Announcements
Announcements

• Last Friday’s Guest Speaker (Kendra Yourtee)
  – Sign thank-you card
  – Take survey: https://tinyurl.com/yay8m24s

• Campus Maps Demos Wednesday!
  – You don’t have to be finished with HW9
  – The first 10 volunteers will receive a special reward
  – Sign up here: https://tinyurl.com/yay092374

• Course evaluations – Please give feedback on this course!
  – You should have received an email from “UW Course Evaluations” with the link
  – https://uw.iasystem.org/survey/195871
Announcements

- Quiz 8 due Thursday 8/16
- Homework 9 due Thursday 8/16
- Final Exam Friday in class (60 minutes)
  - Covers all material after the midterm
  - Final exam review: during section Thursday 8/16
System
Development
Context

CSE331 is almost over…

- Focus on software design, specification, testing, and implementation
  - Absolutely necessary stuff for any nontrivial project

- But not sufficient for the real world: At least 2 key missing pieces
  - Techniques for larger systems and development teams
    - This lecture; yes fair game for final exam
    - Major focus of CSE403 (Software Engineering)
  - Usability: interfaces engineered for humans
    - Another lecture: didn’t fit this quarter
    - Major focus of CSE440 (HCI)
Outline

• Software architecture

• Tools
  – For build management
  – For version control
  – For bug tracking

• Scheduling

• Implementation and testing order
Software Architecture
Architecture

**Software architecture** refers to the high-level structure of a software system

- A principled approach to partitioning the modules and controlling dependencies and data flow among the modules

Common architectures have well-known names and well-known advantages/disadvantages

A good architecture ensures:

- Work can proceed in parallel
- Progress can be closely monitored
- The parts combine to provide the desired functionality
Example architectures

Pipe-and-filter (think: iterators)

Source → pipe → Filter → pipe → Filter → pipe → Filter → pipe → Sink

Blackboard
(think: callbacks)

Layered
(think: levels of abstraction)
A good architecture allows:

• Scaling to support large numbers of ______
• Adding and changing features
• Integration of acquired components
• Communication with other software
• Easy customization
  – Ideally with no programming
  – Turning users into programmers is good
• Software to be embedded within a larger system
• Recovery from wrong decisions
  – About technology
  – About markets
System architecture

• Have one!
• Subject it to serious scrutiny
  – At relatively high level of abstraction
  – Basically lays down communication protocols
• Strive for simplicity
  – Flat is good
  – Know when to say no
  – A good architecture rules things out
• Reusable components should be a design goal
  – Software is capital
  – This will not happen by accident
  – May compete with other goals the organization behind the project has (but less so in the global view and long-term)
Temptations to avoid

• Avoid featuritis
  – Costs under-estimated
    • Effects of scale discounted
  – Benefits over-estimated
    • A Swiss Army knife is rarely the right tool

• Avoid digressions
  – Infrastructure
  – Premature tuning
    • Often addresses the wrong problem

• Avoid quantum leaps
  – Occasionally, great leaps forward
  – More often, into the abyss
Outline

• Software architecture

• Tools
  – For build management
  – For version control
  – For bug tracking

• Scheduling

• Implementation and testing order
Tools
Build tools

• Building software requires many tools:
  – Java compiler, C/C++ compiler, GUI builder, Device driver build tool, InstallShield, Web server, Database, scripting language for build automation, parser generator, test generator, test harness
• Reproducibility is essential
• System may run on multiple devices
  – Each has its own build tools
• Everyone needs to have the same toolset!
  – Wrong or missing tool can drastically reduce productivity
• Hard to switch tools in mid-project

If you’re doing work the computer could do for you, then you’re probably doing it wrong
Code Review

Leah Perlmuter @lrperlmu commented about a minute ago
avoid importing *

Reply...

Leah Perlmuter @lrperlmu commented less than a minute ago
what is this used for?

Reply...

filename = '/home/lrperlmu/research_ws/src/pr2_assist/trp_main/config/script.json'
fp = open(filename)
script_json = json.load(fp)
json_commands = script_json['scripted_tasks']
self.commands = {}
for cmd in json_commands:
Version control (source code control)

• A version control system lets you:
  – Collect work (code, documents) from all team members
  – Synchronize team members to current source
  – Have multiple teams make progress in parallel
  – Manage multiple versions, releases of the software
  – Identify regressions more easily
• Example tools:
  – Subversion (SVN), Mercurial (Hg), Git
• Policies are even more important
  – When to check in, when to update, when to branch and merge, how builds are done
  – Policies need to change to match the state of the project
• Always diff before you commit
Issue tracking

• An issue tracking system supports:
  – The team’s to-do list
    • who will do each work item and when
  – Tracking and fixing bugs and regressions
  – Communicating among team members

• Essential for any non-small or non-short project

• Example tools:
  – cloud hosted: Google Developers, GitLab, GitHub, Bitbucket, Jira, Trello
  – host your own: Bugzilla, Flyspray, Trac
Issue tracking

Need to configure the bug tracking system to match the project
  – Many configurations can be too complex to be useful
A good process is key to managing bugs
  – An explicit policy that everyone knows, follows, and believes in
Outline

• Software architecture

• Tools
  – For build management
  – For version control
  – For bug tracking

• Scheduling

• Implementation and testing order
Scheduling and Scoping
Scheduling and Scoping

“More software projects have gone awry for lack of calendar time than for all other causes combined.”

-- Fred Brooks, *The Mythical Man-Month*

Three central questions of the software business

3. When will it be done?
2. How much will it cost?
1. When will it be done?

- Estimates are almost always too optimistic
- Estimates reflect what one wishes to be true
- We confuse effort with progress
- Progress is poorly monitored
- Slippage is not aggressively treated
Some wry wisdom...

A project expands to fill up the time you have available for it.

Hofstadter’s Law: It always takes longer than you expect, even when you take into account Hofstadter's Law.
SMART goals

The name is cheesy, but it’s a valuable concept

Specific
Measurable*****
Achievable
Relevant
Timebound*****

• Work on HW9
  – when? how much work?
• Work on HW9 by 5pm on Wednesday 8/15
  – how much work?
• Get HW9 mostly done by 5pm on Wednesday 8/15
  – what does “mostly done” mean?
• Get HW9 completely done by 5pm on Thursday 8/16
Milestones in a Software Project

• Milestones are critical keep the project on track
  – Policies may change at major milestones
  – Check-in rules, build process, etc.

• Some typical milestones (names)
  – Design complete
  – Interfaces complete / feature complete
  – Code complete / code freeze
  – Alpha release
  – Beta release
  – Release candidate (RC)
  – FCS (First Commercial Shipment) release
Effort is not the same as progress

**Effort** is the amount of time spent earnestly working on the project
- Can be equated with number of hours
- **Cost** of the project (salary paid to workers) is proportional to effort

**Progress** involves reaching milestones
- Hard to track, because it is hard to make good milestones

• Often lots of effort leads to little progress
  - This is normal! Much experience gained!
    • but for some reason, managers don’t seem to like it
      - (see cost)
  - Be honest with yourself.
    • You can’t just “catch up before anyone notices”
- Need to adjust the schedule
When you know you will miss a milestone...

Change the scope and/or the due date.

• Option A: Later deadline, same amount of work
• Option B: Same deadline, less work
• Option C: Same deadline, same amount of work
• Option D: Later deadline, and more work

• Which of these will set you up for success?
  – only A and B.
• Options C and D are implemented surprisingly frequently, often with painful results.
Dealing with slippage

• People must be held accountable
  – Slippage is not inevitable
  – Software should be on time, on budget, and on function

• Four options
  – Add people – startup cost ("mythical man-month")
  – Buy components – hard in mid-stream
  – Change deliverables – customer must approve
  – Change schedule – customer must approve

• Take no small slips
  – One big adjustment is better than three small ones
It’s a learning process!

• Scoping and time management, like other skills, can be learned!
• Delivering stuff late just means you have not yet learned good time management (growth potential!)
  – might have consequences, but not the end of the world
• Make sure to change your process for the next time

• *Retrospective* – discussing/analyzing past work in order to learn how to improve your (team’s) process
Outline

• Software architecture

• Tools
  – For build management
  – For version control
  – For bug tracking

• Scheduling

• Implementation and testing order
Implementation and Testing Order
How to code and test your design

• You have a design and architecture
  – Need to code and test the system

• Key question, what to do when?

• Suppose the system has this module dependency diagram
  – In what order should you address the pieces?
Bottom-up

• Implement/test children first
  – For example: G, E, B, F, C, D, A
• First, test G stand-alone (also E)
  – Generate test data
  – Construct test driver to run low-level components
• Next, implement/test B, F, C, D
• No longer unit testing: use lower-level modules
  – A test of module M tests:
    • whether M works, and
    • whether modules M calls behave as expected
  – When a failure occurs, many possible sources of defect
  – Integration testing is hard, irrespective of order
Top-down

- Implement/test parents (clients) first
  - Here, we start with A

- To run A, build *stubs* to simulate B, C, and D
  - Also known as *mocking*.
    - Tools: Mockito, PowerMock, ...

- Next, choose a successor module, e.g., B
  - Build a stub for E
  - Drive B using A

- Suppose C is next
  - Can we reuse the stub for E?
Implementing a stub or mock object

• Query a person at a console
  – Same drawbacks as using a person as a driver

• Print a message describing the call
  – Name of procedure and arguments
  – Fine if calling program does not need result
    • More common than you might think

• Provide “canned” or generated sequence of results
  – Often sufficient
  – Generate using criteria used to generate data for unit test
  – May need different stubs for different callers

• Provide a primitive (inefficient & incomplete) implementation
  – Best choice, if not too much work
  – Look-up table often works
  – Sometimes called “mock objects” (ignoring technical definitions?)
Comparing top-down and bottom-up

• Criteria
  – What kinds of errors are caught when?
  – How much integration is done at a time?
  – Distribution of testing time?
  – Amount of work?
  – What is working when (during the process)?

• Neither dominates
  – Useful to understand advantages/disadvantages of each
  – Helps you to design an appropriate mixed strategy
Catching design errors

• Top-down tests global decisions first
  – E.g., what system does
  – Most devastating place to be wrong
  – Good to find early

• Bottom-up uncovers efficiency problems earlier
  – Constraints often propagate downward
  – You may discover they can’t be met at lower levels
Amount of work

- Always need test harness
- Top-down
  - Build stubs but not drivers
- Bottom-up
  - Build drivers but not stubs
- Stubs are usually more work than drivers
  - Particularly true for data abstractions
- On average, top-down requires more non-deliverable code
  - Not necessarily bad
What components work, when?

• Bottom-up involves lots of invisible activity
  – 90% of code written and debugged
  – Yet little that can be demonstrated

• Top-down depth-first
  – Earlier completion of useful partial versions
Regression testing

• Ensure that things that used to work still do
  – Including performance
  – Whenever a change is made

• Knowing exactly when a bug is introduced is important
  – Keep old test results
  – Keep versions of code that match those results
  – Storage is cheap
Perspective...

• Software project management is challenging
  – There are still major disasters – projects that go way over budget, take much longer than planned, or are abandoned after large investments
  – We’re better at it than we used to be, but not there yet (is “software engineering” real “engineering”?)

• Project management is a mix of hard and soft skills

• We’ve only skimmed the surface
  – Next: CSE 403, internship/real world, ???
Announcements
Announcements

• Last Friday’s Guest Speaker (Kendra Yourtee)
  – Sign thank-you card
  – Take survey: https://tinyurl.com/yay8m24s

• Campus Maps Demos Wednesday!
  – You don’t have to be finished with HW9
  – The first 10 volunteers will receive a special reward
  – Sign up here: https://tinyurl.com/yay092374

• Course evaluations – Please give feedback on this course!
  – You should have received an email from “UW Course Evaluations” with the link
  – https://uw.iasystem.org/survey/195871
Announcements

- Quiz 8 due Thursday 8/16

- Homework 9 due Thursday 8/16

- Final Exam Friday in class (60 minutes)
  - Covers all material after the midterm
  - Final exam review: during section Thursday 8/16
Bonus Material!

Download the slides in .pptx format to see material on “hidden slides” not presented in this quarter’s lecture. (Hidden slides not visible in PDF.)