

**SE202**

# **Introduction to Software Engineering**

## **Lecture 6.1 Development**

# Topics

## Development

- Use the right tools

- Selecting algorithms

- Top-down design

- Commenting code

# Programmer's life

- Programmers have collectively spent a huge amount of time programming, fixing bugs in their code, and thinking of ways to avoid similar bugs in the future.



# USE THE RIGHT TOOLS

- Programming language
  - Provide a set of instruction to build software
  - Know the advantages and disadvantages of the programming languages
  - [10 Best Programming Languages to Learn in 2020](#)
  - [TIOBE Index for February 2020](#)
- Hardware
  - Provided fast computers with lots of memory and disk space
  - P.S. Always test applications with hardware that is similar to the end users' machine.
- Network
  - Free access to the Internet if possible
  - Often a quick search can find a solution to a programming problem.

# USE THE RIGHT TOOLS

- Development environment
  - An integrated development environment (IDE)
  - Compiler or interpreter, debuggers, code profilers, class visualization tools, auto-completion when typing code, context-sensitive help, team integration tools, and more.
  - E.g. Eclipse, Visual Studio
- Source Code Control
  - Part of documentation management system
  - Tracking on changing of program code
  - Keep revisions of program code
  - Prevent multiple programmers from modifying the same code at the same time.
- Testing tools
  - Automated testing tools
    - make testing a whole lot faster, easier, and more reliable

# USE THE RIGHT TOOLS

- Source Code Formatters
  - Built into IDE or a separate code formatter
  - Formatting makes code easier to read and understand and fix bugs
  - E.g. standardize indentation, align and reformat comments
- Refactoring Tools
  - Built into IDE or a separate refactoring tools
  - Rearranging code to make it easier to understand, more maintainable, or generally better
  - E.g. defining new classes or methods, or extracting a chunk of code into a new method
- Training
  - Online video training courses, books and in-person training
  - Make programmers more effective

# SELECTING ALGORITHMS

- An algorithm is a recipe for solving a hard programming problem
- Example of hard problems
  - Sorting and arranging pieces of data
  - Quickly locating items in databases
  - Finding optimal paths through street, power, communication, or other networks
  - Encrypting and decrypting data
  - Finding least cost construction and production strategies
- Selecting a good algorithms for more complicated problems
  - Help finding a good solution in short time (in seconds or minutes or hours)

# Characteristics of good algorithms

- Effective
  - The algorithm can really solve your problems
  - If you can't find an algorithm that fits your problem, and you can't adjust your problem to fit the available algorithms, then you may need to write your own algorithm or modify an existing one.
  - If you do need to write your own algorithm or modify an existing one, be sure to perform extra testing to make sure it works correctly



# Characteristics of good algorithms

- Efficient
  - an algorithm must satisfy your speed, memory, disk space, and other requirements
  - E.g. Problems of dividing the 3 treasures (A, B and C) as equally as possible
    - The algorithm → to try every possible division and see which combination gives you the best result.
    - How many possible ways to divide the treasures?

You	Friend
A,B,C	-

# Characteristics of good algorithms

- Predictable
  - Some algorithms produce nice, predictable results every time they run.
  - It's also nice to know that an algorithm eventually finishes
- Simple
  - Ideally an algorithm should be simple
  - Simple code is easy to understand and easy to debug
- Prepackaged
  - If you can find an algorithm that is implemented inside your programming language or in a library, use it.
  - There's no need to write, test, debug, and maintain your own code if someone else can do it for you.



# TOP-DOWN DESIGN (stepwise refinement)

- If you need to write your own algorithm or prepare some code for using the algorithm, consider “stepwise refinement”
- Starting with a high-level statement of a problem, and you break the problem down into more detailed pieces.
- Continue examine the pieces and break any that are too big into smaller pieces until you have a detailed list of the steps you need to perform to solve the original problem.

# Example: PromoteSales ()

- Description: Identify customers who are likely to buy items on sale and send them e-mails, flyers, or text messages as appropriate.
- Add some detail

**PromoteSale()**

1. For each customer:
  - A. If the customer is likely to buy:  IsCustomerLikelyToBuy()
    - i. Send e-mail, flyer, or text message depending on the customer's preferences  
 SendSaleInfo()

**PromoteSale()**

1. For each customer:
  - A. If IsCustomerIsLikelyToBuy()
    - i. SendSaleInfo()

- At this point, you need to write the `IsCustomerLikelyToBuy` and `SendSaleInfo` methods.

`IsCustomerLikelyToBuy()`

1. If (customer earns more than \$50,000) return `true`.
2. If (customer lives within 1 mile of a golf course) return `true`.
3. If (customer is a country club member) return `true`.
4. If (customer wears plaid shorts and sandals with spikes) return `true`.
- ...
73. If (none of the earlier was satisfied) return `false`.

**Assumptions**

### **SendSaleInfo()**

1. Use the customer's `CustomerId` to look up the customer in the database's `Customers` table.
2. Get the customer's `PreferredContactMethod` value from the database record.
3. If (customer prefers e-mail) send e-mail message.
4. If (customer prefers snail-mail) send flyer.
5. If (customer prefers text messages) send text message.

At that point, sit down and write the code.

# PROGRAMMING TIPS AND TRICKS

- Be Alert
- Write for People, Not the Computer
  - Use meaningful names for variables
  - Indent your code nicely
  - Use comments or spell words correctly

# Comment First

- To explain what the code does and not what it should do in **Human Readable Descriptions**
- If the **code is well-written**, the future reader will read the code to **see what it actually does**.
  - Proper use of commenting can make code maintenance much easier, as well as helping make finding bugs faster.
  - Commenting is very important when writing functions that other people will use.



# Write Self-Documenting Code

- Use descriptive names for classes, methods, properties, variables, and anything else you possibly can.
- One exception to this rule is looping variables.
  - Programmers often loop through a set of values and they use looping variables with catchy names like *i* or *j*
- E.g.

```
% Poorly written/Cryptic code in Matlab
```

```
[a1, a2] = slvqd(A,B,C);
```

```
fprintf('we got %f, %f', a1, a2);
```

```
[answer1, answer2] = solve_the_quadratic_formula(A,B,C);
```

```
fprintf('The solutions to  $Ax^2 + Bx + C = 0$ , are %f, %f', answer1, answer2);
```

# Where to Comment

1. The top of any program file.
  - This is called the "**Header Comment**". It should include all the defining information about who wrote the code, and why, and when, and what it should do.
2. Above every function.
  - This is called the function header and provides information about the purpose of this function of the program.
3. In line
  - Any "tricky" code where the code is not self-documenting, should have comments right above it or on the same line with it.

# C file header example

```
/**
 * File:      compute_blackjack_odds.C
 *
 * Author1:   H. James de St. Germain (germain@eng.utah.edu)
 * Author2:   Dav de St. Germain (dav@cs.utah.edu)
 * Date:      Spring 2007
 * Partner:   I worked alone
 * Course:    Computer Science 1000
 *
 * Summary of File:
 *
 *   This file contains code which simulates a blackjack game.
 *   Functions allow the user of the software to play against the
 *   "casino", or to simulate the odds of successfully "hitting"
 *   given any two cards.
 *
 */
```

# C function header example

```
/**
 *
 * void sort( int array[] )
 *
 * Summary of the Sort function:
 *
 *     The Sort function, rearranges the given array of
 *     integers from highest to lowest
 *
 * Parameters    : array: containing integers
 *
 * Return Value : Nothing -- Note: Modifies the array "in place".
 *
 * Description:
 *
 *     This function utilizes the standard bubble sort algorithm...
 *     Note, the array is modified in place.
 *
 */

void
sort( int array[] )
{
    // code
}
```

# Inline comment tips

- To write comments that explain what the program is supposed to be doing is **to write the comments first**.
  - This lets you focus on the intent of the code and not get distracted by how many time the code itself changing!
- Compare the following versions of comments

```
// Loop through the items in the "items" array.
for (int i = 0; i < items.Length - 1; i++)
{
    // Pick a random spot j in the array.
    int j = rand.Next(i, items.Length);
    // Save item i in a temporary variable.
    int temp = items[i];
    // Copy j into i.
    items[i] = items[j];
    // Copy temp into position k.
    items[j] = temp;
}
```

The comments in this code explain what the code is doing, but they're mostly redundant.

```
// Randomize the array.
// For each spot in the array, pick a random item and swap it into that spot.
for (int i = 0; i < items.Length - 1; i++)
{
    int j = rand.Next(i, items.Length);
    int temp = items[i];
    items[i] = items[j];
    items[j] = temp;
}
```

← Tell the code's goal

← Tell how the code does it

# Validate Results

- Murphy's law states, "Anything that can go wrong will go wrong."
  - By that logic, you should always assume that your calculations will fail. Maybe not every single time, but sooner or later they will produce incorrect results
- To catch these problems as soon as possible, you should add validation code to your methods.
  - The validation code should look for trouble all over the place.
  - It should examine the input data to make sure it's correct, and
  - it should verify that the result your code produces is right.
  - It can even verify that calculations are proceeding correctly in the middle of the calculation.
- The main tool for validating code is the assertion.
  - An assertion is a statement about the program and its data that is supposed to be true.
  - If it isn't, the assertion throws an exception to tell you that something is wrong

```

// Use Euclid's algorithm to calculate the GCD.
// See en.wikipedia.org/wiki/Euclidean_algorithm.
private long GCD(long a, long b)
{
    // Verify that a and b are greater than 0.
    Debug.Assert(a > 0);
    Debug.Assert(b > 0);

    // Save the original values for later validation.
    long original_a = a;
    long original_b = b;

    for (; ; )
    {
        long remainder = a % b;
        if (remainder == 0)
        {
            // Verify that the result evenly divides the original values.
            Debug.Assert(original_a % b == 0);
            Debug.Assert(original_b % b == 0);

            return b;
        }
        a = b;
        b = remainder;
    };
}

```

# Exercise: improve commenting code

```
1  //gets all longest strings
2  function longestString(inputArray){
3      //Initialize the longest let to index 0
4      let lenght = inputArray[0].length;
5
6      for(let i=1; i<inputArray.length; i++){
7
8          //check if the current string is longer
9          if(lenght < inputArray[i].length){
10             lenght = inputArray[i].length;
11         }
12     }
13
14     //fillers out any values not equal to the longest String
15     const strs = inputArray.filter(() => {
16         return word.lenght === lenght;
17     });
18
19     return strs;
20 }
```