Almost to the Finish Line...

• CS 326 has been all about software design, specification, testing, and implementation
  – Absolutely necessary for any nontrivial project

• But not sufficient for the real world
  – Software Engineering: Techniques for larger systems and development teams
    • architecture, tools, scheduling, implementation order
  – Usability: interfaces engineered for humans (HCI)

Software Architecture

• High-level structure of a software system
  – Principled approach to partitioning modules and controlling dependencies / data flow among them

• 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 Software Architectures

Pipe-and-filter (think: iterators)

Source → Pipe → Filter → Pipe → Filter → Pipe → Sink

Blackboard
(think: callbacks)

Layered
(think: levels of abstraction)
Good Architectures Support

• Scaling to support large numbers of ______
• Flexibility
  – Adding and changing features
  – Easy customization (Ideally with no programming)
• Versatility
  – Integration of acquired components
  – Communication with other software
  – Software to be embedded within a larger system
• Recovery from wrong decisions
  – About technology… About markets…

Software Architecture

• Have one! Subject it to serious scrutiny!
  – At relatively high level of abstraction
  – Basically lays down communication protocols
• Strive for simplicity
  – 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

Temptations to Avoid

• Avoid feature creep
  – Costs under-estimated
  – Benefits over-estimated
  – A Swiss Army knife is rarely the right tool
• Avoid digressions
  – eg: premature tuning
  • Often addresses the wrong problem

Tools: Build Management

• Building software requires many tools:
  – Swift compiler, simulator, C/C++ compiler, GUI builder, Device driver build tool, Web server, Database, scripting language for build automation, parser generator, test generator, test harness
  – Reproducibility is essential
  – 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.
Tools: Version Control

• You've all been using it
  – Collect work (code, documents) from team members
  – History of changes
  – Synchronize team members to current source
  – Have multiple teams make progress in parallel
  – Manage multiple versions, releases of the software
  – Identify regressions more easily

• Establish policies
  – When to check in, when to update, when to branch and merge, how builds are done, ...

Tools: Continuous Integration

• Build and test every commit
  – Catch errors early
  – Localize bugs to specific change
  – Prevent bad code from spreading

Tools: Bug Tracking

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

• Example tools:
  – GitHub, Bugzilla, Flyspray, Trac, Sourceforge, Google Developers, GitLab/GitHub, Bitbucket, ...
  – https://github.com/stephenfreund/cs326

Tools: Bug Tracking

• Establish good process.
• Make it explicit in a policy.
• Keep it simple!
How Does a Project Become a Year Late?

• It’s not the hurricanes that get you
• It’s the termites
  – Someone missed a meeting
  – Someone’s keyboard broke
  – The compiler wasn’t updated
  – Bad flu season. Or maybe a pandemic...
  – Missing documentation
  – Manager quit

Scheduling

• Must predict time/cost to build software
• Schedule is needed to make slippage visible
  – Must be objectively checkable by outsiders
• Unrealistically optimistic schedules are a disaster
  – Decisions get made at the wrong time
  – Decisions get made by the wrong people
  – Decisions get made for the wrong reasons
• It will always take longer than you expect. Always.

Effort != Progress

• Effort
  – Product of workers and time. (eg: person-months)
  – Easy to track.
• Progress
  – Forward movement toward a destination.
  – Hard to track.
  – No one likes to admit lack of progress...

• Design the development process and architecture
to facilitate tracking progress.

Controlling the Schedule

• Have one!
  – Know effects of slippage
  – Know what to work on when
• Gantt Chart
Milestones

- Verifiable
  - Module 100% coded
  - Unit testing 100% complete
- Non-verifiable
  - 90% of coding done
  - 90% of debugging done
  - Design complete
- Avoid non-verifiable milestones

Typical Milestones

- Design complete / design freeze
- Interfaces complete / feature freeze
- Code complete / code freeze
- Alpha release
- Beta release
- Release candidate (RC)
- FCS (First Commercial Shipment) release

When You Know You’ll Miss Milestone

- Reflect on why. Hold people accountable.
- Four options
  - Same deadline, same amount of work
  - Same deadline, reduced scope of work
  - Later deadline, same scope of work
  - Later deadline, increased scope of work
- Wrong choice made often...
- Take no small slips
  - One big adjustment is better than three small ones

Possible Ways To Shorten Timeline

- Add people
  - Startup cost (“mythical man-month”), communication cost
- Buy components
  - Hard in mid-stream
- Change deliverables
  - Customer must approve
- Change schedule
  - Customer must approve
How to Code and Test Your Design

- You have a design and architecture
- Key question: what to do when?

Bottom-up

- Implement/test children first
  - For example: G, E, B, F, C, D, A
- First, G in isolation. Then E.
  - Generate test data
  - Construct drivers
- Then B, F, C, D.
  - 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

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
- Instead: Automated drivers in a test harness
  - GraphADT, SocialNetworks, CampusPaths,...

Top-down

- Implement/test parents first
- First: A
  - build stubs to simulate B, C, and D
- Then: B
  - Build a stub for E
  - Drive B using A
- Then: C
  - Possibly reuse E, if sufficient, or create new stub
- ...
Implementing a Stub

• Query a person at a console.
• Print a message describing the call.
  – Name of procedure and arguments
  – Fine if calling program does not need result
• Provide “canned” results.
  – UtterKit’s canned responses
• Provide a primitive implementation.
  – Inefficient & incomplete
  – Best choice, if not too much work
  – Look-up table often works

Top-Down vs. Bottom-Up

• Which is Better?

• Neither dominates
  – Understand advantages/disadvantages of each
  – Helps you design an appropriate mixed strategy

Criteria

<table>
<thead>
<tr>
<th>Criteria</th>
<th>Top-Down</th>
<th>Bottom-Up</th>
</tr>
</thead>
<tbody>
<tr>
<td>When Do You Catch Design Errors?</td>
<td></td>
<td></td>
</tr>
<tr>
<td>When Do Visible Components Work?</td>
<td></td>
<td></td>
</tr>
<tr>
<td>How Much Integration Work? (less is better)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Amount of Work?</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Testing Time Distribution?</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Good Practice

• Largely top-down
  – But always unit test modules
• Switch to 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
    • Have something to show early on.
**Perspective...**

- Software project management is challenging
  - Different intellectual demands than programming
  - Mix of hard and soft skills
  - Communication, writing, problem solving, reflection
  - eg: a liberal arts education

- We’ve only skimmed the surface
  - Software Engineering is an entire field within CS

---

**Wrap Up**

“Controlling complexity is the essence of computer programming.”

-- Brian Kernighan

(UNIX, AWK, C, ...)

---

**Goals**

- Primary focus: writing correct programs
  - What does it mean for a program to be correct?
    - Specification (vs Requirements)
  - How do we determine if a program is correct?
    - Reasoning, Verification, Testing
  - How do we build correct programs?
    - Principled design and development
      - abstraction, modularity
      - documentation
  - Will cover both principles and tools.

---

**Outcomes**

- Better at design
- Better at coding
- Better at debugging
- Better at using development tools
- Better at evaluating quality / behavior
- Better at communication

- Essential skills regardless of what you do next
Life After 326...

• System building can be rewarding and fun
  – Never "easy" (but what worthwhile endeavors are?)
  – There are always new challenges
  – It’s even more fun when you’re successful

• Pay attention to what matters
  – Take advantage of the techniques and tools you’ve learned (and will learn!)
  – Make good decisions, not expedient decisions

Life After 326...

• Your next project can be much more ambitious.
  – Be confident but humble
  – Recognize your own strengths and weaknesses
    • We all have both

• Life-long process
  – Like being a good writer of prose
  – Practice is a good teacher
    • Requires thoughtful introspection
    • Don’t learn only by trial and error!
  – Voraciously consume ideas and tools