333 Project

• a simulation of reality
  - building a substantial system
  - in groups of 3 to 5 people

• "three-tier" system for any application you like

• 3 major pieces
  - graphical user interface ("presentation layer")
  - processing in the middle ("business logic")
  - persistent storage / data management

• examples: many web-based services
  - Amazon, Facebook, Instagram, ...
  - news, information services, bots, mashups
  - email, chat, search, code tools, maps, ...
  - cellphone systems are often like this too

• your project
  - make something of roughly this structure
  - but smaller, simpler, defined by your interests
Some local examples

- Point
- PTX
- Events
- Rooms
- ICE
- TigerFinder
- Pursuit of Mappiness
- EasyPCE
- PrincetonCAT
- Find-A
- Tigerbook
Getting started

• right now, if not sooner
  - 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 Fri Mar 6: short meeting of group with bwk (earlier is desirable)
  - to be sure your project idea is generally ok
  - you should have one pretty firm consensus idea, not several vague ones
• Fri Mar 13: design document (before break)
  - ~3-4 pages of text, pictures, etc. A template will be posted
  - overview, initial web page, elevator speech
    project name / title, paragraph on what it is, one person as project manager
  - components & interfaces
    major design choices: web vs. standalone, languages, tools, environment, …
    major pieces, how they fit together
  - 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 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

• classic "waterfall" model: a real process
  specification
    requirements
      architectural design
      detailed design
    coding
    integration
    testing
    delivery

• this is overkill for 333, but some process is essential ...
The academic software life cycle

requirements | design | build | test | debug
Informal process

• conceptual design
  - roughly, what are we doing? make sketches, scenarios, screenshots

• requirements definition ("what")
  - precise ideas about what it should do
  - explore options & alternatives on paper
  - specify more carefully with written docs
  - this should not change a lot once you start

• architecture / design ("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 information, 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
Interfaces

• the boundary between two parts of a program
• a contract between the two parts
• what are the inputs and outputs?
• what is the transformation?
• who manages resources, especially memory and shared state?

• hide design & implementation decisions behind interfaces, so they can be changed later without affecting the rest of the program
  - database system, data representations and file formats
  - specific algorithms
  - visual appearance

• "I wish we had done interfaces better" is one of the most common comments
  - less often: "We thought hard about the interfaces so it was easy to make changes without breaking anything."
Choices (a small and incomplete list)

• user interface
  - browser, desktop, phone, game console, API, ...
  - HTML/CSS/LESS, Javascript, Flash, Jquery, Bootstrap, Swing, ...

• languages
  - C++, Java, C#, Objective C, Perl, Python, PHP, Ruby, Javascript, ...

• server
  - own machine, OIT, CS, Google AppEngine, Amazon AWS, Heroku, ...

• database
  - MySQL, SQLite, Postgres, MongoDB, Redis, ...

• information exchange formats
  - text, JSON, XML, REST, ...

• frameworks
  - Django, Flask, Rails, Express, Google Web Toolkit, ...

• development environments
  - XCode, Eclipse, Visual Studio, ...
“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
Other ways to think about it

• "elevator pitch"
  - what would you say if you were alone in an elevator with Bill Gates for 60 seconds?
    short attention-grabbing description without big words but good buzzwords

• 5-7 slides for a 5-10 minute talk or a poster session
  - what would be the titles and 2-3 points on each slide?

• 1 page advertisement
  - what would be the main selling points?
  - what would your web page look like?

• talk and demo outline for the end of the semester
  - what would you want working for the demo?

• business plan
  - how would you pitch it to a VC or Yagoosoftbook or at Princeton Pitch?
    - what does it do for who?
    - who would want it?
    - what's the competition?
    - what are the stages of evolution or major releases?

• job talk / interview
  - what did you do that's really cool?
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!
2015 Project Schedule

Su Mo Tu We Th Fr Sa
Feb  1  2  3  4  5  6  7  <- you are here
     8  9 10 11 12 13 14
     15 16 17 18 19 20 21
     22 23 24 25 26 27 28
Mar  1  2  3  4  5  6  7  initial talk with bwk
     8  9 10 11 12 13 14  design doc before break
     15 16 17 18 19 20 21  spring break (don't waste it)
     22 23 24 25 26 27 28  weekly project meetings begin
     29 30 31
Apr  1  2  3  4
     5  6  7  8  9 10 11  project prototype
     12 13 14 15 16 17 18
     19 20 21 22 23 24 25  alpha test
     26 27 28 29 30
May  1  2  beta test
     3  4  5  6  7  8  9  demo days: project presentations
     10 11 12 13 14 15 16  Dean's Date: all done
     17 18 19 20 21 22 23
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, (VB|W|C)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 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
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

• **the shell is a programming language**
  - the earliest scripting language

• **string-valued variables**

• **limited regexprs** mostly for filename expansion

• **control flow**
  - if-else
    ```bash
    if cmd; then cmds; elif cmds; else cmds; fi (bash)
    if (expr) cmds; else if (expr) cmds; else cmds; endif (csh)
    ```
  - while, for
    ```bash
    for var in list; do commands; done (bash)
    foreach var (list) commands; end (bash)
    ```
  - switch, case, break, continue, ...

• **operators are programs**
  - programs return status: 0 == success, non-0 == various failures

• **shell programming out of favor**
  - graphical interfaces
  - scripting languages
    e.g., system administration
    setting paths, filenames, parameters, etc
    now often in Perl, Python, PHP, ...
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 things that are really necessary

- this leads to scripting languages