What is Software Engineering For?

- Solve two problems:
  - How do we know what code to build?
    - How do we know the code works?
  - How do we develop the code efficiently?
    - Minimize time
    - Minimize dollars
    - Minimize …
- How do we organize these activities?

How Do We Build Software?

- Consider ad-hoc development:
  - Alternate in ad-hoc manner between:
    - Some thinking about what we need to build
    - Some coding
    - Some talking to customers
    - Some testing
  - This may work for very small prototypes
- For complex software we learned from past mistakes that it is worth to have a systematic approach (software process)

Software Process

- Most projects follow recognized stages
  - From inception to completion
- These steps are a “software process”
  - Arrived at by trial and (lots of) error
  - Represent a good deal of accumulated wisdom
- Process = how things are done
  - In contrast to what is done
  - Consider how you pay your bills
    - How do you make sure systematically you do not forget about them, or overdraft your account?
The Waterfall Process

1. Requirements and Specifications

- Figure out what this thing is supposed to do
  - A written document
  - Req.: a user’s perspective
  - Spec.: more precise/complete
- Purpose:
  - We don’t build the wrong thing
  - Information for planning
- Talk to users, clients, or customers (stakeholders)!
  - But note, they don’t always know what they want

2. Design

- The system architecture
- Decompose system into modules

2. Design

- The system architecture
- Decompose system into modules
- Specify interfaces between key modules
- Much more of how the system works, rather than what it does
3. Implementation

- Code up the design
- First, make a plan
  - The order in which things will be done
  - Usually by priority
  - Also for testability
- Test each module

4. Integration

- Put the pieces together
- A major QA effort at this point to test the entire system

Waterfall Process - Opinions

- The waterfall process was adopted (1970s) from other fields of engineering
  - This is how bridges are built
- But many good aspects
  - Emphasis on spec, design, testing
  - Emphasis on communication through documents
- (I believe) Very little software is truly built using the waterfall process
  - Where is it most, least applicable?
Waterfall Example at NASA

• Space shuttle control software

• Each execution controls $4B equipment, lives, “dreams of a nation”
  – No beta testing
  – 420,000 lines program, had 17 errors in 11 versions
  – Commercial equivalent would have 5000 bugs

• Secret sauce is the process

Waterfall Example at NASA

• A third of the effort before coding starts

• Specifications are written down and negotiated at length
  – Change to add GPS support (1% of code = 7k lines)
    • Spec for the change is 2500 pages!
    • Total spec is 40,000 pages

• Spec is almost pseudo-code
  – Very little flexibility once the spec is set

Waterfall Example at NASA

• When you find a mistake, don’t just fix the mistake, fix what allowed the mistake in the first place
  – Unclear API
  – Insufficient tests
  – Improper use of tools

• Validation and review at all levels
  – 85% of bugs found before formal testing begins

• Process relies heavily on two databases:
  – Revision history
  – Bug database

Waterfall Example at NASA

• Flip-side:
  – 420,000 lines program maintained by 260 people at a cost of $32 million a year
    • That is $8/line of code/year

• Such a process is too expensive for many software products
  – Perhaps overkill too

• But how to reach right compromise …
The Waterfall Model - Drawbacks

- What are the risks with the waterfall process?

The Waterfall Process - Drawbacks

- Relies heavily on being able to accurately assess requirements at the start
- Whole process can take a long time before the first working version is seen
- Little feedback from users until very late
  - Unless they read and understand specification documents
  - And they know what they want
- Problems in the specification may be found very late
  - Coding or integration

An Opinion on Time

- Time is the enemy of all software projects
- Taking a long time is inherently risky
- Why is time so important?

Why Time is Important?

- The world changes, sometimes quickly
- Other people produce competitive software
- Technologies become obsolete
  - Some products are obsolete before they first ship!
- Software usually depends on many 3rd-party pieces
  - Compilers, networking libraries, operating systems, etc.
  - All of these are in constant motion
  - Moving slowly means spending lots of energy keeping up with these changes
The Flip Side: Advantages to Being Fast

• In the short-term, we can assume the world will not change
  – At least not much

• Being fast greatly simplifies planning
  – Near-term predictions are much more reliable

• Unfortunately, the waterfall model does not lend itself to speed . . .

Software Process Control Variables

• Control variables in a software project
  – Time
  – Quality
  – Scope (features)

• If you try to fix all three then the hardest to measure (i.e., quality) will suffer

• Management fixes two and the development team controls the third variable
  Scope is the control variable for agile progr...
Agile Process (cont.)

• 2. Design:
  – Design for expected change

• 3. Implementation:
  – Critical pieces first
  – No premature optimization/generalization
    • Implementation will likely change
  – Can leave some parts unimplemented

Iterate:
  – Show to user/customer the prototype
  – Update the requirements, update design, ...

Agile Processes: Disadvantages

• Main risk is making a major mistake in requirements, spec, or design
  – Because we don’t invest as much time before build 1
  – Begin coding before problem is fully understood

• Trade this off against the risks of being slow
  – Often better to get something working and get feedback on that, rather than study problem in the abstract for too long

Agile – Ready for Change – Simplicity

• Just-in-time design
  – Design and implement what you know right now
  – Put abstractions where you expect change
    • E.g., separate modules for code more likely to change

• No premature optimization
  – It is too hard to predict the performance bottleneck
  – Plus, the implementation will likely change

• Be ready for refactoring the code

Refactoring: Improving Code Design

• Make the code easier to read/use/extend
  – Change “how” code does something
  – Code functionality should not change

• Why refactor? Incremental feature extension often outgrows the initial design
  – Expected because of lack of extensive early design

• But needed often even for waterfall-model code
  – Plan for it and expect it
  – No point in trying to avoid it.
Agile – Ready for Change – Testing

• Frequent refactoring means frequent testing
  – Need to know that you don’t break previous features
  – You have to (re)test all features in future iterations

• Automated testing is a mandatory part of agile

• Agile implementation steps:
  – Develop a new feature AND write its automated tests
  – Refactor AND ensure tests pass

Conclusions

• Important to follow a good process

• Waterfall
  – top-down design, bottom-up implementation
  – Lots of upfront thinking, but slow, hard to iterate

• Iterative, or evolutionary processes
  – Build a prototype quickly, then evolve it
  – Postpone some of the thinking
  – Plan for change:
    • Be ready for frequent re-factoring/re-design
    • Develop automated tests as you go