

System integration and software process

CSE 331

University of Washington

Michael Ernst

Outline

- Architecture
- Tools: Build tools and version control
- Tools: Bug tracking
- Scheduling
- Implementation and testing order

Outline

- Architecture
- Tools: Build tools and version control
- Tools: Bug tracking
- Scheduling
- Implementation and testing order

Architecture

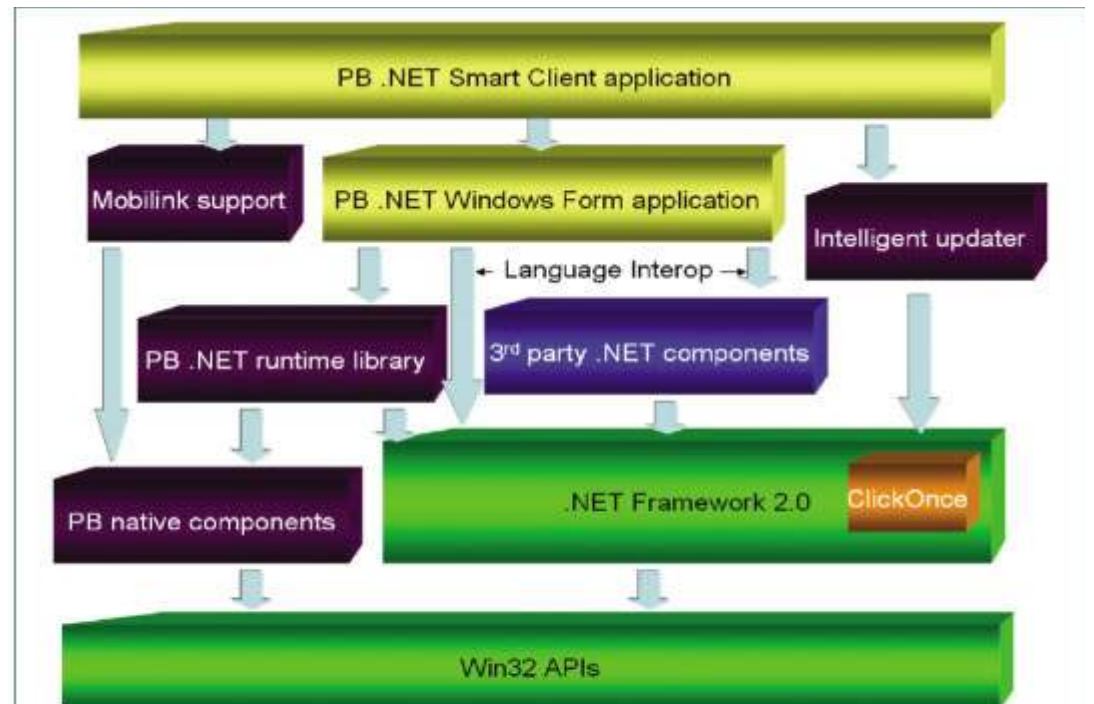
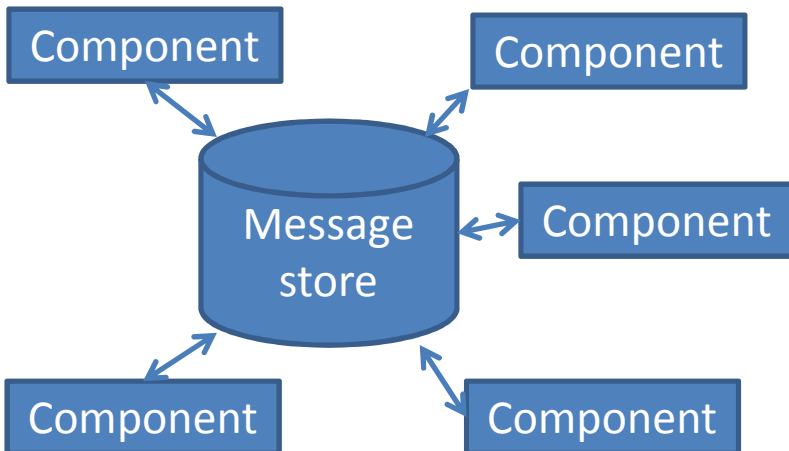
- An **architecture** describes a **partitioning** of the system
 - It indicates dependences on, and data flow between, modules
- A good architecture ensures that
 - Work can proceed in parallel
 - Progress can be closely monitored
 - The parts combine to provide the desired functionality

Example architectures

- Pipe-and-filter (think: iterators)



- Layered (think: levels of abstraction)
- Blackboard (think: callbacks)



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
 - Organizational mission is not the same as the project
 - Build your organization as well as the project
 - Software is capital
 - This will not happen by accident

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

- Architecture
- Tools: Build tools and version control
- Tools: Bug tracking
- Scheduling
- Implementation and testing order

Build tools

- Building software requires many tools
 - Example: Java compiler, 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, missing tool can drastically reduce productivity
- Hard to switch tools in mid-project

Version control (source code control)

- A version control system supports:
 - Collecting work (code, documents) from multiple team members
 - Synchronizing all the team members to current source
 - Let multiple teams make progress in parallel
 - Manage multiple versions, releases of the software
 - Help identify regressions
- 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

Outline

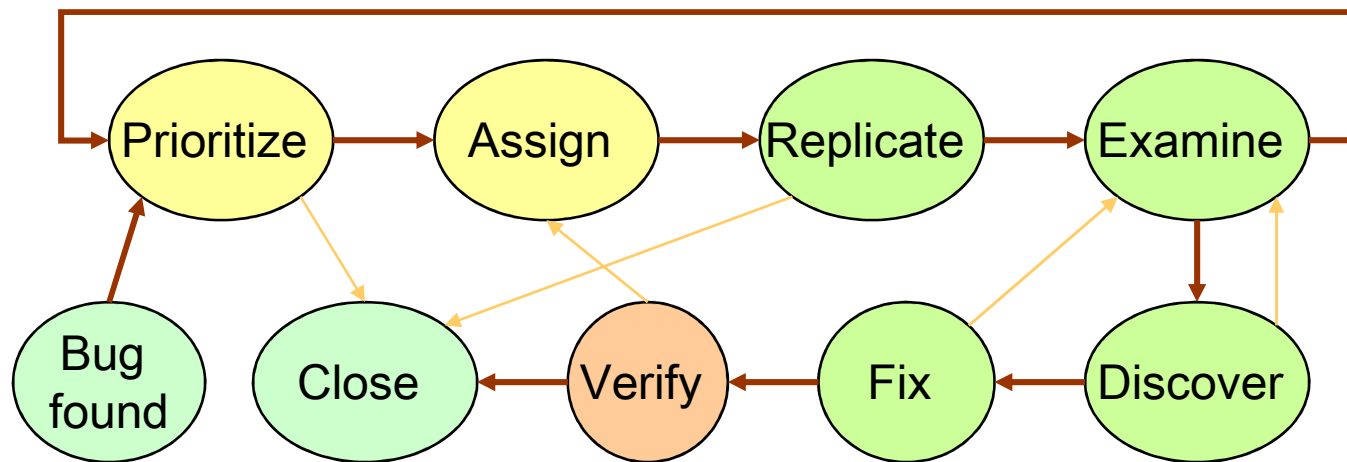
- Architecture
- Tools: Build tools and version control
- Tools: Bug tracking
- Scheduling
- Implementation and testing order

Bug tracking

- An issue tracking system supports:
 - Tracking and fixing bugs
 - Identifying problem areas and managing them
 - Communicating between team members
 - Track regressions and repeated bugs
- Any medium to large size project requires bug tracking software
- Example tools:
 - Bugzilla, Flyspray, Trac, hosted tools (Sourceforge, Google Code, GitHub)

Bug tracking

- Need to configure the bug tracking system to match the project
 - Many make the system too complex to be useful
- A good process is key to managing bugs
 - Need an explicit policy that everyone knows, follows, and believes in



Outline

- Architecture
- Tools: Build tools and version control
- Tools: 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 scheduling paradox
 - Everything takes twice as long as you think
*... even if you **know** that it will take twice as long as you think*

Effort is not the same as progress

- **Cost** is the product of workers and time
 - Easy to track
- **Progress** is more complicated, and hard to track
- People don't like to admit lack of progress
 - Think they can catch up before anyone notices
 - Not usually possible
- 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
 - Design complete
 - Interfaces complete / feature complete
 - Code complete / code freeze
 - Alpha release
 - Beta release
 - FCS (First Commercial Shipment) release

Dealing with slippage

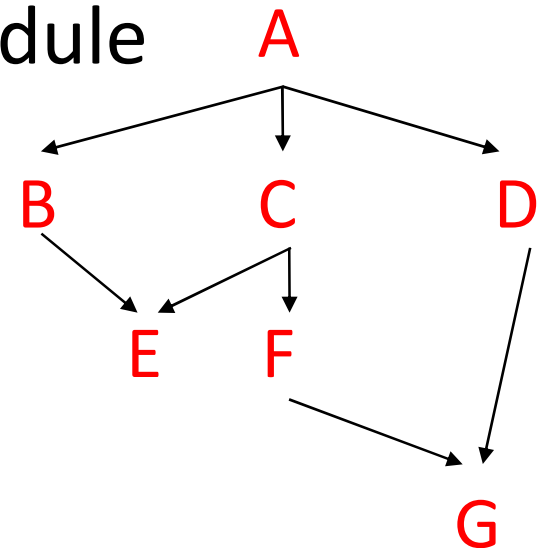
- People must be held accountable
 - Slippage is not inevitable
 - Software should be on time, on budget, and on function
- Four options
 - Add people – startup cost (“mythical man-month”)
 - Buy components – hard in mid-stream
 - Change deliverables – customer must approve
 - Change schedule – customer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

Outline

- Architecture
- Tools: Build tools and version control
- Tools: Bug tracking
- Scheduling
- Implementation and testing order

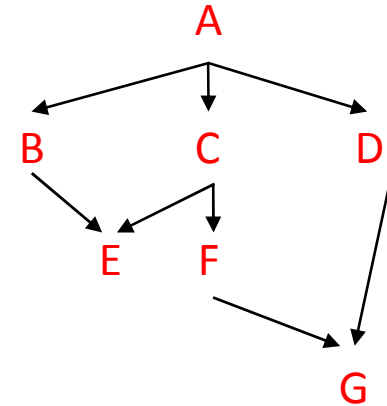
How to code and test your design

- You have a design and architecture
 - Need to code and test the system
- Key question, what to do when?
 - We'll assume an incremental development model
- Suppose the system has this module dependency diagram
 - In what order should you address the pieces?



Bottom-up implementation

- 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

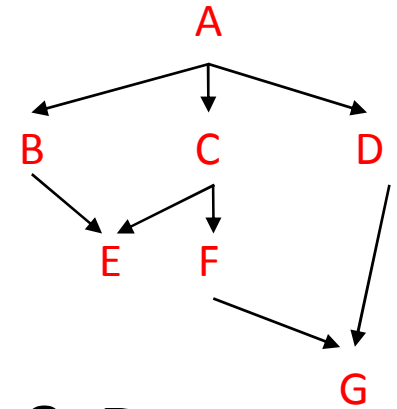
Test harnesses

- 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

Top-down testing



- Implement/test parents (clients) first
 - Here, we start with A
- To run A, build **stubs** to simulate B, C, & 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
 - This is more common than you might think!
- Provide canned or generated sequence of results
 - Very 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

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 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 integration at each step

- Less is better
- Top-down adds one module at a time
 - When error 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

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
 - In business environments this can be an issue

Amount of work

- Always need test harness
- Top-down
 - Build stubs but not drivers
- Bottom-up
 - Build drivers but not stubs
- Stubs 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

One good way to structure an implementation

- Largely top-down
 - But always unit test modules
- Bottom-up
 - When stubs are too much work
 - 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