Lecture #18: Software Engineering

CS106E, Young

In this lecture, we study software engineering. We begin by contrasting programming with software engineering. We look at three ways in which programming in introductory CS classes differs from real world software development – the need to create requirements, the use of teams, and the need to maintain code long term. We look at some important concepts that are used to support teams and maintenance including modularity and encapsulation.

Next we take a look at different processes (also known as lifecycles) used to develop software. We begin by looking at the Waterfall Model, the traditional process used in software development. We consider some alternatives to the Waterfall Model and introduce the concept of Agile Development – a movement that developed in direct opposition to traditional software development. We take a closer look at Scrum, a popular Agile development process. We end by considering the plusses and minuses of Agile vs. traditional methods and where Agile works and where it doesn't work well.

Software Engineering

- Software Engineering is the study of software development.
- This field of study includes the software development process and the methods and tools used to develop software.

Programming is Not Software Development

- Computer science classes, particularly introductory CS classes, generally focus on how to program.
- However, programming is only part of the much wider set of activities that are needed to develop software.
- Let's take a look at some important differences between programming (particularly as experienced in CS classes) and software development.

    Requirements

    - Generally, in most computer science classes you're told exactly what to build (and often are told how to build it).
    - In real life, projects don't start with predefined requirements. There are several different models for how projects might start and how requirements might be developed.
        - In some cases, projects are made from scratch. A team simply brainstorms for interesting or useful ideas for what kind of program might be worth building.
        - In other cases, a technology company is approached by a client in another industry that wants to take advantage of technology, but doesn't necessarily know the best way to do that.
            - The tech company will send in a team of specialist who will carry out a study of the processes currently in use by the client and will determine how computer technology can best be used to improve things.

Teams

- Programming for CS classes is typically done alone. In contrast, almost all real world programming is done as part of a team.

Maintenance

- Usually, once you've finished writing your program for a class assignment, you forget about it.
- In real life, if the project is successful, the program will be supported for quite some time. It may be further enhanced, it may be ported to different platforms.
    - Software development and maintenance is sometimes compared to an iceberg. The casual observer sees just the initial cost of developing the product. However, lying under the water is the long-term cost of maintaining the software.
    - Some estimates put the cost of maintenance as high as 70% of the overall total cost of a software project.

Key Concepts to Support Teams and Maintenance

- As software projects grow in size and complexity, we need special techniques to keep everything under control.
    - The difficulty of developing software goes up exponentially with the project size.

- Just to provide some numbers on how big real world software projects can get:
    - Microsoft's Windows Vista operating system is estimated to contain over 50 million lines of code.
    - I couldn't find corresponding numbers of developers on Vista, but the Microsoft Windows Server 2003 project included 5,000 internal developers and another 5,000 contributing partners.

- Here are some key concepts that software developers use to keep projects manageable. You'll notice that many of these terms overlap, although they all have their own distinct nuances.

    Modularity – We break things into discrete manageable chunks, which are sometimes called modules.

    Encapsulation – We hide the details of different parts of the program. For example, we distinguish between the internal details of a module, which only the writer of that module needs to understand, and the public external interface to that module, which those using the module can see.

    Interface vs. Implementation – Following along similar lines, we distinguish between the interface to a section of code and the implementation of that code.

    Information Hiding – By encapsulating information, we hide the internal details from those who use a module, but who are not writing it. Because the internal details are hidden, they can be changed at a later point in time, without requiring those using the module to change their code.

    Black Box – We sometimes talk about modules as forming a black box. This goes with the previous concepts. The implementer can see the inside implementation details, but everyone else just sees a black box with an interface provided, but no way of seeing internal details.

- The above concepts generally will hold for modules (also known as packages in some programming languages), but we can also consider them as relevant for classes and objects as well as individual methods or functions.
    - In other words, we can think of a single class as encapsulating details and hiding its private instance variables, or we can talk about how a single function forms a black box with an interface specifying its inputs and outputs, but hiding its actual implementation.

- Many of the habits your beginning programming instructors have probably tried to pound into you are actually there to support both teams and long-term maintenance of projects. These include:
    - the importance of modularity,
    - writing clean code, using good variable names, adding comments where appropriate,
    - not using global variables.

The Waterfall Model

The traditional approach to software development is a process or *Software Development Lifecycle Model* called the ***Waterfall Model***. Using this model we carry out the following steps:

Requirements – Determine exactly what it is that we are going to develop.

Design – Determine how we are going to build the product. This is sometimes broken down into two parts.

- Preliminary Design determines the major building blocks of the project and chooses any COTS (Commercial Off The Shelf) components we will purchase instead of developing internally.

- Detailed Design provides a thorough breakdown of modules along with the external interface of those modules. It should be sufficiently detailed to allow individual programmers to know exactly what their coding assignments are.

Coding – Individual programmers code their modules and carry out *Unit Testing* to make sure that their part of the project works correctly.

Testing / Integration – The modules developed by individual programmers are combined together and are tested together. This type of testing involving multiple programmers' code is called *Integration Testing* to distinguish it from the Unit Testing done by individual programmers.

Release – The product is released.

Maintenance – The product is maintained.

Key Features of the Waterfall Model include:

- We determine what we are building before we worry about how we are actually going to implement it.
- There is an emphasis on making sure we get the front-end activities of requirements and design right before moving on to coding.
    - We know that the cost of errors goes up exponentially the further in the process we discover it.
        - If there's a problem in our requirements, for example, it will be extremely expensive to fix it, if we don't discover it until coding.
        - This model tries to make sure we don't get past the requirements phase without ensuring that we have the requirements correct.

- Each phase is followed by some sort of validation and verification activities to ensure that the phase has been done correctly.
- The model emphasizes the importance of design and documentation.

Alternatives to the Waterfall Model

There are some problems with the waterfall model:

- Contrary to the assumptions that the waterfall model is based on, we can't always determine requirements in advance. For example, user interfaces are very difficult to get correct without having users actually try the user interface out.
- For most of the early part of the project the waterfall model results in large amounts of paper being generated rather than code.

We can address these problems either by modifying the waterfall model, or using alternative development approaches.

Modified Waterfall Model – We can modify the waterfall model in a variety of ways. For example, we can add in extensive early prototyping activities in an attempt to make sure we get the requirements correct.

Incremental Development – An alternative approach is to use an incremental development approach. In this model, we begin by designing and building just a subset of our proposed final product. Once we have the subset up and working, we extend the product by building a larger subset. This process continues until the entire product is built. In contrast to the waterfall model, with this approach we have running code early.

Agile Development

- In 2001, several software developers came together and decided they wanted an alternative to the traditional model. They came up with the ***Agile Manifesto***. This document emphasized values they though were important that were not properly being taken into account by the traditional process. This manifesto states:

  We value:
  - individuals and interactions over processes and tools
  - working software over comprehensive documentation
  - customer collaboration over contract negotiation
  - responding to change over following a plan

- They also came up with a list of 12 principles they thought should be followed in software development. You can see the manifesto and principles here:

  http://agilemanifesto.org/

- The manifesto doesn't specify how software should be developed. It's a statement of principles that are in clear conflict with the traditional waterfall model, rather than an actual software process to follow.

- There have been several software processes developed that try to follow the principles stated in the manifesto. Probably the best known are:

  - Extreme Programming
  - Scrum

- Let's take a close look at Scrum

Scrum

- Scrum can be thought of as a specialized version of the incremental development approach, where the increments are kept quite small.
- We develop software using a short cycle called the Sprint.
    - The Sprint can be anywhere from a week up to a month.
    - At the start of the Sprint we choose the features that we will be developing during the current Sprint.
    - We then work on those features for the duration of the Sprint.
    - At the end of each Sprint we should have a stable working product.
    - We then begin the Sprint cycle again, choosing new features to add to the product.

- Sprint also includes a daily meeting called the *Daily Scrum*.
    - This meeting should be kept short (~15 minutes).
    - Each team member will tell
        - what work they've done since the previous Daily Scrum,
        - what work they plan to get done today,
        - if there are any obstacles preventing them from meeting their goals.

- In Scrum we keep a list of product features and functions called the *Product Backlog*.
- During each Sprint we choose items from the Product Backlog for implementation.

Evaluation of Agile Development

- The article "When Agile Projects Go Bad" provides some details from some of the Agile Manifesto signatories on when Agile works and when it doesn't.
    - Agile works best when the team members are all very experienced. In fact, the assumption was that the programmers would be "self-aware" and have "a lot of self-discipline".

    - There is a tendency for inexperienced developers to only get enough information from the customer to do an increment:

      … the customers or users try to tell them what they want, and the developers say 'No, no, no! That's too much information. Just give me the first sentence out of what you said and I'll go build it.' And then the users say 'But no, it's more complicated than that, let me tell you more.' The developers say 'That's all I need, we're doing increments, let me just build that.'

- Inexperienced developers don't focus on the big picture.
    - Because they aren't thinking long term, they may make poor design decisions in a given Sprint that will make future features very difficult to implement.

- These techniques work poorly
    - if there is high turnover,
    - if comprehensive documentation is needed
    - for life-critical situations
    - for some legal situations.

Comparison of Different Approaches

- See "Balancing Agility and Discipline: A Guide to the Perplexed" by Boehm and Tuner

    - Traditional Approaches (including both Waterfall and Incremental Development)
        - are based on traditional engineering fields (e.g., civil engineering)

    - take a systematic approach
    - emphasize "completeness of documentation" and verification at each step

    - key goals are
        - predictability, stability, high assurance

- Agile Methods
    - emphasize rapid value (i.e., quick release of code to the customer),
    - responsiveness to change
        - which may be good or bad
        - Why is being responsive to change bad? Here's a quote from a manager:

          We just spent six months getting a dozen stakeholders to agree on a plan, should we scrap it every time someone wants to make a change?

- Agile only works well for relatively small projects
    - Ken Beck, the creator of Extreme Programming says

      You probably couldn't run an XP project with a hundred programmers. Not fifty, nor twenty, probably. Ten is definitely doable.

    - Team size over 40 is considered problematic.
    - There are reports of larger projects, but they aren't necessarily positive reports. One manager from a 250-person Agile project said *"I would never do [this] again. It was too big."*

- In traditional methods, plans and documentation facilitate communication and coordination across groups.
- Agile doesn't work well if several independent projects need to be closely integrated.

The proper Software Process depends on the type and size of the software being developed
- How easy is it to determine what the end product should look like?
- What is the size and longevity of the product?
    - an iOS app developed by a single person may not need extensive design documentation
    - large teams require more written documentation, face-to-face communication will be insufficient
    - a 5 year product means that we'll have personal turnover, which increases the importance of thorough documentation

- for small projects
    - individual programmer skill is most important in determining overall success rate
- as size increases
    - organization and team issues increasingly becomes important

[Author's note: The comparison section of this document came from some old lectures I've used for my Senior Project class. I've tried to properly credit my quotes in this section of this document. I suspect some of the additional bullet points are close to being taken straight from one or more articles (possibly from the Boehm and Turner book), but unfortunately I didn't do a good job writing down my sources when I originally created these lectures.]