CSE 331
Software Design & Implementation
Topic: Real-world Systems

💬 Discussion: Should I make next lecture a work session?
Reminders

• Will not accept *any* work after Aug. 18 (Friday) at 11pm
• Please fill out course evals ASAP
  • Very valuable to me as a new instructor!

Upcoming Deadlines

• Prep. Quiz: HW9  due Monday (8/13)
• HW9  due Thursday (8/17)
Last Time...

• More Design Patterns!
  • Creational
  • Behavioral
  • Structural

Today’s Agenda

• End-of-quarter timeline
  • Lectures
  • Final Grades
• System Integration
Lecture Timeline

Last few lectures will be content-sparse. Will have lots of work time during class.

- Today's lecture is about the high-level ideas needed to build massive systems
- Next lecture will include information on ethics in CS and a class discussion on related topics
- Friday’s lecture will start with student demos and end with a course wrap-up
Grading Timeline

- All work needs to be submitted to us by **Friday** at 11pm

- Course staff will finish grading HW9 and regrades by **Sunday evening**

- Grades will be posted on Canvas on late **Sunday night**
  - You should check that these are consistent with what you expected!

- Soham will calculate final GPA based on these grades on **Monday**
  - This is when I will account for extra credit and special circumstances

- Grades due to the university on **Tuesday at noon**
What we didn’t do...

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
    • Major focus of CSE403
  – Usability: interfaces engineered for humans
    • Major focus of CSE440 – something you should take!
Outline

• Software architecture

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

• Scheduling and Planning ahead

• Implementation and testing order
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 and disadvantages, just like design patterns

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)

Blackboard (think: callbacks)

Layered (think: levels of abstraction)
Good architecture considers many things!

• 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!
  - Basically lays down communication protocols and a project plan
- Subject it to serious scrutiny
  - At relatively high level of abstraction
- 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 of the organization (but less so in the global view and long-term)
Things to Avoid

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

Premature optimization is the root of all evil
- Don Knuth
Outline

• Software architecture

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

• Scheduling and Planning ahead

• Implementation and testing order
Build tools

- Building software requires many tools:
  - Java compiler/JVM, C/C++ compiler, GUI builder, react/node/framework-du-jour, 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*
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 work in parallel
  – Manage multiple versions, releases of the software
  – Identify regressions more easily
• Example tools:
  – Git, Mercurial (Hg), Buck, Subversion (SVN), ...
• 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 pull and diff before you commit
Bug tracking

• An issue tracking system supports:
  – Tracking and fixing bugs
  – Identifying problem areas and managing them
  – Communicating among team members
  – Tracking regressions and repeated bugs

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

• Example tools:
  JIRA, Bugzilla, Flyspray, Trac, ...
  Hosted tools (GitLab, GitHub, Sourceforge, ...)

CSE 331 Summer 2023
Bug 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
Feature Rollout
Feature Rollout
Feature Rollout

Testing  Canary / Staging  Production
Outline

• Software architecture

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

• Scheduling and Planning ahead

• Implementation and testing order
Scheduling

“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
Scheduling is crucial but underappreciated

- Scheduling is underappreciated
  - Made to fit other constraints
- A schedule is needed to make slippage visible
- Unrealistically optimistic schedules are a disaster

- The great paradox of scheduling:
  - Everything takes \textit{twice as long} as you think
  - Hofstadter’s Law: It always takes longer than you expect, even when you take into account Hofstadter's Law
Effort is not the same as progress

Cost is the product of workers and time
- Reasonable approximation: All non-labor costs (everything but salary/benefits) are zero (!)
- Easy to track

Progress is more complicated and hard to track

- People don’t like to admit lack of progress
  - Progress is mis-estimated
  - Think they can catch up before anyone notices

- Design the process and architecture to facilitate tracking
How does a project get to be one year late?

One day at a time...
• It’s not the hurricanes that get you
• It’s the termites
  – Arjun missed a meeting
  – Sarah’s keyboard broke
  – The compiler wasn’t updated
  – ...

If you find yourself ahead of schedule
  – Don’t relax
  – Don’t add features
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 staff-month")
  – Buy components – hard in mid-stream
  – Change deliverables – customer must approve
  – Change schedule – customer must approve

• How can you build an environment that supports people falling behind and helps people catch up?
Outline

• Software architecture

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

• Scheduling and Planning ahead

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

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

• 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 as discussed earlier
  - Construct test drivers
- Next, implement/test B, F, C, D
- No longer unit testing: using lower-level modules
  - A test of module M tests:
    - whether M works, and
    - whether modules that M calls behave as expected
  - When a failure occurs, many possible sources of defect
  - Integration testing is hard, irrespective of order
Building drivers

• Use a person
  – *Simplest* choice, but also *worst* choice
  – Errors in entering data are inevitable
  – Errors in checking results are inevitable
  – Tests are not easily reproducible
    • Problem for debugging
    • Problem for regression testing
  – Test sets stay small, don’t grow over time
  – Testing cannot be done as a background task

• Better alternative: Automated drivers in a test harness
Top-down

- Implement/test parents (clients) first
  - Here, we start with A
- To run A, build *stubs* to simulate B, C, and D
- 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?
    (Maybe, but maybe need something different)
Implementing a stub

• 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” or fakes
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

• Neither dominates
  – Useful to understand advantages/disadvantages of each
  – Helps you to design an appropriate mixed strategy
Amount of integration at each step

• Less is better

• Top-down adds one module at a time
  – When an error is detected, either:
    • Lower-level module doesn’t meet specification
    • Higher-level module tested with bad stub

• Bottom-up adds one module at a time
  – Connect it to multiple modules
  – Thus integrating more modules at each step
  – More places to look for error
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
- Most of the work is in integration two different modules
One good way to structure an implementation

• Largely top-down
  – But always unit test modules
• Bottom-up
  – When stubs are too much work [just implement real thing]
  – Low level module that is used in lots of places
  – Low-level performance concerns
• Depth-first, visible-first
  – Allows interaction with customers, like prototyping
  – Lowers risk of having nothing useful
  – Improves morale of customers and programmers
  • Needn’t explain how much invisible work done
  • Better understanding of where the project is
  • Don’t have integration hanging over your head
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
  – Disasters usually stem from lack of discipline
  – Always new challenges; we never build the same thing twice
  – 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 [so-called] soft skills

• We’ve only skimmed the surface
  – Next: CSE 403, internship, your startup, ???
Before next class...

1. Start on HW9
   - Let me know if you want to demo extra credit
     • Can be small things like a slightly different layout
     • Can be big things like adding a whole new feature

2. Wrap-up any regrades for HW1-8
   - Won’t accept late work after the last day of class

3. Please fill out course evals! I genuinely care about what you have to say.