Subject :

SoftwareEngineering

Topic: Unit testing

Name of the teacher :

LISNA THOMAS

Academic year:2020-21

Overview

- Testing Fundamentals
- Unit Test Planning
- Test Implementation
- Object Oriented Testing
- References

Terms for Quality Problems

- error - a mistake made by a human (in a software development activity)
- defect (or fault) - the result of introducing an error into a software artifact (SRS, SDS, code, etc.)
- failure - a departure from the required behavior for a system

Testing Fundamentals (1)

- Software Testing is a critical element of Software Quality Assurance
- It represents the ultimate review of the requirements specification, the design, and the code.
- It is the most widely used method to insure software quality.
- Many organizations spend 40-50% of development time in testing.

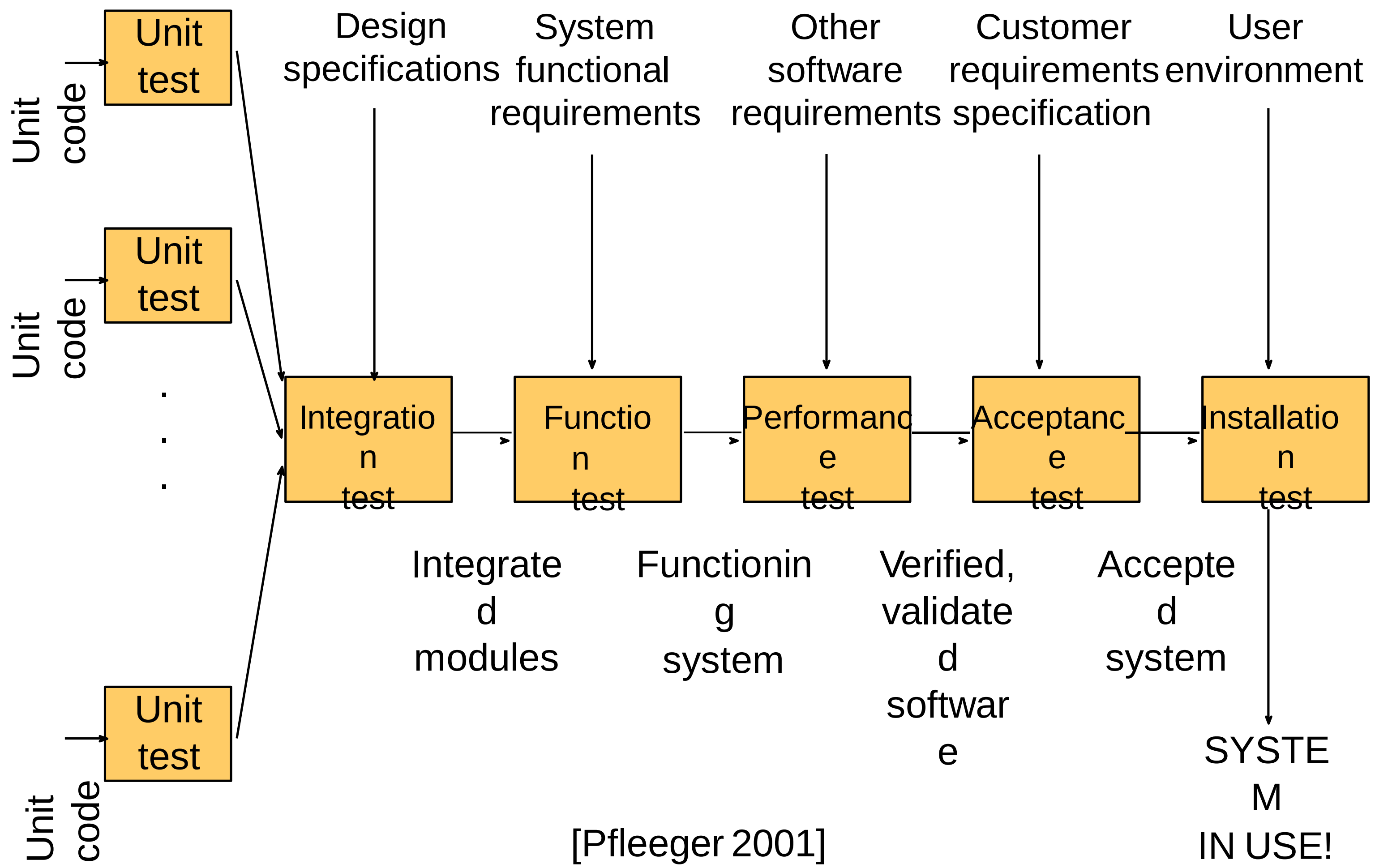
Testing Fundamentals (2)

- Testing is concerned with establishing the presence of program defects.
- Debugging is concerned with finding where defects occur (in code, design or requirements) and removing them. (fault identification and removal)
- Even if the best review methods are used (throughout the entire development of software), testing is necessary.

Testing Fundamentals (3)

- Testing is the one step in software engineering process that could be viewed as destructive rather than constructive.
  - "A successful test is one that breaks the software." [McConnell 1993]
- A successful test is one that uncovers an as yet undiscovered defect.
- Testing can not show the absence of defects, it can only show that software defects are present.
- For most software exhaustive testing is not possible.

Testing Steps



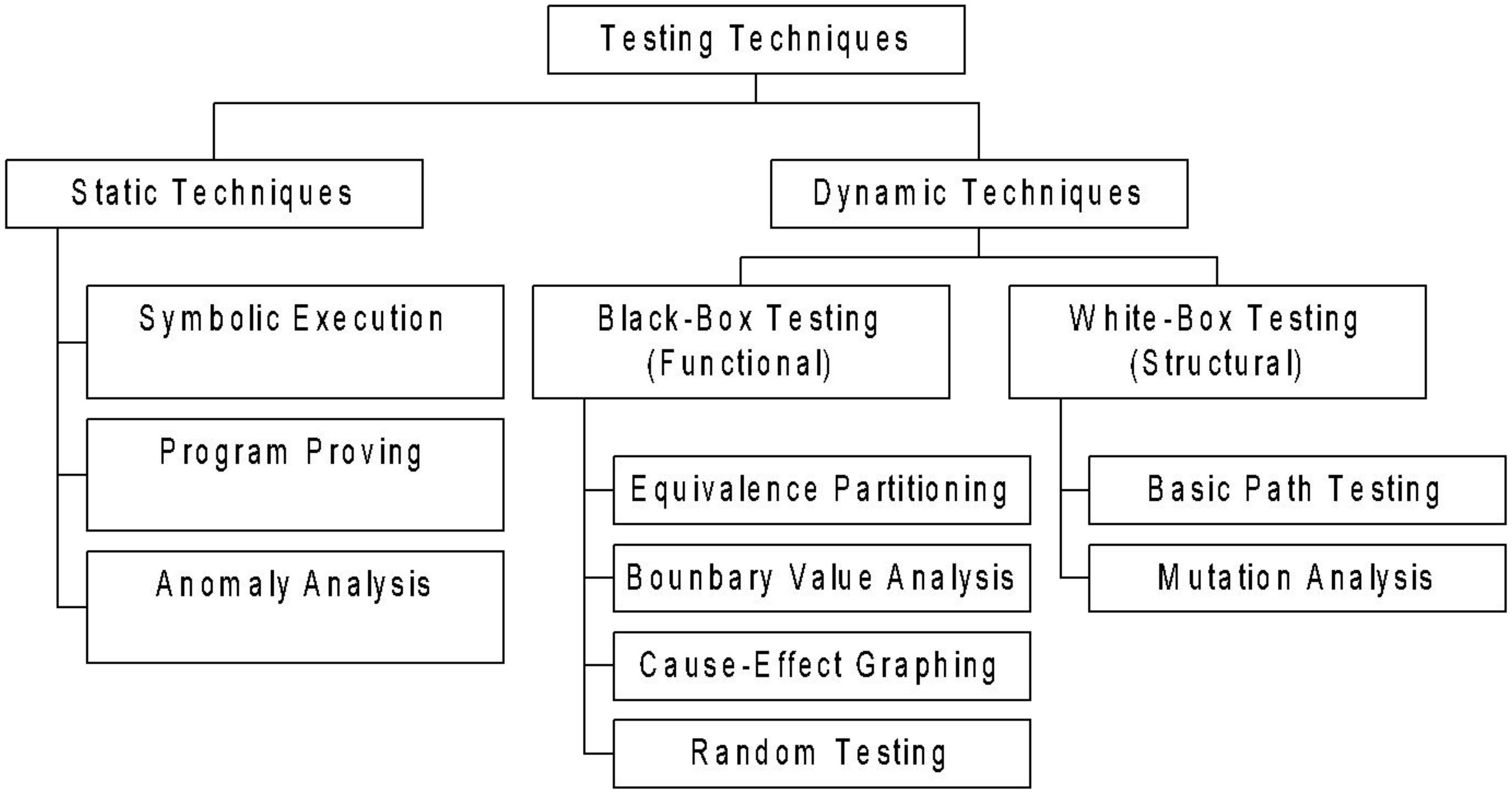
[Pfleeger 2001]

Unit Testing

- Unit Testing checks that an individual program unit (subprogram, object class, package, module) behaves correctly.
  - Static Testing
    - testing a unit without executing the unit code
  - Dynamic Testing
    - testing a unit by executing a program unit using test data
- There is some debate about what constitutes a "unit". Here some common definitions of a unit:
  - the smallest chunk that can be compiled by itself
  - a stand alone procedure of function
  - something so small that it would be developed by a single person

- First, unit testing in general
- Second, the role of unit testing in test-driven design (TDD)
  - Spring 05: We’ve done this already!

Unit Testing Techniques



Static Testing

- Symbolic Execution
  - works well for small programs
- Program Proving
  - difficult and costly to apply
  - does not guarantee correctness
- Anomaly Analysis
  - unexecutable code (island code)
  - uninitialized variables
  - unused variables
  - unstructured loop entry and exit

Dynamic Testing

- Black Box Techniques
  - design of software tests relies on module description to devise test data
  - uses inputs, functionality, outputs in the architectural design
  - treats module like a “black box”
- White Box Techniques
  - design of software tests relies on module source code to devise test data
  - analyze the module algorithm in the detailed design
  - treats module like a “ white box” or "glass box"

Common Testing Terms (1)

- test class
  - a set of input data that is identified for testing
  - Note difference from JUnit term
- test case
  - specific data that is chosen for testing a test class.
- test Suite
  - a set of test cases that serve a particular testing goal
- exhaustive testing
  - test all logical paths in a software module

Example: Exhaustive Testing

```
get(x)
for i := 1.. 20 loop
        if   F(x)   then
                ...
        else if   G(x)   then
                ...
        else if   H(x)   then
                ...
        else if   M(x) then
                                ...
        else
                …
        end if
        get(x)
end loop
```

Suppose in developing a test plan for the code at the left, you decide to create a set of test cases that would allow you to execute all possible combinations of the branches for the 20 iterations of the for loop .

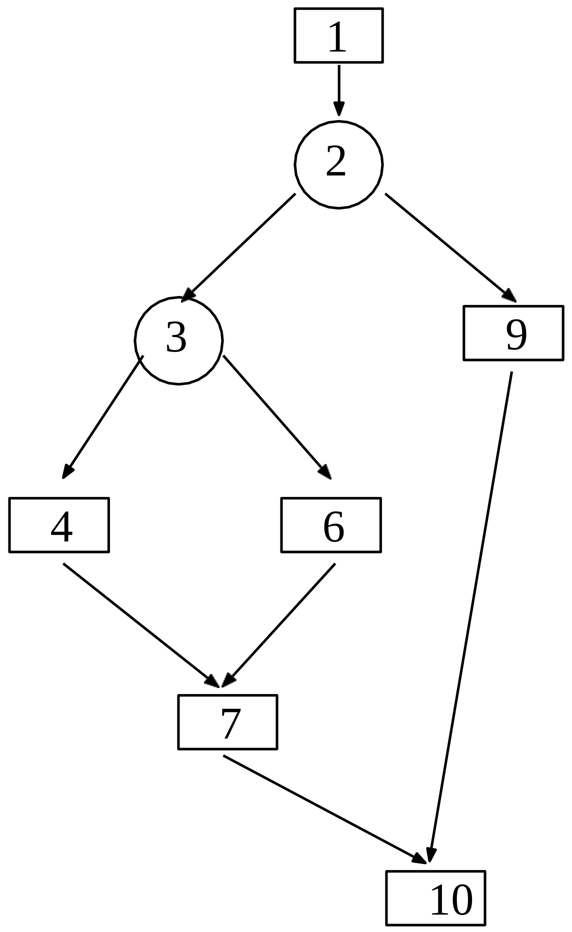
If it takes an average of 1 millisecond to execute a branch of the code (e.g., the instructions associated with the F(X) branch), how long will it take to run the test plan?

Basic Path Testing

- Basic Path Testing is a white box testing technique that consists of the following steps:
  - convert the unit into a “flow graph”
    - A flow graph is a directed graph (for an algorithm) with a "start node" and a "terminal node" (could have multiple terminal nodes, but not ideal).
  - compute a measure of the unit's logical complexity
  - use the measure to derive a “basic” set of execution paths for which test cases are determined.

Flow Graph Example

```
     procedure XYZ is
          A,B,C: INTEGER;
     begin
1.        GET(A); GET(B);
2.        if A > 15 then
3.             if  B < 10  then
4.                  B := A + 5;
5.             else
6.                  B := A - 5;
7.             end if
8.      else
9.          A := B + 5;
10.  end if;
     end XYZ;
```


$$\begin{tikzcd}
 & & \boxed{1} \arrow[d] & \\
 & & 2 \arrow[dl] \arrow[dr] & \\
 & 3 \arrow[dl] \arrow[dr] & & \boxed{9} \arrow[dddl] \\
\boxed{4} \arrow[dr] & & \boxed{6} \arrow[dl] & \\
 & \boxed{7} \arrow[dr] & & \\
 & & \boxed{10} &
\end{tikzcd}$$

Cyclomatic Complexity

- The cyclomatic complexity (McCabe complexity) is a metric, V(G), that describes the logical complexity of a flow graph, G. V(G) can be computed using any of the following formulae:
  - V(G) = E - N + 2
    - where E = number of edges in G and N = number of nodes in G
  - V(G) = R
    - where R = number of bounded regions in G
  - V(G) = P + 1
    - where P = number of predicates
- Studies have shown:
  - V(G) is directly related to the number of errors in source code
  - V(G) = 10 is a practical upper limit for testing

Basic Path Set

- An execution path is a set of nodes and directed edges in a flow graph that connects (in a directed fashion) the start node to a terminal node.
- Two execution paths are said to be independent if they do not include the same set of nodes and edges.
- A basic set of execution paths for a flow graph is an independent set of paths in which all nodes and edges of the graph are included at least once. V(G) provides an upper bound on the number of independent paths needed.

Equivalence Partitioning

- Equivalence partitioning is an approach to black box testing that divides the input domain of a program into classes of data from which test cases can be derived.
- Example: Suppose a program computes the value of the function $\sqrt{(X-1)*(X+2)}$. This function defines the following valid and invalid equivalence classes:

$X <= -2$	valid
$-2 < X < 1$	invalid
$X >= 1$	valid

- Test cases would be selected from each equivalence class.

Boundary Value Analysis (1)

- Boundary Value Analysis is a black box testing technique that where test cases are designed to test the boundary of an input domain. Studies have shown that more errors occur on the "boundary" of an input domain rather than on the "center".
- Boundary value analysis complements and can be used in conjunction with equivalence partitioning.

Boundary Value Analysis (2)

- In previous example, after using equivalence partitioning to generate equivalence classes, boundary value analysis would dictate that the boundary values of the three ranges be included in the test data. That is, we might choose the following test cases (for a 32 bit system):

$X <= -2$	-2^{31}, -100, -2.1, -2
$-2 < X < 1$	-1.9, -1, 0, 0.9
$X >= 1$	1, 1.1,100, 2^{31}-1

Unit Test Planning

- Unit test planning is typically performed by the unit developer
- Planning of a unit test should take place as soon as possible.
  - Black box test planning can take place as soon as the functional requirements for a unit are specified.
  - White box test planning can take place as soon as the detailed design for a unit is complete.
- Why not wait until a unit is coded and compiled, and ready for execution, before writing the test plan?

More Specific JUnit Advice

- Handout from *Pragmatic Unit Testing* (by Andrew Hunt and David Thomas)
- What to test? The Right-BICEP

Right-BICEP

- **Right**: Are the results **right**?
  - Validate results
  - Does the expected result match what the method does?
- If you don't know what "right" would be, then how can you test? How do you know if your code works?
  - Perhaps requirements not known or stable
  - Make a decision. Your tests document what you decided. Reassess if it changes later.

Right-BICEP (cont'd)

- Should your JUnit test hard-code test data to validate expected results?
- Option: use test data file with JUnit test
- See handout (p. 39) for example
  - Not so hard to do this, is it?
- Allows you to do a large range of tests cases as described elsewhere these slides

Right-BICEP

- **Boundary** Conditions
  - See earlier slides, and also consider:
  - Garbage input values
  - Badly formatted data
  - Empty or missing values (0, null, etc.)
  - Values out of reasonable range
  - Duplicates if they’re not allowed
  - Unexpected orderings

Right-BICEP

- Use this acronym, CORRECT, to remember:
  - Conformance
  - Ordering
  - Range
  - Reference
    - Does code reference anything external outside of its control?
  - Existence
  - Cardinality
  - Time (absolute and relative)
    - Are things happening in order? On time? In time?

Right-BICEP

- Check **Inverse** Relationships
  - If your method does something that has an inverse, then apply the inverse
  - E.g. square and square-root. Insertion then deletion.
  - Beware errors that are common to both your operations
    - Seek alternative means of applying inverse if possible

Right-BICEP

- **Cross-check** using other means
- Can you do something more than one way?
  - Your way, and then the other way.   Match?
- Are there overall consistency factors you can check?
  - Overall agreement

Right-BICEP

- Force **Error** Conditions
- Some are easy:
  - Invalid parameters, out of range values, etc.
- Others not so easy
  - See JUnit slide on how to force exceptions
- Failures outside your code:
  - Out of memory, disk full, network down, etc.
  - Can simulate such failures
    - Example: use Mock Objects

Right-BICEP

- Performance
  - Perhaps absolute performance, or
  - Perhaps how performance changes as input grows.
  - Perhaps separate test suite in JUnit
- But… Other, perhaps better ways to do this
  - JUnitPerf: http://www.clarkware.com/software/JUnitPerf.html

Example follows…

GCD Test Planning (1)

- Let’s look at an example of testing a unit designed to compute the “greatest common divisor” (GCS) of a pair of integers (not both zero).
  - GCD(a,b) = c where
    - c is a positive integer
    - c is a common divisor of a and b (e.g., c divides a and c divides b)
    - c is greater than all other common divisors of a and b.
  - For example
    - GCD(45, 27) = 9
    - GCD (7,13) = 1
    - GCD(-12, 15) = 3
    - GCD(13, 0) = 13
    - GCD(0, 0) undefined

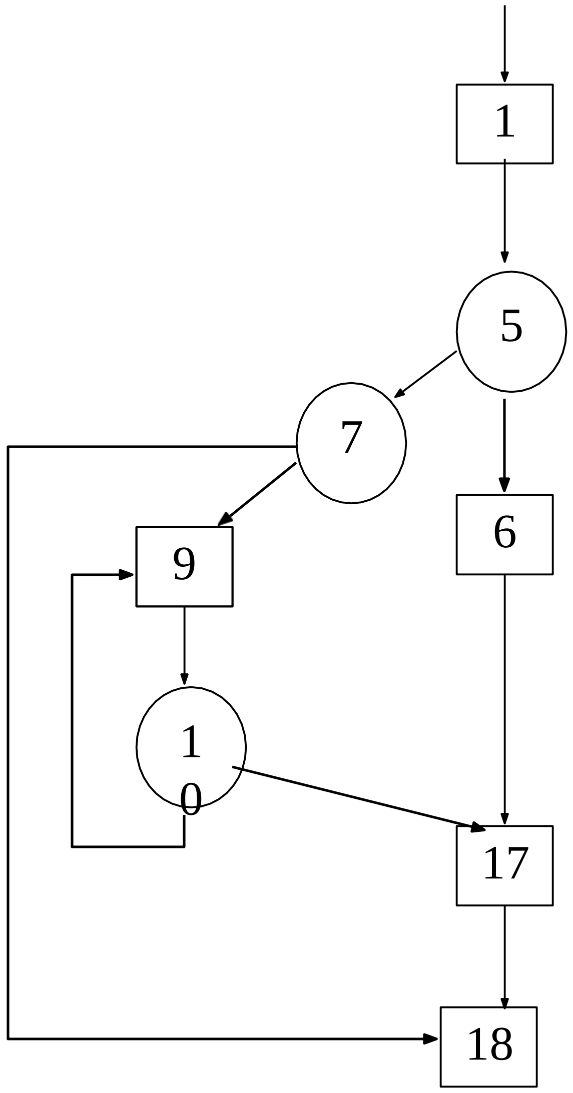
GCD Test Planning (2)

- How do we proceed to determine the tests cases?
  1. Design an algorithm for the GCD function.
  2. Analyze the algorithm using basic path analysis.
  3. Determine appropriate equivalence classes for the input data.
  4. Determine the boundaries of the equivalence classes.
  5. Then, choose tests cases that include the basic path set, data form each equivalence class, and data at and near the boundaries.

GCD Algorithm

note: Based on Euclid's algorithm

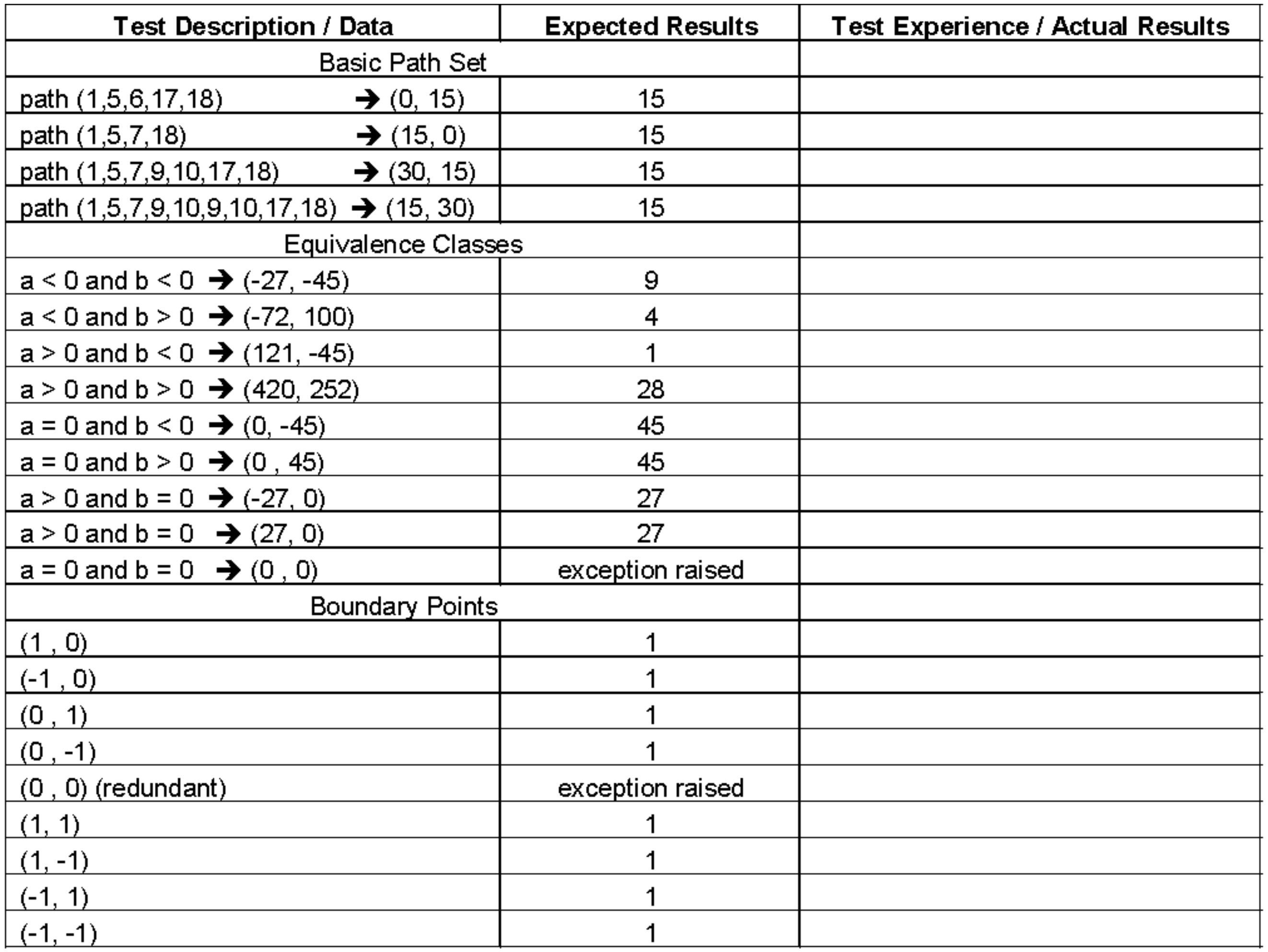
```
1.   function gcd (int a, int b) {
2.       int temp, value;
3.       a := abs(a);
4.       b := abs(b);
5.       if (a = 0) then
6.               value := b;     // b is the GCD
7.       else if (b = 0) then
8.           raise exception;
9.       else
10.              loop
11.                  temp := b;
12.                  b := a mod b;
13.                  a := temp;
14.              until (b =  0)
15.              value := a;
16.        end if;
17.        return value;
18.  end gcd
```



GCD Test Planning (3)

- Basic Path Set
  - V(G) = 4
  - (1,5,6,17,18), (1,5,7,18), (1,5,7,9,10,17,18), (1,5,7,9,10,9,10,17,18)
- Equivalence Classes
  - Although the the GCD algorithm should accept any integers as input, one could consider 0, positive integers and negative integers as “special” values. This yields the following classes:
    - $a < 0$ and $b < 0$, $a < 0$ and $b > 0$, $a > 0$ and $b < 0$
    - $a > 0$ and $b > 0$, $a = 0$ and $b < 0$, $a = 0$ and $b > 0$
    - $a > 0$ and $b = 0$, $a > 0$ and $b = 0$, $a = 0$ and $b = 0$
- Boundary Values
  - $a = -2^{31}, -1, 0, 1, 2^{31}-1$ and $b = -2^{31}, -1, 0, 1, 2^{31}-1$

GCD Test Plan



Test Description / Data	Expected Results	Test Experience / Actual Results
Basic Path Set		
path (1,5,6,17,18) → (0, 15)	15	
path (1,5,7,18) → (15, 0)	15	
path (1,5,7,9,10,17,18) → (30, 15)	15	
path (1,5,7,9,10,9,10,17,18) → (15, 30)	15	
Equivalence Classes		
a < 0 and b < 0 → (-27, -45)	9	
a < 0 and b > 0 → (-72, 100)	4	
a > 0 and b < 0 → (121, -45)	1	
a > 0 and b > 0 → (420, 252)	28	
a = 0 and b < 0 → (0, -45)	45	
a = 0 and b > 0 → (0 , 45)	45	
a > 0 and b = 0 → (-27, 0)	27	
a > 0 and b = 0 → (27, 0)	27	
a = 0 and b = 0 → (0 , 0)	exception raised	
Boundary Points		
(1 , 0)	1	
(-1 , 0)	1	
(0 , 1)	1	
(0 , -1)	1	
(0 , 0) (redundant)	exception raised	
(1, 1)	1	
(1, -1)	1	
(-1, 1)	1	
(-1, -1)	1	

Anything missing?

Test Implementation (1)

- Once one has determined the testing strategy, and the units to tested, and completed the unit test plans, the next concern is how to carry on the tests.
  - If you are testing a single, simple unit that does not interact with other units (like the GCD unit), then one can write a program that runs the test cases in the test plan.
  - However, if you are testing a unit that must interact with other units, then it can be difficult to test it in isolation.
  - The next slide defines some terms that are used in implementing and running test plan.

Test Implementation Terms

- Test Driver
  - a class or utility program that applies test cases to a component being tested.
- Test Stub
  - a temporary, minimal implementation of a component to increase controllability and observability in testing.
  - When testing a unit that references another unit, the unit must either be complete (and tested) or stubs must be created that can be used when executing a test case referencing the other unit.
- Test Harness
  - A system of test drivers, stubs and other tools to support test execution

Test Implementation (2)

- Here is a suggested sequence of steps to followed in testing a unit.
  - Once the design for the unit is complete, carry out a static test of the unit. This might be a single desk check of the unit or it may involve a more extensive symbolic execution or mathematic analysis.
  - Complete a test plan for a unit.
  - If the unit references other units, not yet complete, create stubs for these units.
  - Create a driver (or set of drivers) for the unit, which includes the following;
    - construction of test case data (from the test plan)
    - execution of the unit, using the test case data
    - provision for the results of the test case execution to be printed or logged as appropriate

Unit Testing in TDD

- Motto: "Clean code that works." (Ron Jeffries)
- Unit testing has "broader goals" that just insuring quality
  - Improve developers lives (coping, confidence)
  - Support design flexibility and change
  - Allow iterative development with working code early

Unit Testing Benefits

- Developers can work in a predictable way of developing code
- Programmers write their own unit tests
- Get rapid response for testing small changes
- Build many highly-cohesive loosely-coupled modules to make unit testing easier

Red/Green/Refactor

- The TDD mantra of how to code:
  - Red: write a little <u>test</u> that doesn’t work, perhaps even doesn’t compile
  - Green: Write <u>code</u> to make the test work quickly (perhaps not the best code)
  - Refactor: Eliminate duplication and other problems that you did to just make the test work

Assigned Readings

- http://junit.sourceforge.net/doc/testinfected/testing.htm
- http://junit.sourceforge.net/doc/cookstour/cookbook.htm

Object-Oriented Testing Issues

- The object class is the basic testing unit for system developed with OO techniques.
  - This is especially appropriate if strong cohesion and loose coupling is applied effectively in the class design.
- An incremental testing approach is dictated by
  - The package/class design decomposition.
  - The incremental nature of the development process.
- Information hiding restricts/constrains testing to using a white-box approach.
- Encapsulation motivates testing based on the class interface.

Class Testing (1)

- Class test cases are created by examining the specification of the class.
  - This is more difficult if the class is a subclass and inherits data and behavior from a super class. A complicated class hierarchy can be pose significant testing problems.
- If you have a state model for the class, test each transition - devise a driver that sets an object of the class in the source state of the transition and generates the transition event.
- Class Constraints/Invariants should be incorporated into the class test.
- All class methods should be tested, but testing a method in isolation from the rest of the class is usually meaningless
  - In a given increment, you may not implement all methods; if so, create stubs for such methods.

Class Testing (2)

- To test a class, you may need instances of other classes.
- Interaction diagrams provide useful guidance in constructing class test cases:
  - They provide more specific knowledge about how objects of one class interact with instances of other classes.
  - Messages sent to a class object provide a guidance on which test cases are most critical for that class.

Method Testing (1)

- A public method in a class is typically tested using a black-box approach.
  - Start from the specification, no need to look at the method body.
  - Consider each parameter in the method signature, and identify its equivalence classes.
  - Incorporate pre-conditions and post-conditions in your test of a method.
  - Test exceptions.
- For complex logic, also use white-box testing or static testing.

Method Testing (2)

- For private methods, either
  - modify the class (temporarily) so that it can be tested externally:
    - change the access to public
    - or incorporate a test driver within the class
  - or use static test methods, such as program tracing or symbolic execution

Incremental Testing

- Incremental testing indicates that we test each unit in isolation, and then integrate each unit, one at a time, into the system, testing the overall system as we go.
- Classes that are dependent on each other called <u>class clusters</u>, are good candidates for an increment integration.
- Candidate class clusters:
  - Classes in a package
  - Classes in a class hierarchy.
  - Classes associated with the interaction diagram for a use case.

Integration Test Plan

- An Integration Test checks and verifies that the various components in a class cluster communicate properly with each other.
- With this plan, the emphasis is not on whether system functionality is correctly implemented, but rather do all the cluster classes “fit” together properly.
- A single test plan, for each cluster, which tests the communication between all cluster components is typically sufficient.

Increment Test Planning

- Determine the classes to be tested in an increment.
- Determine the appropriate “class clusters”.
- Develop a unit test plan and test driver for each class.
- Develop an integration test plan and test driver for each class cluster.
- Develop a test script that details the components to be tested and the order in which the plans will be executed.

Testing Tools

- There are a number of tools that have been developed to support the testing of a unit or system.
  - googling “Software Testing Tools” will yield thousands of results.
- JUnit testing (http://www.junit.org/index.htm) is a popular tool/technique that can be integrated into the development process for a unit coded in Java.

References

- [Beck 2004] Beck, K., and Gamma, E. *Test Infected: Programmers Love Writing Tests*, http://members.pingnet.ch/gamma/junit.htm, accessed July 2004.
- [Binder 1999] Binder, R.V., *Testing Object-Oriented Systems*, Addison-Wesley, 1999.
- [Humphrey 1995] Humphrey, Watts S., *A Discipline for Software Engineering*, Addison Wesley, 1995.
- [McConnell 1993] McConnell, Steve, *Code Complete, A Practical Handbook of Software Construction*, Microsoft Press, 1993.
- [Jorgensen 2002] Jorgensen, Paul C., *Software Testing: A Craftsman's Approach*, 2nd edition, CRC Press, 2002.
- [Pfleeger 2001] Pfleeger, S., *Software Engineering Theory and Practice*, 2nd Edition, Prentice-Hall, 2001.
- [Pressman 2005] Pressman, R.S., *Software Engineering: A Practitioner's Approach*, 6th edition, McGraw-Hill, 2005.