Topic 5

What is a good program?

A software engineering point of view

資料結構與程式設計

Data Structure and Programming

Sep, 2010

Why Software Engineering?

- **Software development is hard** !
- **Important to distinguish**

  “easy” systems (*one developer, one user, experimental use only*) **from “hard” systems** (*multiple developers, multiple users, products*)
- **Experience with “easy” systems is misleading**
  - *One person techniques do not scale up*
- **Analogy with bridge building:**
  - Over a stream = easy, one person job
  - Over River Severn … ? (*the techniques do not scale)*

Source: http://www.csc.liv.ac.uk/~igor/COMP201/

Why Software Engineering ?

- The problem is ***complexity***
- Many sources, but ***size*** is key:
  - UNIX contains 4 million lines of code
  - Windows 2000 contains 10^8 lines of code

Software engineering is about managing this complexity.

Source: http://www.csc.liv.ac.uk/~igor/COMP201/

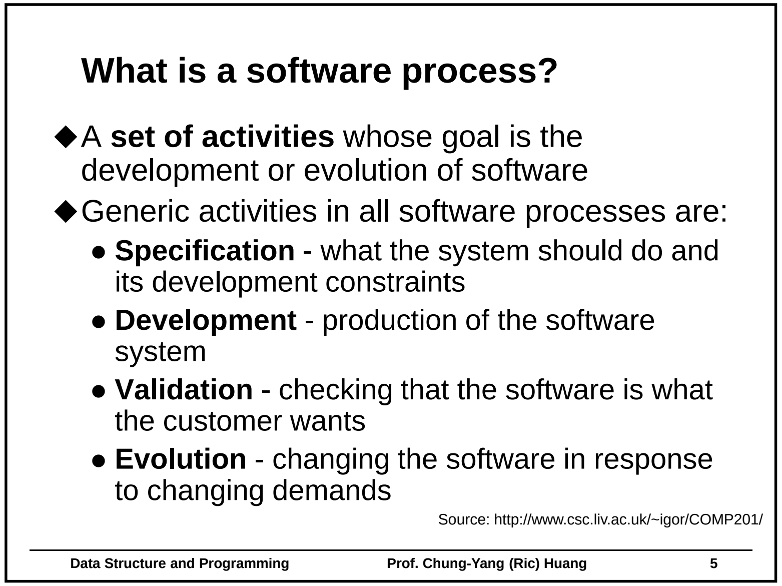
What is software engineering?

Software engineering is an engineering discipline which is concerned with all aspects of software production

Software engineers should

- Adopt a systematic and organized approach to their work
- Use appropriate tools and techniques depending on
  - the problem to be solved,
  - the development constraints and
  - the resources available

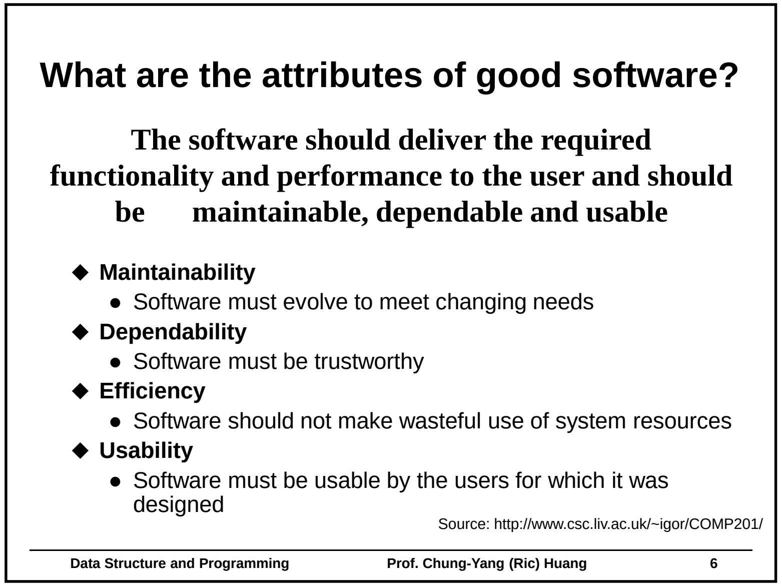
Source: http://www.csc.liv.ac.uk/~igor/COMP201/



What is a software process?

- A **set of activities** whose goal is the development or evolution of software
- Generic activities in all software processes are:
  - **Specification** - what the system should do and its development constraints
  - **Development** - production of the software system
  - **Validation** - checking that the software is what the customer wants
  - **Evolution** - changing the software in response to changing demands

Source: http://www.csc.liv.ac.uk/~igor/COMP201/

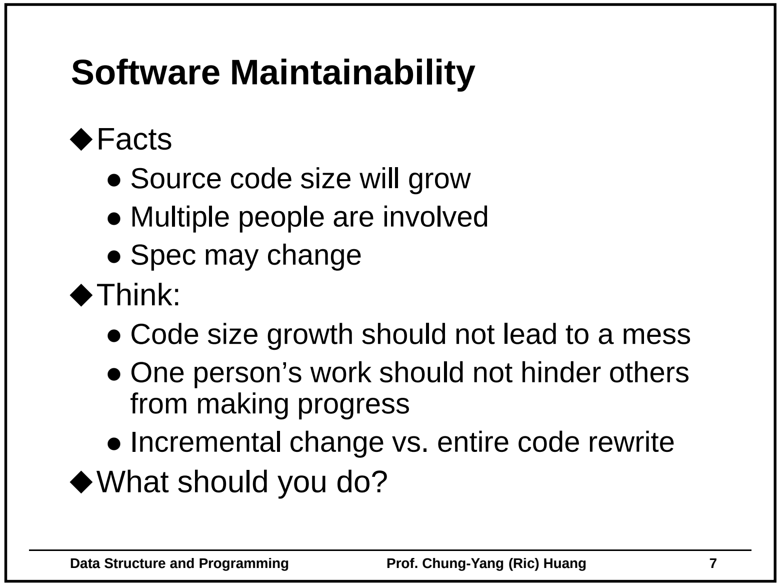


What are the attributes of good software?

The software should deliver the required functionality and performance to the user and should be maintainable, dependable and usable

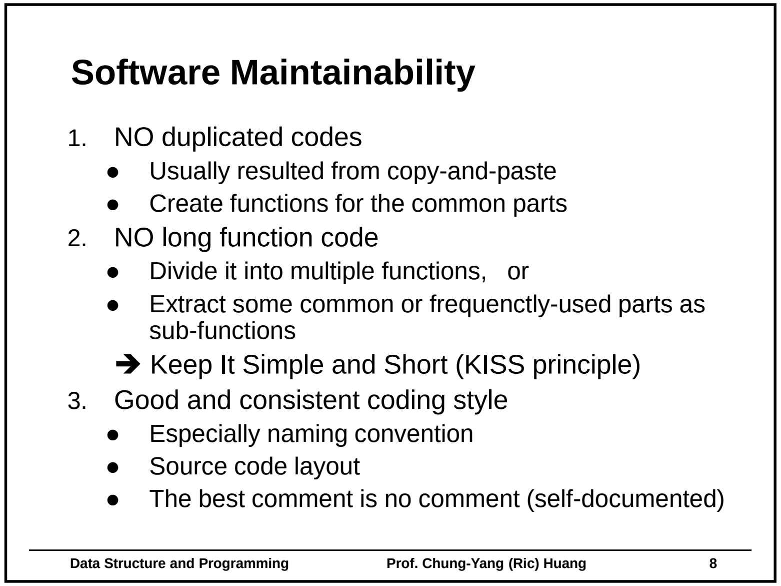
- **Maintainability**
  - Software must evolve to meet changing needs
- **Dependability**
  - Software must be trustworthy
- **Efficiency**
  - Software should not make wasteful use of system resources
- **Usability**
  - Software must be usable by the users for which it was designed

Source: http://www.csc.liv.ac.uk/~igor/COMP201/



Software Maintainability

- Facts
  - Source code size will grow
  - Multiple people are involved
  - Spec may change
- Think:
  - Code size growth should not lead to a mess
  - One person’s work should not hinder others from making progress
  - Incremental change vs. entire code rewrite
- What should you do?



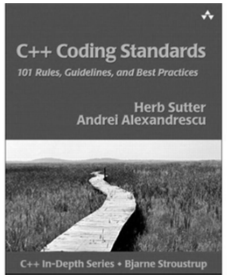
Software Maintainability

1. NO duplicated codes
   - Usually resulted from copy-and-paste
   - Create functions for the common parts
2. NO long function code
   - Divide it into multiple functions, or
   - Extract some common or frequenctly-used parts as sub-functions

   ➔ Keep It Simple and Short (KISS principle)
3. Good and consistent coding style
   - Especially naming convention
   - Source code layout
   - The best comment is no comment (self-documented)

C++ Coding Guidelines

- ◆ The first step in exercising software engineering principle is to follow the coding guidelines in the software development process
- ◆ This is an art.
  No universally correct answer.
- ◆ Google "C++ coding guideline"...


C++ Coding Standards
101 Rules, Guidelines, and Best Practices
Herb Sutter
Andrei Alexandrescu
C++ In-Depth Series • Bjarne Stroustrup

Coding Style --- Naming (FYI)

- ◆ Variable names
  - ● numStudents, isDone...
- ◆ Class names
  - ● LuxuryCar, BinaraySearchTree,...
- ◆ Function names
  - ● checkNumber(), computeScore()...
- ◆ #define / enum constant
  - ● RANGE, MAX_COLORS,...
- ◆ Class data members
  - ● _name, _id, _score,...
- ◆ Static, global variables / functions (optional)
  - ● nameMap_g, count_s, _memMgr_s, checkSum_s()...

Coding Style --- Look and Feel (FYI)

1. Proper indentation
   - 3 (or 2, or 4) spaces right for the codes within a new scope
   - Do not use "tab" ➔ platform dependent ➔ Use "space"
   - Try to turn off "auto indentation"
2. Proper alignment

   -
     ```
     if ((numStudents >= 30) &&
         (numStudents <= 80))
     ```

   -
     ```
     cout << "Hello, today is " << printDate()
          << ". Welcome to my world!!" << endl;
     ```
3. Braces { }

   -
     ```
     void function()
     {
     }
     ```

   -
     ```
     if (boolExpression) {
     }
     ```

Software Dependability

- ◆ Facts
  - Where there is a software, there is a bug.
  - The only way to enhance software dependability is ➔ **Test, and more tests**.
- ◆ Think:
  - What is an "experienced" coder?
  - Experience in:
    - ▪ "Spontaneous coding" (but with GOOD coding styles)
    - ▪ Debugging (to find the bug and to fix the bug)
  - You must get yourself familiar with debugger!!!
- ◆ The ultimate goal
  - When you see the bug, you know the possible cause(s)
  - No overnight (over-the-meal) bug

Software Dependability

- ◆ Regression test
  - ● A systematic mechanism to
    1. Collect and organize testcases
    2. Routinely run the testcases
    3. Make sure the newly added codes can still pass the testcases
    4. Check in new testcases for newly added codes
- ◆ Source code version control
  - ● A tool/database to centralize different versions of source codes
  - ● Differences between different versions are recorded incrementally, with logs and histories for later reference

Software Efficiency

- ◆ Facts
  - ● 80-20 rule
    - ▪ 80% (or more) resources (run time / memory) are consumed or controlled by 20% (or less) of the codes
  - ● Don’t be picky about the efficiency of the 80% less critical codes
    - ▪ Higher priorities: maintainability, dependability
    - ➔ Even though they may have negative effects on the efficiency
    - ➔ However, negligence on the maintainability and dependability may lead to unstructured codes and eventually jeopardize the efficiency
- ◆ What you should do?
  - ● Equip basic instincts about the implied complexity of the data structure and algorithm
  - ● Know when to be picky and when to let go

Software Usability

- ◆ Facts
  - ● Usability factors = usage flow, user interface, ease of use, usage consistency
  - ● 80-20 rule
    - ▪ 80% (or more) of the code is not related to user friendliness
    - ▪ However, 20% (or less) of the code determines how your program is appreciated by others
- ◆ Importance of the minority
  - ● Decisions about the above usability factors determine the architecture of the code/framework
  - ● Later change is hard
  - ➔ Plan at first!!

Discipline & Practice.