Lecture 2:
COS 333 Project Information
and a start on scripting
Project

“a simulation of reality”

a “three-tier” system for any interesting/useful application

groups of 3-5
Some familiar examples

- Point
- PTX
- Events
- Rooms
- ICE
- TigerFinder
- Pursuit of Mappiness
- EasyPCE
- PrincetonCAT
- Find-A
- Tigerbook
- ReCal/ReCal.io
The way to get startup ideas is not to try to think of startup ideas. It's to look for problems, preferably problems you have yourself.

The very best startup ideas tend to have three things in common: they're something the founders themselves want, that they themselves can build, and that few others realize are worth doing.

Paul Graham, co-founder of Y Combinator
www.paulgraham.com
Getting started

• **think about potential projects; form a group**
  talk to TA's & bwk; look at previous projects;
  look around you; check out the external project ideas page

• **by Wed Mar 15: each group meets with bwk (earlier is better)**
  – to be sure your project idea is generally ok
  – you should have one pretty firm idea, not several vague ones

• **Sun Mar 19: design document (before break)**
  – ~3-5 pages of text, pictures, etc.
  – overview of major pieces, how they fit together
  – initial web page, elevator speech
  – milestones: clearly defined pieces either done or not
  – risks

• **must be based on significant thought and discussion**

• **don't throw it together at the last minute**
  – all components of the project are graded
Process: organizing what to do

• you must use an orderly process or it won't work

• this is NOT a successful process:
  – talk about the software at dinner
  – hack some code together
  – test it a bit
  – do some debugging
  – fix the obvious bugs
  – repeat from the top until the semester ends
Process: organizing what to do

classic "waterfall" model: a very formal process

specification
requirements
architectural design
detailed design
coding
integration
testing
delivery

this is overkill for 333,
but some process is essential
Informal process

- **conceptual design**
  - roughly, what are we doing? *scenarios / use cases, sketches, screenshots*

- **requirements ("what")**
  - precise ideas about what it should do
  - explore options & alternatives on paper
  - specify more carefully with written docs

- **design or architecture ("how")**
  - map out structure and appearance with diagrams, prototypes
  - partition into major subsystems or components
  - specify interactions and *interfaces* between components
  - decide pervasive design issues: languages, environment, database, ...
  - make versus buy decisions and what you can use from elsewhere
  - resolve issues of connectivity, access to data, software, etc.

- **implementation ("what by when")**
  - *make prototypes; establish end to end connectivity*
  - get real users as early as possible
  - deliver in stages, so that each does something and still works
  - test as you go: if your system is easy to break, it gets a lower grade
The academic software life cycle

- requirements
- design
- build
- test
- debug
Choices!

**Wire format:**
- XML, JSON, REST, ...

**Business logic:**
- Java, C#, Python, PHP, Ruby, Node, C++, Objective-C, Swift, Perl, Go, ...

**Web client:**
- HTML, CSS Javascript, ...

**Frameworks:**
- jQuery, React, Angular, Vue, ...

**GUI tools:**
- Swing, jQueryUI, Bootstrap, ...

**Server:**
- OIT MyCpanel, AWS, Heroku, Google Cloud, ...

**Web frameworks:**
- Django, Flask, Zend, Rails, Cocoa, Express, ...

**Devel Environ:**
- sh + vi / emacs Eclipse, Xcode, Visual Studio, Android Dev Kit ...

**Plumbing:**
- TCP/IP, OAuth, CAS, ...

**Devel Environ:**
- Git, Github SVN, ...

**Database:**
- MySQL, SQLite, Postgres, MongoDB, ...

**Repository:**
- Git, Github SVN, ...
“Make versus buy”

- you can use components and code from elsewhere
  - copy or adapt open source

- overall project design has to be your own
- so does selection and assembly of components
- so does the bulk of the work

- it's fine to build on what others have done
  - identify what you have used, where it came from

- it’s fine to cooperate with other project groups
  - help each other with insight, knowledge, …
Things to keep in mind

• project management
  – everyone has to pull together, someone has to be in charge

• architecture
  – how do the pieces fit together?
  – make it work like the product of a single mind but with multiple developers
    "Good interfaces make good neighbors"?

• user interface
  – what does it look like?
  – make it look like the product of a single mind

• development
  – everyone has to do a significant part of the coding

• quality assurance / testing
  – make sure it always works
    should always be able to compile and run it: fix bugs before adding features

• documentation
  – internals doc, web page, advertising, presentation, final report, ...

• risks
  – what could go wrong?
  – what are you dependent on that might not work out?
Things to do from the beginning

• think about schedule
  – keep a log of what you did and what you will do next (always current)
• plan for a sequence of stages
  – do not build something that requires a "big bang" where nothing works until everything works
  – always be able to declare success and walk away
• simplify
  – don't take on too big a job
  – don't try to do it all at the beginning, but don't try to do it all at the end
• use source code control for everything
  – Git or equivalent is mandatory
• leave lots of room for "overhead" activities
  – testing: build quality in from the beginning
  – documentation: you have to provide written material
  – deliverables: you have to package your system for delivery
  – changing your mind: decisions will be reversed and work will be redone
  – disaster: lost files, broken hardware, overloaded systems, ...
  – sickness: you will lose time for unavoidable reasons
  – health: there is more to life than this project!
2017 Project Schedule

February

Su Mo Tu We Th Fr Sa
Feb
1 2 3 4
5 6 7 8 9 10 11
12 13 14 15 16 17 18
19 20 21 22 23 24 25
26 27 28

March

Su Mo Tu We Th Fr Sa
Mar
1 2 3 4
5 6 7 8 9 10 11
12 13 14 15 16 17 18
19 20 21 22 23 24 25
26 27 28 29 30 31

April

Su Mo Tu We Th Fr Sa
Apr
1
2 3 4 5 6 7 8
9 10 11 12 13 14 15
16 17 18 19 20 21 22
23 24 25 26 27 28 29
30

May

Su Mo Tu We Th Fr Sa
May
1 2 3 4 5 6
7 8 9 10 11 12 13
14 15 16 17 18 19 20
21 22 23 24 25 26 27

You are here

Start thinking about possible projects & people

Your team should be formed; schedule meeting with bwk
Initial project discussions with bwk by 3/14
Design document due by 3/19
Spring break -- don't waste it
Weekly TA meetings start this week

Project prototype
Alpha test

Last class; beta test
Demo days this week (probably M-W)
Project due by midnight Sunday
Some mechanics

- groups of 3 to 5
  - find your own partners
    - use Piazza for match-making
    - meet potential partners before or after class
  - don't leave this to the end !!

- TA's will be your first-level "managers"
  - more mentoring and monitoring than managing
    - it's your project, not the TA's

- meet with your TA every week after spring break
  - everyone in the group must attend all of these meetings

- be prepared
  - what we accomplished
  - what we didn't get done
  - what we do plan to do next

- these meetings are a graded component
  - this is an attempt to make sure that you don't leave it all to the end
Scripting languages

- originally tools for quick hacks, rapid prototyping, gluing together other programs, ...
- evolved into mainstream programming tools
- characteristics
  - text strings as basic (or only) data type
  - regular expressions (maybe built in)
  - associative arrays as a basic aggregate type
  - minimal use of types, declarations, etc.
  - usually interpreted instead of compiled

- examples
  - shell
  - Awk
  - Perl, PHP, Python, Ruby, Tcl, Lua, ...
  - Javascript
  - Visual Basic, (VBIWIC)Script, PowerShell
  - ...
Shells and shell programming

• shell: a program that helps run other programs
  – intermediary between user and operating system
  – basic scripting language: programming with programs as building blocks
• an ordinary program, not part of the system
  – it can be replaced by one you like better
  – therefore there are lots of shells, reflecting history and preferences
• popular Unix shells:
  – sh  Bourne shell (Steve Bourne, Bell Labs -> ... -> El Dorado Ventures)
    emphasizes running programs and programmability; syntax derived from Algol 68
  – csh  C shell   (Bill Joy, UC Berkeley -> Sun -> Kleiner Perkins)
    interaction: history, job control, command & filename completion, aliases
    more C-like syntax, but not as good for programming (at least historically)
  – ksh  Korn shell (Dave Korn, Bell Labs -> AT&T Labs -> Google)
    combines programmability and interaction
    syntactically, superset of Bourne sh; provides all csh interactive features
    + lots more
  – bash  GNU shell
    mostly ksh + much of csh
  – tcsh  evolution of csh
  – zsh  (written in 1990 by Paul Falstad ‘92)
    evolution of Bourne shell + some bash, ksh, csh
Features common to Unix shells

- **command execution**
  - built-in commands, e.g., `cd`

- **filename expansion**
  - `*` ? `[…]`

- **quoting**
  - `rm '*'` Careful !!!
  - `echo "It's now `date`"`

- **variables, environment**
  - `PATH=/bin:/usr/bin` in ksh & bash
  - `setenv PATH /bin:/usr/bin` in (t)csh

- **input/output redirection, pipes**
  - `prog <in >out, prog >>out`
  - `who | wc`
  - `slow.1 | slow.2 &` asynchronous operation

- **executing commands from a file**
  - arguments can be passed to a shell file ($0, $1, etc.)
  - if made executable, indistinguishable from compiled programs

provided by the shell, not each program
Shell programming

- shell programs are good for personal tools
  - tailoring environment
  - abbreviating common operations
    (aliases do the same)
- gluing together existing programs into new ones
- prototyping
- sometimes for production use
  - e.g., configuration scripts

- But:
  - shell is poor at arithmetic, editing
  - macro processing is a mess
  - quoting is a mess
  - sometimes too slow
  - can't get at some xys that are really necessary

- this leads to scripting languages