CS162
Operating Systems and Systems Programming
Lecture 7

Programming Techniques and Teams

September 21, 2011
Anthony D. Joseph and Ion Stoica
http://inst.eecs.berkeley.edu/~cs162

Four requirements for Deadlock

- Mutual exclusion
  - Only one thread at a time can use a resource.
- Hold and wait
  - Thread holding at least one resource is waiting to acquire additional resources held by other threads
- No preemption
  - Resources are released only voluntarily by the thread holding the resource, after thread is finished with it
- Circular wait
  - There exists a set $\{T_1, \ldots, T_n\}$ of waiting threads
    - » T_1 is waiting for a resource that is held by T_2
    - » T_2 is waiting for a resource that is held by T_3
    - » …
    - » T_n is waiting for a resource that is held by T_1

Techniques for Preventing Deadlock

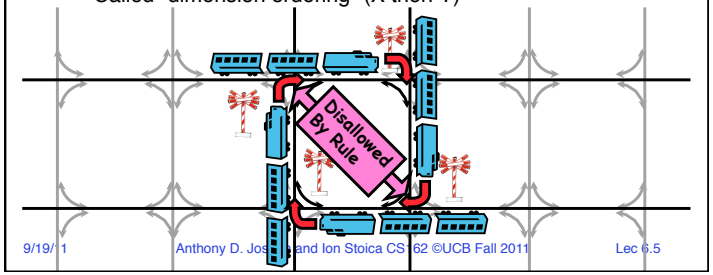
- Infinite resources
  - Include enough resources so that no one ever runs out of resources. Doesn't have to be infinite, just large
  - Give illusion of infinite resources (e.g. virtual memory)
  - Examples:
    - » Bay bridge with 12,000 lanes. Never wait!
    - » Infinite disk space (not realistic yet?)
- No Sharing of resources (totally independent threads)
  - Not very realistic
- Don't allow waiting
  - How the phone company avoids deadlock
    - » Call to your Mom in Toledo, works its way through the phone lines, but if blocked get busy signal
  - Technique used in Ethernet/some multiprocessor nets
    - » Everyone speaks at once. On collision, back off and retry

Techniques for Preventing Deadlock (con't)

- Make all threads request everything they'll need at the beginning
  - Problem: Predicting future is hard, tend to over-estimate resources
  - Example:
    - » If need 2 chopsticks, request both at same time
    - » Don't leave home until we know no one is using any intersection between here and where you want to go!
- Force all threads to request resources in a particular order preventing any cyclic use of resources
  - Thus, preventing deadlock
  - Example (x.P, y.P, z.P,…)
    - » Make tasks request disk, then memory, then…

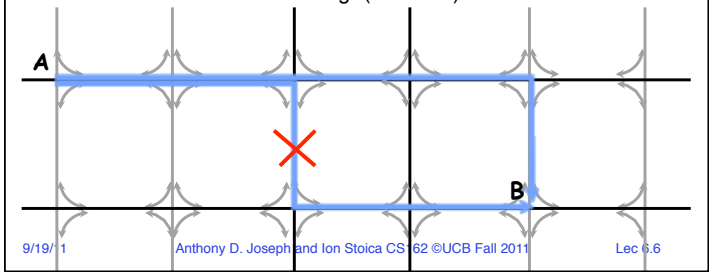
Review: Train Example (Wormhole-Routed Network)

- Circular dependency (Deadlock!)
  - Each train wants to turn right
  - Blocked by other trains
  - Similar problem to multiprocessor networks
- Fix? Imagine grid extends in all four directions
  - Force ordering of channels (tracks)
    - » Protocol: Always go east-west first, then north-south
  - Called "dimension ordering" (X then Y)



Train Example (Wormhole-Routed Network)

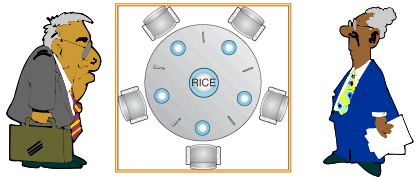
- Circular dependency (Deadlock!)
  - Each train wants to turn right
  - Blocked by other trains
  - Similar problem to multiprocessor networks
- Fix? Imagine grid extends in all four directions
  - Force ordering of channels (tracks)
    - » Protocol: Always go east-west first, then north-south
  - Called "dimension ordering" (X then Y)



Banker's Algorithm for Preventing Deadlock

- Toward right idea:
  - State maximum resource needs in advance
  - Allow particular thread to proceed if:
    (available resources - #requested) ≥ max remaining that might be needed by any thread
- Banker's algorithm (less conservative):
  - Allocate resources dynamically
    - » Evaluate each request and grant if some ordering of threads is still deadlock free afterward
    - » Technique: pretend each request is granted, then run deadlock detection algorithm, substituting ($[Max_{node}]-[Alloc_{node}] \le [Avail]$) for ($[Request_{node}] \le [Avail]$) Grant request if result is deadlock free (conservative!)
    - » Keeps system in a "SAFE" state, i.e. there exists a sequence $\{T_1, T_2, \ldots T_n\}$ with T_1 requesting all remaining resources, finishing, then T_2 requesting all remaining resources, etc..
  - Algorithm allows the sum of maximum resource needs of all current threads to be greater than total resources

Banker's Algorithm Example



- Banker's algorithm with dining philosophers
  - "Safe" (won't cause deadlock) if when try to grab chopstick either:
    - » Not last chopstick
    - » Is last chopstick but someone will have two afterwards
  - What if k-handed philosophers? Don't allow if:
    - » It's the last one, no one would have k
    - » It's 2nd to last, and no one would have k-1
    - » It's 3rd to last, and no one would have k-2
    - » …

Summary: Deadlock

- Starvation vs. Deadlock
  - Starvation: thread waits indefinitely
  - Deadlock: circular waiting for resources
- Four conditions for deadlocks
  - Mutual exclusion
    - » Only one thread at a time can use a resource
  - Hold and wait
    - » Thread holding at least one resource is waiting to acquire additional resources held by other threads
  - No preemption
    - » Resources are released only voluntarily by the threads
  - Circular wait
    - » ∃ set $\{T_1, \ldots, T_n\}$ of threads with a cyclic waiting pattern
- Deadlock preemption
- Deadlock prevention (Banker's algorithm)

Goals for Today

- Tips for Programming in a Project Team
- The Software Process

Note: Some slides and/or pictures in the following are adapted from slides ©2005 Silberschatz, Galvin, and Gagne. Slides courtesy of Anthony D. Joseph, John Kubiatowicz, AJ Shankar, George Necula, Alex Aiken, Eric Brewer, Ras Bodik, Ion Stoica, Doug Tygar, and David Wagner.

The Role of Software Engineering

- Developing software efficiently
  - » Minimize time
  - » Minimize dollars
  - » Minimize …
- First, we'll go through some tips for working in a team
  - [poll]
- Then, we'll talk about more formal processes

Tips for Programming in a Project Team

"You just have to get your synchronization right!"

- Big projects require more than one person (or long, long, long time)
  - Big OS: thousands of person-years!
- It's very hard to make software project teams work correctly
  - Doesn't seem to be as true of big construction projects
    - » Empire state building finished in one year: staging iron production thousands of miles away
    - » Or the Hoover dam: built towns to hold workers

Big Projects

- What is a big project?
  - Time/work estimation is hard
  - Programmers are eternal optimistics (it will only take two days)!
    - » This is why we bug you about starting the project early
- Can a project be efficiently partitioned?
  - Partitionable task decreases in time as you add people
  - But, if you require communication:
    - » Time reaches a minimum bound
    - » With complex interactions, time increases!
  - Mythical person-month problem:
    - » You estimate how long a project will take
    - » Starts to fall behind, so you add more people
    - » Project takes even more time!

Techniques for Partitioning Tasks

- Functional
  - Person A implements threads, Person B implements semaphores, Person C implements locks…
  - Problem: Lots of communication across APIs
    - » If B changes the API, A may need to make changes
    - » Story: Large airline company spent $200 million on a new scheduling and booking system. Two teams "working together." After two years, went to merge software. Failed! Interfaces had changed (documented, but no one noticed). Result: would cost another $200 million to fix.
- Task
  - Person A designs, Person B writes code, Person C tests
  - May be difficult to find right balance, but can focus on each person's strengths (Theory vs systems hacker)
  - Since Debugging is hard, Microsoft has *two* testers for *each* programmer
- Most CS162 project teams are functional, but people have had success with task-based divisions [poll]

Communication

- More people mean more communication
  - Changes have to be propagated to more people
  - Think about person writing code for most fundamental component of system: everyone depends on them!
- Miscommunication is common
  - "Index starts at 0? I thought you said 1!"
- Who makes decisions? [poll]
  - Individual decisions are fast but trouble
  - Group decisions take time
  - Centralized decisions require a big picture view (someone who can be the "system architect")
- Often designating someone as the system architect can be a good thing
  - Better not be clueless
  - Better have good people skills
  - Better let other people do work

Coordination

- More people ⇒ no one can make all meetings!
  - They miss decisions and associated discussion
  - Example from earlier class: one person missed meetings and did something group had rejected
  - Why do we limit groups to 5 people?
    - » You would never be able to schedule meetings otherwise
  - Why do we require 4 people minimum?
    - » You need to experience groups to get ready for real world
- People have different work styles
  - Some people work in the morning, some at night
  - How do you decide when to meet or work together?
- What about project slippage?
  - It will happen, guaranteed!
  - Ex: everyone busy but not talking. One person way behind. No one knew until very end – too late!
- Hard to add people to existing group
  - Members have already figured out how to work together

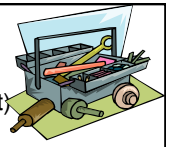
How to Make it Work?

- People are human. Get over it.
  - People will make mistakes, miss meetings, miss deadlines, etc. You need to live with it and adapt
  - It is better to anticipate problems than clean up afterwards.
- Document, document, document
  - Why Document?
    - » Expose decisions and communicate to others
    - » Easier to spot mistakes early
    - » Easier to estimate progress
  - What to document?
    - » Everything (but don't overwhelm people or no one will read)
  - Standardize!
    - » One programming format: variable naming conventions, tab indents,etc.
    - » Comments (Requires, effects, modifies)—javadoc?

Suggested Documents for You to Maintain

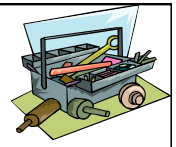
- Project objectives: goals, constraints, and priorities
- Specifications: the manual plus performance specs
  - This should be the first document generated and the last one finished
- Meeting notes
  - Document all decisions
  - You can often cut & paste for the design documents
- Schedule: What is your anticipated timing?
  - This document is critical!
- Organizational Chart
  - Who is responsible for what task?

Use Software Tools



- Source revision control software (CVS, SVN, git)
  - Easy to go back and see history
  - Figure out where and why a bug got introduced
  - Communicates changes to everyone (use RCS's features)
- Use an Integrated Development Environment
  - Structured development model
- Use automated testing tools
  - Write scripts for non-interactive software
  - Use "expect" for interactive software
  - Microsoft rebuilt Vista, W7 kernels every night with the day's changes. Everyone ran/tested the latest software
- Use E-mail and instant messaging consistently to leave a history trail

Integrated Development Environments

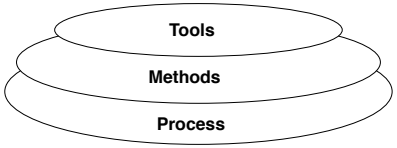


- Structured Compile-Edit-Debug environment
  - Organizes top-level projects, folders, and files in a hierarchical structure
  - Makes it easy to find uses of variables, procedures, …
  - Formats code for easier interaction
  - Interacts with version control infrastructure
- Projects consist of:
  - Files, interdependencies, configurations, version control information, etc.
  - May also manage non-project information:
    - » Global preferences, windows layout, search and navigation history, local change history (like version control, but local changes only)
- Different IDEs support different languages
  - MS Visual Studio (C/C++/C#/.NET), IBM Eclipse (Java)

Test Continuously

- Integration tests all the time, not at 11pm on due date!
  - Write dummy stubs with simple functionality
    - » Let's people test continuously, but more work
  - Schedule periodic integration tests
    - » Get everyone in the same room, check out code, build, and test.
    - » Don't wait until it is too late!
- Testing types:
  - Unit tests: white-/black-box check each module in isolation (use JUnit?)
  - Daemons: subject code to exceptional cases
  - Random testing: Subject code to random timing changes
- Test early, test later, test again
  - Tendency is to test once and forget; what if something changes in some other part of the code?

Software Engineering Layers



- Process: framework of the required tasks
  - e.g., waterfall, extreme programming
- Methods: technical "how to"
  - e.g., design review, code review, testing, etc.
- Tools: automate processes and methods

The Software Process

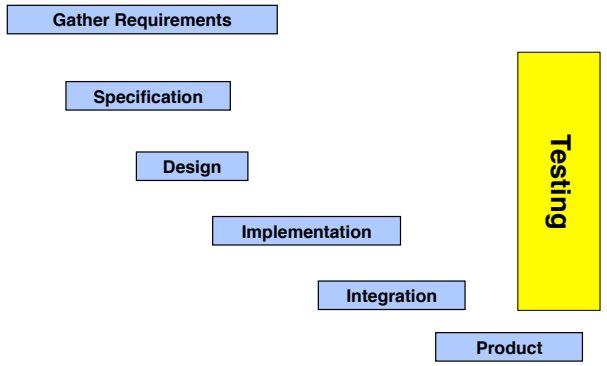
- Most projects follow recognized stages
  - From inception to completion
- These steps are a "software process"
  - Arrived at by trial and (lots of) error
- Process = *how* things are done
  - In contrast to what is done
- Ideal Project (to me)
  - Core functionality is reasonably attainable
  - But extra features are cool and can be implemented as time permits

Administrivia

- Proj 1 individual part and group design due Tue 9/27 11:59P
- Section changes posted to Piazza
- CSUA Hackathon Sept 23-24
  - The goal: code something cool!
  - What: 18 hour coding marathon in teams - great prizes like iPad2's!
  - When: Sept 23-24, i.e. 6pm Friday night to Saturday morning
  - Where: 4th floor Soda, Wozniak Lounge
  - Sponsor: Yahoo!

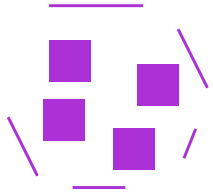
5min Break

Waterfall Process Phases



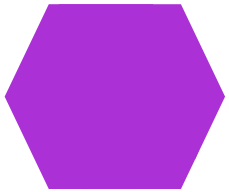
1. Gather Requirements

- Figure out what this thing is supposed to do
  - A raw list of features
  - Written down . . .
- Usually a good idea to talk to users, clients, or customers!
  - But note, they don't always know what they want
- Purpose: Make sure we don't build the wrong thing



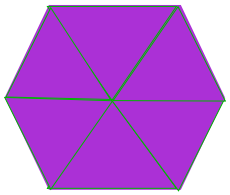
2. Specification

- A written description of *what* the system does
  - In all circumstances
    - » For all inputs
    - » In each possible state
  - Don't assume correct inputs or states!
- Because it covers all situations, much more comprehensive than requirements



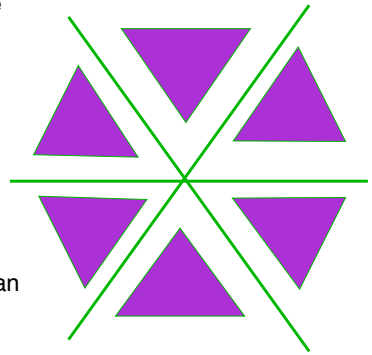
3. Design

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



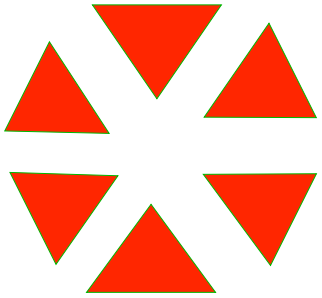
3. Design

- The system architecture
- Decompose system in modules
- Specify interfaces between modules
- Much more of *how* the system works, rather than *what* it does



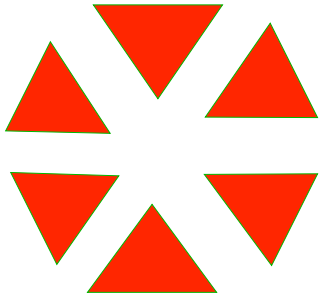
4. Implementation

- 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



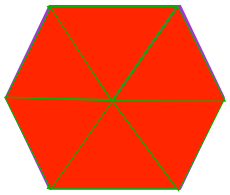
5. Integration

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



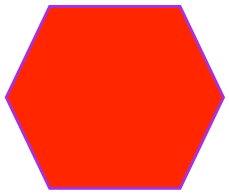
5. Integration

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



6. Product

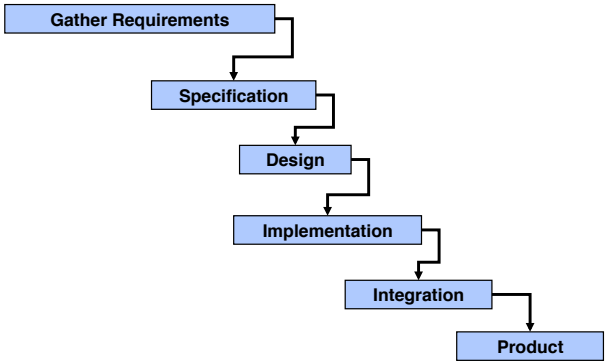
- Ship/Deploy and be happy!
- Actually, start maintenance…



A Software Process

- This is called the *waterfall model*
  - One of the standard models for developing software
- Each stage leads on to the next
  - No iteration or feedback between stages

The Waterfall Model



The Waterfall Model (Cont'd)

- There is testing after each phase
  - Verify the requirements, the spec, the design
  - Not just the coding and the integration
- Note the top-down design
  - Requirements, spec, design
- Bottom-up implementation
  - Implement, integrate, product

The Waterfall Model (Discussion)

- What are the risks with the waterfall model?
- The major risks are (my opinions):
  - Relies heavily on being able to accurately assess requirements at the start
  - Little feedback from users until very late
    - » Unless they understand specification documents
  - Problems in the specification may be found very late
    - » Coding or integration
  - Whole process can take a long time before the first working version is seen

My Opinions

- The waterfall model seems to be adopted from other fields of engineering
  - This is how to build bridges
- Not much software is truly built using the waterfall process
  - Where is it most, least applicable?
- But many good aspects
  - Emphasis on spec, design, testing
  - Emphasis on communication through documents

An Opinion on Time

- Time is the enemy of all software projects
- Taking a long time is inherently risky

"It is hard to make predictions, especially about the future"
- Yogi Berra

Why?

- The world changes, sometimes quickly
- Technologies become obsolete
  - Many products obsolete before they first ship!
- Other people produce competitive software
- 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

Case Study #1

- California DMV software (1987-1993)
- Attempt to merge driver & vehicle registration systems
  - Thought to take 6 years and $8 million
- Spent 7 years and $50 million before pulling the plug
  - Costs 6.5x initial estimate and expected delivery slipped to 1998 (or 11 years)!

Case Study #2

- FBI Virtual Case File system (2000-2003)
  - Trilogy project: thought to take 3 years and $380 million (including PC and networking upgrades)
- Replace FBI's Automated Case Support (ACS) software
  - Developed in-house by the bureau, considered obsolete when deployed in 1995…
- In 2002, Congress granted Trilogy another $123 million
  - In 2004, contractor requests another $50m, FBI pays contractor $16m to salvage system and another $2m to perform external review
- In 2005, FBI scraps project
  - Continues to use "obsolete" ACS…

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…

Something Faster: Rapid Prototyping

- Write a quick prototype
- Show it to users
  - Use to refine requirements
- Then proceed as in waterfall model
  - Throw away the prototype
  - Do spec, design, coding, integration, etc.

Fundamental Assumption

- We do not know much about the final product
  - No matter what we think
  - Environment will change
  - Requirements will change
  - Tools will change
  - Design will change
  - Better to roll with the punches than go for the KO
    - » (A terrible analogy)

Comments on Rapid Prototyping

- Hard to throw away the prototype
  - Slogan "the prototype is the product"
  - Happens more often than you might think!
  - Best way to avoid: write prototype in another language
- But prototyping is so useful
  - Much more realistic to show users a system rather than specification documents

More Comments

- A prototype exposes design mistakes
  - "Man, this is a pain in the butt to code up, even in the prototype"
- Easy to do with web technologies
  - Scripting languages are flexible
  - Browsers are forgiving
  - Much of the glue (CGI, etc.) is already there
  - Ruby on Rails makes everything easy
    - » Even using a DBMS

Opinions on Reality

- Neither of these models is true to life
- In reality, feedback between all stages
  - Specifications will demand refined requirements
  - Design can affect the specification
  - Coding problems can affect the design
  - Final product may lead to changes in requirements
    - » I.e., the initial requirements were incorrect!
- Waterfall model with "feedback loops"
  - Iterative model

What to Do?

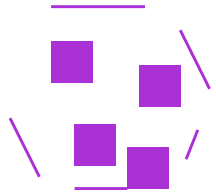
- Accept that later stages may force changes in earlier decisions
- And plan for it!
- The key: Minimize the risk
  - Recognize which decisions may need to be revised
  - Plan to get confirmation/refutation as soon as possible

Iterative Models: Plan for Change

- Use the same stages as the waterfall model
- But plan to iterate the whole cycle several times
  - Each cycle is a "build"
  - Smaller, lighter-weight than entire product
- Break the project into a series of builds which lead from a skeletal prototype to a finished product
- This is the model we use in Berkeley research projects!
  - Also used by Microsoft (internally), Google, Facebook, Twitter, and many others

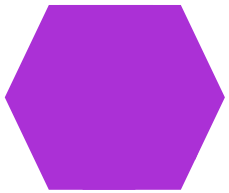
Gather Requirements

- Same idea as before
- Talk to users, find out what is needed
- But recognize diminishing returns
- Without something to show, probably cant get full picture of requirements on the first iteration



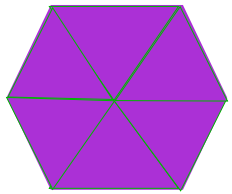
Specification

- A written description of *what* the system does
  - In all circumstances
    - » For all inputs
    - » In each possible state
  - Don't assume correct inputs or states!
- Still need this
  - Worth significant time
- Recognize it will evolve
  - Be aware of what aspects are under-specified



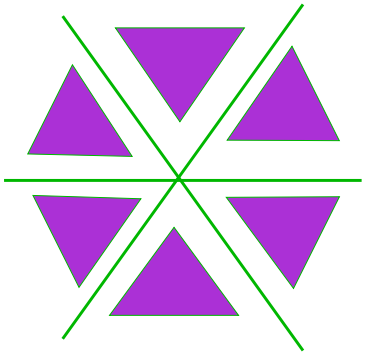
Design

- Decompose system into modules and specify interfaces
- Design for change
- Which parts are most likely to change?
  - Put abstraction there



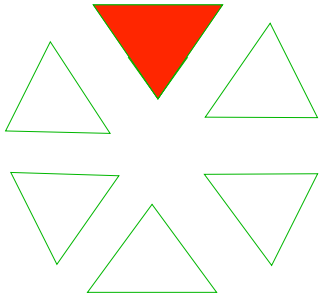
Design

- Decompose system into modules and specify interfaces
- Design for change
- Which parts are most likely to change?
  - Put abstraction there



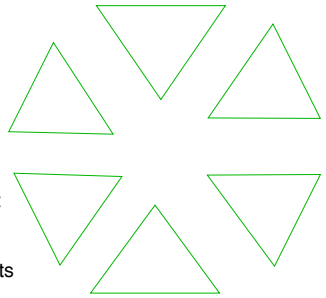
Design

- Plan incremental development of each module
- From skeletal component to full functionality
- From most critical to least critical features
  - Example: Two engineers at Facebook implemented photo sharing with just one feature – tagging



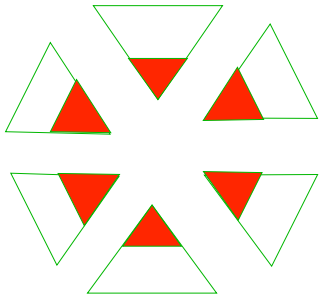
Implementation: Build 1

- Get a skeletal system working
- All the pieces are there, but none of them do very much
- But the interfaces are implemented
- This allows
  - A complete system to be built
  - Development of individual components to rely on all interfaces of other components



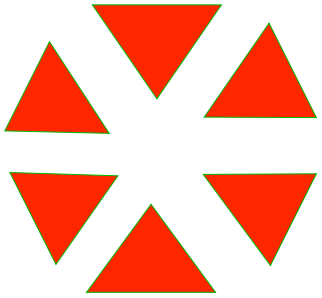
Implementation: Subsequent Builds

- After build 1, always have a demo to show (or product to deploy)
  - To customers
  - To the team
  - Communication!
- Each build adds more functionality



Integration

- Integration and major test for each build
- Stabilization point
- Continues until “last” build
  - But may begin shipping or deploying earlier builds



Advantages

- Find problems sooner
  - Get early feedback from users
  - Get early feedback on whether spec/design are feasible
- More quantifiable than waterfall
  - When build 3 of 4 is done, product is 75% complete
  - What percentage have we completed at the implementation stage of the waterfall model?

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

In Practice

- Most consumer software development uses the iterative model
  - Daily builds
  - System is *always* working
  - Always getting feedback
  - Microsoft, Google, Facebook, Twitter are well-known examples
- Many systems that are hard to test use something more like a waterfall model
  - E.g., unmanned space probes

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 (and ship/deploy it), then evolve it over time
  - Postpone some of the thinking