Integration and Testing Strategy

- Integration
  - problems w/phased integration
  - incremental integration
    - orders and models
- Testing
  - unit testing
  - test driven development
  - in vivo & in vitro testing
  - getting at those hard to reach places

The Integration "Phase" Canard

- Came from large system procurement
  - numerous "Balkanized" contractors
  - manufacturing/assembly is expensive
- Development model
  - specify all components/interfaces up front
  - independently develop each component
  - combine them when they are all finished
- Result
  - a predictable disaster

Incremental Integration

- Process
  - start with dummy versions of every component
    - including makefiles, test cases, configuration data
  - run automatic regular (e.g. daily) builds
  - integrate each (small) change as it is ready
- Benefits
  - system can be built & tested from day one
  - problems are found sooner and more quickly
  - problems are spread out over the schedule
  - more predictable schedule and quality
  - less wasted re-engineering

Incremental Integration Models

- Order of updates
  - top-down
  - bottom-up
  - hybrid combinations
- Granularity of updates
  - continuous integration
    - expose code to others ASAP
  - Train Model integration
    - we will ship no product before it is ready

Integration Sequencing

- integrate updates as they are made
  - but in what order do we do our development?
- this is not a simple question
  - testability: an order that facilitates unit testing
  - modularity: finish one thing, then move on
  - dependencies: some things are needed sooner
  - resources: some are only available at one time
  - learning curve: do simpler things first
  - risk: deal with high-risk problems early
  - uncertainty: leave time to tie down open issues

Unit Testing

- is this component complete & correct?
  - does it perform all required functions
  - does it correctly handle all specified errors
  - does it meet all of its specifications
- some unit testing can be done in isolation
  - simple input/output functionality testing
  - simple error testing
  - test harnesses or dummy components
- some testing requires other components

Test Driven Development

- As a formal methodology
  1. write test case(s)
  2. run test case(s) and confirm failure
  3. write code to implement functionality
  4. re-run test(s) and confirm success
  5. check in the new tests and code
- As a general approach
  - thoroughly test each feature as you write it
  - do all testing with automated test cases
  - save and accumulate all the test cases

approaches to unit testing

- *in vivo* testing (within the living)
  - build a complete program
  - run it with test inputs, observe its behavior
- *in vitro* testing (within glass)
  - build an individual component
  - build a test harness for that component
    - that simulates the rest of the program
    - that passes inputs to the tested component
    - that records the component behavior
  - test the component with the exercise harness

"in vivo" unit testing

- testing whole programs
- advantages
  - we are testing the program we will ship
  - we are exercising the whole program
  - less work wasted on special test harnesses
- disadvantages
  - requires program to be (somewhat) complete
  - some problems may be hard to induce
  - some behavior may be hard to observe

where *in vivo* clearly wins

- externally visible behavior
  - in response to external inputs
- whole-program functionality testing
  - does program support all specified inputs
  - does program produce correct output
- testing response to external errors
  - invalid parameters and requests
  - improper configuration
  - missing or incorrect data files

designing for *in vivo* testing

- design for progressive implementation
  - there is always a complete program
    - but its functionality grows over time
  - start with the highest level functions
- design for maximum observability
  - key components produce observable output
  - key internal state can be displayed
- design for maximum controllability
  - all interesting situations can be created
- often achieved w/diagnostic options

"in vitro" unit testing

- testing components in isolation
- advantages
  - we can completely control program stimulus
  - we can closely observe program behavior
  - problems are easily diagnosed
- disadvantages
  - component may behave differently *in vivo*
  - must build special test harnesses
  - still need other tests for whole program

in vitro routine testing

- test driver
  - reads test case descriptions from config table
  - sets up inputs and environment
  - calls the routines to be tested
  - records all results
- component adaptor includes routines to
  - call target routines with test parameters
  - return test values when called by target
- involves much component specific code

where *in vitro* clearly wins

- when rest of the system isn't yet available
  - independent pieces, parallel development
- testing complex and unlikely error cases
  - things that can't happen in a healthy system
  - highly unlikely combinations
- stressing infrequently used components
  - harness can provide much better exercise
- directly testing deeply embedded behavior
  - harness can observe all interesting behavior

alternatives to *in vitro* testing

- test real components in a dummy system
  - of simulated networked components
- build dummy/test components
  - implement same interface as real component
  - includes test driver to control behavior
  - generate controllable requests and loads
  - generate plausible or erroneous responses
    - including very unlikely error
- Combines benefits of in-vivo and in vitro
  - wider range of whole component tests

alternatives to *in vitro* testing

- diagnostic error injection
  - built in to each component
- design error simulation into components
  - identify important hard-to-induce errors
  - design ways to manually simulate them
    - but higher level s/w will treat them as real
- externally trigger and observe behavior
- these are often left in production code
  - they can be valuable debugging aids

For Next Lecture

- McConnell, chapters 25
- Wikipedia: system testing
- Kaner: Scenario Based Testing
- Kampe: Load and Stress testing
- Kampe: Testing and Bug Discovery
- Kampe: Release Phases & Criteria
- Gnu: gprof execution profiling (skim)

Supplementary Slides

Industrial Strength Error Testing

- diagnostic platforms
  - very realistic simulation of real errors
  - under the control of a software test driver
- firmware error injection
  - transient parity and communications errors
  - solid faults, system resets, power failures
- software error injection
  - resource exhaustion, system overloads
  - wrappers to simulate service/node failures

The Bottom Line

- Quality comes from methodology
  - requirements, design, construction processes
- Confidence comes from testing
  - reviews, unit testing, system testing
- How much confidence do we have?
  - how thoroughly have we tested it?
  - this is limited by its intrinsic testability
- Testability results from careful design
  - if you make it testable, you can test it
  - if you test it well, you can make it good

testing object oriented software

- how do you test one class at a time?
  - You can't. Who said you should?
- OO gives us powerful language features
  - it does not greatly change our programs
  - test OO s/w the same way we test other s/w
- test methods as they become testable
  - as they are implemented
  - as modules that call them are implemented
- design software with this phasing in mind

automation is essential

- tests must be run regularly
  - on each new version of the component
- tests must be run repeatably
  - the exact same tests run every single time
- results must be checked mechanically
  - tired/bored eyes can miss minor errors
- results must be summarized and reported
  - to measure improvement/regression
- these are repetitive, mind-numbing tasks

automated testing framework

- is driven by a database
  - each entry describes a test case
- for each selected test case
  - set up the environment (files, credentials, …)
  - initiate a specified action (command, msg, …)
  - capture all output
  - note the state of all files, databases, etc
  - compare the results with what was expected
  - report any differences as errors/failures
- produce summary of tests run and results

test drivers

- should run w/o human assistance
- are highly application-class specific
  - for command-line-interfaces
    - shell scripts work very well
  - for network servers
    - transaction generators are needed
  - for GUIs
    - capture/replay can be used
    - widget level action and query is better
- but should be designed for generality

Golden Output

- used by many testing frameworks
  - golden output is what system should produce
  - results are compared against golden output
  - differences are reported as errors/failures
- not all differences are errors
  - some output may be incidental
  - we need a way to filter this out
- we must be sure golden output is correct
  - otherwise errors will go undetected
  - it must be carefully reviewed and managed

Data Verification

- some verification is very simple
  - we expect specific files with specific contents
  - such verification is easily automated
- verification can be more complex
  - assertions relating output to supplied input
  - programs must be written to test this
  - the assertions, or the programs can be wrong
- when you design a component
  - think about how to confirm its correctness