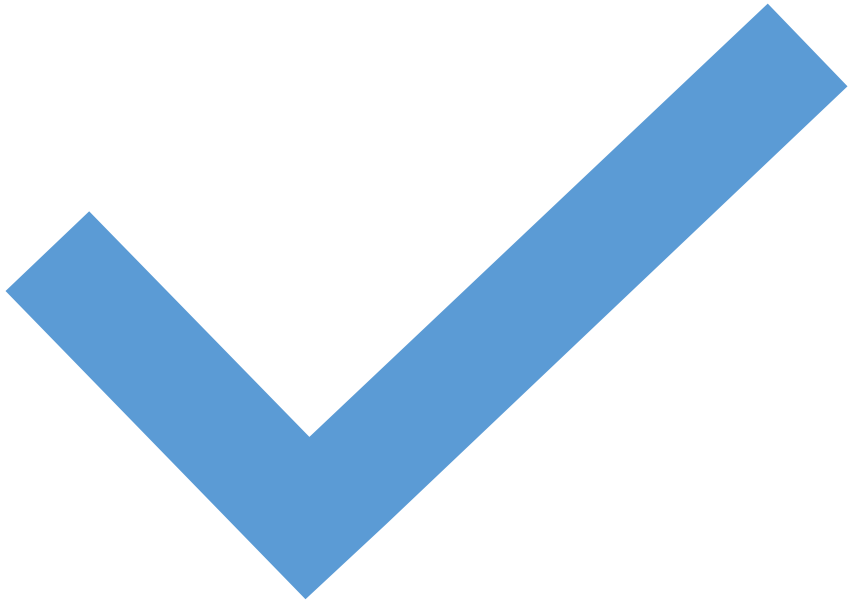


SE401: Software Quality Assurance and Testing

Test Cases

Outline

- Testing
- Test selection
- Writing test cases
- Test Execution

Testing

- There is a massive misunderstanding about testing: that it improves software. It doesn't!
  - Weighing yourself doesn't reduce your weight
  - Going to the doctor doesn't make you healthy.
- Those things help to identify problems that you might choose to resolve. Testing does too.
- The testing does not make the product better, even though it's part of a process that does make a product better.

Test selection

Test cases

- A **test case**, is a set of conditions under which a tester will determine whether an application, software system or one of its features is working as it was originally established for it to do.
- Test cases are often referred to as test scripts, particularly when written – when they are usually collected into test suites.
- A Test Case is a set of actions executed to verify a particular feature or functionality of your software application.

Test cases

- A test case is a description of a specific interaction that a tester will have in order to test a single behavior of the software.
- Test cases are very similar to use cases, in that they are step-by-step narratives which define a specific interaction between the user and the software.
- A typical test case is laid out in a table, and includes:
  - A unique name and number
  - A requirement which this test case is exercising
  - Preconditions which describe the state of the software before the test case
  - Steps that describe the specific steps which make up the interaction
  - Expected Results which describe the expected state of the software after the test case is executed

Test cases

- Test cases must be repeatable.
- Good test cases are data-specific, and describe each interaction necessary to repeat the test exactly.

Writing test cases

- First you must understand the language fundamentals
  - Sizes and limits of variables, platform specific information
- Second, you must understand the domain
- Read the requirements
- Think like a user – what possible things do they want to do
- Think about possible “mistakes”; i.e. Invalid input
- Think about impossible conditions or input
- What is the testing intended to prove?
  - Correct operation – gives correct behavior for correct input
  - Robustness – responds to incorrect or invalid input with proper results
  - User acceptance – typical user behavior
- Write down the test cases

Writing Good Test Cases

- Test Cases need to be simple and transparent
- Create Test Case with end user in mind
- Avoid test case repetition
- Do not Assume
  - Stick to the Specification Documents.
- Ensure 100% Coverage
- Test Cases must be identifiable.
- Implement Testing Techniques
  - It's not possible to check every possible condition in your software application
  - Testing techniques help you select a few test cases with the maximum possibility of finding a defect

Writing Good Test Cases

- Boundary Value Analysis (BVA)
  - testing of boundaries for specified range of values.
- Repeatable and self-standing
  - The test case should generate the same results every time no matter who tests it

Writing a test case

While drafting a test case do include the following information

- The description of what requirement is being tested
- Inputs and outputs or actions and expected results
  - Test case must have an expected result.

- Verify the results are correct
  - ✓Testing Normal Conditions
  - ✓Testing Unexpected Conditions
  - ✓Bad (Illegal) Input Values
  - ✓Boundary Conditions

Writing test cases

- Cover all possible valid input
  - Try multiple sets of values, not just one set of values
  - Permutations of values
- Check boundary conditions
  - Check for off-by-one conditions
- Check invalid input
  - Illegal sets of value
  - Illegal input
    - Impossible conditions
  - Totally bad input
    - Text vs. Numbers, etc.

Writing test cases

- Beware of problems with comparisons
  - How to compare two floating numbers
    - Never do the following:

```
float a, b;
. . .
if (a == b)
```

    - Is it 4.0000000 or 3.9999999 or 4.0000001 ?
    - What is your limit of accuracy?
  - In object oriented languages make sure whether you are comparing the contents of an object or the reference to an object

```
String a = "Hello world!\n"
String b = "Hello world!\n"
if ( a == b )
```

vs.

```
if ( a.equals(b) )
```

Writing test cases

Let's consider the triangle example

- Cover all possible valid input
  - all three possible conditions: **equilateral, isosceles, scalene**
  - Try multiple sets of values, not just one set of values
  - Permutations of values **{3,4,5}, {4,3,5}, {5,4,3}**
- Check boundary conditions
  - Check for off-by-one conditions: **0, MAX_INT**
- Check invalid input
  - Illegal sets of value
    - Wrong format: **not integer**
    - Negative numbers
  - Illegal input
    - Impossible conditions: **{2,3,8}, {2,3,5}** {definition of a triangle}
  - Totally bad input
    - Text vs. Numbers, etc.

Writing test cases

Let's consider the triangle example

- How to test the code?
- Have the code read from the **standard input**
  - **`java Triangle < testcases.txt`**
    or,
  - **`java Triangle`**
    `2 3 3`
    `3 4 5`
- Have the output print to standard output.
  - **`java Triangle > results.txt`**
- Combine the two and we have:
  - **`java Triangle <testcases.txt >results.txt`**
- This assumes we have the code read three numbers on the line, or have the code read three numbers whether they are on one line or more.

Tips for testing

- You cannot test every possible input, parameter value, etc.
  - So you must think of a limited set of tests likely to expose bugs.
- Think about boundary cases
  - positive; zero; negative numbers; infinity; very small
  - right at the edge of an array or collection's size (plus or minus one)
- Think about empty cases and error cases
  - 0, -1, null; an empty list or array
- test behavior in combination
  - maybe add usually works, but fails after you call remove
  - make multiple calls; maybe size fails the second time only

Test Cases – Good Example

Name	TC-47: Verify that lowercase data entry results in lowercase insert
Requirement	FR-4 (Case sensitivity in search-and-replace), bullet 2
Preconditions	The test document TESTDOC.DOC is loaded (base state BS-12).
Steps	1. Click on the "Search and Replace" button. 2. Click in the "Search Term" field. 3. Enter *This is the Search Term.* 4. Click in the "Replacement Text" field. 5. Enter *This IS THE Replacement TeRM.* 6. Verify that the "Case Sensitivity" checkbox is unchecked. 7. Click the OK button.
Expected results	1. The search-and-replace window is dismissed. 2. Verify that in line 38 of the document, the text *this is the search term* has been replaced by *this is the replacement term.* 3. Return to base state BS-12.

Test Cases – Bad Example

Steps	1. Bring up search-and-replace. 2. Enter a lowercase word from the document in the search term field. 3. Enter a mixed-case word in the replacement field. 4. Verify that case sensitivity is not turned on and execute the search.
Expected results	1. Verify that the lowercase word has been replaced with the mixed-case term in lowercase.

Test cases

- **Why we write test cases?**
  - The basic objective of writing test cases is **to validate the testing coverage of the application.**
- Keep in mind while writing test cases that all your **test cases should be simple and easy to understand**.
- For any application basically you will cover all the **types of test cases including functional, negative and boundary value test cases.**

Trustworthy tests

- Test one thing at a time per test method.
  - 10 small tests are much better than 1 test 10x as large.
- Each test method should have few (likely 1) assert statements.
  - If you assert many things, the first that fails stops the test.
  - You won't know whether a later assertion would have also failed.
- Tests should avoid logic.
  - minimize if/else, loops, switch, etc.
  - avoid try/catch
    - If it's supposed to throw, use expected= ... if not, let JUnit catch it.
- Torture tests are okay, but only in addition to simple tests.

Test Execution

- The software testers begin executing the test plan after the developers deliver the alpha build, or a build that they feel is feature complete.
- The alpha should be of high quality—the developers should feel that it is ready for release, and as good as they can get it.
- There are typically several iterations of test execution.
  - First, focus on new functionality
  - Then, regression test to make sure that a change to one area of the software has not caused any other part of the software
  - Regression testing usually involves executing all test cases which have previously been executed
  - There are typically at least two regression tests for any software project

Test Execution

- When is testing complete?
  - No defects found
  - Or defects meet acceptance criteria outlined in test plan

TABLE 8-6. Acceptance criteria from a test plan

1. Successful completion of all tasks as documented in the test schedule.
2. Quantity of medium- and low-level defects must be at an acceptable level as determined by the software testing project team lead.
3. User interfaces for all features are functionally complete.
4. Installation documentation and scripts are complete and tested.
5. Development code reviews are complete and all issues addressed. All high-priority issues have been resolved.
6. All outstanding issues pertinent to this release are resolved and closed.
7. All current code must be under source control, must build cleanly, the build process must be automated, and the software components must be labeled with correct version numbers in the version control system.
8. All high-priority defects are corrected and fully tested prior to release.
9. All defects that have not been fixed before release have been reviewed by project stakeholders to confirm that they are acceptable.
10. The end user experience is at an agreed acceptable level.
11. Operational procedures have been written for installation, set up, error recovery, and escalation.
12. There must be no adverse effects on already deployed systems.

Automating Test Execution

- Designing test cases and test suites is creative
  - Like any design activity: A demanding intellectual activity, requiring human judgment
- Executing test cases should be automatic
  - Design once, execute many times
- Test automation separates the creative human process from the mechanical process of test execution

From Test Case Specifications to Test Cases

- Test design often yields test case specifications, rather than concrete data
  - Ex: “a large positive number”, not 420023
  - Ex: “a sorted sequence, length > 2”, not “Alpha, Beta, Chi, Omega”
- Other details for execution may be omitted
- Generation creates concrete, executable test cases from test case specifications

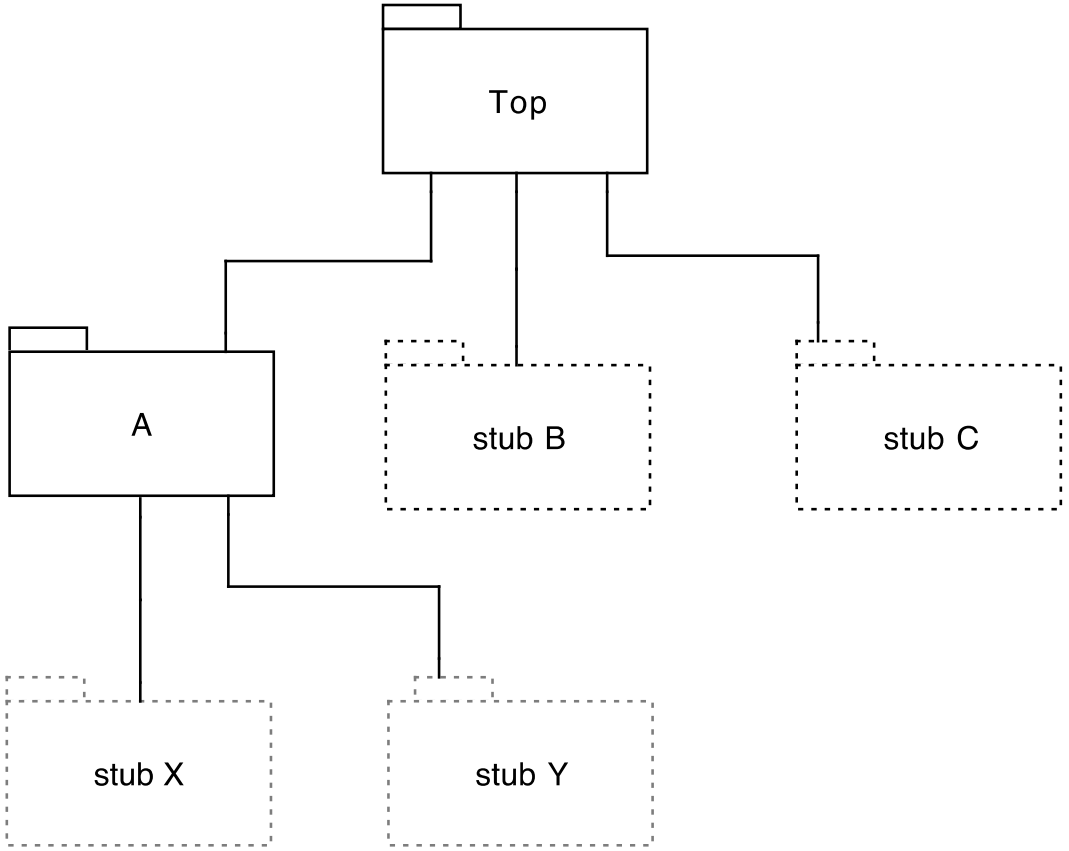
Scaffolding

- Code produced to support development activities (especially testing)
  - Not part of the "product" as seen by the end user
  - May be temporary (like scaffolding in construction of buildings)
- Includes
  - Test harnesses, drivers, and stubs
  - Example:
    - JUnit – test harness
    - Eclipse – IDE, scaffolding, JUnit built in

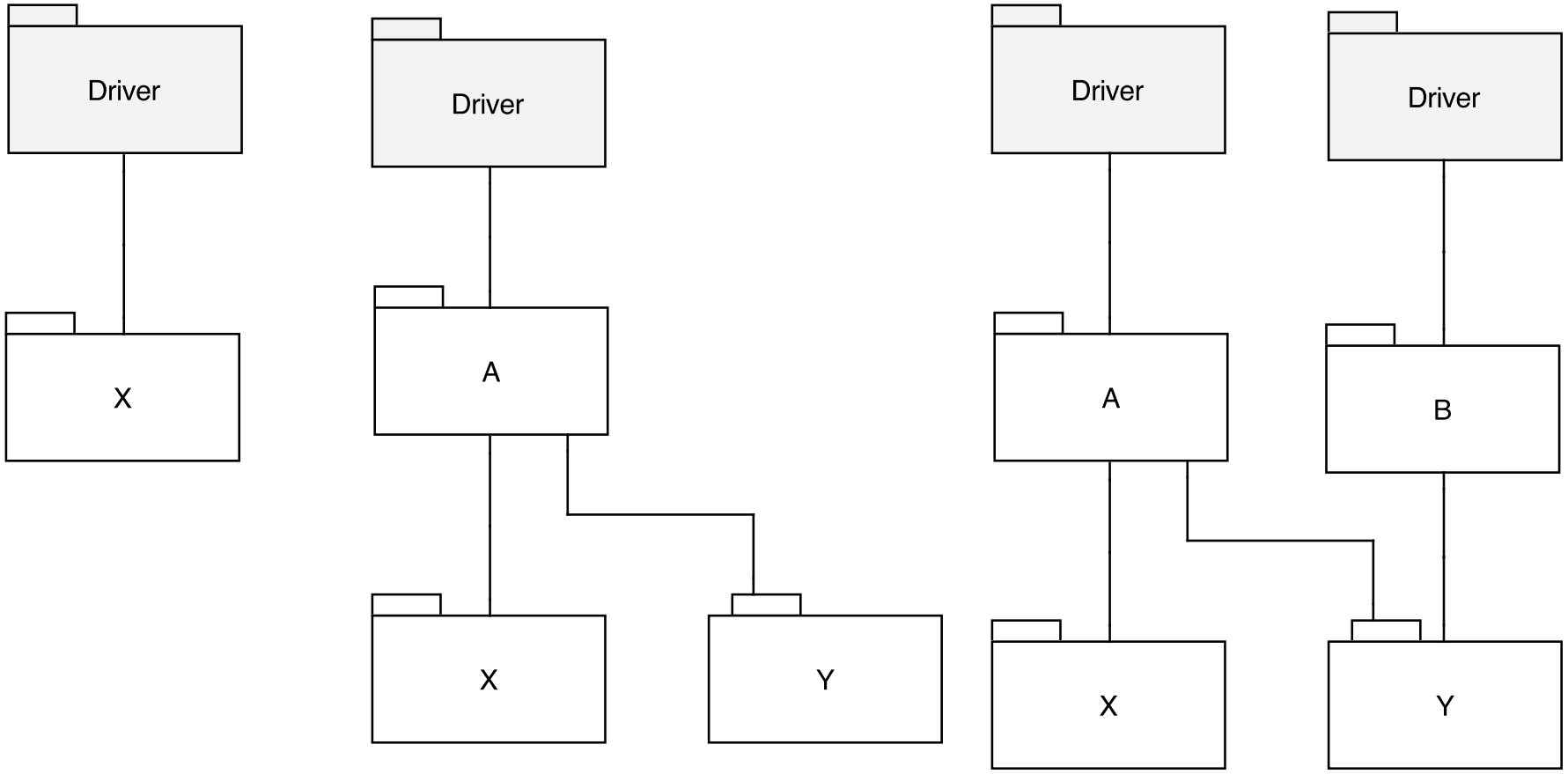
Scaffolding …

- Test driver
  - A “main” program for running a test
    - May be produced before a “real” main program
    - Provides more control than the “real” main program
      - To drive program under test through test cases
- Test stubs
  - Substitute for called functions/methods/objects
- Test harness
  - Substitutes for other parts of the deployed environment
    - Ex: Software simulation of a hardware device

Stubs



Drivers



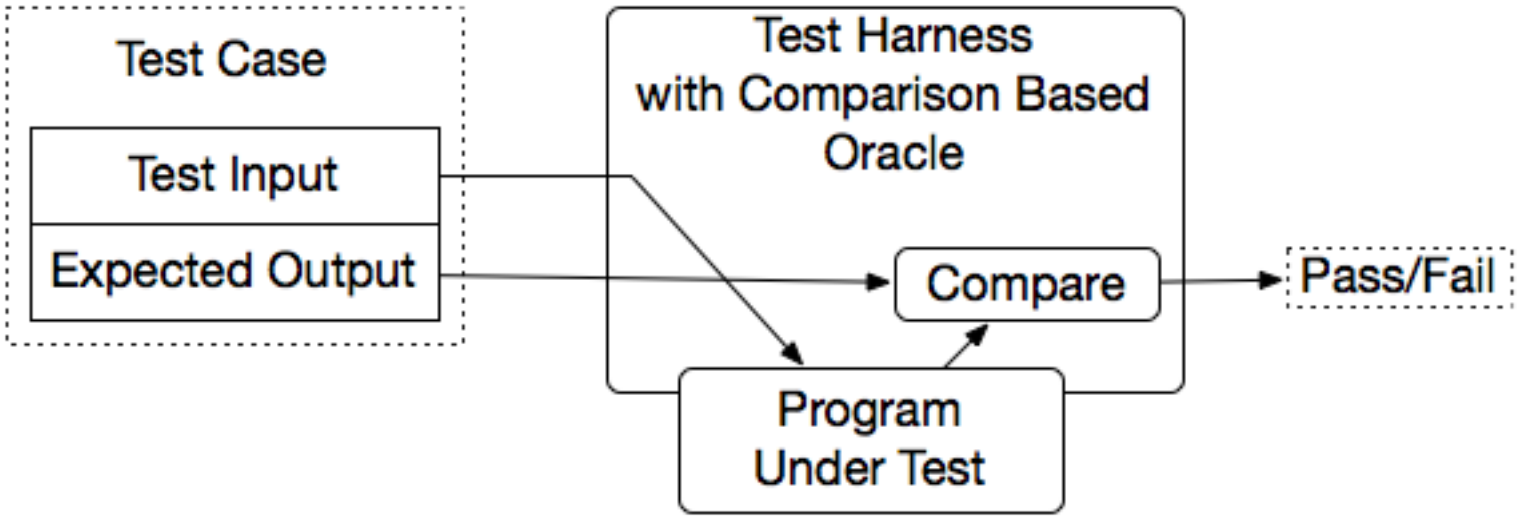
Generic or Specific?

- How general should scaffolding be?
  - We could build a driver and stubs for each test case
  - ... or at least factor out some common code of the driver and test management (e.g., **JUnit**)
  - ... or further factor out some common support code, to drive a large number of test cases from data (as in **DDSteps**)
  - ... or further, generate the data automatically from a more abstract model (e.g., network traffic model)
- A question of costs and re-use
  - Just as for other kinds of software

Oracles

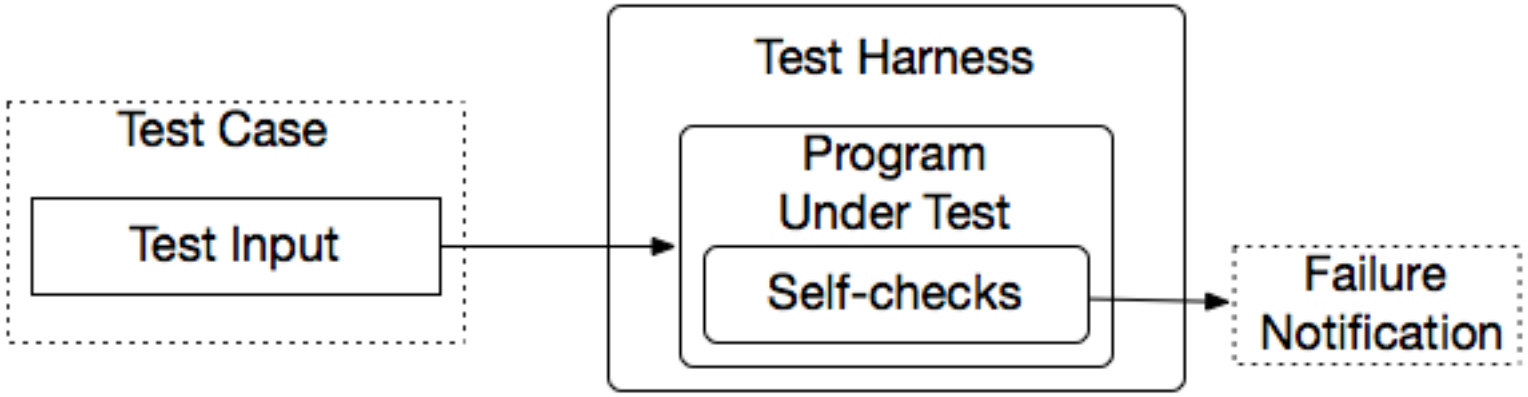
- Did this test case succeed, or fail?
  - No use running 10,000 test cases automatically if the results must be checked by hand!
- Range of specific to general, again
  - ex. JUnit: Specific oracle ("assert") coded by hand in each test case
  - Typical approach: "comparison-based" oracle with predicted output value
  - Not the only approach!

Comparison-based oracle



- With a comparison-based oracle, we need predicted output for each input
  - Oracle compares actual to predicted output, and reports failure if they differ
- Fine for a small number of hand-generated test cases
  - E.g., for hand-written JUnit test cases

Self-Checking Code as Oracle



- An oracle can also be written as self-checks
  - Often possible to judge correctness without predicting results
- Advantages and limits: Usable with large, automatically generated test suites, but often only a partial check
  - e.g., structural invariants of data structures
  - recognize many or most failures, but not all

Capture and Replay

- Sometimes there is no alternative to human input and observation
  - Even if we separate testing program functionality from GUI, some testing of the GUI is required
- We can at least cut *repetition* of human testing
- *Capture* a manually run test case, *replay* it automatically
  - with a comparison-based test oracle: behavior same as previously accepted behavior
    - reusable only until a program change invalidates it
    - lifetime depends on abstraction level of input and output

Defect Tracking

- The defect tracking system records and tracks defects.
- It routes each defect between testers, developers, the project manager and others, following a workflow designed to ensure that the defect is verified and repaired.

Smoke Tests

- A smoke test is a subset of the test cases that is typically representative of the overall test plan.
  - Smoke tests are good for verifying proper deployment or other non invasive changes.
  - They are also useful for verifying a build is ready to send to test.
  - Smoke tests are not substitute for actual functional testing.

Summary

- Goal: Separate creative task of test design from mechanical task of test execution
  - Enable generation and execution of large test suites
  - Re-execute test suites frequently (e.g., nightly or after each program change)
- Scaffolding: Code to support development and testing
  - Test drivers, stubs, harness, including oracles
  - Ranging from individual, hand-written test case drivers to automatic generation and testing of large test suites
  - Capture/replay where human interaction is required