Lecture 5.
Software Engineering
(Part 1)

Prof. Guoyong Shi
shiguoyong@sjtu.edu.cn
Dept of Micro/Nanoelectronics
Shanghai Jiao Tong University
Fall 2015

Acknowledgement

- **Most part of this lecture was borrowed from a series of lectures offered by Prof. Gary Kimura, CS of University of Washington, 2001.**
- **His background**
  - **4 years at DEC**
  - **11 years at Microsoft**
  - **At DEC he prototyped a new PASCAL system compiler**
  - **At Microsoft he prototyped the Windows NT file system**

Contents

- **What and why software engineering**
- **Looming software crisis**
- **Lifecycle model**
- **Group organization**
- **Responsibilities of team member**
- **Software requirements**
- **Software prototyping**

Rumors and Myths on SE

- **Rumors:**
- **Software Engineering is not rocket science and not something to learn from a textbook**
- **Software Engineering is the use of common sense and discipline**

- **Learn that building large software systems is not a mere matter of programming**

Then how to teach Software Engineering?

- **There is not a single right way to teach software engineering**
- **Rule of the thumb: Have to teach SE from experience**
- **Principle: All engineering, including software engineering, is concerned with building useful artifacts under constraints**

Class Project

- **Learn SE while you implement small C++ programs**

- **To have hands-on feeling**
- **To build up your coding confidence**
- **To learn how to manipulate "complexity"**
- **To learn from Internet resources**

Several Definitions of Software Engineering

- The practical application of scientific knowledge to the design and construction of computer programs and the associated documentation required to develop, operate, and maintain them [Boehm].
- The systematic approach to the development, operation, maintenance, and retirement of software [IEEE].
- The establishment and use of sound engineering principles (methods) in order to obtain economically software that is reliable and works on real machines [Bauer].
- Multi-person construction of multi-version software [Parnas].

Why Study Software Engineering?

- **Most complex systems need software**

- **<u>However, building software without discipline is crazy</u>**

- **<u>Building a large complete software project is hard</u>**

- **There is a perceived crisis in our ability to build large-scale software**

Scale of Software

Code sizes due to Jon Jacky: KLOC = 1000 lines of code; MLOC = 1,000,000 lines of code

Bar code scanners	10-50KLOC
4-speed transmissions	20KLOC
ATC ground system	130KLOC
Automatic teller machine	600KLOC
Call router	2.1MLOC
B-2 Stealth Bomber	3.5MLOC
Seawolf submarine combat	3.6MLOC
NT 4.0	10MLOC
NT 5.0 (NTFS alone)	60+LMLOC (250K source lines or 100KLOC)

Size of Software

As the size of the software system grows, the dominant discipline changes (due to Stu Feldman)

Code Size (LOC)	Dominant Discipline
10^3	Mathematics
10^4	Science
10^5	Engineering
10^6	Social Science
10^7	Politics
10^8	Legal?

Software needs coordination among people

- **Therefore, most complex systems require many people to build**
- **Hence the critical need for coordination**

The Software Crisis

- **“We are unable to produce or maintain high-quality software at reasonable price and on schedule.”**
- **“Software systems are like cathedrals; first we build them and they we pray. [Redwine]"**

To some degree this is accurate

- **Some so-called software “failures” are often management errors**
- **(the choice not to do something that would have helped)**

- **In some areas, in particular safety-critical real-time embedded systems, we may indeed have a looming crisis**

Why is it hard?

- **There is no single reason software engineering is hard—it's a "wicked problem"**

- **Lack of well-understood representations of software makes customer and engineer interactions hard [Brooks]**
  - **Norman Augustine [Wulf]: "Software is like entropy. It is difficult to grasp, weighs nothing, and obeys the second law of thermodynamics; i.e., it always increases." [Law XXIII]**

In Your Programming Class

- **In your programming class you mostly implemented carefully defined specifications (by your instructor)**

Example

- **main.cpp - this is the main driver program.**
- **From this code we make calls to the 2DPlotter classes and functions.**
- **2DPlotter.h, 2DPlotter.cpp - these contain a skeleton of the specification and implementation of the 2D plotter as a C++ class.**
- **You will complete the implementation by adding more functions and variable definitions.**
- **Remember to document your implementation and present to the class!**

(continued)

- **Write a Makefile to automatically build your program whenever you make some changes.**

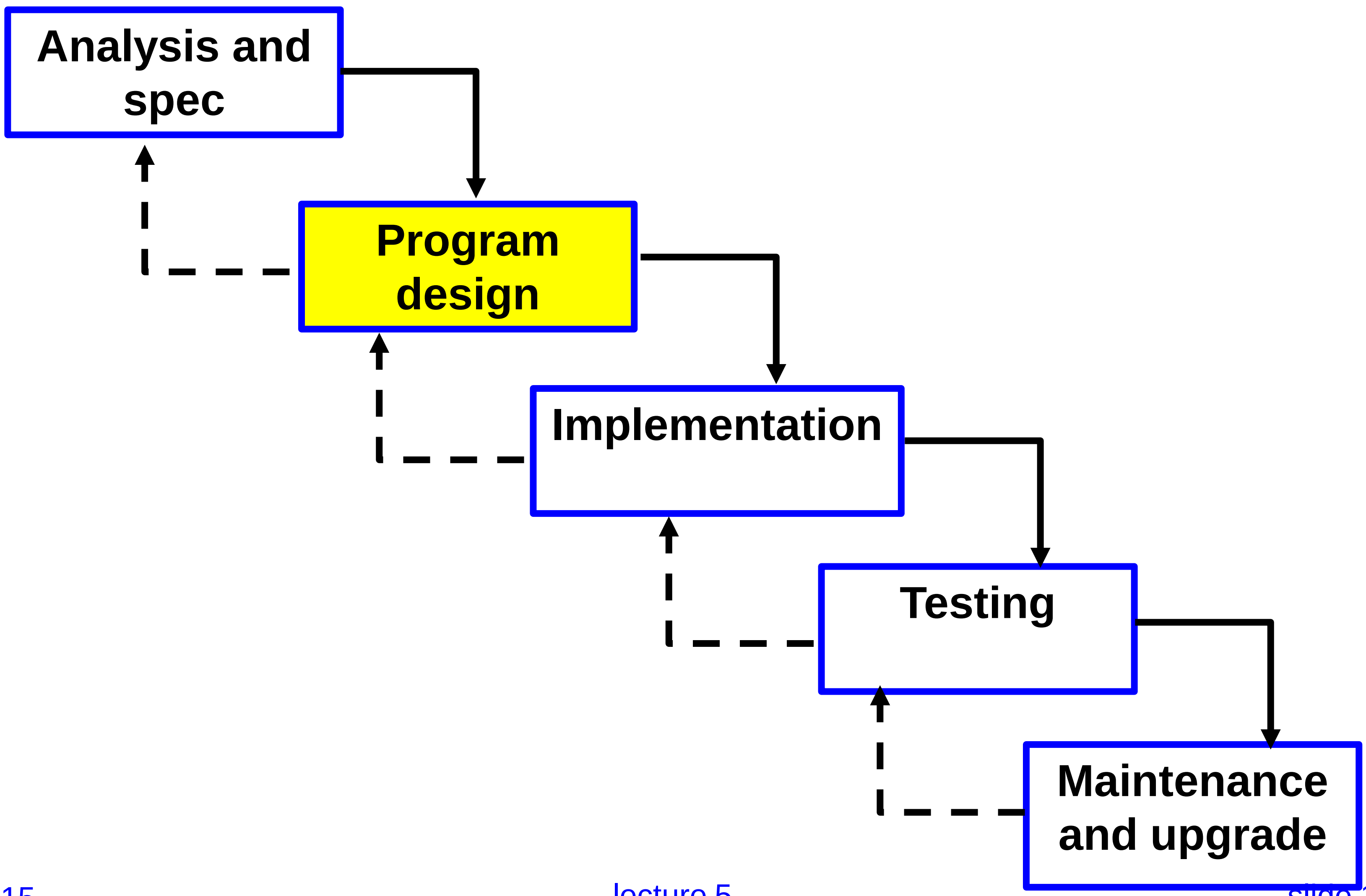
- **You were given...**
  (1) …the specification (in another slide)
  (2) …the design (discussed in class)
  (3) …hints about the implementation
  (4) …some partial code
  (5) …pointer to some graphics libraries to use

Software lifecycle

- **A software engineering lifecycle model describes how one might put structure on the software engineering activities**

- **The classic lifecycle model is the waterfall model (roughly due to Royce in 1956), which is structured by a set of activities and is inherently document-driven**

Waterfall Model describing software lifecycle



Hints from the Lifecycle Model

- **The cost of fixing errors at later lifecycle phases is much higher than at earlier stages**

- **A software lifecycle account for more than programming**

- **Also pay attention to the feedback between phases**

Why need a model

- **Software programs are large complicated beasts**
  - **Windows NT started out small.**
  - **Today not one single person can grasp all of NT.**
- **Use a model to recognize that Software Engineering is more than just programming**
- **Recognize product phases in a life cycle**
  - **Various requirements specification phases**
  - **Design phases**
  - **Coding and testing phases**
  - **Maintenance phase (bug fixes and revisions)**
- **Dividing in phases means management**

More reasons to have a model

- **A model helps you to recognize and define the division of labor (i.e., management)**
  - **Individual responsibilities**
  - **How big should a team be**
  - **Parallel work efforts**
- **Provides a structure for communication between different parties**

- **Documentation is vital**
  - **Comments in the code is not sufficient**
  - **Dave Cutler's NT design workbook is now part of the Smithsonian (a history museum in US)**

A good model means good management

- **A paradigm （范例，样式） that adds discipline and order to software development**
- **Provides a formal mechanism to clarify, track, and modify the product requirements throughout the product life cycle**

- **Even you code totally by yourself, it is still good to be aware of the lifecycle model**
- **Because one day another person might pick up your code**

More goals of a good model

- **Compel engineers to want to use it**
  - **Convinces them that they will build a better product**

- **Keep everyone organized**

  - **Recognize that Software Engineering is a process of iterative refinement**
  - **Allow for alternate designs and implementations**

Lessons from the models

- **Just as Software Engineering is full of compromises, so is using a Software Engineering model**

- **So take these models with a grain of salt and use only those parts that most suit your situation**

Product requirements

- **Needs a document listing the product requirement.**

- **Some necessary items are:**
  - **Describe its general function and purpose**
  - **Describe how it will be used by the customer**
  - **Describe what is required for the customer**
  - **Describe various aids to the customer**
  - **Describe hardware and software requirements**

Group Organization

- **22 students in one group – for example**

- **Not everyone will write shipping code**
  - **Manager, secretary, and group organizer (1 - 2)**
  - **Program management (4 - 5)**
  - **Software Developer (5 - 6)**
  - **Tester (7 - 8)**
  - **Documentation (3 – 4)**

Manager's Responsibilities

- **<u>Organizing</u> the whole thing**
- **Understanding the whole project**
- **Ensuring that everyone knows their part and milestones (communication)**
- **Catching up the schedule**
- **Not doing the work, but knowing how each part fits in**

Program Manager's (PM) Responsibilities

- **Defining the product**
- **Identifying customer needs**
- **Questioning the need or appropriateness of the design**
- **Working through all the usage scenarios**
- **Looking outside the "box"**

- **If you are alone in your team, you take the responsibility of everything**
- **But imagine you are taking the roles in turn**

Developer's Responsibilities

- **Designing the architecture and coding the product**
- **Working with PM to ensure you are building what they defined**
- **Adding APIs as needed by the test group**

Tester's Responsibilities

- **Unit or component testing**
- **Correctness tests**
- **End (terminal, extreme condition) cases**
- **Error checking**
- **Stress tests (how large problem it can solve)**
- **Independent code review leading to targeted tests**
- **Interaction with other systems**

Documentation Responsibilities

- **Keeping track of all the design documentation**
- **Complete end user documentation**
- **Quick guides and on-line help.**

Software Requirements

- **Two words: "Risks" and "Constraints"**
- **Specifying requirements**
  - **One person's requirement is usually someone else's design**
  - **Expect unintended side affects (i.e., customers will use the system in ways you can never imagine)**
- **How to write a requirement**
  - **It is an iterative process,**
  - **a good requirements writer bridges the gap between customer and implementer**

What should be in a Requirement

- **Remember requirement could be changing.**

- **Expect the requirements (goals) to change, due to customer changes, market place changes, technological changes**
- **Expect the team to change during the product cycle.**
- **One of the hardest tasks is to replace people in the middle of a project**

Back to our programming assignment

More on the Programming Assignment

- **Write a plan with milestones**
  - **Must have a time schedule (in weeks)**
- **Lifecycle model**
  - **Division of labor is important (if you are alone then division of time)**
  - **Making sure you have a roadmap**
- **Requirements**
  - **Important to write this down**
  - **Keep this realistic**
  - **Expect them to change**

More on the Assignment

- **You must learn to design**
  - **Design your code and components**
  - **Design possible extensions**
- **Be aware of testing**
  - **Coding and component testing**
  - **Integration and system testing**
- **Deployment and maintenance**
  - **How do you want the user to use your software ?**

How to build and use software prototypes?

What is prototyping

- **Building models that demonstrate properties of the real product**
- **Building something faster and cheaper than the real product**

Why prototype

- **To understand**
  - **How people will use and interact with the product**
  - **How to build the real product**
- **To tweak the design before it is too late**
  - **Change requirements**
  - **Change interface**
  - **Change architecture**
- **The goal is to convey enough information <u>to judge the design</u> and the product development process**

Examples of prototyping

Sample of prototypes:

- **A subset of an API set**
- **A non-fully featured app**

What is missing from a software prototype

- **The code is missing**
  - **Typically most of the error handling is missing**
  - **May not be extensible, maintainable or just well designed**
  - **Typically not fully featured**

- **The documentation, testing, performance, and support considerations are missing**

When to do, when to start, and when to stop?

- **Finish the prototype when you've learned what you wanted to know**

- **But resist jumping into the coding phase before you're really ready**