
CSE 331

Software Design & Implementation

Hal Perkins

Winter 2015

System Integration and Software Process

(Based on slides by Mike Ernst, Dan Grossman, David Notkin, Hal Perkins)

Administrivia – same as last time

- HW9 due tonight
 - Usual late days apply if any left
- We want to show off projects on Friday – please let us know if we can use yours!
- No more quizzes (everyone has enough to do already)
- Final exam Monday, **8:30 am** (sigh), here
 - Review Q&A Sunday, 2 pm, EEB 037
- Course evals: please fill them out sometime this week

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
 - *Usability*: interfaces engineered for *humans*
 - Another lecture: didn't fit this quarter
 - Major focus of CSE440

Outline

- Software architecture
- Tools
 - For build management
 - For version control
 - For bug tracking
- Scheduling
- 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/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)



Layered (think: levels of abstraction)

Blackboard (think: callbacks)

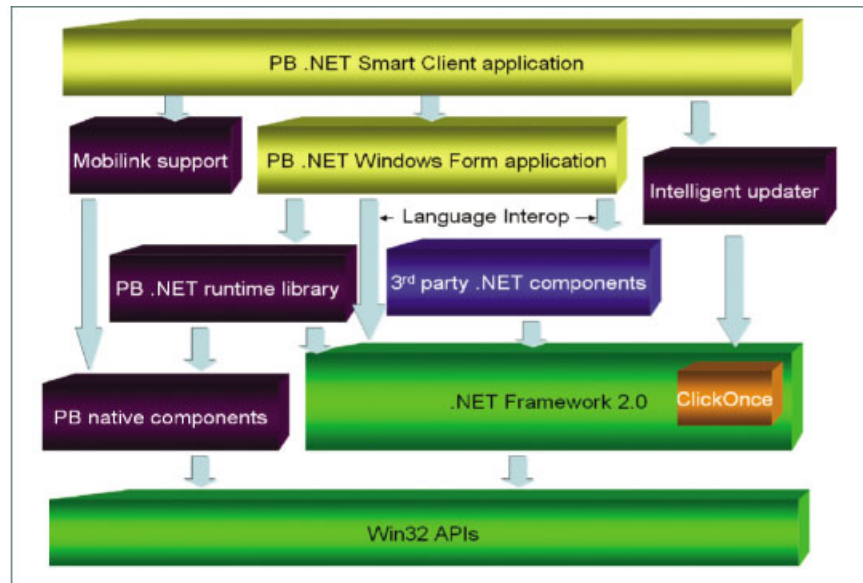
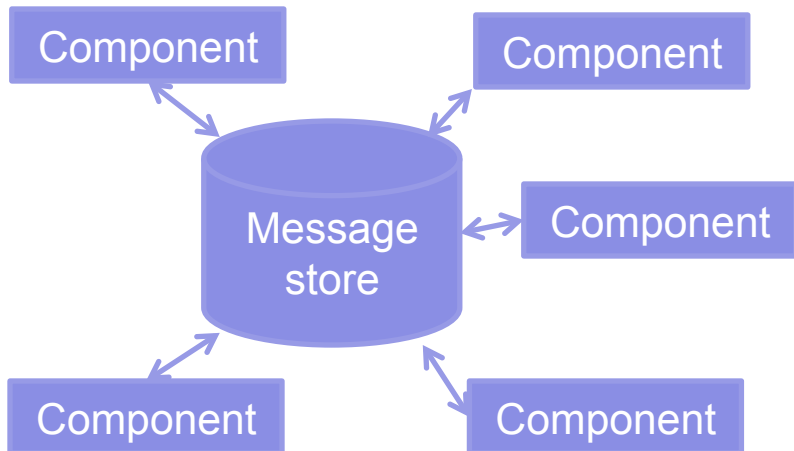


FIGURE 1 | ARCHITECTURAL DIAGRAM OF A POWERBUILDER SMART CLIENT APPLICATION

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

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*

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

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:
Bugzilla, Flyspray, Trac, hosted tools (Sourceforge, Google Developers, GitHub, Bitbucket, ...)

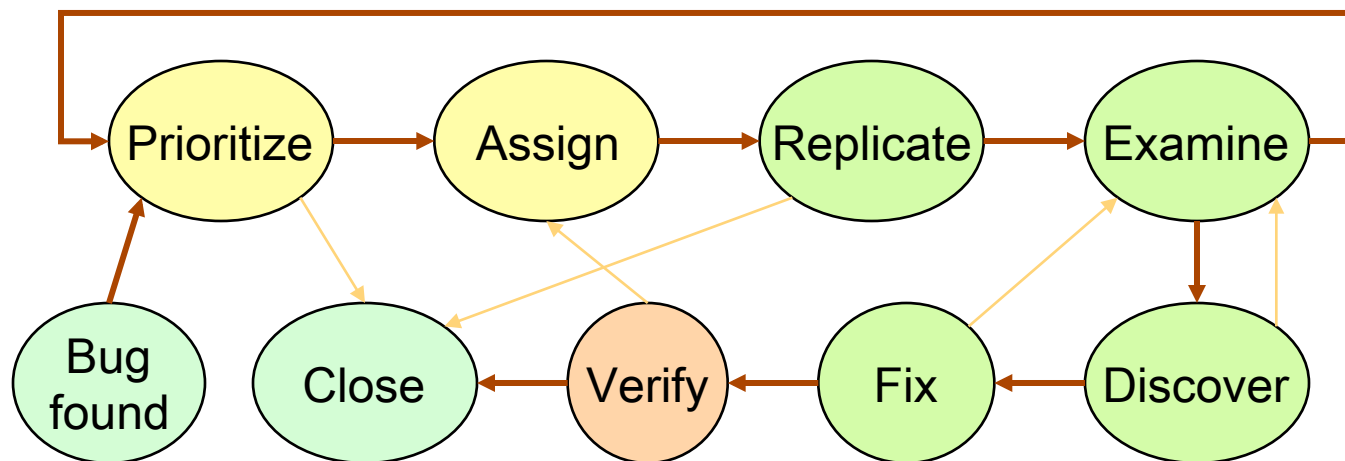
Bug tracking

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



Outline

- Software architecture
- Tools
 - For build management
 - For version control
 - For bug tracking
- Scheduling
- 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
 - 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
- The great paradox of scheduling:
 - Hofstadter's Law: It always takes longer than you expect, even when you take into account Hofstadter's Law
 - But seriously: 2x longer, even if think it will take 2x longer

Effort is not the same as progress

Cost is the product of workers and time

- Reasonable approximation: All non-people costs (mostly salary) are zero (?!)
- Easy to track

Progress is more complicated

- Hard to track
- People don't like to admit lack of progress
 - Think they can catch up before anyone notices
 - Assume they (you) are wrong
- 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
 - Tom missed a meeting
 - Mary's keyboard broke
 - The compiler wasn't updated
 - ...

If you find yourself ahead of schedule

- Don't relax
- Don't add features

Controlling the schedule

- First, you must have one
- Avoid non-verifiable milestones
 - 90% of coding done
 - 90% of debugging done
 - Design complete
- 100% events are *verifiable milestones*
 - Module 100% coded
 - Unit testing successfully complete
- Need *critical path* chart (Gantt chart, PERT chart)
 - Know effects of slippage
 - Know what to work on when

Milestones

- 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

Dealing with slippage

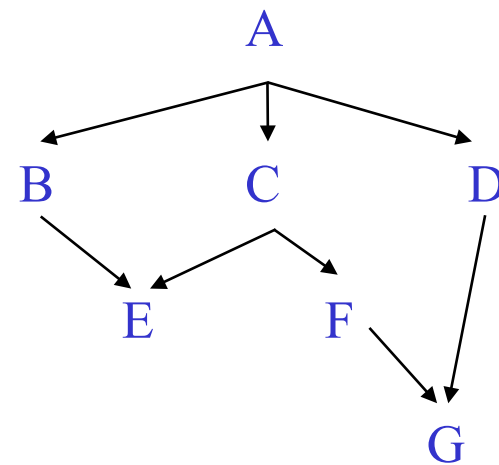
- People must be held accountable
 - Slippage is not inevitable
 - Software should be on time, on budget, and on function
- Four options
 - Add people – startup cost (“*mythical man-month*”)
 - Buy components – hard in mid-stream
 - Change deliverables – customer must approve
 - Change schedule – customer must approve
- Take no small slips
 - One big adjustment is better than three small ones

Outline

- Software architecture
- Tools
 - For build management
 - For version control
 - For bug tracking
- Scheduling
- Implementation and testing order

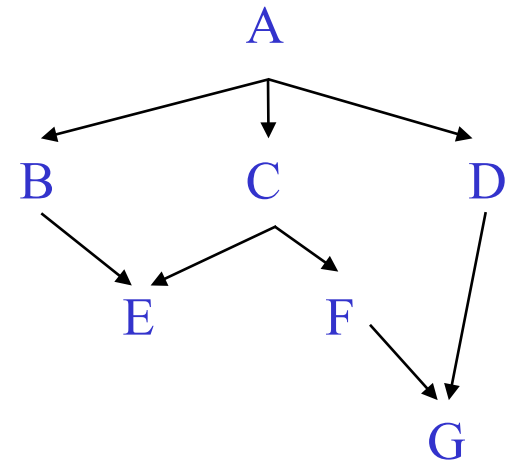
How to code and test your design

- 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 as discussed earlier
 - Construct drivers
- 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

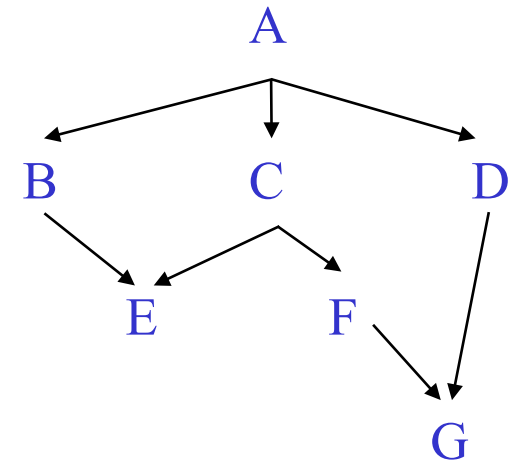


Building drivers

- 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?



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*” (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

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

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
- On average, top-down requires more non-deliverable code
 - Not necessarily bad

Distribution of testing time

- Integration is what takes the time
- Bottom-up gets harder as you proceed
 - You may have tested 90% of code
 - But you still have far more than 10% of the work left
 - Makes prediction difficult
- Top-down more evenly distributed
 - Better predictions
 - Uses more machine time (could be an issue)
 - Because testing overall (even if stubbed) functionality

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

Test harnesses

- Goals:
 - Increase amount of testing over time
 - Facilitate regression testing
 - Reduce human time spent on testing
- Take input from a file
- Call module being tested
- Save results (if possible)
 - Including performance information
- Check results
 - At best, is correct
 - At worst, same as last time
- Generate reports

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
 - CSE 403 is the next step, but just a start