INTRODUCTION TO SOFTWARE ENGINEERING

the need for software engineering

1. What are the origins of software engineering?
   " Software engineering was born in the late 1960s as a result of the
   "software crisis".
   " Third generation computers (e.g., IBM 360) accelerated the crisis by
   greatly increasing the power of computers and access to computers.
   " Many "software metrics" are discouraging to say the least.

2. For example, a famous study by the Comptroller General of the United
   States (1979) presented the following metrics relating to US federal
   software projects. The Comptroller General found that:
   " 2% worked on delivery
   " 3% worked after some corrections
   " 45% delivered, but never successfully used
   " 20% used, extensively reworked, and eventually abandoned
   " 30% were paid for, but never delivered.

3. [Discussion of “two views of software failure”. Standish Report can be
   found at www.cs.nmt.edu/~cs328/reading/Standish.pdf ]

4. Another impetus for software engineering are software engineering
   disasters. There are numerous examples. These include:
   " Boeing 757 crash in Colombia several years ago (cockpit design; human
   factors)
   " Boeing 737 crash in England (cockpit design; human factors)
   " Mariner I space probe: destroyed on launch due to programmer error
   " Recent Mars probe failed due to a confusion between metric system
   and English system
   " Airbus 320 accidents (controversy about fly-by-wire)
   " Aegis system shoot-down of an Iranian airbus
   " Paris police sent out 10,000 traffic citations accusing people of
   serious crimes.
   " Y2K fiasco
   " CONFIRM fiasco
   " Therac-25 accidents

1
real world software

5. Here are some characteristics of real world software that distinguish it from the kind of software that is developed in academia (e.g., as homework assignments). Real-world software is likely to have:
   - naive users
   - formal business contracts
   - lifetime in years
   - software that is likely to evolve and to change
   - development in teams, not by individuals
   - careful specification prior to product design
   - need for user-friendly interfaces
   - need for various forms of documentation (including user manuals)
   - individual creativity may be frowned upon (i.e., egoless programming) [e.g., The “You touched it last” syndrome”].
   - software that is incredibly complex as compared to academic software
   - formalized product reviews and quality assurance measures
   - formalized, rigorous testing procedures
   - product may need to run on different platforms
   - reliability may be a matter of life or death; or the cost of failure might lead to an economic catastrophe
   - issues of security may be extremely important [CIA = Confidentiality, Integrity, Availability]
   - ethical issues (like protecting privacy) may be extremely important
   - many different kinds of people might have a vested interest in a product, its costs, and its efficacy (vendors, clients, users, the penumbra)

6. The above list suggests the need for engineering techniques in the production of software. But, traditional engineering practices do not necessarily transfer over to the production of software.
   - software is not manufactured in the traditional sense
   - software costs are concentrated in engineering (development) and not in production
- software does not wear out (in the physical sense)
- software has no replacement parts
- software maintenance is a difficult problem
- much software is custom-built
- the causes of software failure may be difficult to detect or to remove
- estimating the probability of failure can be very difficult
- many legal issues have yet to be resolved (e.g., intellectual property rights, liability)

the concept of quality

7. Here is a list of eleven qualities that we would like to find in software. This is an attempt to describe "quality software":
   - correctness (it satisfies its specifications)
   - timeliness (it is developed in time to benefit various constituencies)
   - robustness (it can function under abnormal situations)
   - reliability (this is defined to be correctness + robustness)
   - adaptability (graceful evolution; the cost of maintenance is acceptably low)
   - reusability (software components can be utilized in other projects)
   - interoperability (software can be combined with other software systems in a convenient manner)
   - efficiency (various cost metrics are acceptably low)
   - portability (can be moved to new hardware platform with acceptably low cost; can be moved to a new compiler...)
   - security (cannot be accessed, modified, or damaged by unauthorized users)
   - efficacy (complete and efficient functionality relative to the problems that the system is designed to solve)
   - user-friendliness (rewards of using the system are commensurate with the effort required to both learn and to use the system in practice)

8. Different constituencies will have different perspectives on what constitutes "quality". For example, vendors and clients might differ about user-friendliness, or timeliness.
9. Typical constituencies (each with their own perspective on a software project) include:
   - vendors
   - clients (sponsors)
   - users
   - developers
   - maintainers
   - the penumbra ("stake-holders")

10. Discussion of constituencies versus qualities handout.

   why is software development so difficult?

11. User expectations: User expectations increase as the technology becomes more and more sophisticated.

12. Communications: Communications among the various constituencies is a difficult problem. Sometimes different constituencies speak completely different languages. For example, developers may not have the domain knowledge of clients and / or users. Among team members, the number of communications paths varies as n-squared (where n is the number of team members). Thus, the larger the project, the more difficult the communications problems become. [e.g., the Confirm fiasco].

13. The mythical man-month factor: Adding personnel to a project may not increase productivity. Adding personnel to a late project will just make it later. Quoting Brooks: "There is no way a baby can be produced in one month by using nine women."

14. Project characteristics:
   - size / complexity
   - novelty of the application
   - response-time characteristics
   - security requirements
   - user interface requirements
   - reliability / criticality requirements
hardware complexity (e.g., is this a networked application?)

15. Characteristics of individual team members:
   - Ability to perform work (productivity / quality). E.g., one study showed a 25:1 ratio between the most productive and the least productive programmers.
   - Prior experience or training (knowledge of application area).
   - Ability to communicate (speaking, listening, writing)
   - Ability to work in a group (50% of a programmer's time is spent interacting with team members)
   - "Problem programmers"
   - Managerial issues (corporate culture; for example, commitment to quality)


17. Barry Boehm, a famous software engineer, lists the following basic skills as highly desirable in entry-level programmers. They should have the ability to:
   - express themselves clearly in English
   - develop and validate software requirements and design specifications
   - work with application areas (domain knowledge)
   - perform software maintenance
   - work with suitable standards and project management techniques
   - work in groups

   programmer psychology

15. In the Killer Robot scenario (onlineethics.org/cases/robot/robot.html) programmer psychology is a major issue. These notes summarize facts about programmer psychology. These notes will emphasize three desirable characteristics for a computing professional:
   a. commitment and ability to learn new technologies
   b. ability to handle stress
   c. ability to work in a group (leading to "ethics of speech" considerations)
16. Bass and Dunteman (1963) in an influential paper on group dynamics, identified three base personality types (in a team project context). Here are the three personality types that they identified and the self-descriptions of team members of each type:
   a. task-oriented: self-sufficient, resourceful, aloof, introverted, not aggressive, competitive, independent
   b. interaction-oriented: low need for autonomy and achievement; not aggressive, considerate, helpful
   c. self-oriented: disagreeable, aggressive, competitive, introverted jealous, dogmatic

17. These three groups are more generally characterized (from outside) as follows:
   a. task-oriented: motivated by intellectual challenge
   b. interaction-oriented: motivated by desire for human interaction and involvement with others
   c. self-oriented: motivated by desire for personal success

18. Most programmers are task-oriented. Bass and Dunteman found that heterogeneous groups performed better than homogeneous groups. Thus, a group which consists just of task-oriented people might experience difficulties that a mixed group might avoid.

19. Here are some other findings concerning programming teams:
   a. it is best for team members to be involved in every stage of the project
   b. group structures should not be allowed to determine the software structure (e.g., don't design a four-pass compiler because you have four team members)
   c. small groups are more desirable than large groups; superfluous team members must be avoided.

   egoless programming

20. Weinberg introduced the idea of "egoless programming" in 1970. Since then, he has been a major voice in the study of software productivity and team work.
21. The basic idea behind egoless programming is to consider a software project as a product of a group and thus to lessen individual attachment to one's work. Egoless programming implies the following attitudes:
   a. errors are normal and are to be expected (thus, do not avoid exposure)
   b. frequent quality assurance reviews in a group setting are important
   c. emphasis on co-operative effort and a collective goal

22. Henry Ledgard has expanded upon Weinberg's ideas in a series of books on team work in the software industry. Ledgard distinguishes between a group (where egos are at work) and a team (where egoless programming is being practiced). According to Ledgard, a group is characterized by:
   a. individualism
   b. drifting responsibilities
   c. frequent organizational meetings (that don't accomplish anything)
   d. members who have other responsibilities
   e. members who work in relative isolation

23. In contrast, he characterizes a team as follows:
   a. a collective ego
   b. a collective goal
   c. individuals with assigned tasks and no "responsibility drift"
   d. frequent quality reviews with few organizational meetings
   e. members whose fundamental responsibility is to the team's project
   f. frequent work reading (quality assurance reviews)

24. Ledgard lists the following problems that can arise within a team; problems that need to get fixed:
   a. responsibility drift (a sure sign that a team is heading for dissolution and disaster)
   b. talent overload
   c. negativism (the "sniper")
   d. superiority (the "prima donna")
   e. bickering (lots of snipers, the end is near)
   f. singling out (the "scape goat")
   g. complaining
   h. boasting
   i. crisis ("war room mentality")
j. dictatorship
k. group think (overly cohesive group; everyone thinks alike)

team organization and structure

25. There are various ways in which a team can be organized. Three important possibilities are:
   a. democratic team
   b. hierarchical team
   c. chief programmer team (requires a “superstar” programmer)

26. Democratic teams are viable if:
   a. the team is small
   b. the goal is readily attainable
   c. members are equally talented
   d. individual members can engage in give and take

27. Dangers of a democratic team:
   a. responsibility drift
   b. time wasted in the democratic process
   c. individual responsibilities may be too vague or may be difficult to enforce

28. A hierarchical team may have a structure such as the following:
   a. Top level: project leader
   b. Middle level: technical leaders
   c. Bottom level: 5-7 staff members (including programmers) under each technical leader

29. Advantages of a hierarchical team:
   a. centralized decision making
   b. well-defined team structure
   c. technical leader and his/her staff may be fairly democratic (which leads to high job satisfaction)

30. Disadvantages of a hierarchical team:
   a. Project leader and / or technical leaders may not be capable of delegating authority.

8
b. Low morale among bottom level staff if they do not have access to the big picture and if they feel like second class citizens

31. Harlan Mills introduced the idea of the "chief programmer team" that was used on several famous software projects. This requires a "super programmer" with great technical skill. This approach is generally used on large projects only. A chief programmer team might be organized with the following kinds of personnel:
   a. the chief programmer, who
      o makes all critical technical decisions,
      o implements critical software components
      o does most of the system design work
      o is responsible for delegating work
   b. an assistant programmer who backs up the chief programmer (no one is immortal)
      o liaison with client and other outside groups
      o may do some design / code work
   c. 2-5 programmers
      o design, implement, and test source code
   d. administrator who performs day to day administrative tasks
   e. librarian who maintains system documents
      o program listings, documentation, test plans, test runs
      o configuration management problems
   f. toolsmith who writes specialized tools that team members need

32. A chief programmer team has centralized decision making and a well-defined team structure, but low-level programmers may not have much job satisfaction.

some notes on stress

33. Many factors are making work in information technology more stressful than ever:
   a. change is stressful, and we are in a period of rapid technological change
   b. there is a shortage of qualified workers (although this is controversial), so employers are trying to induce workers to work enormous hours; the personal cost can be very high
c. the complexity of computer systems and the costs of failure are increasing

34. The following quote from Chogyam Trungpa is helpful: "For the warrior, every moment is a challenge to be genuine, and each challenge is delightful. When you let go properly, you can relax and enjoy the challenge." [From Shambhala: The Sacred Art of the Warrior]

35. Here are some recommendations for reducing stress:
   a. set and negotiate reasonable goals
   b. maintain the attitude of a student (a humble and open attitude)
   c. be wary of perfectionistic tendencies
   d. find effective techniques for controlling anger and hostility (e.g., reframe your situation)
   e. get plenty of exercise
   f. avoid the use of tobacco, drugs, alcohol, and food as a mood regulator; use exercise instead
   g. get adequate sleep
   h. investigate the value of stress-reduction methods, such as prayer, meditation and yoga
   i. nurture close and supportive relationships
   j. reserve time for pleasurable activities
   k. be assertive when assertiveness is called for [apparently, people who live to be 100 have this characteristic]
   l. develop time management skills
   m. know yourself

   software process models

36. There are two interpretations for the phrase "software process model:
   o descriptive (a description of how software is created based upon real-world studies)
   o prescriptive (a suggestion as to how to create software)

37. Some of the standard software process models are:
   o the life-cycle or waterfall model [What is the fundamental problem with this approach.]
38. In fact, there are many software process models, but these are the most important general models. Many models are variations on the fundamental ideas contained in these models.

39. The life-cycle model describes a software project as consisting of the following steps:
   - requirements analysis and specification
   - design
   - implementation
   - testing
   - adaptation (maintenance)

40. In practice, these steps are not sequential. There are feedback loops from latter steps to earlier steps. An important consideration is that errors found late in the process are more expensive to fix than those found early on.

41. Studies have shown that the adaptation stage (often called maintenance) is the most expensive. Here are typical metrics relating to the relative costs of the various stages:
   - analysis and design 13%
   - implementation 7%
   - testing 13%
   - adaptation 67%

42. What kind of inferences can you make from the above data?

43. A major drawback of the life-cycle model is that clients and users will not get to see a working version of the system until near the end of the development process. The very act of playing with a system often suggests new possibilities to sponsors and to users. However, at this late stage, the clients have already signed off on the requirements in the form of a formal business contract.
requirements analysis and specification

44. Requirements analysis and specification is the process by which sponsors and developers reach a formal agreement concerning the exact nature of a software system. The output (or, deliverable) is a requirements document that becomes part of a contractual agreement between clients and developers.

45. In general, this process is very difficult and far from straightforward. Conflicts may arise among various groups and constituencies (see for example the article by Walz et al. in our coursepack).

46. Recall that incomplete requirements were a major factor in "impaired projects" according to the Standish Report.

47. A requirements document typically describes:
   - goals: these are neither observable nor quantifiable; inspirational
   - functional requirements: these relate to the behavior of the system and the services provided
   - non-functional requirements: these describe constraints under which the system will be operating

48. One property of a requirement is that it is either satisfied or it is not. One must be able to state clear criteria for deciding whether a given requirement (whether functional or non-functional) is satisfied or not. Otherwise, it is not a requirement.

49. Requirements must be stated in a language that both clients and developers can understand. Usually, this is a mixture of natural language and some diagrammatic notations.

50. Non-functional requirements include:
   - hardware issues (platform, operating system, network environment)
   - security issues
   - performance issues (e.g., response times)
   - interoperability issues (ability to integrate with other software)
   - human factors issues (ease of use; ease of learning)
software design

51. Software design involves developing the architectural (structural) framework for the proposed system. A design describes a system in terms of software components. Architectural design is higher level than detailed design, which deals with the inner structure of software components. Some authors consider detailed design an aspect of implementation (see next section).

52. In an object-oriented system, architectural design will involve classes and their relationships, although higher level architectural components are being introduced into some systems. Detailed design will include algorithms (but not programs) and more detailed decisions concerning data structures.

53. Here are some design objectives for software components (e.g., classes) taken from Bertrand Meyer's *Object-Oriented Software Construction*. Software components should:
   - correspond to syntactic units in the implementation language (like classes in C++)
   - be intelligently coupled (e.g., global variables are often considered a poor form of coupling)
   - be highly cohesive (e.g., each function should have a single purpose <this is called functional cohesion> or its component tasks should be related to a common data store <communicational cohesion>.
   - support information hiding
   - have explicit interfaces
   - have clear interface specifications (pre- and post-conditions)
   - be designed with software reuse in mind.

54. The above considerations generally apply to functions as well as to classes (in an object-oriented system).

implementation

55. Implementation is generally the least time-consuming stage of the software life-cycle. It involves:
o the choice of an implementation language (although this might be specified in the requirements)
o detailed design (implementing the functions; some consider this part of the design process)
o coding (writing the program)
o unit testing (testing software components as they are generated)

[As we shall see, unit tests are coded first in XP]

56. Here are some desirable features for an implementation language (based upon Meyer, cited above):
o elegance and simplicity
o defense in depth
o portability
o separate compilation
o data abstraction (information hiding and encapsulation)
o support for generics (templates in C++)
o support for software reuse

testing

57. The goal of software testing is to detect as many defects as possible in a system before it is delivered, installed, and made operation. The goal is not to prove that a program is bug-free, since this is a theoretical impossibility (e.g., the halting problem).

58. Testing is one aspect of validation and verification:
o Validation: Are we building the right system?
o Verification: Are we building the system right?

59. Validation: Is this the system that the customer wants?

60. Verification: Assuming that we know what the customer wants, will our system meet the customer's specifications?

61. Validation and verification activities occur throughout the life cycle. These take the form of quality assurance reviews of various kinds:
o requirements reviews
o design reviews
- code reviews (done by individual programmers on their own code)
- peer reviews (or code inspections)
- structured walkthroughs (one formal approach to code inspections)
- informal program verification activities

62. Testing can either be:
   - static testing: tests that do not involve the execution of the program; for example, the analysis of the code by a static analyzer
   - dynamic testing: tests that involve executing the software using suites of test data

63. Dynamic testing takes two very general forms:
   - white box: design test suites to traverse every path in a flowchart of the code; this requires a detailed knowledge of program structure and the coding details.
   - black box: design tests using knowledge of the kinds of data that might cause trouble, but without any knowledge of the program structure.

64. Testing can either be:
   - bottom-up: test lower level routines, and then integrate
   - top-down: test upper level routines with subroutine stubs

   maintenance

65. Maintenance involves activities that seem to replicate earlier life cycle activities:
   - analyzing requirements, because they might change
   - evaluating architectural design in view of changing requirements
   - coding (writing new code or modifying existing code)
   - testing changes
   - updating documentation

66. Maintenance requires detailed knowledge of program structure and of the code itself.
67. Some experts suggest that development teams and maintenance teams should be separate since this encourages more careful documentation and design.

68. The maintenance team consists of analysts and programmers. A lot of the work of maintenance involves interacting with clients (sponsors and users) to see what is needed. The main responsibilities of these responsible for maintenance are (based upon Shari Pfleeger, *Software Engineering*):

- to understand the system
- to locate information in the system documentation
- to keep system documentation up to date
- to extend existing functions to accommodate new or changing requirements
- to add new functions to the system
- to find the source of errors in the system
- to correct errors identified in the system
- to answer questions about the way the system works
- to restructure the design
- to rewrite the code
- to delete modules that are no longer useful
- to manage changes to the system as they are made

69. Four fundamental kinds of maintenance have been identified (Pfleeger):

- corrective maintenance: fixing problems as they arise from day to day
- adaptive maintenance: fixing problems caused by other attempts to fix programs
- perfective maintenance: responding to changes in the environment or attempting to improve the system performance
- preventive maintenance: proactive prevention of errors and/or crashes

70. Here are some metrics on the percent effort involved in the four types of maintenance (Pfleeger):

- perfective: 50%
- adaptive: 25%
- corrective: 21%
- preventive 4%