Computer programs desinged accomplish a specific task (for example: word processing software 4 Internet)
Software program goes through several phases
(software engineering specializes in this area)
three fundamental stages:

1. Development: The program is created
(Solving a Problem)
Use: start using and discover problems
3 Maintenance: The program can be modified to fix the problem or to enhance it.

Software Development phase:

1. Problem analysis and specification
2. Design
3. Implemantation (coding)
4. Testing and debugging

1. Problem analysis and specification:
First and most important step.
Analysis requirement:
- Understand the problem (whole):
- Understand the problem requirements
(functional and non-functional requirements)
- Divide the problem into sub problems (if complex), analyze each sub problem, and understand each subproblem's requirements.

2. Design :

- Design an algorithm to solve the problem or subproblem.
- Algorithm :
  - Step-by-step problem-solving process
  - Solution obtained in finite amount of time
- Structured design :
  - Dividing problem into smaller subproblems
  - Also known as : top-down design, stepwise refinement, and modular programming

The solutions of all subproblems are then combined to solve the over all problem.

- object-orinted Design (OOD)

Steps in OOD are :

- Identify the component called objects
- Determine how objects interact with one another
- object specification: relevant data and possible operations on that data.

- object-oriented programming (OOP) language.

Programming langage that implements OOD.

- object-oriented design principles (advantages) :
  - Encapsulation : The ability to combine data and operations in a single Unit (in C++ using classes)
  - Inheritance - the ability to create new data types from existing types.
  - polymorphism - the ability to use the same expression to denote different operations.

3. Implementation :

- write and compile programming code

To implement the classes and functions discovered in the design phase.

The user needs to know only how to use the function and what the function does (The user does not need to know the specific details of how the function

is implemented).

the information about how to use the functions must be provided as a part of the documentation using specific statements, called preconditions and post conditions

Preconditions: A statement specifying the condition(s) that must be true befor the function is called.

Postconditions: A statement specifying what is true after the function call is completed.

4. testing and debugging:

- The term testing refers to
  - Testing program correctness
  - verifying program works properly.
- The term debugging refers to finding and fixing the errors, if they exist (bugs removing).

- Verification: refers to checking that program is correct (Are we build the product right).
- Validation: checking the matching with problem's specification (Are we build the right product).

Program (Algorithm) Verification methods:

1. Mathematical analysis (enough for small program)
2. Runing a program through a test cases:

test cases: a series of specific tests (set of inputs, User actions, other initial condition, and the expected output)

Test case categorization:

Categorize test cases into separate Categories, Called equivalence Categories.

example: function that takes an integer as input and return true if the integer is nonnegative, and false otherwise:

. we can form two equivalence Categories:

- negative numbers and nonnegative numbers.

Types of testing:

- black_box testing
- white_box testing

Black_box testing: it is based on inputs and outputs (internal structure of the algorithm or function is not important)

The test cases for black-box testing are usually selected by creating equivalence categories.

example: suppose that the function isWithinRange returns a value true if an integer is greater than or equal to 0 and less than or equal to 100.

- In black_box testing, we use values called "Boundary Values", as well as general values from the equivalence categories:

So: Boundary Values are:

-1, 0, 99, 100 and 101,

and the test values might be:

-500, -1, 0, 1, 50, 99, 100, 101 and 500

White_box testing:

relies on the internal structure and implementation of a function.

The objective is to ensure that every part of the function is executed at least once.

example: if statement in the program:

we must check all cases.

Maintenance: The program is updated and modified.

Example :

```
// Precondition : The value of inches must be nonnegative.
// post condition : if the value of inches is < 0 , the
// function return -1.0 ; otherwise, the function
// returns the equivalent length in centimeters.
   double inchesToCentimeters ( double inches)
     {
        if (inches < 0)
          {
           cerr << " The given measurment must be non-
                    negative." << endl;
           return -1.0 ;
          }
        else
           return 2.4 * inches ;
     }
```

or, we can use c++'s assert statement to validate the input.

```
// Precondition : The value of inches must be nonnegative
// postcondition : if the value of inches is < 0, the
// function return -1.0 ; otherwise ; the function
// return the equivalent length in centimeters.
   double inchesToCentimeters ( double inches)
   {
        assert (inches >= 0);
        return 2.54 * inches;
   }
```

if the assert statement fails, the program will terminate.

* you must use the preprocessor directive #include <cassert> to use the function assert.