COSC 340: Software Engineering

Introduction

Michael Jantz
(adapted from slides by Ravi Sethi, University of Arizona)
The term 'software engineering' was originally used by a joint NATO conference which met in 1968.

The group met to discuss what was known as the 'software crisis', which was a term they used to describe many of the issues in software development at the time.

Edward Dijkstra described the cause of the crisis in this way. He said:

"The major cause of the software crisis is that the machines have become several orders of magnitude more powerful! To put it quite bluntly: as long as there were no machines, programming was no problem at all; when we had a few weak computers, programming became a mild problem, and now we have gigantic computers, programming has become an equally gigantic problem." -- Edsger Dijkstra, The Humble Programmer (EWD340), Communications of the ACM

This crisis led to a number of issues, including:

Projects that ran over budget or over time
Software that was very inefficient or low quality
Software that did not meet requirements
Projects that were unmanageable or hard to maintain
And software projects that were never delivered

So, in 1968, the NATO science committee met to discuss current practices and give recommendations going forward. They called the workshop ‘Software Engineering’ to highlight the need for making software development an engineering discipline based on scientific principles and rigorous practices.

Another quote from a workshop participant sums up the state-of-the-art at the time:

(software is built like Wright brothers build airplanes)

"Today, we tend to go on for years, with tremendous investments, to find that the system, which was not well understood to start with, does not work as anticipated. We build systems like the Wright brothers built airplanes — build the whole thing, push it off the cliff, let it crash, and start over again."

Since then, software engineering has progressed from an aspiration to an established field. There is a growing body of knowledge, incorporating hard-won lessons from both successful and unsuccessful projects.

Professional societies, ACM and IEEE, have published curricula for undergraduate programs and courses. Thousands of books have been written about the subject.

To me, software engineering is still sort of a softer science, but there are differences of opinion about whether software engineering has truly matured into an engineering discipline. In this course, I’ll try to teach you about the state of the art in software engineering – including its underlying principles and what experts regard as its best practices and essential concepts.
So, let's start by looking at how the United States government classifies software engineers.

The website for the US Bureau of Labor Statistics says that software engineers:

(read each point)

Any others?
The BLS also says there is growing need for software engineers.

So, if you want to be a software engineer after you graduate, and these trends hold up, you should be in pretty good shape.

Employment is expected to grow by 17% over the next 10 years (which is much faster than the average for all occupations – only 7%)

And software engineers get paid pretty well – with a median wage for software systems developers of $100,690 in 2015. The top 10% earned over $150,000. For application developers, the median wage is only slightly less at $98,260.

I should note these are not starting median salaries – but probably closer to mid-career median salaries. The BLS does not publish starting salary numbers – but I think it's closer to $60,000 / year.

However, if you do want to be a software engineer, you need a number of important skills and qualities, including

- Employment expected to grow 17% (2014 – 2024)
  - Much faster than the average for all occupations
- Median annual wage for software systems developers was $100,690 in 2015
  - Top 10% earned more than $153,710
  - Median for application developers was $98,260
- Qualities and Skills
  - Analytical, creativity, problem solving
  - Communication, customer-service, inter-personal
  - Big picture, attention to detail
Analytical, creativity, and problem-solving skills. These are important for breaking down a problem and designing a solution.

A lot of people think that that's all you need – but building software is all about making people's lives easier and it's often done in teams – so you will also need communication, customer-service, and inter-personal skills.

And the last set of skills it lists are 'big picture' – which actually just means being able to think abstractly – and also attention to detail.

I think these two are the ones that separate the average software engineer from the really valuable engineers. If you have these skills, you'll be able to not only write code to support some necessary functionality, but these are the people that can start with only an idea or vision of what they want done, and bring it to reality.

(all of these skills take practice and can be mastered)
Last time we talked about how the term Software Engineer came about, and we also talked a little bit about what software engineers do.

In this course, I’m going to try to teach you about the practice of Software Engineering and how to be a good Software Engineer. So, let’s start by trying to define the term Software Engineering.

In 2004, the IEEE and ACM published curriculum guidelines for undergraduate degree programs in software engineering. They noted then that there are "still differences of opinion about the meaning of the term"

They highlighted three different definitions of the term at the time:

Bauer said software engineering is “The establishment and use of sound engineering principles (methods) in order to obtain economically software that is reliable and works on real machines“.

In the early 70’s, emphasis was on sound engineering principles (trying to relate the creation of software to more established engineering fields).
Another definition from the Software Engineering Institute at says that "Software engineering is that form of engineering that applies the principles of computer science and mathematics to achieving cost-effective solutions to software problems”

(‘Applying principles’ -- more appealing to academics)
Also note – ‘cost-effective’ – dealing with cost, estimating timelines, and satisfying customers is an essential part of software engineering.

And another from IEEE defines software engineering as "The application of a systematic, disciplined, quantifiable approach to the development, operation, and maintenance of software”

This one brings in another set of concepts – we want a ‘systematic, disciplined, quantifiable’ approach so that we can apply it repeatedly and evaluate it with metrics. It also brings in different aspects of the software development life-cycle – not just designing and writing code, but also the operation and maintenance of software.
Here is our definition from your text. It also tries to succinctly capture several of the elements of software engineering.

(read it)

First, it says SW is both an art and science – it is a creative activity – but we want to be able to apply it in a systematic way and evaluate it objectively.

Then, it brings in some of the aspects from the other definitions we’ve already seen.

(developing reliable software – focus on operability and maintenance after the initial design is done)  
(address customer needs – always about developing solutions for people in the real world)  
(and like other engineering disciplines – our budget is not unlimited – so we must consider cost and scheduling constraints)
This diagram captures four key areas or aspects of software engineering that these definitions touch on.

Customers, teams, artifacts, and organizations. Keep these in mind because we’ll be revisiting them throughout the course.
At the top of the figure are customer needs, which drive requirements.

Requirements define what a software system must do and what it must not do.

More generally, requirements are based on the needs of stakeholders, that is, anyone who with a stake in the system.

If you were designing software for a hospital – who might be the stakeholders?
Doctors, nurses, administrators, patients

Different stakeholders can have different, perhaps conflicting, ideas about what they need. And so these ideas have to be elicited and reconciled into a prioritized set of requirements.

#Another challenge with requirements is that they often change. For example, for the software for the space shuttle, NASA and its contractors made over 2,000 requirements changes over six years, before the first flight in 1981.
On the left, we have teams. Teams are responsible for developing and delivering the system. Their development activities include design, coding, testing, and maintenance.

In the 1970's, the prevailing wisdom was similar to "measure twice, cut once". That is, always gather detailed requirements and do a careful design before implementing the system.

The problem with that approach is that if the requirements change – it results in a lot of wasted effort. For the space shuttle, those 2000 requirement changes resulted in huge budget over-runs. Their costs went from 20 million to 200 million over the course of 6 years.

To the right in the figure, we have technology. Technology takes two forms, artifacts and tools. An artifact is an output from a development activity; examples include code, tests, documents, and services delivered from the cloud. A tool is a piece of software that is used to develop other software. What are examples of tools? Programming environments, languages, compilers, debuggers, testing tools.

At the bottom of the figure, we have the organizational context of the software project. So, these are constraints the project will face during development. All software projects face cost and schedule constraints – but there are other constraints as well. For instance, if the company's staff does not have the necessary skills to develop software they need, that's an organizational constraint. There might also be legal or regulatory constraints, such as safety regulations or trade restrictions.

And in the middle we have the Art & Science of software engineering, which links the concepts of requirements, development, technology, and organizational constraints together. They are strongly connected.
Also in the middle, we have processes. A process, or software development process, are the rules for organizing activities during software development. Processes can vary from project to project, even within the same organization.

One example of a process is incremental delivery, where a system is delivered a piece at a time, starting with a minimal working system with enough functionality to get customer feedback. The feedback is then used to make corrections and decide what to deliver in the next iteration.

We’ll study processes in the next chapter.

#For example, consider security and privacy. For most people, privacy of personal health information is a requirement. The US government has regulations for protecting individually identifiable health information. There are many development activities that are done to protect privacy – such as bug fixes to fix security holes in the code. And secure systems must be deployed to keep hackers out. So, requirements, constraints, development, and technology all go together.
Let's talk about one more definition before concluding our introduction.

In general, methods for writing programs for personal or friendly use are not sufficient for building software products. Programming is an essential part of software engineering, but SW engineering is not the same as programming.

One way you could think of it is that software engineering is "Multi-person development of multi-version programs"

So, multi-person implies the need for coordination and communication between people. The larger the project, the greater the need for coordination. Multi-person projects benefit from designs that are modular – so that work can be distributed to different team members and their efforts combined to produce the system.

Multi-version just implies the need to design, code, test, and maintain multiple versions. This is important for building products that can be used on different platforms. Versioning can also help build products that can be maintained and continue to be useful for a long time.
Such products benefit from the sharing of designs and components across versions. Multi-version development can take at least two forms: a family of versions, and a sequence of releases.