CS211 Lecture: Course Intro; Introduction to Software Engineering

last revised July 28, 2004

Objectives:

1. To introduce the course requirements and procedures.
2. To set programming in the larger context of software development/engineering.
3. To introduce the Software Engineering Code of Ethics
4. To introduce the software lifecycle
5. To introduce various software development models (build and fix, waterfall, incremental)
6. To introduce the notion of a design methodology as embracing a process and a notation.

Materials:

1. Syllabus
2. Term project requirements handout
3. Transparency from Gilbert and McCarty: Figure 2.10, 2.11 on same page
4. Transparencies from Gilbert and McCarty: Figures 2.12, 2.13, 2.14
5. Transparencies from Schach: Figures 3.1, 3.4
6. Transparencies from Lethbridge Figures 11.1, 11.2, 11.3, 11.4

I. Preliminaries

A. Distribute, go over syllabus.

B. Distribute, go over term project. Encourage students to begin thinking about team formation now - note Preliminary Milestone due date.

C. Writing exercise - have half of the class do 1 and half do 2:

1. Suppose you have asked someone to develop a piece of software to meet a specific need that you have. How would you decide whether they’ve done a quality job?

2. Suppose you were asked by someone to develop a piece of software to meet a specific need that they have. What process would you follow (what steps would you perform)?

Discuss class answers to each
II. Software Engineering

A. This course is a continuation of CS112; but it will differ from CS112 in some important ways.

1. In CS112, a great deal of your mental energy was invested in learning about programming, and more specifically in learning how to program in Java.

2. While programming in Java is part of the content of this course too, we want to step back from programming per se to take a broader approach to software development - an entire process of which programming is just one part. Software development is the process of producing a product that will fulfill some real need for someone. It includes, therefore, matters like:

   a) Understanding the need that is to be met. (Note: in terms of loss of money and time, the worst software disasters are the products that do exactly what they were designed to do - except that what they were designed to do wasn't really what was needed!)

   b) Designing a user interface that will make the software easy and pleasant to use, and has documentation and/or help facilities of various sorts to help someone learn to use the software.

   c) Being sure that the software is correct and doesn't do unexpected things (like crashing)

   d) Being sure that the software can be changed to keep up with changes in the user's needs.

   etc.

B. Developing quality software is not easy.

1. Software systems are among the most complex systems ever attempted by humanity. There is still much to be learned about how to do this well.

2. Most large-scale software projects exhibit one or more of the following problems to an unacceptable degree:

   a) The software is delivered late.

   b) The budget is exceeded.
c) The software contains undetected errors. (Note: these are commonly called "bugs". Dijkstra has pointed out that calling them bugs rather than errors is a way of avoiding taking responsibility for them.)

d) The software is difficult to maintain/modify - fixing one error often introduces two more.

e) The software does not really meet the user's needs.

f) The software is hard or confusing to use.

C. The discipline that seeks to address these problems in a systematic way is called software engineering. Its goals are to:

1. Produce software that meets the needs of users

2. Produce correct software on time and on budget.

3. Produce software that can be maintained and modified to keep abreast of changing needs. For software that is used over a period of years, the cost of keeping it current in the face of changing needs often exceeds the cost of originally developing it.

Meeting these goals is not easy, and probably never will be, because the complexity of modern software makes its development one of the greatest intellectual challenges ever faced by humanity. However, applying known principles can help.

D. Software Engineering compared/contrasted with more traditional engineering professions.

1. Software is certainly not like physical engineered artifacts.

How? (ASK CLASS)

a) For most physical artifacts, the bulk of the cost is in the manufacturing, not the design. For example, if one builds a bridge and then attempts to build another just like it, the second bridge costs almost as much to build as the first. However, “manufacturing” software is cheap - the cost of producing a new copy (say on a CD) is miniscule.
b) Most physical artifacts are costly to change once they have been produced; but making changes to a piece of software is often a matter of editing and recompiling. (Of course, making correct changes is not necessarily easy!).

c) Physical artifacts wear out and need to be maintained or ultimately replaced - but software never wears out.

d) It is often possible to tell, by looking closely at a physical artifact, that it is defective. Faults in software are often much less obvious until they manifest themselves in some sort of error.

e) A key difference is reflected in the existence of the “open source” movement. Open source software is software whose source code is made publicly available; in general, one who acquires open source software is free to modify it to suit his/her own purposes (often with the proviso that he/she share these modifications with the wider community.)

(1) For example, Linux is an open-source operating system, and much of the software designed for Linux platforms is open source. The same is true of the kernel of Mac OS X (Darwin).

(2) OTOH, Microsoft has been a leading opponent of open source software.

   It’s hard to imagine an equivalent to open source in the more traditional engineering disciplines - the last thing anyone would want is thousands of people making individual modifications to a bridge! However, proponents of open source point out that such software is often more reliable, because many eyes looking at the code find more of the problems. (Linux is a much better operating system for servers than Windows products, IMO)

f) A profound - and subtle difference - has to do with mathematical foundations.

(1) Continuous mathematics - the calculus - is the mathematical foundation of traditional engineering.

(2) However, discrete mathematics is really the foundation of “software engineering”.

(3) In this distinction lies a profound difference between failure modes of the two entities. Physical systems often have slight
errors; catastrophic failure is relatively rare. Software systems are prone to crashes, or total failures.

2. There is a movement within the software field to move software engineering to be like other engineering disciplines - including licensing of software engineers just as other engineers are licensed.

   a) This is more pronounced in Europe than in the U.S.; but in our country Texas has already set up a process for licensing software engineers. (Note that our textbook reflects something of a European perspective - its publisher is McGraw-Hill of the UK)

   b) OTOH, there are those who argue that going in this direction is a mistake; and there are those within the software field who argue that producing software is more of a craft than an engineering discipline.

   c) The ACM - the major professional organization in the field of Computer Science - has gone on record as opposing licensing of software engineers as engineers, in part because of the differences between software development and traditional engineering. (See Communications of the ACM 45.11, November, 2002, p. 91)

3. One key characteristic of any profession is the expectation that its practitioners will perform their work in accordance with ethical expectations appropriate to the profession. In the case of Software Engineering, those ethical expectations have only been recently formalized in a document developed jointly by the ACM and the IEEE/CS, entitled “Software Engineering Code of Ethics and Professional Practice”.

   a) This is an assigned reading in the syllabus.

   b) The fact that this document is very recent (1998) is evidence of the relative youthfulness of software engineering as a profession like the other engineering disciplines, or professions like law or medicine.

4. Despite the differences, there is much to be learned from other engineering disciplines about the process of producing quality software - though I would resist the notion that software engineering is just another form of engineering.

E. There are several key sets of terms used to describe different kinds of software projects.
1. The text noted that software can be classified into three broad categories

   a) **Custom** software is developed to meet the requirements of a specific user. Normally, a given piece of custom software is used only by one company. (Example: the software that manages the financial records of a corporation.)

   b) **Generic** software is developed to meet the perceived needs of a market, and is sold to many users. (Example: a particular word-processor or spreadsheet package or game). Sometimes this is called COTS, which stands for Commercial Off-the-Shelf Software.

   c) **Embedded** software is incorporated into some other product, and everyone who purchases the product also ends up using the software, though probably without being aware of it. (Example: a DVD player or the ABS system on a car)

2. Another classification is to classify software as:

   a) **Data-processing** software is used to support the various business functions of a company. This includes systems like payroll systems, customer order systems, airline reservation systems, etc. etc. Often, some or all of the processing can be done without direct interaction with a human user, so response time is not critical.

   b) **Interactive** systems involve direct interaction with a human user (e.g. computer games, word-processors, etc.)

   c) **Real-time** systems control mechanical systems, and must respond to events within tight time constraints. (E.g. you would be very unhappy if your car took 5 seconds to start to accelerate after you press the gas pedal.).

      (Note: sometimes these are called hard real time systems, and the term soft real time systems is used for interactive systems where the time constraints aren’t as inflexible.)

3. The text also noted that software **projects** are of three general kinds (p. 13):

   a) “Those that involve modifying an existing system” - **evolutionary projects.**
b) “Those that involve starting to develop a system from scratch” - *Greenfield projects*

c) “Those that involve building most of a new system from existing components, while developing any missing details” - *framework projects*.

Though there are profound differences between these three types of projects, the same basic approach is used for each.

4. The text also noted that, in any software project, a key concept is the notion of a stakeholder. A stakeholder is someone who has a legitimate stake in the outcome of the project.

a) The book identified four types of stakeholders.

(1) Users - those who will eventually use the software.

(2) Clients - those who decide to have the software developed, and pay for doing so.

(3) Developers - those who actually produce the software.

(4) Development managers - those who oversee the work of the developers.

b) For different kinds of software projects, there may be different relationships between these categories of stakeholders - *e.g.*

(1) The users of the software may be the same as the clients - or may be employees of the client. (*E.g.* in the case of the software Gordon uses for registration, billing etc. Gordon is the client, but faculty and staff are users.)

(2) The users of the software may be customers of the client - *e.g.* if a firm uses an e-commerce web site, it is the client and its customers are the users.

(3) The developers may be part of the client organization, or may be contracted by the client to produce the software for them, or the client may purchase “off the shelf” software that the developers have produced for a market they believe exists.

(4) In some cases, one individual may be user, client, developer, and development manager for a project - *e.g.* if you or I write software for our own personal use.
III. The Software Lifecycle

A. The process of developing any piece of software involves a number of activities - though not necessarily performed in a rigid sequence.

1. TRANSPARENCY: Gilbert and McCarty Figure 2.10 (leave 2.11 covered)

2. These activities are, in fact, common to any process that results in producing a product.

TRANSPARENCY: Figure 2.11

B. The major activities involved in any software development project include the following:

NOTE: Leave transparency of figure 2.10 up

1. Requirements analysis - the goal of this activity is to understand the need - to determine what is needed.

a) It is very easy to get this part wrong. Some of the worst software disasters that have occurred in the industry have resulted from misunderstanding of what is really needed.

TRANSPARENCY: Figure 2.12

b) Often times, this activity is considered to be two separate activities

(1) Requirements elicitation. Sometimes this leads to the creation of a specification for the software - which is a formal statement of what the software will do, and may serve as a legal contract between the software developer and the client. (This is particularly the case with custom software; rarely true with generic software.)

(2) Analysis proper

c) We will talk about analysis some in this course, but it will not be a major emphasis. Industrial-strength analysis requires expertise both in software development and in the problem domain - e.g. doing analysis for a particular business domain requires business expertise in that area; doing analysis for software to be used to control laboratory instruments requires scientific expertise, etc. We will discuss analysis activities as a single unit.
2. Design - the goal of this activity is to determine how the requirements are going to be met. Design is a broad area that encompasses a large number of issues, like:

a) System design is typically part of creating an embedded system - the partitioning of functionality between hardware and software. (This is usually not an issue with custom or generic software.)

b) User interface design is typically part of creating custom or generic software - how will users interact with the software? (This is usually not an issue with embedded software.)

c) Software structure (architecture) - how will the overall task be broken up into component parts?

d) If the software uses a database, then the database will need to be designed. (This is primarily an issue with data-processing software)

e) Design (called detailed design) is also a part of the next activity. Design will be a major emphasis of this course

3. Implementation - translating the design into reality. This involves:

a) Detailed design of the individual components identified in the overall design phase (e.g. the individual classes in an object-oriented design)

b) Coding the design in a suitable language (e.g. Java).

This was the focus of CS112, but more advanced programming concepts will also be considered in this course.

c) Testing each component as it is implemented

d) Integrating the various components together, and testing the result.

e) Note that our book calls this phase Programming. Programming, rightly understood, includes all of the activities I have listed for implementation, but is sometimes misunderstood as referring just to coding. I have used the term implementation to stress that much more than just writing code is involved here.
4. Quality Assurance - also known as Verification and validation - ensuring that the resulting software is built correctly (verification) and does the right thing (validation).

a) Sometimes this activity is called “testing”. While testing is a major means of doing verification and validation, it is not (and should not be) the only means of doing verification and validation.

b) We have listed it as a distinct phase, because there needs to be a time of extensive testing near the end of most projects. However, the fundamental concepts of quality assurance should pervade the entirety of a project.

5. Deployment / Maintenance

a) Once the software is delivered by the developer to the client, it is put into use by the users. Frequently, this leads to the discovery of the need for changes.

b) Software maintenance refers to the activity of modifying an existing piece of software. Maintenance is of three general types:

(1) Corrective maintenance - fixing errors that were not caught before the software was delivered - i.e. to make the software fulfill its original requirements.

*Example:* The program crashes or freezes when a certain feature is used in a particular way, or the result produced by a certain operation is incorrect or incomplete

(2) Adaptive maintenance - dealing with changing requirements. As a piece of software is used over time, external changes in the environment in which it is being used may change the tasks the software is required to perform

*Example:* Tax return preparation software must undergo adaptive maintenance whenever the tax code changes (i.e. most election years!)

(3) Perfective maintenance - adding new features not part of the original release, or improving the user interface.

c) Note that software maintenance is quite different from hardware maintenance.
(1) Software doesn't wear out. (There is no such thing as “bit rot”).

(2) The purpose of maintenance of a mechanical device such as a car is to bring it back to its original condition when delivered. Software maintenance involves improving the condition of the software in some way.

d) Note that the book referred to this activity simply as “deployment” because significant maintenance often gives rise to a whole new project.

6. Though not properly an activity in the creation of software, we should note that there is an end to the lifecycle for any given piece of software, commonly called retirement or obsolescence, when a particular piece of software is no longer maintained and stops being used. This occurs when either

a) The original need for the software no longer exists

b) It is expedient to develop a whole new piece of software rather than continuing to maintain an old piece of software.

7. The book included “modeling” as a distinct activity between design and implementation. Really, though, modeling is part of both the analysis and design portions of the life-cycle, not a distinct activity.

C. These activities are not watertight nor are they done in a rigid sequence.

1. Some of the things we need to do may involve elements of both analysis and design, or of both design and programming

2. Verification and validation needs to be done throughout the entire process - not saved for the end.

3. With OO approaches, in particular, the line between analysis and design tends to become a bit blurred, as we shall see.

D. Your semester project will involve you in activities of all six kinds

1. A fair amount of the requirements analysis work has already been done for you - there is a quite detailed statement of the system requirements in the project document. (In a real project, requirements elicitation would be a major activity early in the project.)
2. Use case development (milestones 1-1, 2-1) is an analysis activity.
3. The development of a class diagram (milestone 1-2) is a structural design activity.

4. Much of the user interface has been designed for you. However, you will do some user interface design for milestone 1-3.

5. CRC cards (milestones 1-2, 2-1) and the creation of sequence diagrams (milestones 1-3, 2-1) are design activities.

6. Coding (milestones 1-4, 2-2, and 3-2) is an implementation activity.

7. Test plan creation and execution (milestones 2-3, and 3-2) are verification and validation activities.

8. Your software will no doubt be retired to a hallowed spot on your server volume when the end of December rolls around. (I don’t actually have a contract to sell your projects to a local video store and get enough money to go to Bermuda for Christmas break :-) ) As a result, you won’t have an opportunity to perform maintenance activities. However, working on someone else’s project for iteration 3 will give you experience with a key issue involved in maintaining someone else’s work - namely figuring it out!

IV. Software Lifecycle Models

A. Although the activities we have listed need to be part of any well-done project, there are many different models of how these activities relate to one another.

B. One approach to software development, which I am not advocating, is called build and fix

1. TRANSPARENCY - Schach Figure 3.1

2. At one point in time, this was the prevalent way that software was built in the software industry. This approach worked somewhat successfully when computer memory was small and programs were of inherently limited size. But as memory capacities grew and programs became larger, this approach broke down rapidly.

3. Build and fix is still the way most student computer programs are developed - at least in introductory courses. This was not a bad approach for CS112 - but hopefully you will learn a much better way of doing things in this course.
4. Your book discussed a somewhat similar (and also undesirable) approach called the *opportunistic approach*.

**TRANSPARENCY - Lethbridge Figure 11.1**

C. The model that replaced build and fix is called the *waterfall model*.

1. In this model, the various activities are sequential. Each is done to completion before we move on to the next activity; and once we move on, we avoid going backward if at all possible (just as water goes down a waterfall but never goes back up.)

**TRANSPARENCY: Gilbert and McCarty Figure 2.13**

2. Actually, it is not possible to carry out a significant software development process with a totally one-way flow of activity. The author of our text gives a diagram that shows the reality of the waterfall model more clearly:

**TRANSPARENCY: Lethbridge Figure 11.2**

3. The waterfall model has both advantages and disadvantages.

   a) A major advantage of the waterfall model is that it is efficient. Changing a requirement is relatively inexpensive during the requirements analysis phase; more expensive during the design phase; and potentially extremely expensive in the later phases.

   b) A major disadvantage of the waterfall model is that it is often very hard to fully understand the requirements for a piece of software early in its development. Missed requirements are quite common, even when an effort is made to do a thorough job of requirements analysis before moving on to the next phase.

   c) A second disadvantage of the waterfall model is that nothing is available for use until the end of the process, which can one or more years from start to finish. This can be years!

4. The waterfall model is sometimes called the “traditional waterfall model”, because there was a time when this model was strongly advocated as *the* right way to produce quality software. It was, at the time it was introduced, a major advance over the prevalent “build and fix” approach.
D. Recognizing the difficulty of fully capturing requirements, many projects are done now using *iterative, incremental development*. This is not a single model, but rather a family of models. In incremental models, after some initial analysis and design, a subset of the complete functionality is implemented, and then additional capabilities are added incrementally. (Indeed, in many cases the earlier version can be used even as it is being extended.) These models are called incremental because they grow the software by increments, rather than trying to implement everything at once; and they are called iterative, because they repeat the basic life-cycle activities for each increment.

1. *TRANSPARENCY*: Schach Figure 3.4

2. Variations:

   a) *TRANSPARENCY*: Lethbridge figure 11.3

   b) *TRANSPARENCY*: Lethbridge figure 11.4

3. Again, there are advantages and disadvantages to these models

   a) An advantage of an incremental model is that the client gets to begin making some use of the software fairly early, rather than having to wait for everything to be completed.

   b) Another advantage of an incremental model is that experience with using the first part of the software implemented can help to refine the requirements for subsequent parts.

   c) A significant disadvantage is that an incremental model can degenerate to build and fix or an opportunistic approach if the developer is not careful. The key lies in planning what features are to be developed for each increment.

E. Finally, a variant of incremental development that has attracted a lot of interest in the OO world is called Extreme Programming (XP - no relationship to Microsoft’s name for its new version of Windows!) I have placed links to a set of pages discussing XP on the course page. We will be using one of the key ideas of XP extensively in the course project: pair programming. (Pair programming did not originate with XP, but XP makes it one of its key components.)

F. What approach does the text develop?
ASK

The book develops an iterative approach.

G. Of course, the various lifecycle models we have looked at are not precise, well defined concepts, but rather reflect alternatives in a broad continuum of possibilities, both in terms of formality (emphasis upon a spelled-out style of documentation) and in terms of granularity (extent to which the whole project is done at once, rather than in smaller pieces.)

TRANSPARENCY - Gilbert and McCarty figure 2.14

H. What model are we using for the course project?

1. **ASK**

2. The model we are using is closest to the phased-release model described in the text, but not identical with it. We will also use some ideas from Extreme Programming - though not the XP model per se.

   a) The requirements have been specified up-front. This is not typical of real software development projects, but is inherent in the nature of a course project like this.

   b) The fact that you will only implement the requirements over several increments - and even then will only implement a subset of the requirements - is also characteristic of a phased-release approach - though, of course, in a real project you would implement all of the requirements.

   c) However, it is not the case that the result of the initial iteration is a usable system, since we won’t add the ability to create customers and rentable items until iteration 2! The phased release model normally produces a usable (albeit not fully-featured) system in the first iteration. The approach we are taking is a compromise dictated by pedagogical considerations.

   d) We will make use of pair-programming, which is a key element of the XP model.

   e) Another key idea in XP is “test first” - one develops test cases before implementing a requirement. This is explicit in milestones 1-1 and 2-1.
V. This course is primarily concerned with your learning a **methodology** of software design.

A. A methodology is a way of doing design, and consists of a **process** and a **notation**.

1. A process is a series of steps that you go through, starting with a vague idea of what it is you need to produce and ending with a complete piece of software. Each step along the way results in the creation of some model of the finished product, or a portion of it.

2. A notation is a language for recording the models.

3. Although a particular methodology embraces a specific process and notation, processes and notations are actually independent of one another - you can often use different notations with the same process, and different processes with the same notation.

   *Example*: An author who speaks several languages may use the same process for writing a piece of work, regardless of whether the work is being written in English, or French, or braille.

   *Example*: Different authors may use very different processes for producing a piece of work, even though the finished product is written in English in all cases.

B. There are actually many different methodologies one can use to produce quality design. Some of them are quite formal and elaborate, and some are quite informal. The methodology we will develop has both informal and formal elements to it.

1. We will develop an informal and simple design process in this course.

2. However, we will use a quite formal notation scheme known as UML (Unified Modeling Language) for recording our design work, and another formal notation scheme (Java) for actually producing our programs.

C. There is no such thing as “the right methodology” for software development. Different projects call for different methodologies, and often there are multiple methodologies that can be successfully applied for a given project. What is important, though, is to make use of some appropriate methodology, rather than just “hacking away to a solution”!