Software Testing

Andy Podgurski

Electrical Engineering & Computer Science Department
Case Western Reserve University

Software Testing

- Employs *inductive inference* to validate software
  - Making a *general conclusion* based on *specific examples*
- Software is executed on a *sample* of inputs – the *test set* or *test suite*
- Its behavior is evaluated for *conformance to requirements*

Testing Terminology

- *Failure*: runtime deviation from required behavior
- *Defect, fault, or bug*: A flaw in a program that can cause it to fail
- *Error*: human action that results in a defect or fault
- *Correctness*: property that program absolutely satisfies functional requirements
- *Reliability measure*: measure of extent to which behavior of deployed software conforms to requirements
  - e.g., *frequency of failures*
  - Depends on *operational (field) usage*

See also www.istqb.org/downloads/send/20-istqb-glossary/186-glossary-all-terms

Broad Categories of Testing

- *Synthetic testing*
  - Tester or tool generates test data
- *Operational/field testing*
  - Software tested in field or with inputs *captured* in field
  - Prerequisite for *measuring reliability*
- *Simulation testing*
  - Approximation to field testing
  - Used when field testing is infeasible
  - Requires probabilistic *simulation model*

Goals of Testing

- *Reliability assessment*
  - Important for release/deployment decisions
  - Often done *subjectively*, based on *in-house* testing
  - Can be done *statistically*, based on *field* testing
- *Reliability improvement*
  - Defect identification and repair

Limitations of Testing

- Testing *cannot* generally demonstrate program *correctness*
  - Result proved using techniques of *Computability Theory*
    - e.g., reduction from *Halting Problem*
- Testing is *not* certain to reveal all defects
  - Even *well tested* software typically has *latent defects*

Alternatives to Testing

- *Inspection*
  - Feasible and very beneficial
  - Inspectors *overlook* many defects, however
- *Static program analysis*
  - Effective for finding *violations* of *implicit programming rules*
- *Program verification (formal verification)*
  - Intended to produce *proof of correctness*
  - Does *not* address erroneous specifications
  - *Manual* verification *impractical* for large programs
  - Can be *partially automated*:
    - *Automatic theorem proving*
      - May not succeed, even for correct program
    - *Finite-state model checking*
      - Checks program *properties* weaker than correctness (e.g., no deadlock)
      - Computationally intensive
      - *Abstraction* required to reduce number of states
- *Reliability estimation*
  - Employs *statistical sampling methodology*
  - Based on *field testing*

Testing Costs

- Principal costs of traditional testing:
  - *Analysis of specification and program*
  - *Test data generation*
  - *Program execution*
  - *Evaluation of program behavior (or determining correct output in advance)*
- Test generation and evaluation often require much *manual effort*
  - Developer/tester time is *expensive*

Test Oracle

- Means of determining if *test outcome* is *correct*
- Automated, complete oracle requires* a *correct implementation* of requirements
  - *"Gold" program – generally not available*
- Typically, humans check test output or determine expected results *manually*
  - Actual results can often be *compared automatically* to expected ones
- *Partial checks* can often be automated
  - E.g., check that output of sorting routine is in proper order

*Usually

Testing Phases

- *Unit Testing*
  - Routine or module tested
  - Analysis and evaluation are simplest
  - *Driver* and *stubs* often needed
  - Supported by frameworks like xUnit
- *Integration Testing & Subsystem Testing*
  - Units integrated into subsystem
  - *Interactions* between units tested
  - Subsystem tested as a whole
  - Analysis and evaluation are usually harder
  - Driver and stubs often needed
- *System Testing*
  - Subsystems integrated
  - Entire system tested
  - Analysis and evaluation are usually hardest
- *Field Testing*
  - System used in field by *ordinary users*, as they wish
  - One or more sites
  - Extended duration (e.g. months)
  - Users report problems
- *Acceptance Testing*
  - Testing by *customer*
  - Basis for accepting software
- *Regression Testing*
  - Retesting after maintenance
  - Involves some tricky issues

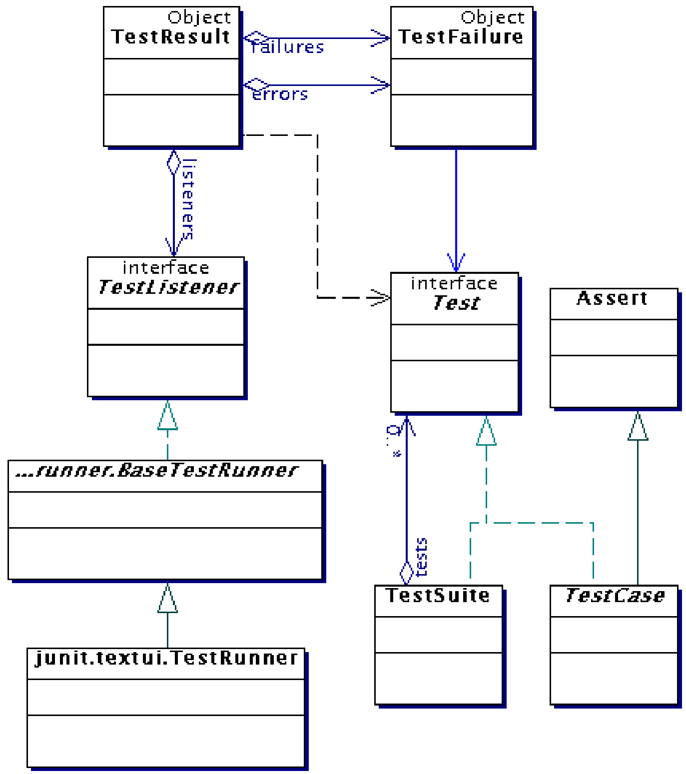
↑ Who conducts these phases?

Unit Testing Frameworks

- These facilitate
  - Creating, organizing, and running automated tests
  - Presenting and summarizing test results
- SUnit [Beck, 1989] gave rise to the xUnit family of frameworks

Example: JUnit Testing Framework (circa ~2008)

- *TestRunner* runs tests and reports *TestResults*
- Tests extend abstract class *TestCase*
- *Assertions* used to check behavior of unit under test
- Tests combined into *TestSuite*

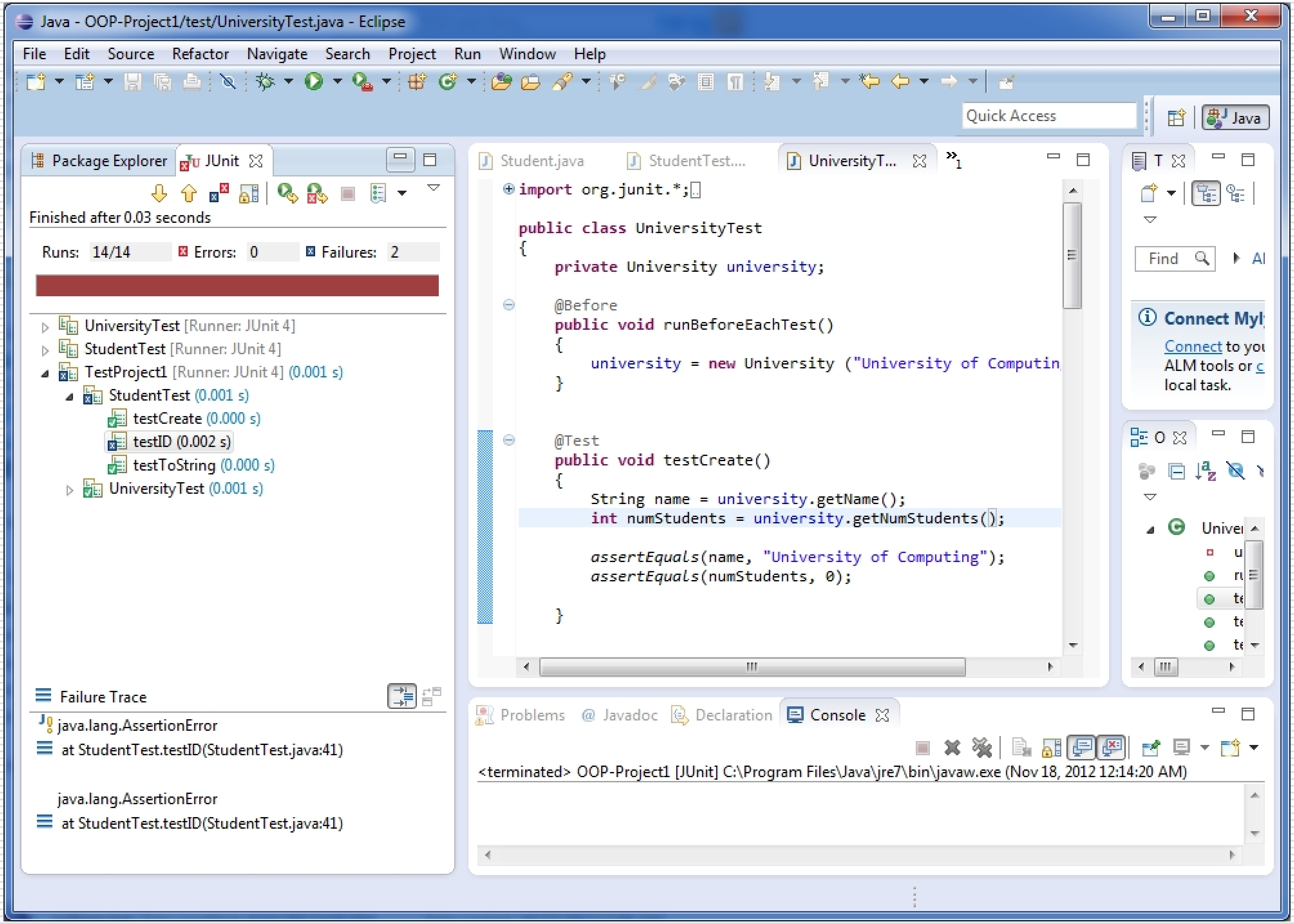


Example: JUnit 4 Test

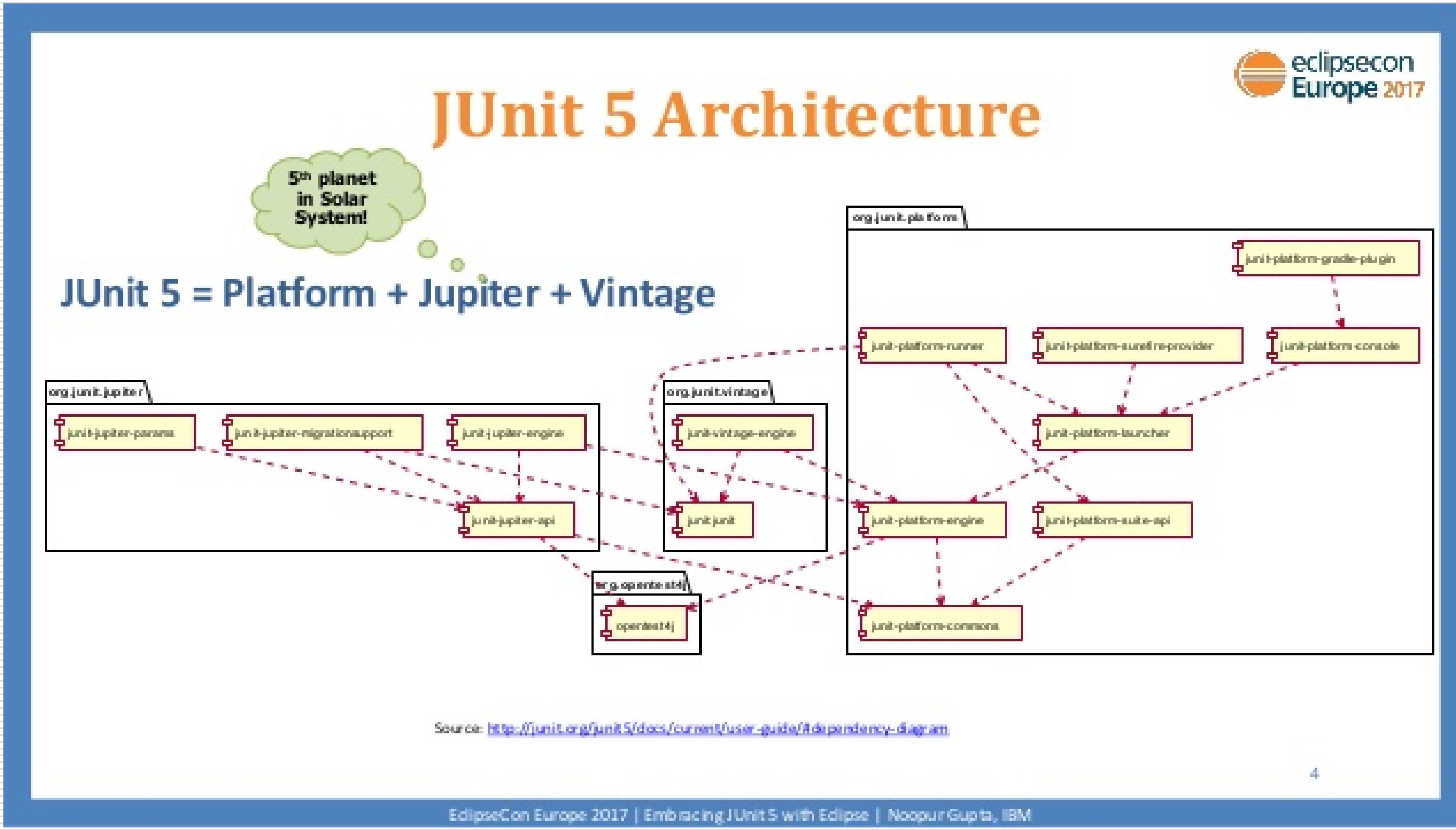
```
public class Calc {
  public long add(int a, int b) {
    return a+b;
  }
}

import org.junit.Test;
import static org.junit.Assert.assertEquals;

public class CalcTest {
  @Test
  public void testAdd() {
    assertEquals(5, new Calc().add(2, 3));
  }
}
```

www.javabeat.net/2008/10/junit-4-0-example/

Junit in Eclipse



oopbook.com/junit-testing/junit-testing-in-eclipse/



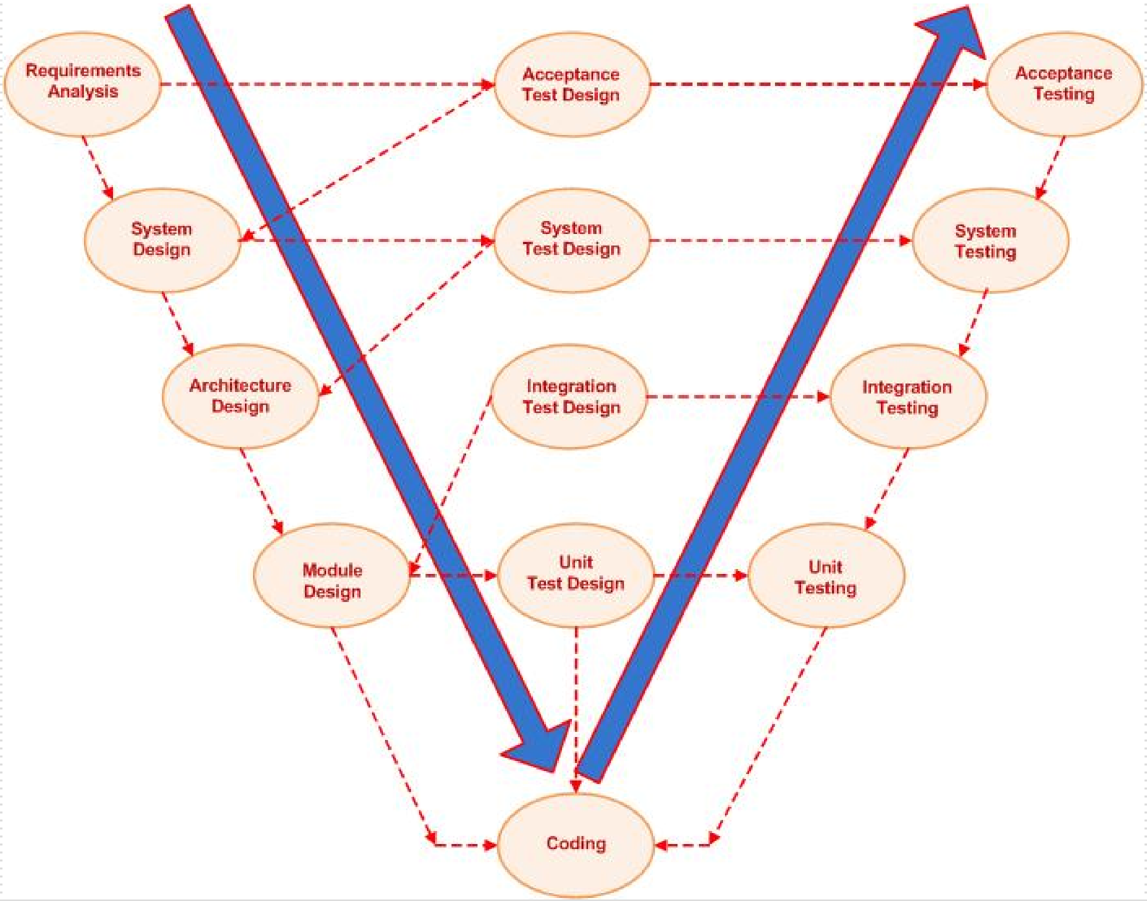
junit.org/junit5/docs/current/user-guide/

When to Test

- Principle: *Test each component as soon as it becomes possible*
- Upon completing component, developer has greatest:
  - *Motivation* to test it
  - *Focus, recall* and *understanding*
- Cost of repair is *least*
- Early testing is facilitated by *tool support*
  - Obviates need to write drivers, stubs

Testing in V-Model of Software Development

influences



There are many variants of the V-Model.

Synthetic Testing

- Goal: select test data likely to *reveal defects*
- Sources of information used:
  - *Knowledge of typical programming errors*
  - *Knowledge of application domain*
  - *Requirements specification document*
  - *Code*
- Used to define *testing criteria*
  - Conditions that must be satisfied by test set
- Tests data generation *typically manual*, but can be automated in some cases

Types of Synthetic Testing

- *Functional testing*
  - Synonyms: specification-based, black box
- *Structural testing*
  - Synonyms: code-based, coverage, glass box
- *Boundary and special values testing*
  - Special case: load testing
- *Interaction testing*
- *Model-based testing*
  - *State-based testing*
- *Fault-based testing*
- *Equivalence-partition and subdomain testing*
- *Random testing*
- *Hybrid techniques*

Functional Testing

- *Exercises each specified functional requirement at least once*
  - *Each case in requirement exercised at least once*
- Motivation: any requirement may be
  - *Improperly specified or implemented*
  - *Neglected (unimplemented)*
- *Most common* type of testing in practice
- May be applied to *any component with a specification*
  - Method, class, subsystem, system

Example: Functional Testing of Blood Analyzer Software Component

Requirements:

29.A. PROFILE EDITOR

- 29.A.1. The user shall be able to define a collection of related blood tests for the purpose of scheduling convenience and grouping results.
- 29.A.2. The user shall be able to edit existing profile definitions.
- 29.A.3. The user shall be able to view profile definitions.
- 29.A.4. The user shall be able to delete profile definitions.
- 29.A.5. The user shall be able to uniquely name profiles.
- 29.A.6. The user shall be able to assign profile name aliases
- 29.A.7. The user shall be able to assign assays.
- 29.A.8. All result demographic screen presentations shall have user selectable fields.

Basic functional test criteria:

- Create test collections
- Edit profile definitions
  - Exercise each editing operation
- View profile definitions
- Delete profile definitions
- Name profiles and use their names
- Assign and use profile name aliases
- Assign assays
- Select fields of demographic screen presentations

See also: http://www.math-cs.gordon.edu/courses/cs211/ATMExample/InitialFunctionalTests.html

Example: Functional Tests of Automated Teller Machine

1. Verify the slot for ATM Card insertion is as per the specification
2. Verify that user is presented with options when card is inserted from proper side
3. Verify that no option to continue and enter credentials is displayed to user when card is inserted correctly
4. Verify that font of the text displayed in ATM screen is as per the specifications
5. Verify that touch of the ATM screen is smooth and operational
6. Verify that user is presented with option to choose language for further operations
7. Verify that user asked to enter pin number before displaying any card/bank account detail
8. Verify that there are limited number of attempts up to which user is allowed to enter pin code
9. Verify that if total number of incorrect pin attempts gets surpassed then user is not allowed to continue further- operations like blocking of card etc gets initiated
10. Verify that pin is encrypted and when entered
11. Verify that user is presented with different account type options like-saving, current etc
12. Verify that user is allowed to get account details like available balance

artoftesting.com/manualTesting/atm.html

Question: Is the list above complete?

Functional Testing cont.

- *Multiple tests* may be created for *each* requirement
- May focus on input *subdomains*, *boundary conditions*, and *special values*
  - e.g., invalid inputs

Example: Multiple Functional Tests of Feature

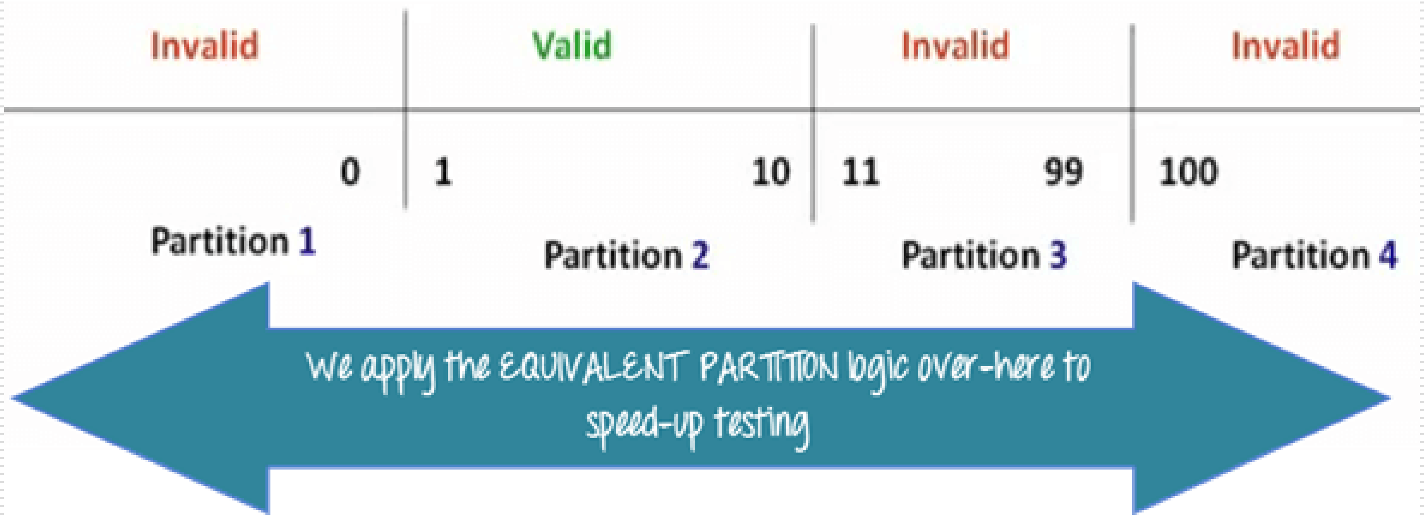
	Author	Test Engineer 1			
Test Id	Description	Test Steps	Expected Result	Actual Result	Status
1	Test the "Save" button with valid values in mandatory fields	Step:1 Enter valid values in "Name" Step:2 Enter valid values in "Email" Step:3 Click "Save"	The information should be saved		
2	Test the "Save" button with invalid values in mandatory fields	Step:1 Enter invalid values in "Name" Step:2 Enter invalid values in "Email" Step:3 Click "Save"	The information should not be saved. System should request the user to enter valid values.		
3	Test the "Save" button by not entering any data in fields	Step:1 Click "Save"	The information should not be saved. System should request the user to enter values.		

http://www.codentest.com/images/testcases.jpg

Equivalence Partition and Subdomain Testing

- Techniques for *economizing* on test effort
- Involve dividing input domain of program into *subdomains*
  - Inputs in each subdomain are treated *similarly* by the spec or program
  - *One* or *a few* test cases are selected from each subdomain
- *Disjoint* subdomains can be viewed as *equivalence classes*
  - These form a *partition* of the input domain
- Many testing techniques can be viewed as forms of subdomain testing

Example: Partition Testing



www.guru99.com/equivalence-partitioning-boundary-value-analysis.html

Category-Partition Testing

1. *Decompose* functional specification into *functional units* that can be tested independently
2. Determine general *categories* of *input parameters* and *environmental variables*
3. *Partition* each category into *separate choices*
4. Determine *constraints* among choices of different categories
5. Write the *test specification*
6. Use a *generator* to produce *test frames* from the test specification
7. For each generated test frame, create a *test case* by selecting one element from each choice in the frame

Test Specification for FIND Command with Category Partitions

Categories

Choices

```
Parameters:
    Pattern size:
        empty                                   [property Empty]
        single character                        [property NonEmpty]
        many character                          [property NonEmpty]
        longer than any line in the file        [property NonEmpty]

    Quoting:
        pattern is quoted                       [property Quoted]
        pattern is not quoted                   [if NonEmpty]
        pattern is improperly quoted            [if NonEmpty]

    Embedded blanks:
        no embedded blank                       [if NonEmpty]
        one embedded blank                      [if NonEmpty and Quoted]
        several embedded blanks                 [if NonEmpty and Quoted]

    Embedded quotes:
        no embedded quotes                      [if NonEmpty]
        one embedded quote                      [if NonEmpty]
        several embedded quotes                 [if NonEmpty]

    File name:
        good file name
        no file with this name
        omitted


Environments:
    Number of occurrences of pattern in file:
        none                                    [if NonEmpty]
        exactly one                             [if NonEmpty] [property Match]
        more than one                           [if NonEmpty] [property Match]

    Pattern occurrences on target line:
        #  assumes line contains the pattern
        one                                     [if Match]
        more than one                           [if Match]
```

FIGURE 4. Specification with Property Lists and Selector Expressions

From “The Category Partition Method for Specifying and Generating Functional Tests” by T.J. Ostrand and M. J. Balcer

Category-Partition Testing cont.

- Question: what fundamental Computer Science concept is used in forming categories and choices?

Partition Testing Example

Develop a program *loan* for use by ABC Bank to process applications by its customers for personal loans, based on their employment and credit card details. In order to evaluate an application, the program will accept the following details from the applicant. The evaluation criteria are not specified here.

- Employment Status: Either "Employed" or "Unemployed".
- Type of Employment (if the applicant is working): Either "Self-Employed" or "Employed by Others".
- Type of Job (if the applicant is working): Either "Permanent" or "Temporary".
- Monthly Salary S (if the applicant is working): Either "$\$0 < S \leq \$2000$", "$\$2000 < S \leq \$3000$", or "$S > \$3000$".
- Type of Applicant: Either "Cardholder" or "Non-Cardholder".
- Type of Credit Card (if the applicant is a cardholder): Either "Gold" or "Classic".
- Credit Limit (applicable only to a classic card): Either "\$2000" or "\$3000".

It should be noted that there is no credit limit for a gold card.

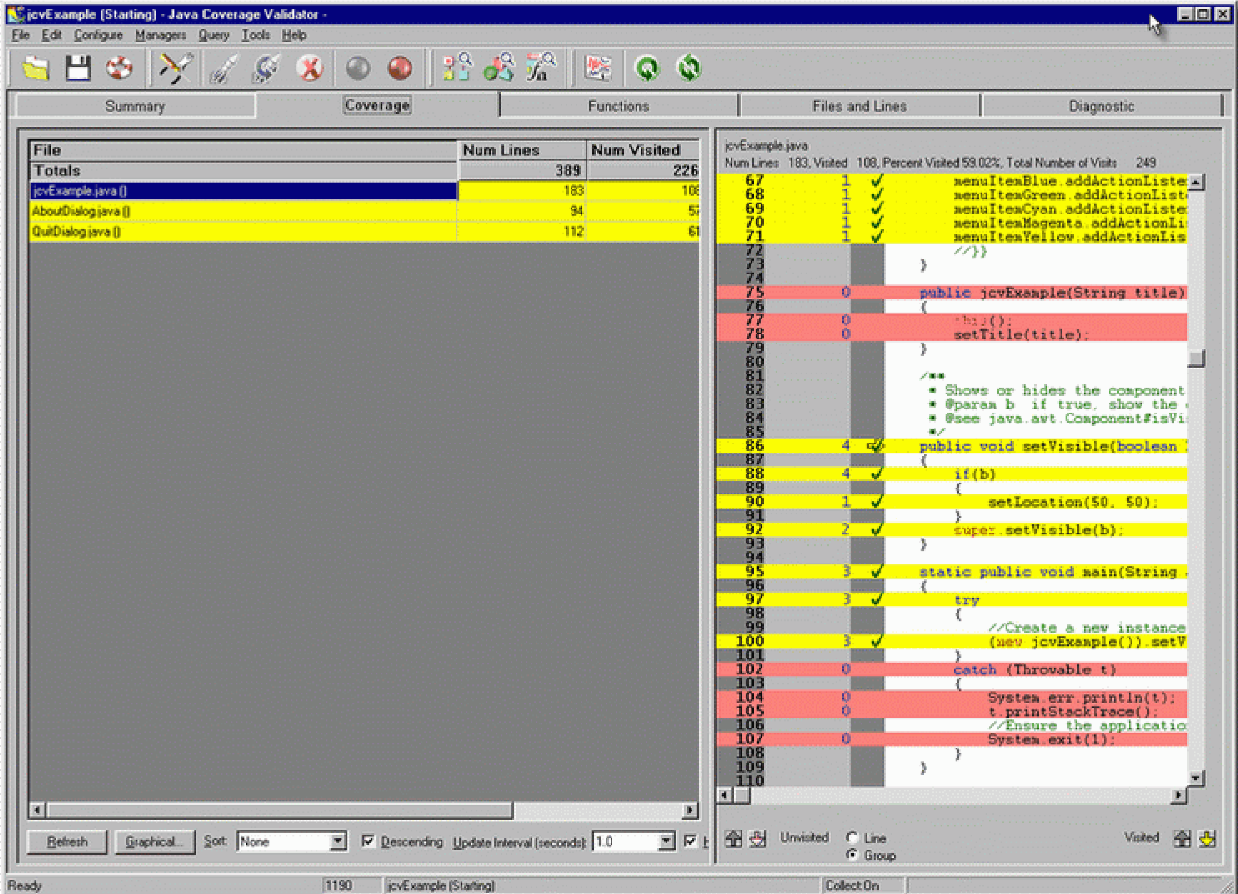
- *Cartesian-product* of sets of attribute categories defines partition of input domain
- It may not be practical to vary more than one attribute at a time

Example from T. Y. Chen, P.-L. Poon, and T. H. Tse, "A Choice Relation Framework for Supporting Category-Partition Test Case Generation", IEEE Transactions on Software Engineering, vol. 29, no. 6, June 2003.

Structural Testing

- Exercises or *covers* each program element of a certain type, e.g.,
  - Statements or basic blocks
  - Conditional branches
  - Control flow paths (in control flow graph)
  - Definition-use (data flow) chains
- Motivation:
  - *Any such element may be defective*
  - *Simply executing it may trigger failure*
- *Percent coverage* achieved by test set is one measure of testing *completeness*
  - *Inadequate* by itself

Example: softwareverify.com Java Coverage Validator

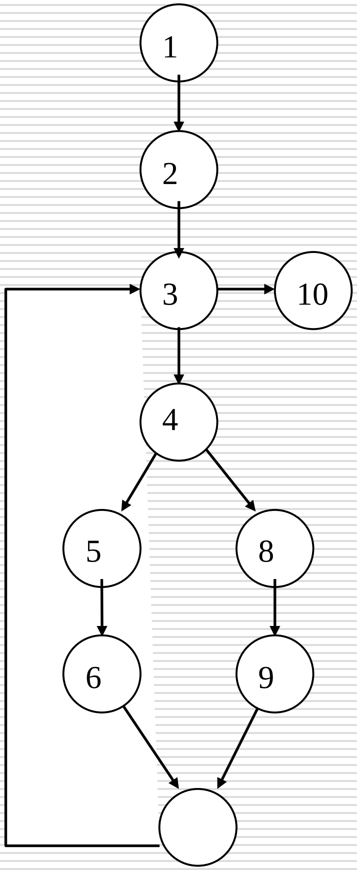


Control Flow Graph (CFG)

- Depicts *potential control flow* between program statements or instructions
- Each *simple statement* and *decision point* is represented by a *vertex*
- Each *potential branch* is represented by a *directed edge*
  - *Decision vertex* has ≥ 2 outgoing edges
- Control flow coverage criteria may be defined in terms of a CFG

Example: Control Flow Graph

```
input(x, y)
z ← 1
while y > 0 do
    if even(y)
        y ← y div 2
        z ← x * x
    else
        y ← y - 1
        z ← z * x
output(z)
```



Types of Control Flow Coverage

- *Statement coverage*: has each statement been executed by test set?
- *Branch coverage*: has each conditional branch been executed?
- *Condition coverage*: has each Boolean subexpression of branch condition evaluated both to true and false?
- *Function coverage*: has each function in the program been called?

Types of Control Flow Coverage continued

- *Linear Code Sequence and Jump (LCSAJ) coverage*: has every LCSAJ been executed?
- *Entry/exit coverage*: has every possible call and return of each function been executed?
- *Loop coverage*: has every possible loop been executed zero times, once, and more than once?
- *Path coverage*: has every complete control flow path been executed?

Question: How many paths could there be?

Relationships Among Some Control Flow Coverage Criteria

Coverage Criteria	Statement Coverage	Decision Coverage	Condition Coverage	Condition/ Decision Coverage	MC/DC	Multiple Condition Coverage
Every point of entry and exit in the program has been invoked at least once	•	•	•	•	•	•
Every statement in the program has been invoked at least once	●					
Every decision in the program has taken all possible outcomes at least once		●		•	•	•
Every condition in a decision in the program has taken all possible outcomes at least once			●	•	•	•
Every condition in a decision has been shown to independently affect that decision's outcome					●	•[8]
Every combination of condition outcomes within a decision has been invoked at least once						●

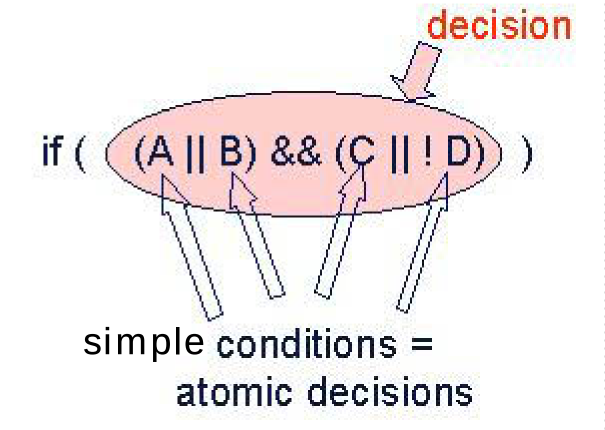
●/• : Statement on left defines/satisfied-by criterion above

From library-dspace.larc.nasa.gov/dspace/jsp/bitstream/2002/12640/1/NASA-2001-tm210876.pdf

Modified Condition/Decision Coverage (MC/DC)

For a set of tests to satisfy MC/DC:

- *Each possible outcome of each decision occurs*
- *Each possible outcome of each simple condition in a decision occurs*
- *Each entry and exit point is invoked*
- *Each simple condition in a decision is shown to independently affect the outcome of the decision*



Required by Federal Aviation Administration (FAA)*

*www.faa.gov/aircraft/air_cert/design_approvals/air_software/cast/cast_papers/media/cast-10.pdf

Example:MC/DC Coverage

```
int myFunc (bool c1, bool c2, bool c3)
{
    bool d1 = c1 or c2;
    bool d2 = d1 and c3;
    if (d2)
        return 1;
    else
        return -1;
}
```

c1	c2	d1 = c1 or c2
F	F	F
F	T	T
T	F	T
T	T	T

d1	c3	d2 = d1 and c3
F	F	F
F	T	F
T	F	F
T	T	T

Test Suite

Subset of all possible input tuples which satisfies MC/DC criteria

For Example: `{TFF, FTF, FFT, TTT} // (c1 c2 c3)`

se.inf.ethz.ch/old/teaching/2009-S/0276/slides/fiva.pdf

Criticism of MC/DC

- *Syntactic rearrangements* of decisions that preserve expression meaning can dramatically alter the number of tests needed to satisfy MC/DC coverage.

Measuring Coverage Achieved By Functional Tests*

- Common industrial practice
- If coverage is not adequate it is best *not* to just create tests to boost it
  - Such tests are often *contrived*
- Instead the *requirements spec* should be examined:
  - *Correct omissions* in spec
  - *Construct new functional tests* to exercise new cases
  - Iterate process

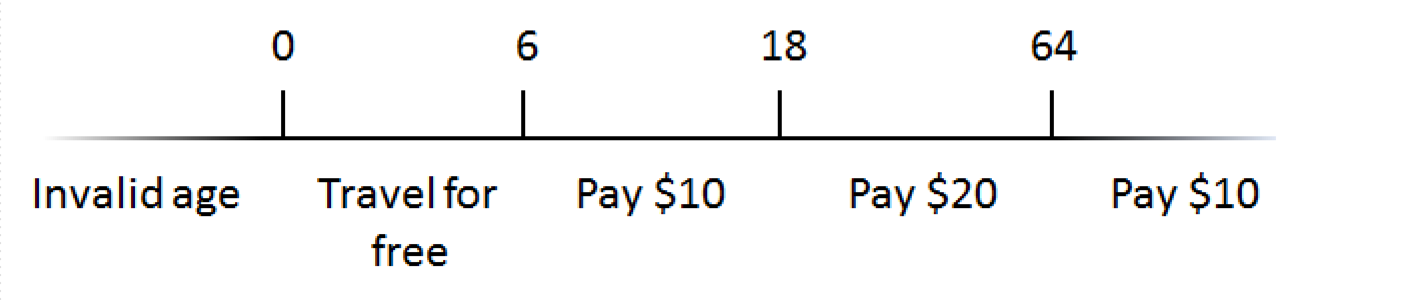
*See “The Craft of Software Testing …” by Brian Marik, Prentice Hall.

Boundary and Special Values Testing

- Involves testing with boundary or special values of variables
  - Often *mishandled* by programmers
- Examples:
  - Numbers: 0, 1, -1, MIN, MAX, MIN – 1, MAX + 1, e, π
  - Collections: empty, full
  - Strings: NULL, empty, single-element, very large
  - *Invalid* inputs
- Used *in conjunction with* functional and structural testing
- Often guided by *conditions in spec or code*, e.g.,
  - X < 0 suggests trying X = -1, 0, 1
  - p != null suggests trying null and non-null values of p
- Common boundary cases should be exercised *even if not explicit*
  - May be *neglected* by program
  - e.g., adding to full data structure, removing from empty one
- *Loops* should be iterated 0, 1, intermediate number, and large number of times
  - *Boundary-interior testing*

Example: Boundary-Value Testing [Nupponen]

- Consider ticketing system with rates dependent on age:



- Decision boundaries are at ages 0, 6, 18, and 64
- Each should be tested with values <, =, > boundary

www.conformiq.com/2013/12/equivalence-class-partitioning-and-boundary-value-analysis-as-black-box-test-design-heuristics/

BOUNDARY VALUE ANALYSIS (BVA)

AGE Enter Age *Accepts value 18 to 56

BOUNDARY VALUE ANALYSIS		
Invalid (min -1)	**Valid (min, +min, -max, max)**	**Invalid (max +1)**
17	18, 19, 55, 56	57

Name Enter Name *Accepts characters length (6 - 12)

BOUNDARY VALUE ANALYSIS		
Invalid (min -1)	**Valid (min, +min, -max, max)**	**Invalid (max +1)**
5 characters	6, 7, 11, 12 characters	13 characters

Example: Boundary Conditions in Requirements Specifications

- "3.I.1. Automatic dilutions in the range of 1:2 to 1:500 shall be performable on the system in increments of 2, 5, 10, 20, 100, 200, and 500."
  - Try dilutions from 1:1 to 1:501 at each increment exactly and ±1
- "5.d.1(g) Replicates - The scheduler may receive a worklist entry that specifies up to a maximum of twenty (20) replicates are to be performed on the test/sample pair."
  - Try entries with 0, 1, 20, and 21 replicates

Load Testing

- Involves “stressing” software with *high loads*, e.g.,
  - Large input files
  - Complex inputs
  - Many users
  - Many service requests
  - High network traffic
  - Rapid event-sequences
  - Short deadlines

Interaction Testing

- Involves testing interactions between variables, objects, events, statements, or components
- May reveal defects not revealed by simple coverage
- Number of possible interactions *grows rapidly* with the number of elements involved
  - *It may be practical to test only 2-way or 3-way interactions*

Example: Pairwise Test Configuration

- Exercises all *2-way interactions* between configuration parameters

Test Case	OS	CPU	Protocol
1	Windows	Intel	IPv4
2	Windows	AMD	IPv6
3	Unix	Intel	IPv6
4	Unix	AMD	IPv4

R. Kuhn et al, Combinatorial Software Testing, IEEE Computer, Aug. 2009

Combinatorial Interaction Testing

Combinatorial algorithms used to construct *covering arrays* for *n-way interactions*

Example: 3-way covering array: covers all 3-way interactions of 10 binary parameters, with 13 tests; any 3 columns contain all 8 possible values of 3 parameters

Parameters

Test	1	2	3	4	5	6	7	8	9	10
1	0	0	0	0	0	0	0	0	0	0
2	1	1	1	1	1	1	1	1	1	1
3	1	1	1	0	1	0	0	0	0	1
4	1	0	1	1	0	1	0	1	0	0
5	1	0	0	0	1	1	1	0	0	0
6	0	1	1	0	0	1	0	0	1	0
7	0	0	1	0	1	0	1	1	1	0
8	1	1	0	1	0	0	1	0	1	0
9	0	0	0	1	1	1	0	0	1	1
10	0	0	1	1	0	0	1	0	0	1
11	0	1	0	1	1	0	0	1	0	0
12	1	0	0	0	0	0	0	1	1	1
13	0	1	0	0	0	1	1	1	0	1

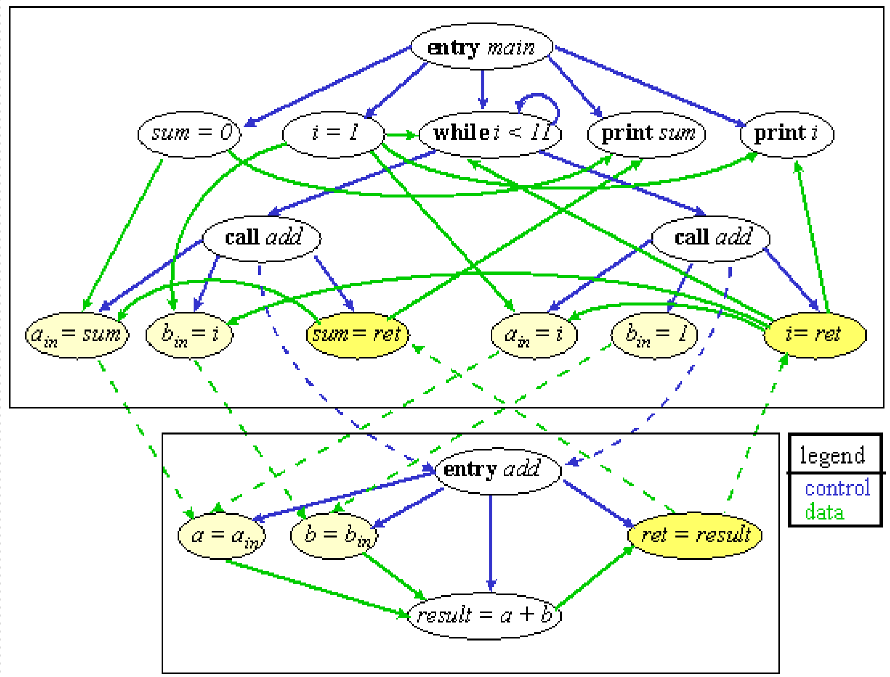
http://csrc.nist.gov/groups/SNS/acts/documents/kuhn-kacker-lei-hunter09.pdf

Data Flow and Dependence Testing

- *Data flow analysis* and *dependence analysis* find interactions between *program elements*
  - *Data dependences* (inverse data flows)
  - *Control dependences*
- Represented by *program dependence graph* (PDG)
- A number of testing techniques exercise specific dependence patterns

Example: Program Dependence Graph

```
void main() {
    int i = 1;
    int sum = 0;
    while (i<11) {
        sum = add(sum, i);
        i = add(i, 1);
    }
    printf("sum = %d\n", sum);
    printf("i = %d\n", i);
}

static int add(int a, int b) {
    return(a+b);
}
```



Example: All-Uses Data Flow Testing Criterion [Rapps & Weyuker]

- Execute a *definition-clear subpath* from each *variable definition* to each *use* it reaches and each *successor node* of the use
- Note: some apparent data flows may be *infeasible* (non-executable)
  - In general, feasibility *cannot* be determined automatically (Why?)

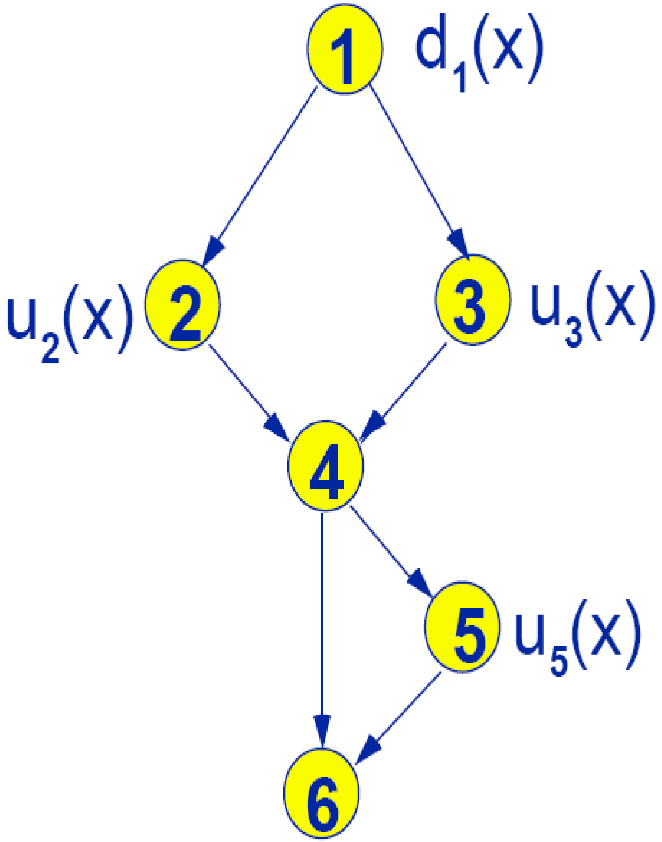
All-Uses

Requires:

- $d_1(x)$ to $u_2(x)$
- $d_1(x)$ to $u_3(x)$
- $d_1(x)$ to $u_5(x)$

Satisfactory Paths:

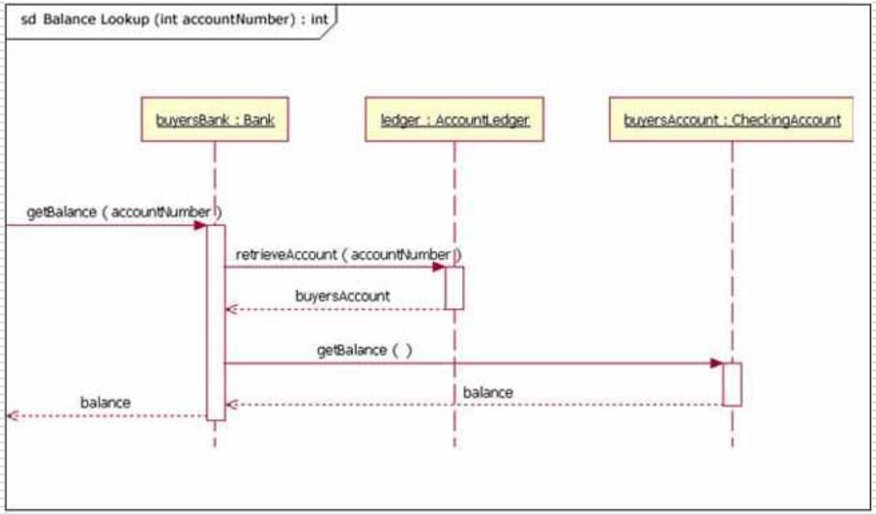
- 1, 2, 4, 5, 6
- 1, 3, 4, 6



From www.cs.txstate.edu/~rp31/slidesSQ/07-dataFlowCovR.pdf

Object-Oriented Interaction Testing

- Exercises *object-interaction scenarios*



- *Mock objects* may be used to check behavior of object under test

Sequence diagram from www.ibm.com/developerworks/rational/library/3101.html

Mock Objects

- A *mock object* substitutes for an object invoked by unit under test:
  - It is *simpler* than the real object.
    - It allows you to *set up private state* for testing.
    - Methods may store values in fields, check assertions, or do nothing.
- Mock frameworks permit specification of *expectations*, e.g., of
  - Which *methods* of a mock will be invoked and in what order
  - What *parameters* will be passed
  - What *values* will be returned

Mock Objects continued

- The mock *verifies* that its expectations are satisfied by the tests
- Mocks do *behavioral verification*, not just state verification

Example Test with jMock Object and Behavior Verification

```
public void testFillingRemovesInventoryIfInStock() {
  //setup - data
  Order order = new Order(TALISKER, 50); // Whiskey!
  Mock warehouseMock = new Mock(Warehouse.class);
  //setup - expectations
  warehouseMock.expects(once()).method("hasInventory")
    .with(eq(TALISKER),eq(50))
    .will(returnValue(true));
  warehouseMock.expects(once()).method("remove")
    .with(eq(TALISKER), eq(50))
    .after("hasInventory");
  //exercise (fill order from mock warehouse)
  order.fill((Warehouse) warehouseMock.proxy());
  //verify
  warehouseMock.verify();
  assertTrue(order.isFilled());
}
```

martinfowler.com/articles/mocksArentStubs.html#TestsWithMockObjects

When to Use Mock Objects [Allen]

- The real object *does not yet exist*
- It has *nondeterministic behavior*
- It is *difficult to set up*
- It has behavior that is *hard to trigger*
- It is *slow*
- It is a *user interface*
- It uses a *callback*

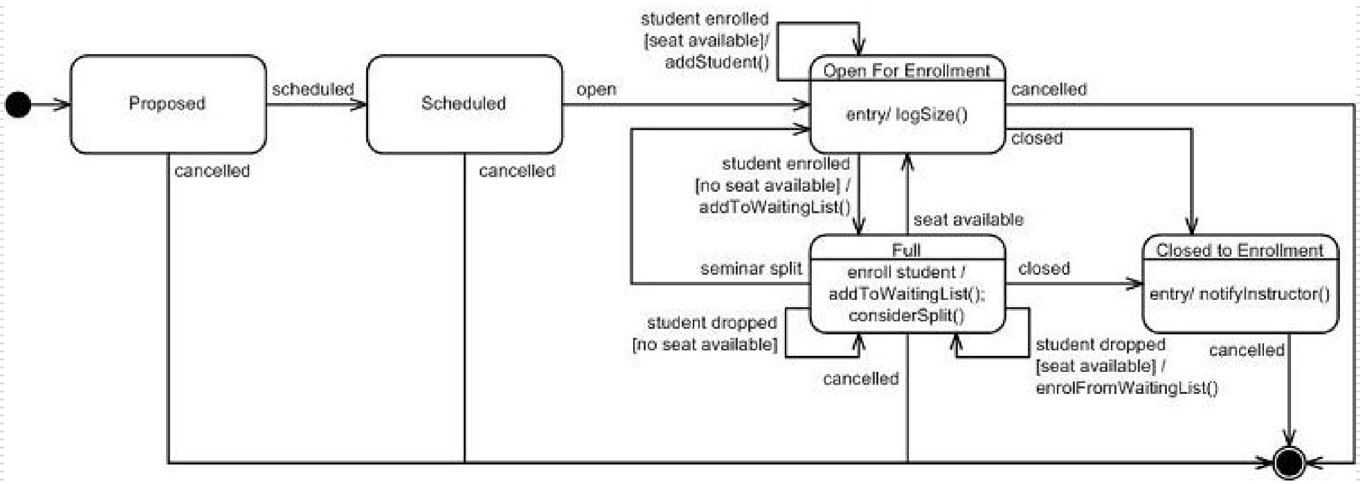
K.S. Allen, odetocode.com/blogs/scott/archive/2008/05/01/mocks-its-a-question-of-when.aspx

Model-Based Testing

- Popular name for any type of testing guided by a *behavioral model*, e.g.,
  - *State machine model*
  - *Activity diagram*
  - *Data flow diagram*
  - *Dependence graph*
  - *Markov model*
  - *Simulation model*
- Limitation: *model omissions*

State-Based Testing

- Various testing criteria are based on *finite-state machine models*, e.g.,
  - *State coverage*
  - *Transition coverage*

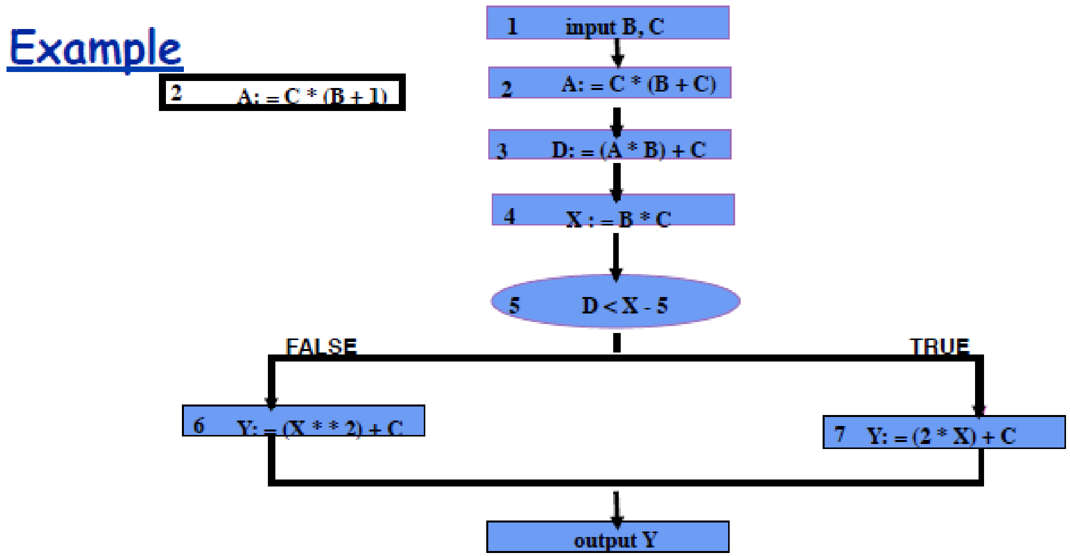


From /www.agilemodeling.com/artifacts/stateMachineDiagram.htm

Fault-Based Testing

- Involves selecting test cases to reveal a *specific kind* of programming fault, e.g.,
  - *Incorrect choices of arithmetic, relational, or logical operators*
  - *Erroneous variable substitutions*
  - *Incorrect constants*
- Test data is selected to *distinguish* supposed fault from supposed correct code
  - E.g., (a = 1, b = 0) distinguishes (a && b) from (a || b)

Example: Fault-Based Testing



	module	t. c.		exp	*a*	*d*	*x*	*d < x - 5*	*y*	*outpu*
		b	c							
B+C	faulty	1	-3	-2	6	3	-3	F	6	6
B+1	correct	1	-3	2	-6	-9	-3	T	-9	-9

ORIGINATION, COMP. TRANSFER (NODE 2)
DATA DEPENDENCE TRANSFER (NODES 3, 5)
CONTROL DEPENDENCE TRANSFER (NODE 6)
FAILURE AT NODE 8

laser.cs.umass.edu/courses/cs521-621.Fall10/documents/09-FaultBasedTesting-6.pdf

Mutation Testing

- Small changes called *mutations* are automatically *injected* into a program, one at a time, creating *mutant* versions.
- Then tests are run.
- A mutant is *killed* if it produces *different output* than the original for some test.
- Otherwise, the mutant *lived*.
- The *quality* of the tests can be gauged from the *percentage of mutations killed*.

Random Testing (Fuzz Testing)

- Test data *generated pseudo-randomly*
  - Variant: *method call sequences* selected randomly
- Test generation is *inexpensive* and *unbiased*
- In studies, often reveals many defects
- *Unlikely* to achieve good coverage
  - Why?
- *Not* a good approximation to operational testing without a realistic simulation model

Directed Random Testing

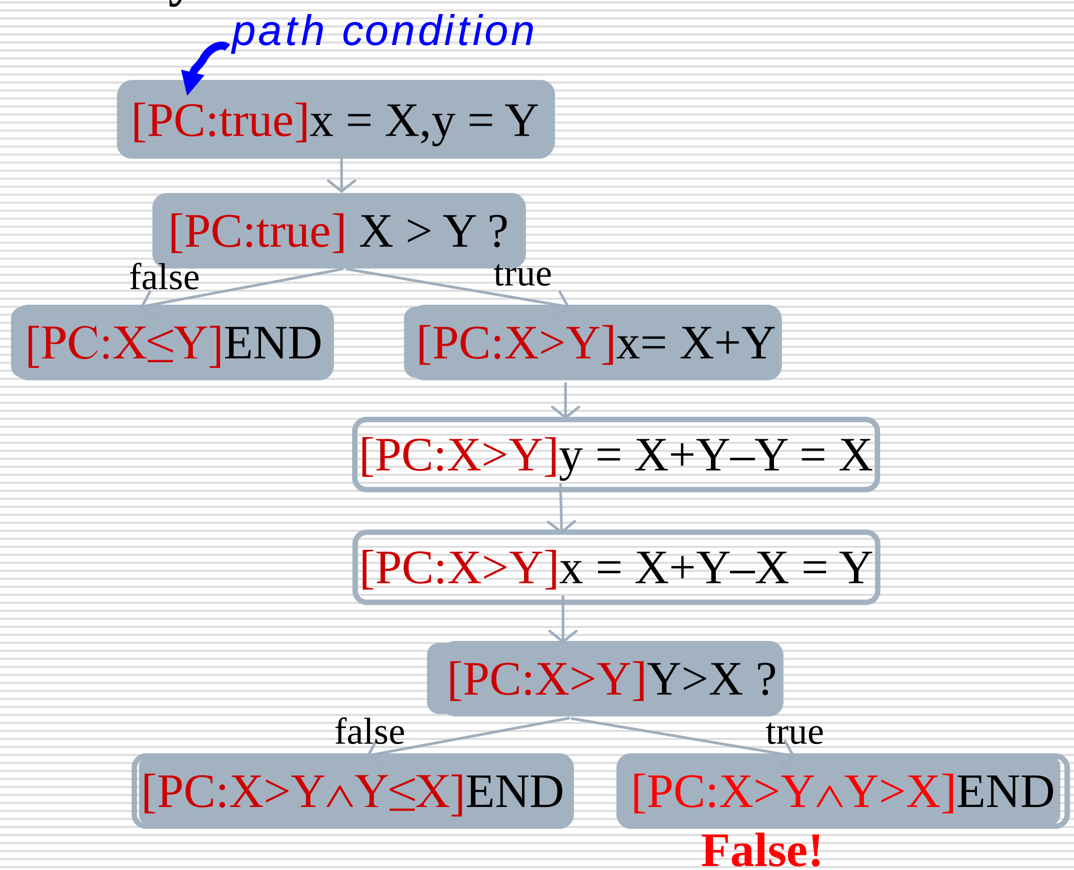
- Seeks to enhance coverage by executing alternative *control paths*
- Employs *symbolic execution* to extract *path conditions* involving input values
  - e.g., <x0 >= y0, 2*x0 == x0 + 10>
- Uses automatic *constraint solver* to generate test data
  - Computationally *expensive*
  - *Solver* may *fail*
  - See Pasareanu's slides

Example: Symbolic Execution [Parsareanu]

Code that swaps 2 integers:

```
int x, y;

if (x > y) {

  x = x + y;

  y = x – y;

  x = x – y;

  if (x > y)

    assert false;

}
```

Symbolic Execution Tree:



Solve path conditions → test inputs

Missing Cases

- An important case may be *neglected* in a specification, program, or both
- If a case is neglected by a program but is addressed in its specification, this can be revealed by functional testing
- Techniques for revealing omissions from *both* spec and program:
  - *Boundary/special values testing*
  - *Independent testing by application expert*
  - *Random testing*
  - *Operational (field) testing*
  - *Mining code base to discover programming rules and rule violations*

Choosing Among Synthetic Testing Techniques

- Diversity of techniques highlights the *variety* of possible faults
- In choosing techniques, developers must often rely on *judgment and experience*
- Best choice(s) may be *context-specific*
- Companies should *collect and analyze data* about the *effectiveness* of different techniques for revealing faults
  - *Causal inference methods* needed to address potential study biases

Test Case Design & Review

- Functional testing is *creative*
- Benefits are highly dependent on *expertise*
- Ideally, testing criteria should be identified and specified:
  - *During requirements analysis and specification*
  - *During architectural design*
  - *As each component is designed*
  - *During design and code reviews*
- During specification, design, and code reviews, relevant *test cases should also be reviewed*:
  - Exploits *team expertise*
  - Permits management *oversight*
  - Permits testing *priorities* to be set
  - *Educates* inexperienced personnel

Regression Testing

- It is standard practice to *reuse* self-checking test cases
  - *Reduces costs* for analysis, test generation, and test evaluation
- Some tests may be *omitted*:
  - *Unnecessary*, *redundant* or *similar* tests
  - Identified by analyzing *code changes* and/or *execution profiles* of tests
  - May be done *automatically*
    - Regression *test suite reduction or filtering*
      - e.g., *greedy coverage maximization*
    - Test case *prioritization*

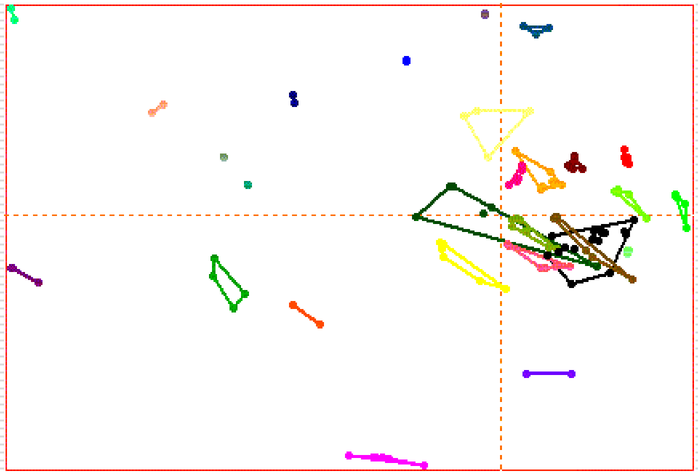
Regression Testing cont.

- *Additional tests* may be needed
  - To cover *new features or code*
  - To *"refresh"* test suite
  - Underutilized option: *capturing and reusing end-user inputs*
    - Issue: cost of checking complex outputs
- *Risks* of reusing test cases too often:
  - Sample becomes *biased* due to fixes and developer learning
  - Can *perpetuate misleading results**

*Podgurski, Andy, and Elaine J. Weyuker. "Re-estimation of software reliability after maintenance." *Proc. of 19th Intl. Conf. on Software Eng*. ACM, 1997.

Example of Test Suite Reduction: Cluster Filtering

- Tests clustered based on execution profiles
- 1+ representative tests selected from each cluster
- Ensures different behaviors are exercised



Multidimensional scaling display of GCC function-call profiles (dots) and clusters

D. Leon and A. Podgurski, "A comparison of coverage-based and distribution-based techniques for filtering and prioritizing test cases," *14th Intl Symp on Software Reliability Engineering, 2003.*

Tool Support in Testing

- Code analysis
  - e.g., control flow analysis, data flow analysis
- Test generation
  - *No general method* exists for satisfying testing criteria
  - *Restricted* techniques do exist, getting better
- Test case management and bookkeeping
- Test scripting
- Support-code generation, mocking
- Code-coverage measurement
- Capture/replay of executions
- GUI testing
- Memory error detection