CPS122 Lecture: Introduction to Software Development; The Software Lifecycle

last revised December 13, 2019

Objectives:

1. To set programming in the larger context of software development.
2. To introduce the Software Engineering Code of Ethics
3. To introduce basic terms/concepts of Software Engineering
4. To introduce the software lifecycle
5. To introduce UML

Materials:

1. Exercises from chapter 1 of Lethbridge/Laganiere
2. Software Engineering Code of Ethics (online + handout of short form)
3. Projectable of Table 1.2 - page 6 in Britton and Doake
4. Projectable of "Tree Swing"
5. Projectable of Wheels Example Requirements - page 305 in Britton/Doake
6. Online access to requirements for AddressBook and ATM example systems.
7. Projectable of UML diagram structure

I. Introduction - A Disciplined Approach to Software Development

A. At the start of the course, we noted that this course deals with the larger context known as software development, of which programming is an important part - but by no means the only part.

   A disciplined approach to developing software draws on ideas from physical branches of engineering.

B. A key emphasis we will keep coming back to is the notion of <u>quality.</u>

   1. Before thinking specifically about software quality, let's consider the larger question - how do you assess the quality of any thing you buy? What do you look for?

      ASK

2. The reading discussed a number of attributes of quality.

   Discuss Exercise E-3 on page 13 of Lethbridge/Laganiere

3. Developing quality software is not easy.

   a) 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.

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

      (1)The software is delivered late.

      (2)The budget is exceeded.

      (3)The software contains undetected errors. (Note: these are commonly called "bugs". Edgar Dijkstra has pointed out that calling them bugs rather than errors is a way of avoiding taking responsibility for them.) It's not that some insidious problem crept into our code - it's that we made a mistake!

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

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

      (6)The software is hard or confusing to use.

4. While quality is an issue with any product of human design, it is a particular issue with software. We do not expect bridges or buildings to collapse - but we are not surprised when a piece of software "crashes". We would be unhappy if we had to shut off and restart our car in the middle of an interstate, but we get used to the idea of periodically rebooting a computer ...

C. A disciplined approach to software development is intended to address issues like this. Historically, this discipline drew heavily on ideas from the engineering of physical systems, and is sometimes called software engineering.

   Its goals are to:

   1. Produce software that really 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. However, in some profound ways software is certainly not like physical engineered artifacts.

   How? *(ASK CLASS)*

   1. 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 DVD or via download) is miniscule or almost nothing at all.

   2. 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!).

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

4. 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.

5. 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.)

   a) 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).

   b) 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 may eyes looking at the code find more of the problems. (e.g. Linux is a much better operating system for servers than Windows products, IMO)

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

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

   b) However, discrete mathematics is really the foundation of "software engineering".

c) 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.

7. 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 development is just another form of engineering.

E. As the reading pointed out, there are three broad categories of software - though the distinction is not hard and fast and not everything needly fits this sort of classification.

Discuss Exercise E-1 in Lethbridge/Laganiere p 6

F. There are also three broad categories of software development projects.

1. Greenfield projects start from scratch.

2. Evolutionary projects involve making changes to existing software.

3. Framework projects start with a framework that is extended or a set of components that are connected to meet a specific need.

II. Ethical Issues

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.

A. In considering the ethical ramifications of a decision, one crucial concept is the notion of a stakeholder. A stakeholder is someone who has a legitimate stake in the outcome of the project.

1. The assigned reading classified stakeholders into four major categories. What were they?

   ASK

   a) Users - those who will eventually use the software.

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

   c) Developers - those who actually produce the software.

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

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

   a) Relationship between clients and users

      (1) In the case of custom software

         (a) 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.)

         (b) Or, 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.

      (2) In the case of generic software, there may or may not be a client per se. A client may buy the software for its people to use (such as the site licenses Gordon has with Microsoft), or the users may buy the software from the developers directly.

(3) In the case of embedded software, the client may be the manufacturer of the product in which the software is embedded, and the users of the software may not even be aware that they are using it!

b) The developers may be part of the client organization, or may be contracted by the client to produce the software for them, or may sell generic software to a client or directly to users.

c) 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.

Discuss Exercise E-2 on page 11 of Lethbridge/Laganiere

B. In the case of Software Engineering, ethical expectations were formalized about a decade ago in a document developed jointly by the ACM and the IEEE/CS, entitled "Software Engineering Code of Ethics and Professional Practice"

Distribute short form handout of code of ethics - allow time for class to read

1. Comments/Questions?

   ASK

2. Who are the stakeholders in mind behind each of the points? Are there any in mind beyond the four we noted earlier? Why are they considered stakeholders?

   ASK

C. Show full form online (link from course web site)

The fact that this document is fairly recent (1998) is evidence of the relative youthfulness of software development as compared to professions like law or medicine.

III. Major Activities in Software Development

A. The development of a medium to large scale piece of software involves quite a number of different activities.

1. What do you think they include?

   ASK

2. Actually, different writers classify the activities slightly differently. This is the classification used by the authors of your main text.

   PROJECT Table 1-2 in Britton and Doake

3. What fraction of the total effort do you think is expended on the actual writing of code (programming, in the narrow sense)?

   ASK

   (About 1/6)

B. We will look at the individual activities briefly now, but in more detail over the course of the semester.

I will use a listing that is more fine-grained than what appeared in the reading.

1. Establishing Requirements - at some point in the process, it is vital is to spell out *exactly 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.

   PROJECT: "Tree Swing"

   b) Often, requirements are formalized in terms of some sort of requirements document that explicitly lists the requirements.

Sometimes this includes 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.)

(1)One of the strengths of the main textbook for this course is that it is built around a continuing example: a case study called "Wheels" based on software to support a bicycle-rental business. An example of a (fairly informal) statement of requirements for this system is found on page 305 of the book

PROJECT Britton/Doake page 305

(2)We will also be referring to two other online examples throughout the course. One is a fairly simple system for maintaining a personal address book.

PROJECT Requirements for AddressBook system

(3)The other online example is a bit more complex - software for controlling an ATM

PROJECT Requirements for ATM System

c) Both readings noted there is a spectrum in how requirements are approached.

(1)At one end of the spectrum is approaches that stress identifying as many as possible of the requirements for a project as early as possible.

A danger to this approach is that it is often not possible to fully identify requirements early - and requirements may change as the project is going along.

(2)At the other end of the spectrum are approaches in which requirements are identified incrementally - and new

requirements may become known as early versions of the project are used and users say "it would be nice if ...'

A danger to this approach is that significant design changes may be necessitated by newly-discovered requirements.

(3)The nature of the project may suggest where on this spectrum a given project may fall. A small to medium size project may benefit from the flexibility of a more incremental approach - especially if it is hard to get a handle on all the requirements up front. Large projects may call for identifying requirements early as much as possible, since design changes to support new requirements may have major costs.

2. Analysis activities focus on *understanding* the need

a) In object-oriented software development, probably the most important analysis activity is the identification of use cases, which are formal statements of how the software will actually be used, and which serve to drive the whole rest of development.

b) 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 this some, but not at great length.

3. Design - here the goal 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

4. The reading included an activity known as "modeling" - but actually various models are developed in conjunction with the other activities, as we shall see.

5. Implementation refers to actually translating the design into reality. In the case of software, 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 is what is sometimes called "programming" in the more narrow sense of the term.)

   c) Testing each component as it is implemented

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

6. Installation (what the reading called Deployment)

   a) Installation includes everything needed to support the use of the software by the users, including documentation and training.

b) For large systems running on multiple computers, this may involve deploying distributing the system to multiple places.

7. 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. Software maintenance refers to the activity of modifying an existing piece of software.

(1)Maintenance is of three general types:

(a)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

(b)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!)

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

(2)Note that software maintenance is quite different from hardware maintenance.

(a) Software doesn't wear out. (There is no such thing as “bit rot”).

(b) 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.

b) The table in the book book referred to this stage simply as “installation” because significant maintenance often gives rise to a whole new project.

8. 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 there is an activity at the end of development 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) The fundamental concepts of quality assurance should pervade the entirety of a project - not confined to a burst of testing at the end of development.

9. Though not properly a stage in the creation of software, we should note that there is an end of life 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.

IV. The Software Lifecycle

A. We will look at the activities themselves in more detail over the course of the semester. For now, we will focus on the notion of a systematic process for carrying them out.

It is important to keep in mind the distinction between the activities themselves and the process for carrying them out. At one point, it was common to think of these as discrete steps in the software development process. Though modern OO development approaches do not view them in this sequential way; it remains the case that there are certain things which need to be done.

B. Traditionally, the various activities have been organized using one of several software lifecycle models.

1. One approach is referred to - somewhat tongue-in-cheek - as "build and fix".

a) This is a totally non-systematic approach.

(1) One begins work on writing code almost right away. Some analysis and design may be done as needed, but it is not uncommon for someone to write code without really understanding what he/she is doing.

(2) Once a draft of the code exists, it is tested. Problems are identified and fixed - which in turn gives rise to new problems ...

(3) The process is continued until the product is judged satisfactory.

(4) The dominant attitude behind this approach is epitomized by the words of a project manager who was cited in a talk I

heard - “We’re going to have a lot of debugging to do on this project, so we’d better get started coding as soon as possible”

b) Of course, build and fix may be an appropriate model for small-scale or exploratory programming project. However, it quickly becomes a very bad idea for projects of any size - whether larger school projects or in “the real world”

c) Problems with this approach?

ASK

2. At the other extreme, one alternative is to follow a fairly strict sequence: first requirements are identified, then analysis and design are done, then the design is implemented, and then the finished product is tested (though testing is done throughout the process as well).

a) In this model, the various things that need to be done are regarded as sequential steps. 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. This approach is often called the *waterfall model*, because just as water goes down a waterfall but never goes back up, so the process aims to carry out each stage and then move on.

b) Though this approach can be very efficient, it runs into some serious problems of its own. What are they?

ASK.

(1)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.

(2)It is not possible, in practice, to carry out a significant software development process with a totally one-way flow of

activity. Sometimes later work necessitates clarification of issues considered earlier.

(3)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)Changes in the external environment can result in changes to the requirements for a piece of software. (Example: years ago I developed a software system for one aspect of the work of the registrar's office which, among other things, had to keep track of student grades. While I was working on the project, the faculty voted to change from straight letter grades (A, B ...) to plus-minus grading (e.g. A, A-, B+, B ...)

c) 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.

3. 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.

a) An incremental approach has at least three major advantages:

(1)The client gets to begin making some use of the software fairly early, rather than having to wait for everything to be completed.

(2)Experience with using the first part of the software implemented can help to refine the requirements for subsequent parts.

(3)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.

b) Carrying this further leads to a family of approaches known as *agile approaches*.

(1)These approaches are sometimes referred to as "low ceremony" approaches, because there is little emphasis on any sort of formal documentation.

(2)Agile approaches have been very successful for projects characterized by changing or hard to pin down requirements.

(3)Agile approaches are generally not used with large scale projects, or ones that are life-critical, though.

(4)A significant danger with agile approaches is that they can degenerate to build and fix.

C. For pedagogical reasons, this course will look at the activities of the software lifecycle in sequence one at a time - but this should not be construed as advocating the waterfall model! The course project will use an iterative approach.

V. **UML**

As we have already noted, this course is not simply about software development, but about a particular approach to software development called object-oriented software development. We said something about this earlier, and will have a lot more to say about this later, but for now I want to note one emphasis that will occur quite a bit in this course - the use of a formalism called the Unified Modeling Language (UML).

A. UML is a set of diagramming conventions that allow one to represent a software system by a collection of models. The first version of UML was adopted by the Object Management Group (OMG) - a consortium of companies - in 1997. The second major version (UML 2.0) was adopted in 2004. The most recent version is known as UML 2.2.

B. UML is a graphical language - that is, its vocabulary is composed of graphical symbols. In this respect, it is international in scope. (Example: my experience at OOPSLA design fest).

C. UML is called the unified modeling language for historical reasons. Prior to UML, there were a number of different graphical notations that were in wide use. UML represents a unification of these notations in a single system that is now widely used.

D. As you.can see from the schedule in the syllabus, we will make extensive use of UML in this course. Currently, UML defines 14 types of diagrams that can be used in the design of software. These are not simply diagrams used to document software - they are tools that can be used as part of the design process.

They are categorized into two broad categories, with a major subcategory under one of them

PROJECT Diagram structure

1. Structural diagrams are used to depict the component parts of a system and how they relate one another. We will talk about four of them:

   a) Class diagrams

   b) Object diagrams

   c) Component diagrams (in conjunction with Architectural Design)

   d) Deployment diagrams (ditto)

2. Behavioral diagrams depict how the system behaves. We will talk about two kinds of behavioral diagrams as well as the subcategory of Interaction diagrams.

   a) Use case diagrams

   b) State machine diagrams

3. Interaction diagrams form a subcategory of behavioral diagrams. We will talk about

   a) Sequence Diagrams

   b) Communication Diagrams

4. By the way - notice that this diagram is actually a UML class diagram!

5. Note well, though, that our goal is not learn how to draw various diagrams, but rather to understand how to think about problem solving in various ways. The diagrams are merely tools to that end!

E. A number of companies produce software tools for working with UML diagrams. (CASE tools)

1. Such tools frequently support:

   a) The creation of the various kinds of UML diagrams

   b) Forward engineering - converting a UML diagram into code.

   c) Reverse engineering - converting code into a UML diagram.

   (These two together are often called "round-trip engineering".

2. "Industrial strength" versions of these tools tend to be quite expensive - in excess of $100 per user. But they do tend to be extensively used in industry, where the time savings they engender justifies the cost.

   There are community editions available for some of these tools (that don't support round-trip engineering).

3. For class exercises, we will be using a tool called astah. While this is a commercial product that normally costs $299, students can

download a copy for free at `astah.net`. There is also a community edition accessible from the same site that is free to all for non-commercial use.

F. To give you practice with the activities of the software lifecycle, and with the use of UML, for much of the semester you be working in teams on a larger scale project.

   1. We will discuss the requirements for this at a later time. In this case, for pedagogical reasons, you will be given the requirements up front - i.e. you will not actually seek to discover requirements by working with users. For pedagogical reasons, the approach will resemble the waterfall model - though we're not actually advocating that!

   2. If you are a CS major, as a senior you will be involved in a senior project that does entail discovering requirements from an actual user. Typically, a senior project uses a more agile approach.