Benjamin Fenster
CS-205 (Damon)
20 December 2004

Verification and Validation

I. Introduction
   a. Verification
      i. Does the code do what the designs say it should do? Hard!
      ii. Code inspection
      iii. Testing
      iv. Formal methods
   b. Validation
      i. Does it do what the customer wants? Even harder!
      ii. Requirements reviews (with the customer)
      iii. System testing (customer uses the system and approves it)

II. Testing
   a. Dominates verification & validation
   b. Done primarily by QA / "Test Group"
   c. Setting up Conditions: Hardware, amount of data, input values
   d. Check Results (Test Oracle)
   e. Testing Stages
      i. Unit testing. Done by developer. Test functions, classes
      ii. Integration Testing
         1. No obvious line between this and unit testing
         2. At some point it's done by QA rather than by developers.
      iii. Function Tests: All "functions" (features) work as expected
      iv. Performance Tests
         1. Always done by QA
         2. Run on realistic machines with realistic loads
      v. Stress Tests. At what point does it break?
      vi. Acceptance Tests
         1. Users try at development site
         2. Won't pay until it passes the acceptance test
      vii. Installation Tests
         1. Run old system and new system together. Compare answers.
         2. Do this for several months. If everything works out, switch systems!
   f. Types / "Kinds" of Tests
      i. Input / Output Tests – What everyone considers testing. Put X in, get Y out.
      ii. UI Tests
      iii. Monte Carlo Tests – Generate random queries for DB, random instruction sequences, random timing, … Won't have output prediction but can make sure the system doesn't die completely.
      iv. Load Tests
      v. Recovery Tests: Kill it, see what happens when it recovers
   g. Goals Depend On:
      i. Kind of program you're designing (747 navigation vs. Duke Nukem 74)
      ii. Expectations (alpha, beta, release)
      iii. Existing tradeoff between quality x time x features
   h. Approaches
      i. Depends on goals
      ii. Depends on technologies
         1. OO
         2. GUI
         3. Threads
         4. Distributed Computing – much, much harder
         5. All introduce their own challenges
   i. Basic Questions
      i. What kinds of bugs do you expect? Where?

ii. How important are different types of bugs??
iii. What's your reaction to types of bugs?
iv. Do nothing? Document it? Fix it?

III. Unit Testing
  a. Goal is to find bugs
  b. The challenge is to break the code
  c. Two types of bugs
    i. Correctly handle correctly input
    ii. Gracefully handle incorret input
  d. Want to help identify where the bugs are
  e. Black Box Tests. I know only what it's supposed to do
  f. White Box Tests. Consider implementation; test based on that
  g. Writing Tests
    i. Ideally: Enough tests to identify, find all bugs
    ii. Reality: Find as many as ou can
  h. Scaffolding
    i. You may be relying on other code that's not necessarily reliable
    ii. Need comparison functions for a sort algorithm
    iii. If you're using a database, maybe just return raw data
    iv. Functionst o parse input (if text-based)
    v. Functions to display output (usually as text)
  i. Creating Tests
    i. Start with black box tests (test before coding)
    ii. Make sure the expected / mainstream cases work
    iii. Then worry about boundary cases
    iv. Finally consider error cases
    v. Then do white box tests based on the code structure

IV. Integration Testing
  a. Bigger and bigger pieces.
  b. At some point feels like you're using the application

V. When do You Stop?
  a. Need a test coverage metric
  b. How much of the program has been tested?
  c. Easiest metric: Percent of statements executed
    i. Can't always test every statement though.
    ii. Unreachable code.
    iii. Timing based
    iv. Error conditions (simulate power failure, or whatever)
    v. Good test suite will execute maybe 95% of statements
  d. Consider flow of control metrics
  e. Strength of a Metric: Metric A is stronger than B iff metrics with full coverage under B have full coverage under A
  f. All Branches
    i. Every branch is either taken or not taken
    ii. Stronger than all statements
    iii. Goal is to cover about 85% – 95% of branches
    iv. There are programs that will report what you've missed
  g. All Paths
    i. Even combinations of branches can hurt too
    ii. Not possible to test all paths through the program
    iii. Can test all within a function
    iv. Realistic goal is about 40% coverage. Not encouraging.
    v. We don't really care about all paths though. We care which ones change data others will use.
  h. Data Flow
    i. *Definition* of a value at each assignment

        ii. *Used* when on right of an equal sign
        iii. *Killed* when overwritten
        iv. Def-Use pair is a definition and usage of the same value on a path that contains no kills of that value.
        v. Allows some new metrics
        vi. All DU Paths
        vii. All Uses
            1. For every DU pair, at least one kill-free path
            2. Still hard to test
        viii. All Predicates
            1. When concerned about what determines decision-making
            2. All uses, but only usages in predicates
        ix. All Computations – Opposite of All Predicates
        x. All Definitions. Does every definition get used somewhere?
        xi. All these *look* interesting but nobody's actually using them so far.
    i. New Approaches
        i. None of these metrics really seem to work
        ii. Common Bugs Are
            1. Off by 1
            2. > instead of >=
            3. Used .x instead of .y
            4. Control flow, data flow don't check these bugs
        iii. Mutation Coverage
            1. Assume the program is almost correct
            2. Look for slight variations
            3. Define simple transformations (> to <, exchanging datatypes, …)
            4. Have your tests differentiated between these variations?
            5. Tools will tell you how many things you've tried.

VI. Factors in Testing
    a. Language Differences. Language can have a huge impact on testing
    b. Automated vs. Manual Checking
        i. Some checks are automatic: type checking, null pointer
        ii. Some checks must be manual: bounds in C++
        iii. Automated is better – more efficient, more accurate
    c. Static vs. Runtime
        i. Static: Done at compile-time (type checking is the regular example)
        ii. Runtime: Have to run the program to see what happens
    d. Automated + Static – Best of all worlds.
    e. No checking at all – Fast and stupid.
    f. Runtime – Somewhere in between
    g. Object Oriented
        i. Design style is different
            1. Complexity is in interaction, not in any one class
            2. Suggests integration testing is more important
            3. This may or may not be true
        ii. Information hiding makes testing harder
        iii. Polymorphism is immensely harder to test
        iv. Exceptions
        v. Coverage metrics
        vi. Encapsulation
            1. Tests have to be friends
            2. Changes how you structure tests
        vii. Polymorphism
            1. Even if your class works, will someone else's subtype work?
            2. Unbounded number of legal subtypes
            3. Need to clarify boundaries on what can be changed

        4. Want to write "abusive" subclasses to test limits
        5. Integration Testing now includes different subclasses
        6. Test for invalid downcasts
        7. Make sure you've refined methods you need to refine (did you *replace* the method with one with a different parameter list instead of *refining* it)
    viii. Exceptions
        1. Runtime exceptions in Java aren't statically checked!
        2. No easy way to test these.
    ix. Coverage
        1. Ongoing work to get new coverage metrics for OO
        2. What combinations of subclass / superclass?

VII. System Testing
- a. Review
    - i. Unit Testing: Does the code implement the code design?
    - ii. Integration Testing: Does the code design work for the system design?
    - iii. System Testing: Does the system clearly satisfy the *requirements*?
    - iv. Acceptance Testing: Do the requirements satisfy the user?
- b. Introduction to System Testing
    - i. Much longer. Days / weeks / months
    - ii. Tests from the user's standpoint
    - iii. Includes *everything*. Code, help, documentation, clip art, et cetera
    - iv. Collect a list of bugs, submit to development (who fixes them, probably runs some unit tests), get new version and start the system test from scratch
    - v. Also provides some background for resolving problems users find (provides a baseline for customer support later in taking calls)
- c. Function Testing
    - i. Test features
    - ii. Based on documentation. Go through the manual; test everything it says should work (starting with the examples – if those don't work you're in trouble)
- d. Partition Testing
    - i. Split input into distinct classes (including what state the program's in)
    - ii. Develop a case for each such partition
    - iii. Example: Searching in Word Processor
        1. Partition on simple word to search: not in document, at beginning, at end
        2. Partition on document type: empty document, short, multiple pages, multiple documents (i.e. multiple files)
        3. Barely in Partition (search string is the entire document)
        4. Error Cases: empty, blank, random garbage, et cetera
        5. Illegal Actions: writing to full disk, overwrite open document
    - iv. Lots of possibilities. Want a way out, a good error message, …
    - v. Try everything you can envision – some user will attempt it
    - vi. Copy / Paste Example
        1. Copy some simple string, (whole page, single character, …)
        2. Change target of paste: empty document, beginning, end, …
        3. Copy, copy, paste
        4. Copy, paste, paste
        5. Paste from another program; copy to another program
        6. Not enough memory to copy.
        7. Pasting on read-only target
- e. Stress Testing
    - i. Figure out what to stress, then how to do it
    - ii. May be very hard to create the situation
    - iii. Consider what needs to be accurate
        1. All valid data?
        2. Random data?
        3. What does each connection need to do?

        4. Your stress data need to be realistic for the test to be meaningful
   f. Distributed Systems
      i. Hardest to test. Lots of timing-driven bugs
      ii. Critical race (stuff happening in the wrong order can screw everything up)
      iii. Deadlock: Possible whenever you've got locks
      iv. Timing between processors, timing across network
      v. May be a very small chance of having the wrong timing; hard to repeat
   g. Monte Carlo Testing
      i. Hard to confirm accuracy of results
      ii. Basically don't want it to deadlock or die
      iii. Need some valid data. May have some canned SQL queries to pick at random.
      iv. May need to consider protocol
      v. Probabilistic Machines
         1. Modeled as finite state machine
         2. Open DB first → In transaction
         3. Tx → read → Tx   65% of the time
         4. Tx → delete → Tx   7% of the time
         5. Tx → commit → open   2% of the time…
         6. Setup a dozen of these to run for a month; gives a pretty good indication of how the system will really fare.
      vi. Checking Results
         1. No one client knows the answer
         2. Maybe what you INSERTed was DELETEd by someone else.
         3. If you're testing a new version you might run the same queries on the old system too and make sure the answers agree. That obviously doesn't apply if you're implementing a new feature.

VIII. JUnit
   a. Create subclasses of TestCase to hold tests
   b. Each test is one method
   c. assertTrue(), assertNull(), … to report results
   d. setup(), teardown() to open connections, et cetera that are common to many tests
   e. Collect tests in a TestSuite
      i. Can see results in GUI
      ii. Control which tests you're doing, et cetera
   f. Actually looks for these methods / classes at runtime

IX. GUI Testing
   a. Describe Scenario
      i. Don't want to require someone to click mouse every time you want to test
      ii. Some tools record your actions once, then just repeat (may mean test is tied to layout though – that's problematic)
      iii. Could have a programmer write a script
   b. Playback
      i. Feed fake events into the system
      ii. May not work if layout changes even slightly
   c. Output
      i. Could capture bitmap of the window, but it's really hard to compare. Rendering may be different from your oracle to the new test.
      ii. In Java, could somehow examine panels. Harder to do, and doesn't tell you if the screen still looks the same.
      iii. Spurious Differences
         1. Date / Time – *supposed* to be different
         2. Host name, OS version, …
         3. Can tell comparison software to ignore those parts
   d. Timing Issues
      i. Output may appear only briefly, or after delay
      ii. If you speed up or slow down, test may fail.

e. Exception Situations
    i. "You've got mail."
    ii. Cover up what's behind it and won't go away until you click
    iii. Expensive tools will let you capture, then dismiss (and warn you that *potentially* important messages appeared).
f. Platform
    i. Graphics card, firmware version, …
    ii. Different browsers may render very differently.
    iii. Likewise for different versions of the JVM
    iv. Window managers, themes, skins, fonts
g. Opportunities
    i. Represent the UI as a state chart – finite number of menu options, et cetera
    ii. Generate all possible combinations
    iii. Robot (java.awt.Robot)
        1. Controls UI
        2. Describe actions programmatically
        3. Also used to make tutorials.
        4. Robot is attached to a Display
        5. Mouse move() just like someone's using the mouse – really moves across the screen, indistinguishable from a person to the application
        6. Timing Support
            a. Might want to wait for dialog to popup
            b. Can pause for a certain length of time, or wait for all events to finish processing
            c. Can wait after a specific event or automatically after all events
        7. Captures screen, but nothing sophisticated at that end.

X. Other Testing Issues
a. Automated Test Suites
    i. Goal is to avoid paying monkeys to test everything manually
    ii. Ideally when the developer checks in a change, the system will run tests on everything that's affected, notify the developer if something fails.
    iii. Often have "too many" tests to run them all, so pick some every night
    iv. Build or Buy
        1. Expensive! So most places start with shell scripts, then build on them
        2. Wide range of commercial products
b. Test Generation
    i. With formal specifications of the code, could generate tests automatically
    ii. Active research in this area
    iii. So far, generates too many tests to ever run (for black box testing)
c. Fault Injection
    i. Error cases can be hard to test
    ii. Can easily fill up disk and then test writes
    iii. Harder to drop packets randomly, or make the disk fail for some particular write
    iv. Tools will simulate those things though! "Fault Injection."
    v. Dropped packets, lost connection, allocation failures, all simulated without actually destroying the rest of the system.
d. Test Plans
    i. Last major document
    ii. Really addresses strategy / tactics – planning an invasion, while requirements are about negotiating peace.
    iii. Specifies: What will be tested, how, when will it be done
    iv. Implicitly or explicitly: what *won't* be tested?
    v. Not particular tests, but "Monte Carlo testing on this part of the system for T time"
    vi. Exit Criteria
        1. When are you done?
        2. You'll never (almost) get rid of *all* bugs

3. What kinds are acceptable?
4. What kinds of things will make us go back and rethink the test suite?
5. How much of a change would make you restart the whole testing process from square 1?
6. Everybody has to agree on the exit criteria: QA, Senior Developers, Product Managers,, Marketing

vii. See [CS-205-2004-3-LECTURE-24:38] for a big ol' list of contents of test plans.

XI. Formal Methods

a. Precise, "mathematical," unambiguous notations for describing software
b. Justifiable for programs that can kill (pacemaker, aircraft control, et cetera)
c. Approach
    i. System: What are you doing?
        1. Describing the design or the code
        2. Prove that his function does what it says.
    ii. Goals: What do you want to accomplish?
    iii. Traditionally want to *prove* the goals are met
    iv. Much easier to find counterexamples (indicates the presence of a bug, without really providing specifics)
d. Tool Support
    i. Proving
        1. Theorem Provers
        2. Case Example: 2 people, 3 months, full time, 1 proof
        3. *Much* more people-intensive than the name suggests
    ii. Counterexamples: Can be done with tools automatically
    iii. Dimensions
        1. Design vs. Code
        2. Proving vs. Model Checking
        3. All four cross combinations exist
    iv. Code
        1. Start with invariants
        2. Partial correctness
            a. Can't solve the Halting Problem
            b. Means we can't guarantee code will execute after a loop, say
            c. *Can* say, "If this code executes, it's definitely correct"
            d. Written in { }
        3. Complete Correctness
            a. Written in [ ]
            b. We're certain the code *will* execute, *and* it's correct.
e. Hoare Triples
    i. From Tony Hoare (invented Virtual Memory, monitors, …)
    ii. Have precondition, statement, postcondition
    iii. Assignment
        1. Substitute right-hand side for left-hand side in precondition to get postcondition
        2. {i > 10} i = i + 10 {i > 11}
    iv. If Statement
        1. Split into two cases
        2. One case where you include the if condition in your precondition
        3. The other case for the else
    v. Loops: Hardest
        1. "Unroll" the loop
        2. Take "precondition and loop condition" for the first iteration
        3. Generate the loop invariant (precondition for an iteration)
        4. Usually uses loop control variable.
        5. That becomes the postcondition for the next iteration
    vi. See [CS-204-2004-3-LECTURE-25:11]

XII. Model Checking

a. Originally aimed at checking hardware
b. Differs from traditional static analysis
    i. Static can *prove* that types are correct (e.g. *no* type errors)
    ii. Hard to prove more than that
c. Spin. Original model checker for code
d. When talking about design, need a formal notation
e. Z/np
    i. Pronounced Zed (developed at Oxford)
    ii. Most widely used formal notation (few thousand people)
    iii. Focus on structural issues
        1. Can this structure of code have a cycle?
        2. Showed it was possible for IPv6 to have a cyle, in fact
        3. Is this field unique?
    iv. Tools: nitpick, ladybug
    v. Three kinds of things in np
        1. Scalar Objects (like enums)
        2. Sets of scalar objects
        3. Relations
    vi. Schemas Describe
    vii. Top-level types declared as set of scalar objects
    viii. Associations + AAttributes
        1. In UML thinking about a particular instance
        2. In np, an attribute is a function mapping, say, credit ratings to customers
        3. Boolean attributes isPrepaid : set order
        4. (Inclusion in the set means the attribute is true)
    ix. Constraints
        1. name = [declarations | constraints]
        2. Limit what's allowed
        3. Domain restriction (set domain)