Introduction to Software Development Activities

This module covers the various software development activities involved in creating a software product as part of a business organization. The primary focus is on what is commonly referred to as the **software development lifecycle**. Additionally, the importance of team communication and process organization are covered. For many students this course is their first opportunity to move beyond the role of computer programmer to that of a software engineer.

What is it like to be a Software Engineer?

This is the primary question this entire course strives to answer.

You may have been led to believe that majoring in Computer Science and Software Engineering is all about programming, being an expert at new technologies, and being faster and more knowledgeable than your peers.

While being a competent programmer is an important aspect to being a successful software engineer, this is not the whole story. Most software today is developed for a business purpose. Therefore it must meet customer requirements, be dependable and secure, and be maintainable. Additionally, software is typically created by teams.

To be successful as a Software Engineering requires process organization and communication, two things that, if we're being honest, you haven't been trained or taught to do much of at all.

Student development processes typically consist of reading directions and jumping into coding, followed by receiving a grade on that assignment. While this has some faint echoes of software requirements, development, and feedback, it is a far cry from the steps you should expect from a professional development process.

Communication likewise seems to be fairly difficult for Computer Science students, both in written and verbal form. This is to be expected when the main thing we have been training you for is conveying a sequence of algorithmic instructions to a machine.

The result of all of this is that students tend to walk out of university with varying degrees of technical aptitude, but ill suited for jumping into a modern development process at a real company.

This class is intended to change that.

Why is communication important?

In the real world, you are expected to communicate with a variety of people concerning a variety of topics.

You need to be able to:

- Read a description of what someone wants you to do, be able to understand what they want, as well as be able to offer feedback when that description has problems.
- To be receptive to changes in those requirements, but also able to constructively offer your opinion on those changes.
- Express how your software works to your peers so that your systems can mesh together.
- Express what your software does and be able to organize demonstrations to those that might not understand much about technology.
- Exchange feedback with testers about where an issue is occurring within your system and establish steps to reproduce a problem.
- Have a back and forth with user experience designers, customers, and your managers about what is technically feasible to implement.
- Communicate with a team about your progress on a particular feature, as well as be able to seek help or offer it when you or they encounter problems.
- Some companies will have pair programming, in which all coding is done with a partner, or code reviews where a team scrolls through your code, offering feedback and asking you questions.
- And countless other situations that will require you to have to express your thoughts and ideas to other people.

Many of these scenarios are integrated into this course, and you will get practical experience with each of these, if you plan on being a productive member of your team.

What about Process Organization?

For some, an ideal work process might be:

- You are told exactly what to develop
- You develop the thing
- You give them the thing
- You are done

While this is sort of how academic programming assignments work, this is almost never how any real development is done.

In reality software needs to be worked on in a way that allows for:

- Teammates to have access to your code
- Teammates and managers to be informed as to where you are at in the project.
- Teammates to be able to pick up your work when you are unavailable or leave the company.
- Customers to give feedback on projects before they are finished.
- Many, many changes to be made in the requirements to the project all throughout the development process.

You may have noticed links between these activities and the types of communication you are expected to do. That is because a good software development process incorporates and creates channels for all of those types of communication previously discussed.

What happens in a software development process?

While the organization of the various activities may vary, most or all software development processes share a set of common themes.

- Establishing a Vision
- Establishing Requirements
- Architectural Design
- User Experience Design
- Implementation
- Integration
- Testing
- Validation
- Evolution
- Maintenance

This set of structured activities is commonly referred to as the **Software Development Lifecycle**.

The times at which these activities take place, and who does that activity may change from process to process, but in every process these events are happening.

We will go into detail for each of these activities in future readings, but for now here is a quick overview of each one.

Vision

Every software project starts with an idea, but just an idea won't get you very far in making an actual product.

In order to allow others to clearly grasp that idea and work towards implementing it, the idea needs to be expanded into something called a vision.

A vision is basically just an explanation of what the project is, who it is for, what they will get out of it, and what the project does, as well as identifying any similar products that already exist.

Though this typically is created by someone in upper management, in this class, you and your team are responsible for establishing your vision for your app idea. This can help get the entire team on the same page regarding what everyone is setting out to accomplish. Without a concrete shared vision, it is much more likely that individual team members will end up with something they didn’t want, which could be a disaster for all involved.

Requirements

Have you ever been assigned a programming task and been unsure what you were being asked to accomplish?

At their core, requirements are all about defining what a piece of software has to do, and sometimes how well it must accomplish it. Requirements do not typically define how software accomplishes an action, but rather just that action.

Requirements have a large variety of ways to be defined, from long documents to small stories that can fit on an index card, and the method of defining requirements typically depends on which software development process you use.

The main function of requirements is a communication bridge between project stakeholders (whoever is asking for a product and its features such as a customer or a boss) and the developer who needs to build those requirements into a system. What the stakeholder wants is translated into a series of actionable objectives for the developer, sometimes by an intermediary called a business analyst.

These requirements are by no means always perfect, and a good developer will be able to ask the person writing the requirements clarifying questions and raise concerns that might cause changes to the requirements before development begins.

Even after development starts, requirements are not set in stone, and the project stakeholders may change their mind about what they expect the product to do, or how well it must perform. In addition, developers may discover that certain requirements are not possible, and need to communicate this to stakeholders to manage expectations.

A product's functional or performance requirements are just one part of the chain. One must also design how code is to be structured as well as how a product looks and is experienced. These two very different aspects of design are covered in the sections to follow.

Architectural Design

Have you ever been given code and had no idea how your part of the solution was supposed to incorporate the other parts of the code?

This type of design is essentially concerned with describing how the different pieces of the code, usually being written by many different people, are going to work together.

As with everything else, the time at which this occurs, the amount of time spent on it, and the tangible outcomes that this stage produces (documents, diagrams, etc.) depends on the type of development process being used.

Types of activities that happen during this phase might involve the creation of flowcharts, UML diagrams, description of interfaces, or the use of design patterns, as well as a variety of other things.

Flowcharts help developers understand where their piece of the work comes to fit into a larger whole. UMLs and descriptions of interfaces allow developers to agree on how different methods and objects can be used before they are even created, allowing developers to block each other's progress less.

Design patterns might be a new term for you, and these refer to software archetypes that appear over and over across many different projects. Using the names of these design patterns can give developers not only some useful reusable software, but also give an extra bit of vocabulary during the design stage. For example, a developer can say that they will be using a Singleton, and the others will know exactly what that means, how it is structured, and how it behaves.

While architectural design could be considered as primarily dealing with the "backend" of software projects, experience design is more concerned with the "frontend".

User Experience Design

A software application may have some of the most efficient, most beautiful, most architecturally well designed code the world has ever seen, but if no one can figure out how to use it, then it was all for nothing.

User Experience Design is all about coming up with the way the users are going to actually use your application, as well as how they feel while they are using it. It typically involves several steps of hypotheses and testing before the interfaces reach their final stages.

User experience designers might come up with page layouts, interface elements like buttons or sliders, color themes, fonts, as well as help decide which actions are available on what screens.

They may be developers themselves or have developers assist in bringing implementations of their design into the actual application.

Implementation

While this may be the aspect of software development you are most familiar with, coding in industry can be quite different from what you are used to.

For instance, many modern companies require much more teamwork in coding than what you would be familiar with in an academic setting. This teamwork can range from pair programming, in which all code is accomplished with a partner, or code reviews in which you sit down with your team and go over all the of recently created code together to check for problems. Those of you that have always coded alone might be somewhat uncomfortable with these practices.

Another part of implementation that you may not be prepared for is the near ubiquitous use of version control systems (VCS) in industry. These systems, including Git, allow users to collaborate and manage changes to a code base. In my experience teaching this class, students have rarely used any sort of VCS before, partially because team projects are rare in C.S. courses. It is extremely useful to get some hands on experience on Git before entering industry, allowing you to make your beginner mistakes when the stakes are much lower.

Integration

Making a large software application isn't just about developing brand new code that implements the requirements.

It also heavily relies on identifying where you can use existing code that already meets those requirements. This might be code that you wrote, someone on your team wrote, someone in your company wrote, or even incorporating open source software written by an entirely different organization.

Incorporating existing code can not only save a lot of time, but also decrease the likelihood that new bugs are introduced into the system when used correctly.

Note that this is different from stealing other peoples work or cloning existing software without permission.

Testing

Students tend to come in with only a vague understanding of what it means to really test software.

In terms of practical experience, you have run your code before and compared it to a professor's provided output. While this is a start, there is much more to software testing.

At its core, testing involves ensuring that the software system you have developed fully meets the requirements of the project as you know them. But there are a lot of means by which to do this.

All sorts of processes and technology stacks are built around creating tests, automatically running tests and producing easy to read results, creating and providing test data, and everything else you can possibly think of involving tests.

Software testing is such a big and difficult problem that it is itself a 40 billion dollar a year industry, with various companies providing the technologies and services I just listed.

Software Testing could easily be its own class and some important topics include:

- Test Driven Development: a coding process that involves writing tests first followed by code that passes those tests
- Unit testing: frameworks for creating many small tests that each validate one little part of the system
- Integration Testing: Bigger tests that combine software components and test them as a group
- Automated testing and continuous deployment: tools that allow you to run all of these tests each time you make changes to the code, to make you aware of problems as fast as possible.

One of the biggest hurdles that students must overcome is waiting a long period of time in between testing their code. You will notice that many of these tools are specifically aimed at testing small amounts of functionality, it is far easier to find a bug when you've just added it then after you've added hundreds of lines of code.

Validation

Have you ever submitted an assignment that you thought you did really well on, only to get a poor grade on it because it was not actually what the teacher wanted?

Even with good requirements, you aren't always going to get exactly what the stakeholders want on the first try. In fact you are likely never going to get what the stakeholders want on the first try. Stakeholders often don't know what they want, and when you show them something they asked for a week ago they might decide they want something completely different instead.

Validation is all about getting different concrete artifacts (something the stakeholders can look at and form an opinion on) in front of stakeholders and ensuring that this is lining up with their vision for the project, and receiving feedback.

Evolution

Sometimes this feedback requires many changes to be made to the project, and as you will learn in subsequent readings, the later a change happens in a project the harder and more expensive it is to make that change. We therefore typically want either a very concrete and established vision and set of requirements, or a system that can easily accommodate change and the ability to get feedback as much as possible.

Evolution refers to the process that must occur in order to accommodate change in a project. A single change can sometimes involve a wide array of different members of a project effort including stakeholders, designers, testers, analysts, and of course developers.

Projects can often fail when they become so over complicated that they are no longer able to respond to changes in requirements. This requires software engineers to constantly try to reduce the complexity of their systems using tools like proper architectural design and refactoring to improve the quality of completed code.

Maintenance

Almost none of you have submitted a project and have been expected to make changes to it, or fix problems, or help others use it months or years after you actually developed the thing.

However, this set of activities, referred to as software maintenance, can often provide the majority of software work. In reality, we aren’t constantly making new systems, but rather supporting ones that were released a long time ago. How many software products do you know of which are still on version 1.0?

When actually providing real products to people, these updates can be absolutely critical to their lives. This is why it is so important when a Windows operating system becomes “No longer supported”. Thousands of products and business might have to either upgrade (which can cost millions or billions of dollars), shut down, or risk being vulnerable to constantly discovered exploits and problems.

This is one of the hardest aspects of software engineering to give you practical experience on from within a classroom, simply because there isn't enough time in a semester to require you to support these projects, just a few months to develop them. However, having this long term project at least gives you a taste of working on the same code, or other peoples code for a long period of time.

Summary

In this reading, we have established the importance of communication and development processes as well as identified the primary activities of Software Engineering:

- Establishing a Vision
- Requirements
- Architectural Design
- User Experience Design

- Implementation
- Integration
- Testing
- Validation
- Evolution
- Maintenance

This set of structured activities, known as the software development lifecycle, happen in every software project, although they may happen in different orders depending on the software development process, which we will learn about in the next reading.