Software Engineering Tour

- This is a tour of modern software engineering ideas -- from Waterfall to Agile, with the addition of a bunch of my own random thoughts.

Optimism

- Even today, software and hardware systems have enormous untapped potential. Email, the web, Google, spam, Flikr ... none of these things existed 5 years ago the way they do now. There's a great imbalance: lots of neat, un-solved problems vs. not that many people good at building software systems.

Challenges

- Hardware, languages, frameworks... these seem to be improving actively
- System complexity -- building something big yet reliable out of complex parts -- making best use of the relative scarcity of programming ability (CS108)
- 80/20 joke -- the first 80% of the features take 80% of the time, the last 20% also takes 80% of the time. Point: there are many details to get right moving something from demo-alpha state to really done -- more than you would think. This is probably another reason that software schedules can look ok until near the end, only then realizing that there's a lot more to do.
- Product design complexity -- creativity to think of Google Maps, Flikr (CS147)
- Vendor lock-in / Lack of standards -- why can't my camera and my phone and Google maps exchange data? Vendors tend to want to "lock in" your data to make a lock in a selling opportunity without winning it through competition. The lesson of the Internet email/HTTP/HTML is that you get tremendous synergies when you free the data from being tied to any one vendor. Free data standards enable competition and un-imagined new applications. Collectively, we all benefit enormously from competition, but individual organizations naturally try to avoid competing on their merits if they can.

Process Fadishness

- Silver bullet -- process, language, guru
- Driven by the scariness of software dev -- uncertain, difficult
- Process Mania
  - Danger: get caught up in the vocabulary, activities, diagrams, etc. of the process itself. Remember, it's just a tool.

"Silver Bullet" Design School

- Some people want there to be a "solution" -- a silver bullet -- to the problem of software. The solution, X, might be a language, or a coding style, or a particular guru or a system. A Silver Bullet person really believes in X, and so their view tends to be about how to use X more. In my experience, Silver Bullet people can be passionate advocates for X. There is not a lot of discussion of cost/benefit tradeoffs. (I'm deeply not a Silver Bullet person, and I'm probably doing a shabby job of presenting it.)

"It Depends" Design School

- In contrast, the "It Depends" school is skeptical that any particular technology is going to make a huge difference. All projects will require project-specific decisions along the way, considering the cost/benefit of the various available approaches. The ideal solution will combine multiple approaches, and no one approach is going to make the solution easy. We have to invest in thinking about the specifics of the problem to make decision. There is no shortcut. (Disclaimer: obviously I'm a member of the It Depends school.)
- The It Depends school does not provide any great, insightful answer about anything. It's more of a shrug and an admission that there is no simple answer.

- In any case, there are many proven, valuable techniques available (OOP, unit testing, GC, ...), and of course we should use them where they make sense. Evaluate the available techniques to decide how well they fit the problem at hand.

Scheduled vs. Extreme Programming Style

- There is a traditional branch of software engineering that deals with...
  - Making a big list of requirements
  - Building a development/test schedule from the requirements
  - Managing the progress of the project vs. the schedule
- That process is oriented towards finishing the project with all its features, but not necessarily hitting the deadline.
- We will study the more modern "agile" style which will hit the deadline, but we won't know exactly which features are in and which are out until the end ("agile" is the new umbrella term for what used to be called the "extreme" programming style).

Fred Brooks Quotes

- The Mythical Man Month was a landmark book in thinking about the process of building software
- "Adding more software engineers to a late software project makes it later"
  - There are high internal costs of communication within a team - **diminishing returns**
  - This is why we care so much about modularity
  - What is the project output curve as a function of n, where n is the number of engineers? It's definitely less than linear. sqrt(n)? log(n)?
- "Plan to throw one away"
  - The initial design is limited by lack of foresight
  - You can whip out the low quality version, a "prototype", observe the details, and so do a good job on the 2nd version.
- "Second System Effect"
  - On their first project, people are scared, and so they proceed carefully, and it all works
  - On the second project, they are too confident, and so everything goes awry
- "There's no silver bullet"
  - There's no technique or guru or process that's going to make software easy. There is an irreducible complexity in there.

Common Themes

- There are a few basic themes that run through all modern software engineering disciplines. Here they are as interpreted by my own biases...

1. The 3 (or 4) Variables

- Fundamental quantities to tradeoff
  - Time
  - Features ("Scope")
  - Quality
  - Cost (team size, tools, ...)
- Tradeoff
- Quality can slip most easily -- too intangible

2. Modularity

- A simple idea, but one of the most important things to get right. Modern software systems are as large as they are because they are built in a modular, layered way.
- Independent development -- team development, working in parallel, can change code in one module with limited risk of upsetting the other modules
- Library code -- the ultimate payoff for modularity

2a. Modularity -> Ownership / Delegation

- A super-basic management technique is to have particular people (or teams) own particular modules.
- Advantage: people can talk to people on other teams about public interfaces, but worry about particular implementations on their own.
- Advantage: the module forms a sort of unit of delegation -- I want such and such interface, and I trust you to implement it.
- Advantage: Can work well with people's sense of pride and ownership
- Advantage: modularity aids parallel development
- Disadvantage: loose the two-heads-are-better-than-one advantages of many people seeing how many things are implemented
- Disadvantage: modules may fail to work together well, since people can focus on their area of ownership, and not so much on their public and the project as a whole
- Disadvantage: debugging can be hard, since people don't tend to understand each other's modules
- Nonetheless, ownership/modularity is a good starting point when thinking about a software project.

3. Simplicity vs. Generality and Frameworks

- Write it simple at first. Improve it later if that makes sense.
- Generality has a cost -- build the more general, more sophisticated version only if it has real value for the problem at hand.
- Kent Beck / Extreme programming quote: "Do The Simplest Thing That Could Possibly Work"
- Embracing simplicity goes against some nerd instincts
  - We love fancy, elegant, intricate, high-tech solutions, so we may build such solutions because they are pleasing, but where their costs are greater than their benefits.
  - Consider phrasing it this way: "Do the most unsophisticated thing that could possibly work." The word "unsophisticated" may go against your instincts, but that's largely what simplicity is about.
- Alternate phrasing: Do not build in **anticipation** of need. The future is too uncertain.
- Frameworks
  - In the interest of generality, we sometimes want to build a general **framework** first, and then use it to solve the problem at hand. This can be a trap -- we are ignorant of the real issues of the problem, so our framework is mismatched.
  - The Agile/Simplicity style does not write the framework first.
  - Suppose you are solving problems A, B, and C. The three problems look somewhat similar. Solve problem A first. Later, solve problem B, notice some code overlap with A -- maybe factor some of it out. Later, solve problem C, and again notice common code. Now with concrete knowledge of A, B, and C, now factor out the common code into a nice framework. Point: it's hard to speculate what the common code will look like early on. Don't write the framework first.

Aside: "Strategic Conquest" Generality Story

- Once upon a time, there was a moderately complex but really enjoyable Mac game called Strategic Conquest that involved controlling pieces that moved around a rectangular world.
- A friend of mine -- we'll call him Brian -- decided it would be fun to build a Java version of the game.
- He started building the game.
- It became clear that he would need data structures to support the grid-world and the pieces in it.
- Being a good computer scientist, it occurred to him that the Grid and Piece interfaces could be done in a general way, so they would also support, say, Chess or Checkers or any number of other sort of grid/piece type games.
- So he began designing and implementing Grid and Piece in a really general way.
- He worked on this for a while, the project got deeper and messier, and eventually he lost interest and it died.

"Strategic Conquest" Generality Lessons

- Building the truly general forms of Grid and Piece is **hard** -- who knows what the interface needs to support? This requires the designer to speculate about some future space of uses of the class where the

future use is unknown. In contrast, having Strategic Conquest as a nice, concrete, spelled out use today makes the design much easier.

- Maybe the more general form ends up being a big win in the future. But we don't know if that future will ever come, or if the design will be appropriate when it does come. We are paying a cost now for an uncertain benefit in the future.
- Conclusion: we are skeptical of speculative generality. It is safer to do the generalization at the time in the future when the needs are concrete (like the A/B/C story above).
- Note that general frameworks can be enormously valuable. This is more a question of when the general framework is built.

4. Just Enough Design

- How much is the right amount of design?
- The right amount of design is a coherent plan, detailed enough to enable the independent modules to fit together later
- Early design time is the opportunity to make some fundamentally smart decisions
  - Some basic structural decisions are hard to "retrofit" later -- need to think enough to make a good choice the first time.
- Limited by ignorance of the future
- Symptom of too little design
  - The pieces don't fit together well, since no one is looking at the big picture. Large scale decisions are made poorly.
- Symptom of too much design
  - Easy to get caught up arguing back and forth about circle/arrow diagrams on the whiteboard. The value of the discussion is limited by the great ignorance of how things will play out in the future. Can only get so far with speculation and thought experiments. Discussions will work better later, with more concrete data from real code writing. Get to the coding!

Aside: Waterfall Method

- The "waterfall method" refers to a design discipline that attempts to form a really complete design picture before actual coding. It is not widely used
- Very organized steps
  - 1. Gather requirements
  - 2. Make a paper design, really detail out how its going to work
  - 3. Then start coding, testing. ....
- The "waterfall" only goes one way -- once we're done with (2), we don't change the design.
- Of course the problem with the waterfall method is exactly the ignorance of the future. How well can the thought experiment in step (2) forecast how details are going to play out as the code is actually written and integrated in (3)?
  - Of course it is possible to design/forecast accurately if you put in enough time, but that could be **a lot** of time in step (2).
- Possibly appropriate for high-safety systems such as flight-control systems, where the designers want to understand the correctness of the system at the design/on-paper level, not just by observing it to work.

5. Test Code + Fast Feedback

- Don't leave the app in a broken state too long
- Fast feedback loop:
  - Put effort into having test code for every feature
  - Add features in small steps.
  - Integrate the change back quickly.
  - The new code may break things in some way.
  - Run all the tests again to make sure the "improvement" works right
- The 2 year project
  - Never have a 2 year deadline. Have a series of 4 month deadlines, each with a concrete, runnable deliverable (aka "drive to release")

  - 2 year projects have a tendency to never get finished -- the app gets so broken, it never quite gets put back together.
- Discipline
  - Aim to keep quality high at all times -- requires real discipline.
  - Quality can be an intangible that slips quietly.
  - e.g. nightly build with automatic test harness that emails out bug reports.
  - e.g. Safari checking/performance story
- vs. Innovation
  - These techniques are great for shipping something predictable on a deadline. If you want to do something truly innovating, you need a strategy that leaves time to experiment widely without deadlines and without the cost of keeping quality at the production level.

Aside: Making Feedback Work

- Part of the power of "testing" is bringing the results of the test back to the developer -- "closing the loop", so the feedback can help them make better decisions.
- e.g. Tinderbox -- http://tinderbox.mozilla.org/showbuilds.cgi?tree=Firefox
  - Show build state, integrating everyone's check-ins
- "Smoke test" -- a quick test that checks if the app works at all -- can be integrated with a tinder-box type feedback. Also, can build/use unit tests.
- Also, can have automatic feedback for performance
  - http://wiki.osafoundation.org/bin/view/Projects/PerformanceProject

Refactoring

- Martin Fowler

Code that smells bad

- Suppose you are trying to implement a feature or fix a bug, and you notice something ugly or insufficient (aka "smells bad") in another part of the code.
- Fix the ugly part at that time.
- Having the concrete feature/bug task at hand helps to guide how the fix should work.
- The fix is not done as a **speculation** for the future -- it really is needed.

Put in test code

- Make sure there are tests before "fixing"

Make the improvement ("refactor")

- Do not fix the whole program.
- Fix one annoying aspect of the code
- e.g.
  - Factor out the duplicated code
  - Improve the API
  - Rationalize the parameters

Run tests -- check the "improvement"

- Make sure the improvement really works

vs. From Scratch

- The initial design was imperfect
- The subsequent coding was imperfect (e.g. under a deadline, making tradeoffs)
- The result is some code that is not pleasing (this is especially true if the code was written by someone other than you)
- There is a strong instinct to throw all the code out and "start from scratch with a clean design". Engineers find the "start from scratch" very appealing.
- According to Refactoring, you very rarely want to throw it all out and start from scratch.

- There's a lot of experience that says that the "from scratch" project can go awry.
- Eventually, the from-scratch version is called for, but not nearly as soon as most programmers will want to do it.
- From Scratch can make sense if the new version is so different from the old version, that going in little refactoring steps looks too costly.

Baked in Cases

- Some bugs are found in code from real production use with real data.
- One of the reasons the old source is messy is that it has been tweaked to deal with cases that occurred in the real world -- cases that are "baked in" to the old source.
- e.g. the mouse drag code has some special logic in it to deal with the case where the user does a drag on a two-monitor system, where the drag starts on a monitor with one color depth and crosses over to a monitor of another color depth.
- There is near zero chance that the from-scratch version is going to get that case right.
- You will see things in an old source base that are so obscure, you know the only reason they were figured out because there were thousands of uses, and one of them hit upon a weird situation and filed a bug.

Remember: Value of Old Code

- Suppose you have a messy old system that has been in use with real users and real data for several years.
- Swapping in a new "clean" design will expose significant problems -- there's going to be a real cost there.
- The messy old code has value in baked into it from real field use, training, etc. There is no substitute.
- Code does not "wear out" like some kind of machine!

Refactoring -- Heal the code

- NO: throw it all out and write the clean version from scratch
  - Too costly and risky
  - Doesn't work out that well half the time anyway
- YES: improve it incrementally, piece by piece
- We cannot fix the whole thing, however we can improve it piece by piece.
- This is a low-risk, low-investment strategy -- very incremental, never makes huge changes to the source base.
- We cannot make it perfect, but we can make it better.
- Repeated long enough, the code might actually get pretty good.
- Disadvantage: refactoring is unlikely to lead to a radical improvement in the architecture -- it's inherently incremental.
- Disadvantage: it might be like buying 20 slices of bread for $0.50 a piece (refactoring), instead of just buying a whole new loaf for $3.00 (from scratch). Refactoring is inefficient in a sense, but it is low risk.

Refactoring is Popular

- Many projects regard their sources as not as good as they could be
- Many projects are not in a position for a risky, time-consuming from-scratch re-write
- Refactoring looks like a very appealing compromise

Summary

- Ugly, bug-prone code is irritating
- We want to do it right, but the "from scratch" method is expensive and has a good chance of failing (failing to work and/or failing to be perfect)
- Buy as much improvement as we can afford under our deadline -- step by step
- Use fast-feedback to check that all changes are good changes.
- Overall, can be a little inefficient, but it's low risk -- prevents you from doing something truly stupid.

Aside: Mozilla/Firefox and Refactoring

- The Mozilla Firefox project is a huge success, creating all sorts of competition and value in the previously moribund browser category.
- It's also a study in Refactoring vs. From Scratch.
- 1998 -- The Netscape browser sources are a mess. The decision is to start from scratch. The project is open sourced in 1998 under the name "Mozilla".
- 2000 -- The Mozilla project builds a lot of frameworks, in particular the XUL system for interface construction. By 2000, Mozilla was still not especially usable. It was derided as "bloated" "slow" "what were they thinking".
- 2002 -- Mozilla 1.0 ships, and it's actually pretty usable. Say it took about 4 years of From Scratch effort to get to something pretty good.
- 2004 -- Firefox 1.0 ships. It's cleaner than Mozilla, but in fact built on exactly the source code that Mozilla built up/invested. Firefox is super popular. Firefox has a cleaned up, focused GUI compared to Mozilla -- that's probably what enabled its breakthrough popularity. However, Firefox is also quite dependent on the Mozilla framework investment.

Mozilla/Firefox Lessons

- Starting from scratch, it took 4 years to get to something good. A Refactoring person would say that this shows the danger of from-scratch.
- However, Firefox is good now in large part because of all that careful investment in 1998-2002. A from-scratch person could argue that there was no way to "evolve" to get the XUL system in little steps from the old sources. Maybe it was just an investment cost that had to be paid to enable something like Firefox later on.
- I happened to talk to Mitchell Baker on exactly this question, and got another From Scratch angle: nobody **wanted** to work on the old, 1998 sources. They were messy and awful. Even if that was the top-down edict, you could not force people to be excited about it.

Extreme Programming -- Agile

- Kent Beck (much overlap with Refactoring)
- XP -- the most "extreme" form of fast-feedback coding
- More recently, all these techniques go under the term "Agile" as "extreme" sounded too scary

1. A Customer

- The "customer" is the representative and advocate for the hands-on use of the software when it is done
- The customer provides input to help with decisions about features or bugs -- what is the end user's view. Conceptually, the customer provides a
- There will be many little decisions that go into a project, and the customer keeps those decisions grounded with respect to how the system will actually be used
- Avoid the trap where a system is completed and has all the check-off items, but its end users hate it
- The customer is also motivating. It's more fun to work on a project where you have a concrete sense that it is solving someone's problem.

2. Testing (like refactoring)

- Write test code before implementation
- Test should identify weird cases -- odd conditions that occur to the programmer when thinking through the design.
- Writing tests is easier and requires less imagination than writing the solution. Since you're going to write them both, might as well write the tests first.
- Writing the tests can help get the programmer thinking through the issues, so writing the solution is a little easier.
- The public exposed API of a class is a vital part of a class. Writing the tests first can show us that some parts of the API are clumsy or inconvenient for solving common cases. The unit tests are an opportunity to make the programmer act like a client, to see how things look from the client side.

- Note: it's a lot harder to have an automated test for a GUI system, though there is some work in "robot user" systems that click through the GUI to do testing. -- the "Robot" AWT class, and there's s open source project called "abbot" (a better 'bot).

3. Incremental Dev / Simplicity (like refactoring)

- Make the smallest improvement that makes sense
- Test it immediately, iterate until all the tests pass, add new tests for new quirks
  - Ideally, the change is made and re-integrated all in one day
- Keep quality high by never allowing it to get low
- The build-test system should be really easy -- suppose there's a command that builds, runs the tests, and prints the result.
  - Write the tests first
  - Start writing code -- iterate
  - Keep going until the test output is right

4. Pair programming

- Collective code ownership -- everyone has some understanding of everything
- All coding is done in teams of two.
- One person can code, one person can observe, and try to think about context and improvement.
- This is the most controversial part of Extreme Programming -- XP believers swear by it while many others are not convinced.
- One advantage of pair programming may be simple social pressure -- we are less likely to write awful code if someone is watching (may also limit time spent surfing the web!)

5. Refactoring

- Fix ugly things ("crufty") when they need to be fixed because they are blocking some other work.
  - e.g. maybe an ugly design is exactly one which makes adding a later feature hard. Fix the ugly part when you add the new feature.
  - note: nice way to combat the "ignorance" problem of designing things right before you have built anything.
- Ugly parts of the code are not allowed to stay ugly

Summary

- Very incremental. Keeps the project in a "near shippable" state at almost all times -- makes small, incremental improvements.
- Avoid big, unpredictable slips -- the project never gets very far from the shippable state
- Not good for projects with more than 10 people
- The incremental style is (maybe) a modest "tax" on progress, but it gives predictable, high quality results with whatever time is available.
- You don't know how far you will get ahead of time. However, you do no that when the deadline approaches, whatever you have will be of high quality.