Software Engineering

CS169

Spring 2009

Administrivia

- If you are **enrolled**, you need do nothing
- If you are on the **waiting list**, follow the normal procedures
- Discussion sections?
  - Cancelled!
  - Focus is on team projects
    - You will have team meetings instead
- Pick up class account forms after next lecture

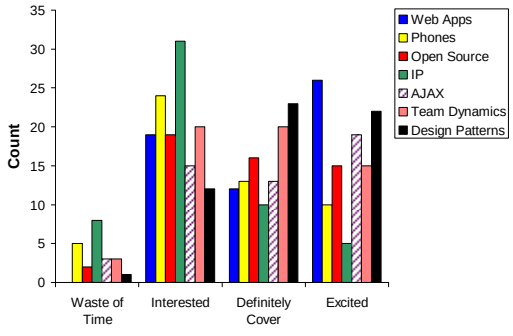
Course Staff

- Eric Brewer, brewer@cs
  - 623 Soda Hall
  - Office hours: Tu 5-6pm, Th 10-11am
  - Founded Inktomi (Yahoo! Search), usa.gov
- RJ Honicky, honicky@cs
  - CS Grad Student
  - Office Hours TBA
- Bonnie Kirkpatrick, bbkirk@cs
  - CS Grad Student
  - Office Hours TBA

Course Communication

- All class materials will be on bSpace
  - Lecture notes, handouts, papers to read, etc.
  - So why should you come to lecture then?
- Read the class site and the forum
  - "Projects" topic is a good way to create teams
  - Ask questions in the "General questions" topic
  - Preferred for most questions over email
  - [replaces ucb.class.cs169 newsgroup]

Class Survey



Academic Honesty

- Policy on web site
  - Expected to cooperate on projects
  - … but not on homeworks/exam
- Default penalty: D in class

Course Structure

- Lectures
  - Course taught mostly from notes
    - Supplemented by readings
  - Programmer's view of software engineering
    - Technology issues over business issues
- Homework
  - TBD
- Midterm exam (no final)
- Project …

The Project

- A BIG project
  - Can be (almost) anything
  - Web app, phone app, desktop app, combo…
- Done in teams of 5-7 students
  - You do everything
  - Design, code, and test in several assignments
- Be prepared for a lot of work (and fun, and satisfactions, …)

One of My Opinions

- Good software engineering can be learned
  - But it is hard to teach
  - Most people only learn through experience (i.e. mistakes)
- How can you get that experience?
  - Do a project, in a team
  - Hear from other projects
  - Each project will present ?? times to the class

Project Timeline

- Project nominations
- Project selection, team assignments
- Requirements and specification
- Project design & plan
- Design review
  - Done by other teams
- Iterative implementation
- Presentation and Demo

What is Software Engineering?

- Your thoughts here

What is Software Engineering?

- As defined in IEEE Standard 610.12:
  - The application of a ***systematic***, disciplined, ***quantifiable*** approach to the ***development***, ***operation***, and ***maintenance*** of ***software***; that is, the application of engineering to software.

An Opinion

- The IEEE definition is really pretty good
- But it is descriptive, not prescriptive
  - It doesn't say how to do anything
  - It just says what qualities S.E. should have
  - As a result many people understand SE differently

End-User License Agreement

- From Microsoft Office (just tiny part):
  - Can't disassembly or reverse engineer
  - If it does something bad, you have only one "remedy"
    - Money back or return product (you pay shipping)
  - Never entitled to any "damages"
    - Even breach of contract, failure to support product
    - Even admitted problems
  - Product is "as is and with all faults"
  - Any *implied* warranty is not valid
    - … doesn't matter if we said it would work

What is Software Engineering?

- Often compared to civil engineering
  - building a bridge
- A surprisingly good analogy
  - Size matters: a dog house vs. a skyscraper
  - Team effort with careful planning
  - Difficulties to change designs?
  - Penalties for failures?
  - Many terms come from this metaphor: building, scaffolding, architecture, components, …

But, a software revolution is in progress…

- Old:
  - Desktop software released every year or two
  - Physically distributed on CD
  - Hard to update, hard to test with all configurations
- New:
  - Applications in the "cloud"
  - Access via browsers, phones, …
  - **Easy to update** every day or every hour
    - Small penalty for errors, just fix them quickly

"Penalties" drive the process

- Medical equipment, air traffic control
  - Errors cost lives (Therac-25 coming later)
- Traditional apps:
  - Errors are hard to fix, therefore long lived
  - Cost $$, reputation
- Web apps:
  - Errors are easy to fix (limits the penalty!)
  - Can also test ideas live with small random groups
- Free apps: users are tolerant

Different penalties => different processes

Tradition SE

- Several fixed steps
- Heavy specification
- Extensive testing
- Controlled release
- Physical Distribution
- Often large teams

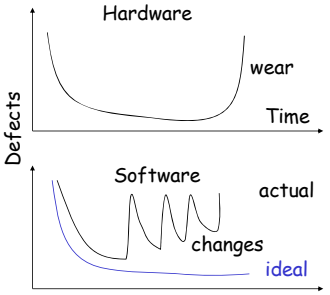
"Agile" SE

- Many small iterations
- Limited specification
  - React to previous version
- Easy to change course
- Some testing (but less)
- Frequent simple releases
- No distribution
- Typically small teams

Software Engineering vs. Civil Engineering

- Software generally unable to reuse components…
  - E.g. standard pipes, light bulbs, outlets, etc.
  - Exception: really big components
    - Databases, Apache, Ruby on Rails
    - Must be big enough to have their own staff, agenda
- Software generally doesn't work
  - Customers rarely demand it !
- Software is much easier to fix
  - (but harder than people think)

Software vs. Hardware Reliability Curve



- Harware wears out
- Software changes
  - or its environment changes
  - called "bit-rot"

Software Engineering Myths: Management

- "We have books with rules. Isn't that everything my people need?"
  - Which book do you think is perfect for you?
- "If we fall behind, we add more programmers"
  - "Adding people to a late software project, makes it later"
- "We can outsource it"
  - If you do not know how to manage and control it internally, you will struggle to do this with outsiders

Software Engineering Myths: Customer

- "We can refine the requirements later"
  - A recipe for disaster if you can't change easily.
- "The good thing about software is that we can change it later easily"
  - As time passes, cost of changes grows rapidly
  - Depends on the size of the project, contracts, distribution, ???
  - This is really somewhere between laziness and rationalization…

Software Engineering Myths: Practitioner

- "Let's write the code, so we'll be done faster"
  - This is an open question!
  - "The sooner you begin writing code, the longer it'll take to finish"
  - Writing tests first has shown value
  - Question is do you:
    - specify then implement? OR
    - implement in iterations?
- "Until I finish it, I cannot assess its quality"
  - Software and design reviews are more effective than testing (find 5 times more bugs)
- "There is no time for software engineering"
  - But is there time to do it over?

My List: What is Software Engineering For?

- We want to build a system
- How will we know the system works?
- How do we develop system efficiently?
  - Minimize time
  - Minimize dollars
  - Minimize …

Problem 1: How Do We Know It Works?

- Buggy software is a huge problem
  - But you likely already know that
- Defects in software are commonplace
  - Much more common than in other engineering disciplines
- Examples (see "Software Crisis" reading)
- This is not inevitable---we can do better!

What is It?

- But how do we know behavior is a bug?
- Because we have some separate specification of what the program must do
  - Separate from the code
  - Like a blueprint for a building…
- Thus, knowing whether the code works requires us first to define what "works" means
  - A specification

Teams and Specifications

Principle #1:

Communication is hard.

In any conversation, the participants will have (slightly) differing interpretations of what was said.

Teams and Specifications (Cont.)

- Principle #1 is devastating for software development
- People will
  - Discuss what to do
  - Divide up the work
  - Implement incompatible components
  - Be surprised when it doesn't all just work together

What Can We Do?

- Write specifications
  - *Write down* what it is supposed to do
  - Make sure everyone understands it
  - Keep the specification up to date
- This does not solve the problem completely
  - There are always ambiguities, contradictions
  - These lead to bugs
  - But the problem is reduced to manageable size

Summary of Problem #1

- A specification allows us to:
  - Build software in teams at all
  - Check whether software works
- Actually checking that software works is hard
  - Code reviews
  - Static analysis tools
  - Testing and more testing
  - We will examine this problem closely

Problem #2: How Do We Code Efficiently?

- Assume we want to minimize time
  - Usually the case
  - Time-to-market exerts great pressure in software
- How can we code faster?
  - Obvious answer: Hire more programmers!

Parallel Development

- How many programmers can we keep busy?
  - As many as there are independent tasks
- People can work on different modules
  - Thus we get parallelism
  - And save time
- What are the pitfalls?

Pitfalls of Parallel Development

- The problems are the same as in parallel computing
- More people = more communication
  - Which is hard
- Individual tasks must not be too fine-grain
  - Increases communication overhead further
- Inherent sequential constraints
  - E.g., pipeline architecture

Interfaces

- The chunks of work must be *independent*
  - But work together in the final system
- We need interfaces between the components
  - To isolate them from one another
  - To ensure the final system works
- The interfaces must not change (much)!
  - Otherwise, development is not parallel

Defining Interfaces

- What are interfaces?
- They are just specifications!
- But of a special kind
  - Interfaces are the boundaries between components
    - And people

Defining Interfaces

- Specifying interfaces is most important
  - Interfaces should not change a lot
  - Effort must be spent ensuring everyone understands the interfaces
  - Both things require preplanning and time
- But often we can stop at specifying interfaces
  - Let individual programmers handle the internals themselves

Software Architecture

- To define interfaces, we must decompose a system into separate pieces with boundaries
- How do we do this?
- Your thoughts

My Opinions

The decomposition of a system is driven by:

- What it does
- How we build it
- Who builds it

Decomposition: What the System Does

- The application itself often dictates natural decomposition
- A compiler is a pipeline consisting of
  - Lexer
  - Parser
  - Type checker
  - Optimizer
  - Etc.

Decomposition: How We Build It

- Buildings need scaffolding during construction
- So does software!
- Two areas in particular:
  - Lots of extra code that is not really part of the final product
  - Influence of third-party subsystems
- Test harnesses, stubs, ways of building and running partial systems

Decomposition: Who Builds It

- Software architecture reflects the structure of the organization that builds it
- Often, 5 developers = 5 components

Summary of Problem #2

- Efficient development requires
  - Decomposing system into pieces
  - Good interfaces among pieces
- The pieces should be large
  - Don't try to break up into too many pieces
- Interfaces are specifications of boundaries
  - Must be well thought-out and well communicated

Conclusions

- Software engineering boils down to several issues:
  - Specification: Know what you want to do
  - Design: Develop an efficient plan for doing it
  - Programming: Do it
  - Validation: Check that you have got what you wanted
- Specifications are important
  - To even define what you want to do
  - To ensure everyone understands the plan

Conclusions (Cont.)

- Is that all?
- NO!
- Why?
  - Because specifications do change!
  - Because you were wrong about what you wanted
  - Because the world changes
  - We'll talk about this next time . . .