Object-Oriented Software Engineering

Practical Software Development using UML and Java

Chapter 10:

Testing and Inspecting to Ensure High Quality

Lecture 3

www.lloseng.com

Testing

Functional testing

- When should we test?
  - Separate phase for testing? OR
  - At the end of every phase? OR
  - During each phase, simultaneously with all development and maintenance activities

Terminology

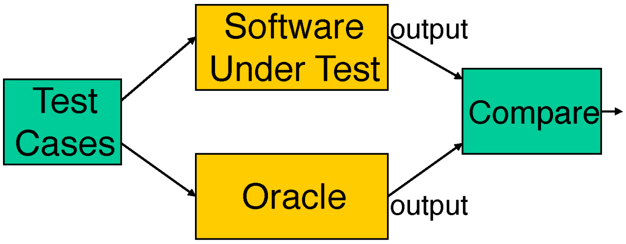
- Program - Sort
- Specification
  - Input: p - array of n integers, n > 0
  - Output: q - array of n integers such that
    - q[0] ≤ q[1] ≤ … ≤ q[n]
    - Elements in q are a permutation of elements in p, which are unchanged
  - Description of requirements of a program

Tests

- Test input (Test case)
  - A set of values given as input to a program
  - Includes environment variables
  - {2, 3, 6, 5, 4}
- Test set
  - A set of test cases
  - { {0}, {9, 8, 7, 6, 5}, {1, 3, 4, 5}, {2, 1, 2, 3} }

Oracle

- Function that determines whether or not the results of executing a program under test is as per the program's specifications



Test Cases

Software Under Test

Oracle

output

output

Compare

Failure?
Success?

Problems
- Correctness of Oracle
- Correctness of specs
- Generation of Oracle
- Need more formal specs

Correctness

- Program Correctness
  - A program P is considered with respect to a specification S, if and only if:
    - For each valid input, the output of P is in accordance with the specification S
- What if the specifications are themselves incorrect?

Errors, defects, faults

- Often used interchangeably.
- Error:
    - Mistake made by programmer.
    - Human action that results in the software containing a fault/defect.
- Defect / Fault:
    - Manifestation of the error in a program.
    - Condition that causes the system to fail.
    - Synonymous with bug

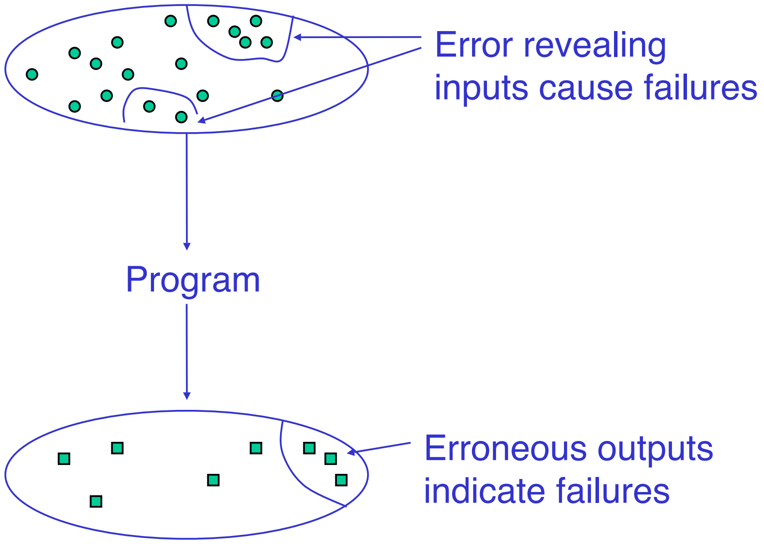
Failure

- Incorrect program behavior due to a fault in the program.
- Failure can be determined only with respect to a set of requirement specs.
- For failure to occur, the testing of the program should force the erroneous portion of the program to be executed.

—The frequency of failures measures the *reliability*

—An important design objective is to achieve a very low failure rate and hence high reliability.

Errors and failure



Inputs

Error revealing inputs cause failures

Program

Outputs

Erroneous outputs indicate failures

Testing Objectives

- Testing is a process of executing a program with the intent of finding an error.
- A good test is one that has a high probability of finding an as yet undiscovered error.
- A successful test is one that uncovers an as yet undiscovered error.

The objective is to design tests that systematically uncover different classes of errors and do so with a minimum amount of time and effort.

Secondary benefits include

- Demonstrate that software functions appear to be working according to specification
- That performance requirements appear to have been met.
- Data collected during testing provides a good indication of software reliability and some indication of software quality.

Testing cannot show the absence of defects,
it can only show that software defects are present.

Test Case Design

- Designing good test case is often as difficult and work intensive as coding the original program

  …this is why most of us don't do it, but we should
- Black box testing -- testing that code conforms to a design
- White box testing -- testing that code conforms to its specification

---

Execution-based testing

- Execution-based testing is a process of **inferring** certain behavioral properties of a product, based, in part, on the results of executing the product in a **known environment** with **selected inputs**.
- Depends on environment and inputs
- How well do we know the environment?
- How much control do we have over test inputs?
  - Real time systems

Reliability

- Probability of failure-free operation of a product for a given time duration
- How often does the product fail?
  - mean time between failures
- How bad are the effects?
- How long does it take to repair it?
  - mean time to repair
- Failure behavior controlled by
  - Number of faults
  - Operational profile of execution

Robustness

- How well does the product behave with
  - range of operating conditions
  - possibility of unacceptable results with valid input?
  - possibility of unacceptable results with invalid input?
- Should not crash even when not used under permissible conditions

Performance

- To what extent does the product meet its requirements with regard to
  - response time
  - space requirements
- What if there are too many clients/ processes, etc...

Levels of testing

- Unit testing
- Integration testing
- System testing
- Acceptance testing

Functional testing

Learning objectives

What is functional testing?

How to perform functional testing?

What are clues, test requirements, and test specifications?

How to generate test inputs?

What are equivalence partitioning, boundary value testing, domain testing, state testing, and decision table testing?

What is functional testing?

When test inputs are generated using program specifications, we say that we are doing functional testing.

Functional testing tests how well a program meets the functionality requirements.

The methodology

The derivation of test inputs is based on program specifications.

Clues are obtained from the specifications.

Clues lead to test requirements.

Test requirements lead to test specifications.

Test specifications are then used to actually execute the program under test.

Specifications-continued

Two types of pre-conditions are considered:

- Validated: those that are required to be validated by the program under test and an error action is required to be performed if the condition is not true.
- Assumed: those that are assumed to be true and not checked by the program under test.

Preconditions for sort

Preconditions for sort

Validated:

N>0

On failure return -1; sorting considered unsuccessful.

Assumed:

The input sequence contains N integers.
The output area has space for at least N integers.

Post-conditions

A post-condition specifies a property of the output of a program.

The general format of a post-condition is:

if condition **then** effect-1 {**else** effect-2}

Example:

For the sort program a post-condition is:

if N>0 **then** {the output sequence has the same elements as in the input sequence and in ascending order.}

Incompleteness of specifications

Specifications may be incomplete or ambiguous.

Example post-condition:

if user places cursor on the name field **then** read a string

This post-condition does not specify any limit on the length of the input string hence is incomplete.

Ambiguous specifications

It also does not make it clear as to

> whether a string should be input only after the user has placed the cursor on the name field and clicked the mouse or simply placed the cursor on the name field.

and hence is ambiguous.

Clues: summary

Clues are:

- Pre-conditions
- Post-conditions
- Variables,
  - e.g. A is a length implying thereby that its value cannot be negative.
- Operations,
  - e.g. “search a list of names” or “find the average of total scores”
- Definitions,
  - e.g. “filename(name) is a name with no spaces.”

Clues

- Ideally variables, operations and definitions should be a part of at least one pre- or post-condition.
- However, this may not be the case as specifications are not always written formally.

Test requirements checklist

- Obtaining clues and deriving test requirements can be tedious.
- Make a checklist of cluse to keep it from overwhelming you.
- Derive test requirements from the clues checklist.

White box testing

- Testing control structures of a procedural design.
- Can derive test cases to ensure:
  1. all independent paths are exercised at least once.
  2. all logical decisions are exercised for both true and false paths.
  3. all loops are executed at their boundaries and within operational bounds.
  4. all internal data structures are exercised to ensure validity
- Why do white box testing when black box testing is used to test conformance to requirements?
  - Logic errors and incorrect assumptions most likely to be made when coding for "special cases". Need to ensure these execution paths are tested.
  - May find assumptions about execution paths incorrect, and so make design errors. White box testing can find these errors.
  - Typographical errors are random. Just as likely to be on an obscure logical path as on a mainstream path.

"Bugs lurk in corners and congregate at boundaries"

White box testing

- Control flow testing
  - **Conditions Testing --** Condition testing aims to exercise all logical conditions in a program module. Focus on testing each condition in the program.Strategies proposed include:
    - Branch testing - execute every branch at least once.
    - Domain Testing - uses three or four tests for every relational operator.
    - Branch and relational operator testing - uses condition constraints
  - **Loop testing**
    - **Simple Loops** of size n:
      - Skip loop entirely
      - Only one pass through loop; then two passes
      - m passes through loop where m < n.
      - (n-1), n, and (n+1) passes through the loop.
    - **Nested Loops**-
      - Start with inner loop. Set all other loops to minimum values.
      - Conduct simple loop testing on inner loop.
      - Work outwards
      - Continue until all loops tested.

Black box testing

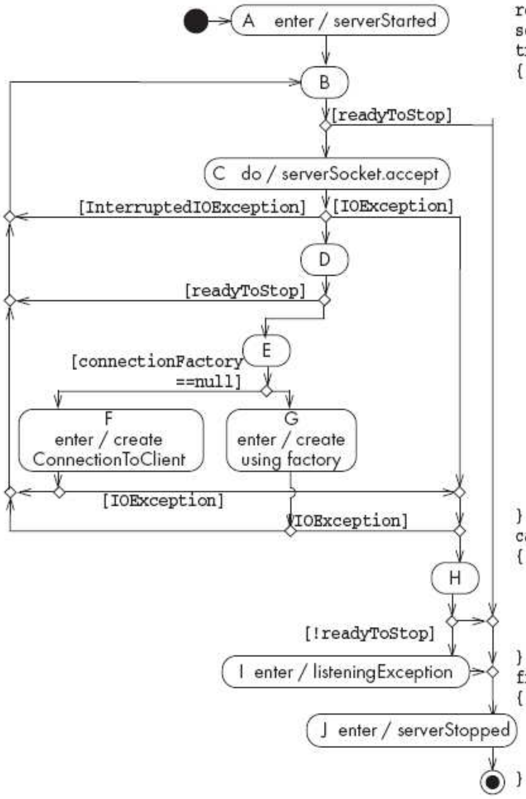
- Focus on functional requirements Attempts to find:
  - incorrect or missing functions
  - interface errors
  - errors in data structures
  - performance errors
  - initialisation and termination errors.•
- **Equivalence Partitioning --** Divide the input domain into classes of data for which test cases can be generated. Attempting to uncover classes of errors. Based on equivalence classes for input conditions. An equivalence class represents a set of valid or invalid states. An input condition is either a specific numeric value, range of values, a set of related values, or a boolean condition.
- **Boundary Value Analysis --** Large number of errors tend to occur at boundaries of the input domain. BVA leads to selection of test cases that exercise boundary values.. Rather than select any element in an equivalence class, select those at the ''edge' of the class.

Flow graph for white-box testing

To help the programmer to systematically test the code

- Each branch in the code (such as if and while statements) creates a node in the graph
- The testing strategy has to reach a targeted coverage of statements and branches; the objective can be to:
  - —cover all possible paths (often infeasible)
  - —cover all possible edges (most efficient)
  - —cover all possible nodes (simpler)

Flow graph for white-box testing



```
readyToStop= false;
serverStarted();     // A
try
{
  while(!readyToStop)  // B
  {
    try
    {
      Socket clientSocket = serverSocket.accept();  // C
      synchronized(this)
      {
        if (!readyToStop) // D
        {
          if (connectionFactory == null) {  // E
            new ConnectionToClient(         // F
              this.clientThreadGroup, clientSocket, this);
          } else {
            connectionFactory.createConnection(    // G
              this.clientThreadGroup, clientSocket, this);
          }
        }
      }
    }
    catch (InterruptedIOException exception) { }
  }
}
catch (IOException exception)
{
  if (!readyToStop)   // H
  {
    listeningException(exception);  // I
  }
}
finally
{
  readyToStop = true;  // J
  connectionListener = null;
  serverStopped();
}
```

Black-box testing

Testers provide the system with inputs and observe the outputs

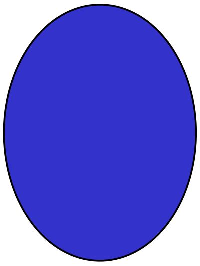
- They can see none of:
  - The source code
  - The internal data
  - Any of the design documentation describing the system's internals

Equivalence partitioning

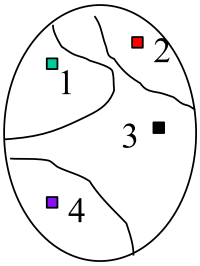
- Why?
  - Input domain is usually too large (e.g. infinite) for exhaustive testing.
- How?
  - Partition into a finite number of sub-domains and select test inputs.
  - Each sub-domain is an equivalence class and serves as a source of at least one test input.

Equivalence partitioning

Input domain

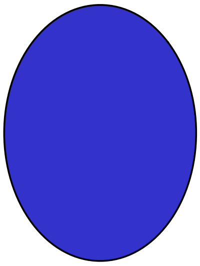


Input domain partitioned into four sub-domains.



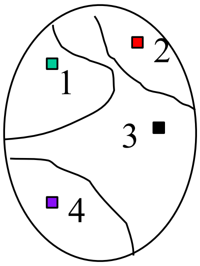
Equivalence partitioning

Input domain



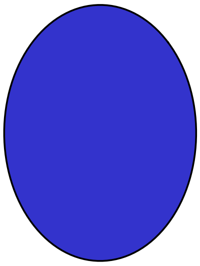
Too many
test inputs.

Input domain
partitioned into four
sub-domains.



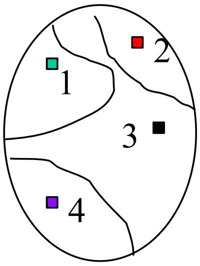
Equivalence partitioning

Input domain



Too many
test inputs.

Input domain
partitioned into four
sub-domains.



Four test inputs, one
selected from each
sub-domain.

How to partition?

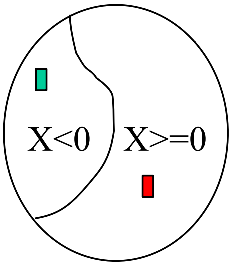
- Inputs to a program provide clues to partitioning.
- Example:
  - Suppose that program P takes an integer input X
  - For X < 0 perform task T1 and
  - for X >= 0 perform task T2.

How to partition?

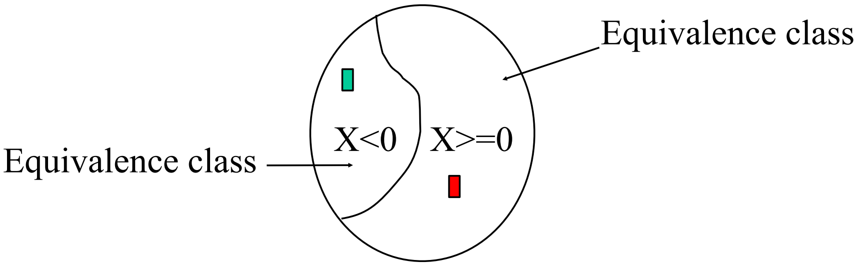
- The input domain is prohibitively large as X can assume the whole range of integer values.
- However, we expect P to behave the same way for all X < 0.
- Similarly, we expect P to perform the same way for all values of X>=0.
- We therefore partition the input domain of P into two sub-domains.

Two sub-domains

Two sub-domains



Two sub-domains



Equivalence class

Equivalence class

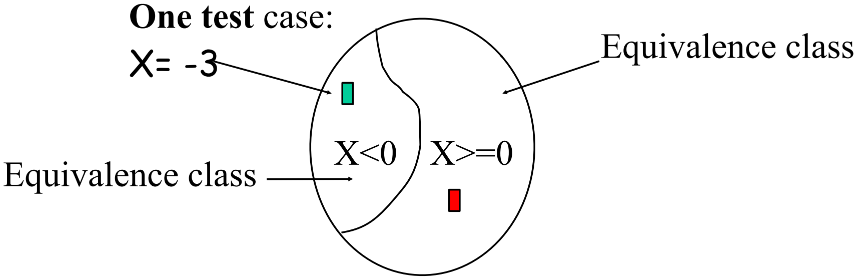
X<0

X>=0

Two sub-domains

One test case:

X= -3



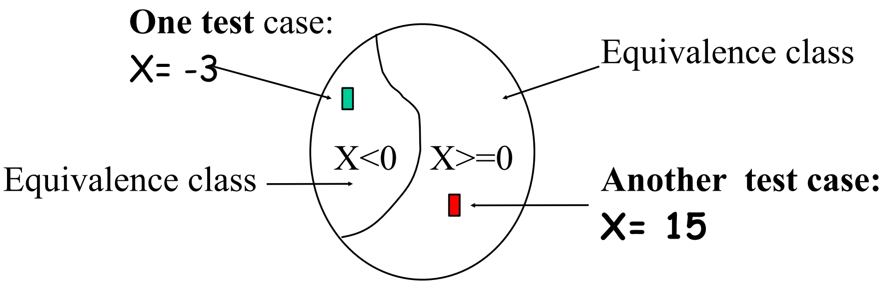
Equivalence class

Equivalence class

X<0

X>=0

Two sub-domains



One test case:
X= -3

Equivalence class

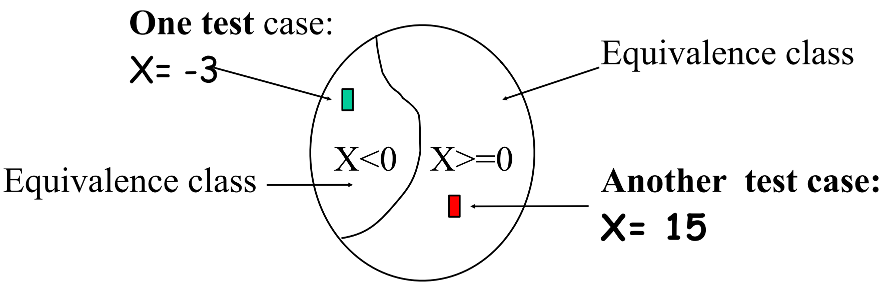
X<0

X>=0

Equivalence class

Another test case:
X= 15

Two sub-domains



One test case:
X= -3
Equivalence class
X<0
X>=0
Equivalence class
Another test case:
X= 15

All test inputs in the X<0 sub-domain are considered equivalent. The assumption is that if one test input in this sub-domain reveals an error in the program, so will the others.

This is true of the test inputs in the X>=0 sub-domain also.

Non-overlapping partitions

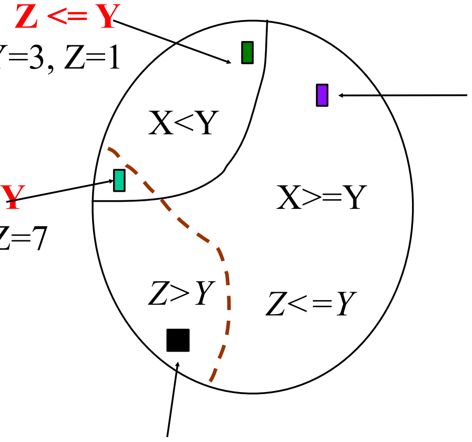
- In the previous example, the two equivalence classes are non-overlapping. In other words the two sub-domains are disjoint.
- When the sub-domains are disjoint, it is sufficient to pick one test input from each equivalence class to test the program.

- An equivalence class is considered covered when at least one test has been selected from it.
- In partition testing our goal is to cover all equivalence classes.

Overlapping partitions

- Example:
  - Suppose that program P takes three integers X, Y and Z.
  - It is known that:
    - $X < Y$
    - $Z > Y$

Overlapping partitions



X < Y, Z <= Y
X=2, Y=3, Z=1

X >= Y, Z <= Y
X=15, Y=4, Z=1

X < Y, Z > Y
X=3, Y=4, Z=7

X >= Y, Z > Y
X=15, Y=4, Z=7

Overlapping partition-test selection

- In this example, we could select 4 test cases as:
  - X=4, Y=7, Z=1 satisfies X < Y
  - X=4, Y=2, Z=1 satisfies X >= Y
  - X=1, Y=7, Z=9 satisfies Z > Y
  - X=1, Y=7, Z=2 satisfies Z <= Y
- Thus, we have one test case from each equivalence class.

Overlapping partition-test selection

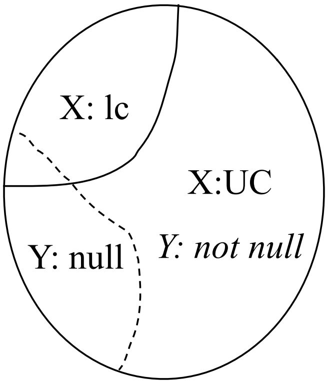
- However, we may also select only 2 test inputs and satisfy all four equivalence classes:
  - X=4, Y=7, Z=1 satisfies X<Y and Z<=Y
  - X=4, Y=2, Z=3 satisfies X>=Y and Z>Y
- Thus, we have reduced the number of test cases from 4 to 2 while covering each equivalence class.

Partitioning using non-numeric data

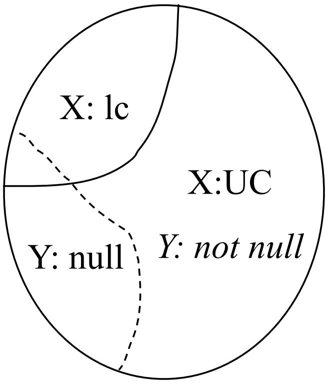
- In the previous two examples the inputs were integers. One can derive equivalence classes for other types of data also.
- Example 3:
  - Suppose that program P takes one character X and one string Y as inputs.
  - P performs task T1 for all lower case characters and T2 for upper case characters.
  - Also, it performs task T3 for the null string and T4 for all other strings.

Partitioning using non-numeric data

Partitioning using non-numeric data



Partitioning using non-numeric data



X: lc

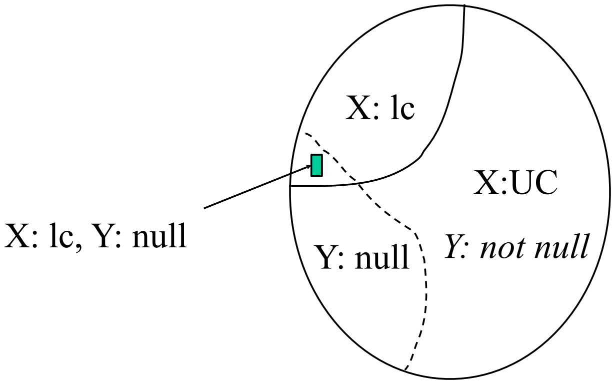
X:UC

Y: null

Y: not null

lc: Lower case character
UC: Upper case character
null: null string.

Partitioning using non-numeric data



X: lc, Y: null

X: lc

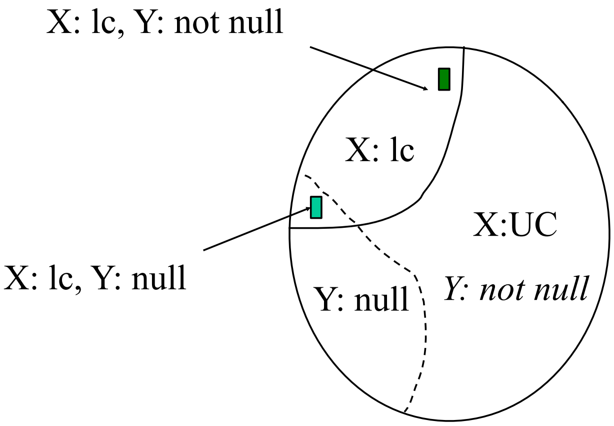
X:UC

Y: null

Y: not null

lc: Lower case character
UC: Upper case character
null: null string.

Partitioning using non-numeric data



X: lc, Y: not null

X: lc

X:UC

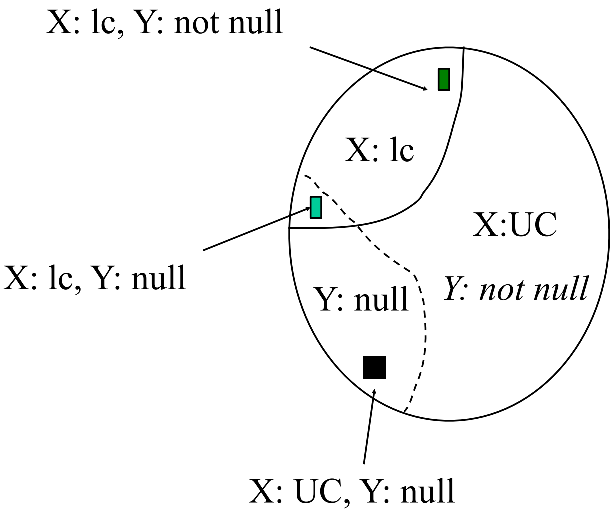
X: lc, Y: null

Y: null

Y: not null

lc: Lower case character
UC: Upper case character
null: null string.

Partitioning using non-numeric data



X: lc, Y: not null

X: lc

X:UC

X: lc, Y: null

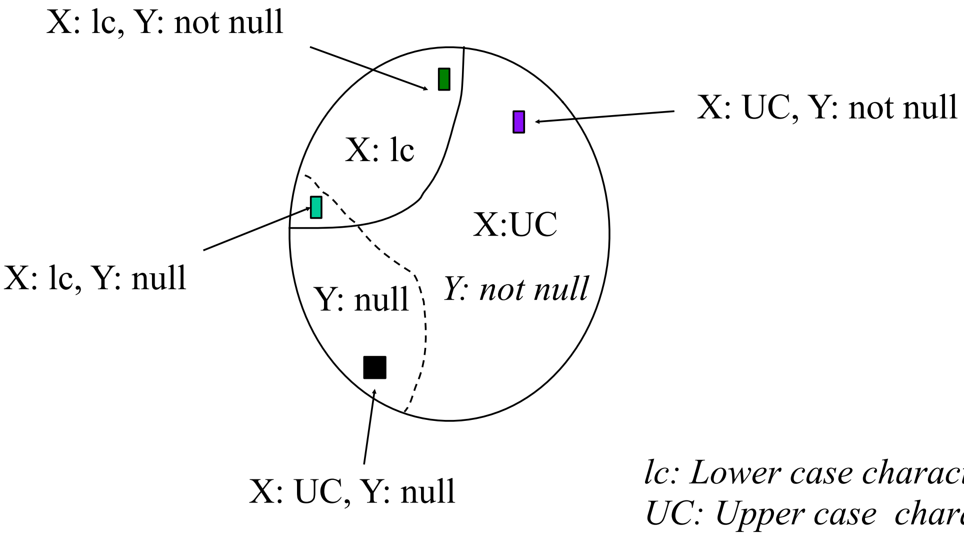
Y: null

Y: not null

X: UC, Y: null

lc: Lower case character
UC: Upper case character
null: null string.

Partitioning using non-numeric data



X: lc, Y: not null

X: UC, Y: not null

X: lc

X:UC

X: lc, Y: null

Y: null

Y: not null

X: UC, Y: null

lc: Lower case character
UC: Upper case character
null: null string.

Non-numeric data

- Once again we have overlapping partitions.
- We can select only 2 test inputs to cover all four equivalence classes. These are:
  - X: lower case, Y: null string
  - X: upper case, Y: not a null string

Guidelines for equivalence partitioning

- Input condition specifies a range: create one for the valid case and two for the invalid cases.
  - e.g. for a <= X <= b the classes are
    - A <= X <= b (valid case)
    - X < a and X > b (the invalid cases)
- Input condition specifies a value: create one for the valid value and two for incorrect values (below and above the valid value). This may not be possible for certain data types, e.g. for boolean.
- Input condition specifies a member of a set: create one for the valid value and one for the invalid (not in the set) value.

Sufficiency of partitions

- In the previous examples we derived equivalence classes based on the conditions satisfied by the input data.
- Then we selected just enough tests to cover each partition.
- Think of the advantages and disadvantages of this approach!

Boundary value analysis (BVA)

Another way to generate test cases is to look for boundary values.

Suppose a program takes an integer X as input.

In the absence of any information, we assume that X = 0 is a boundary. Inputs to the program might lie on the boundary or on either side of the boundary.

BVA: continued

This leads to 3 test inputs:

X = 0, X =- 20, and X = 14.

Note that the values -20 and 14 are on either side of the boundary and are chosen arbitrarily.

Notice that using BVA we get 3 equivalence classes. One of these three classes contains only one value (X=0), the other two are large!

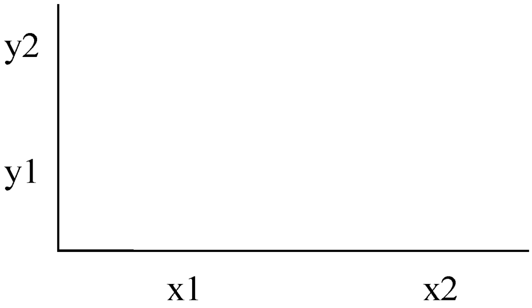
BVA

BVA

- Now suppose that a program takes two integers X and Y and that $x1 <= X <= x2$ and $y1 <= Y <= y2$.

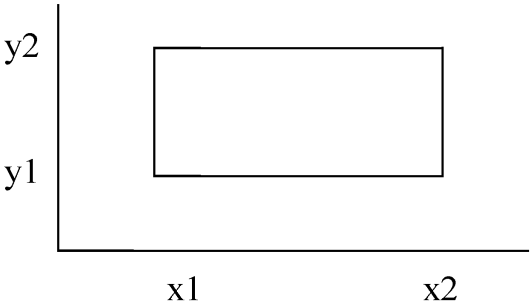
BVA

- Now suppose that a program takes two integers X and Y and that x1 <= X <= x2 and y1 <= Y <= y2.



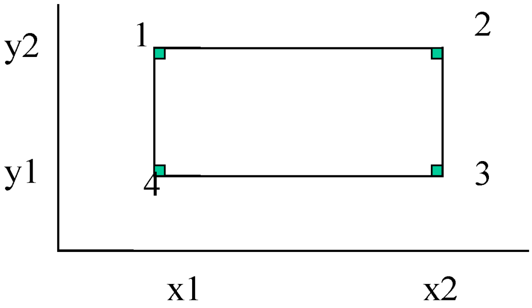
BVA

- Now suppose that a program takes two integers X and Y and that x1 <= X <= x2 and y1 <= Y <= y2.



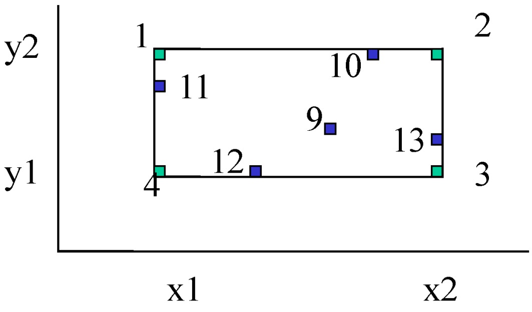
BVA

- Now suppose that a program takes two integers X and Y and that x1 <= X <= x2 and y1 <= Y <= y2.



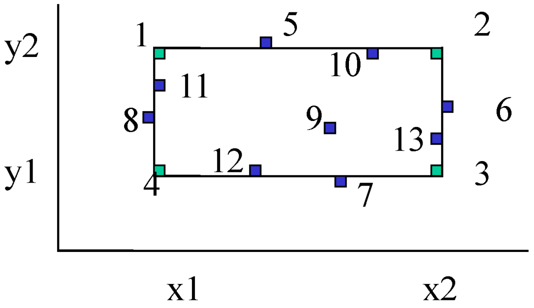
BVA

- Now suppose that a program takes two integers X and Y and that x1 <= X <= x2 and y1 <= Y <= y2.



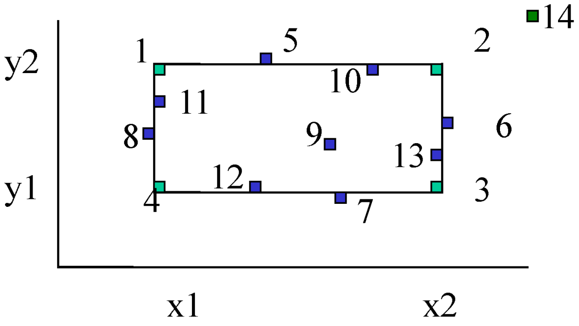
BVA

- Now suppose that a program takes two integers X and Y and that x1 <= X <= x2 and y1 <= Y <= y2.



BVA

- Now suppose that a program takes two integers X and Y and that x1 <= X <= x2 and y1 <= Y <= y2.



BVA-continued

In this case the four sides of the rectangle represent the boundary.

The heuristic for test selection in this case is:

Select one test at each corner (1, 2, 3, 4).

Select one test just outside of each of the four sides of the boundary (5, 6, 7, 8)

BVA-continued

- Select one test just inside of each of the four sides of the boundary (10, 11, 12, 13).
- Select one test case inside of the bounded region (9).
- Select one test case outside of the bounded region (14).

How many equivalence classes do we get?

BVA -continued

In the previous examples we considered only numeric data.

BVA can be done on any type of data.

For example, suppose that a program takes a string S and an integer X as inputs. The constraints on inputs are:

length(S) <= 100 and a <= X <= b

Can you derive the test cases using BVA?

BVA applied to output variables

Just as we applied BVA to input data, we can apply it to output data.

Doing so gives us equivalence classes for the output domain.

We then try to find test inputs that will cover each output equivalence class.

Summary

- Specifications, pre-conditions, and post-conditions.
- Clues, test requirements, and test specifications.
- Clues from code.
- Test requirements catalog.
- Equivalence partitioning and boundary value analysis.

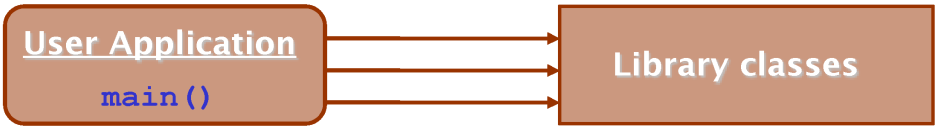
JUnit, a testing framework

- JUnit is an open-source testing framework (Gamma, Beck) that provides:
  - classes for writing Test Cases and Test Suites
  - methods for setting up an cleaning up test data (“fixtures”)
  - methods for making assertions
  - textual and graphical tools for running tests
- It simplifies and automates the task of writing repeatable unit test.

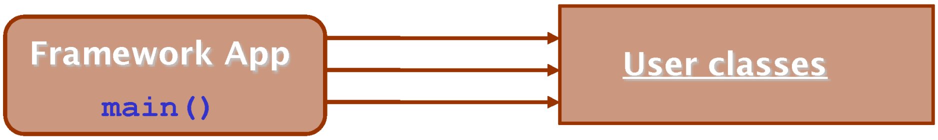
- JUnit distinguishes between failures and errors:
  - A failure is a failed assertion, i.e., an anticipated problem that you check
  - An error is a condition you didn’t check for.

Frameworks vs Libraries

- In traditional application architectures, user-code uses library functionality by invoking methods defined in library classes



- A framework reverses the usual relationship between generic and application code. Frameworks provide both generic functionality and application architecture:



- Essentially a framework says: "Don't call me, I'll call you"
  (Frameworks are much harder to write)

JUnit example

- Testing a FileReader class

```
class FileReaderTester extends TestCase {
    public FileReaderTester(String name) {
           super(name);
    }
    FileReader _input;
}
```

NB placing tests in a separate file implies a tradeoff.

- Separate the test code from the application code (+)
- No space overhead in the final application (+)
- No access to private/protected members (-)
- One more file to manage (-)

Setting up a test case

- The TestCase class provides two methods for manipulating test fixtures: `setUp` creates the objects used by the TestCase, `tearDown` removes them

```
class FileReaderTester...
   protected void setUp() {
        try {    _input = new FileReader(“data.text”);
        } catch (FileNotFoundException e)  {
                throw new RuntimeException(“Unable to open file”);}
   }
   protected void tearDown() {
        try {    _input.close();
        } catch (IOException e)  {
                throw new RuntimeException(“Error closing file”);}
   }
}
```

NB make sure to reset all static variables to their initial state in tearDown.

Writing tests

- A test file: “data.text”

  `‘Bradman 12 45 16\EOF’`

- A test

```
public void testRead() throws IOException {
    char ch = ‘&’;
    for (int i=0; i < 4; i++)
        ch = (char) _input.read();
    assert(‘d’==ch);
 }
```

- Invoking the test

```
class FileReaderTester
     public static Test suite() {
              TestSuite suite = new TestSuite();
              suite.addTest(new FileReaderTester(“testRead”));
              return suite; }
```

Running tests

- The main method

```
class FileReaderTester...
   public static void main(String[] args) {
      junit.textui.TestRunner.run(suite());
   }
```

- A run

```
.
Time 0.11

OK (1 tests)
```

Failures

- A deliberate bug:

```
public void testRead() throws IOException {
   char ch = '&';
   for (int i=0; i < 4; i++)
      ch = (char) _input.read();
   assert('#'==ch);
}
```

- Results

```
.F
Time: 0.22

!!!FAILURES!!!
Test Results:
Run: 1Failures: 1 Errors: 0
There was 1 failure:
1) FileReaderTester.testRead test.framework.AssertionFailedError
```

Failures (bis)

- A deliberate bug:

```
public void testRead() throws IOException {
   char ch = ‘&’;
   for (int i=0; i < 4; i++)
      ch = (char) _input.read();
   assertEquals(“fourth char read”,‘#’,ch);
 }
```

- Results

```
.F
Time: 0.22
!!!FAILURES!!!
Test Results:
Run: 1Failures: 1 Errors: 0
There was 1 failure:
1) FileReaderTester.testRead:
   fourth char read expected “#” but was “d”
```

Failures (bis^2)

- An error:

```
public void testRead() throws IOException {
   char ch = ‘&’;
   _input.close();
   for (int i=0; i < 4; i++)
      ch = (char) _input.read();
   assertEquals(“fourth char read”,‘#’,ch);
 }
```

- Results

```
....
Run: 1 Failures: 0 Errors: 2
There was 1 error:
1) FileReaderTester.testRead: java.io.IOException: Stream closed
```

NB always start by triggering an error, to be sure a test is actually run.

Testing Style

“The style here is to write a few lines of code, then a test that should run, or even better, to write a test that won’t run, then write the code that will make it run.”

- write unit tests that thoroughly test a single class
- write tests as you develop (even before you implement)
- write tests for every new piece of functionality

“Developers should spend 25-50% of their time developing tests.”

What to test?

- FileReader returns -1 at the end of a file

```
public void testReadEnd() throws IOException {
   char ch = ‘&’;
   for (int i=0; i < 16; i++)
      ch = (char) _input.read();
   assertEquals(-1,ch);
 }
```

- Invoking the test

```
class FileReaderTester
    public static Test suite() {
           TestSuite suite = new TestSuite();
           suite.addTest(new FileReaderTester(“testRead”));
           suite.addTest(new FileReaderTester(“testReadEnd”));
           return suite; }
```

Shortcut

- Instead of using the suite() method, write

```
public static void main (String[] args) {
  junit.textui.TestRunner(new TestSuite(FileReaderTester.class);
}
```

tests are extracted by reflection

What to test?

- Test boundary conditions

```
public void testReadBoundaries() throws IOException {
   assertEquals(“read first char”, ‘B’, _input.read());
   int ch;
   for (int i=0; i < 15; i++)
       ch =  _input.read();
   assertEquals(“read last char”, ‘6’, _input.read());
   assertEquals(“read at end”,-1,_input.read());
 }
```

What to test?

- Test special conditions

```
public void testEmptyRead() throws IOException {
     File empty = new File(“empty.text”);
     FileOutputStream out = new FileOutputStream(empty);
     out.close();
     FileReader in new FileReader(empty);
     assertEquals(-1, in.read());
 }
```

What to test?

- Test exceptions

```
public void testReadAfterClose() throws IOException {
     _input.close();
     try {
             _input.read();
             fail(“no exception for read past end”);
     } catch (IOException io) {}
 }
```

8 rules of testing

1. Make sure all tests are fully automatic and check their own results
2. A test suite is a powerful bug detector that decapitates the time it takes to find bugs
3. Run your tests frequently – every test at least once a day.
4. When you get a bug report, start by writing a unit test that exposes the bug
5. Better to write and run incomplete tests than not run complete tests
6. Think of boundary conditions under which things might go wrong and concentrate your tests there
7. Don't forget to test exceptions raised when things are expected to go wrong
8. Don't let the fear that testing can't catch all bugs stop you from writing tests that will catch most bugs

– Martin Fowler, Refactoring

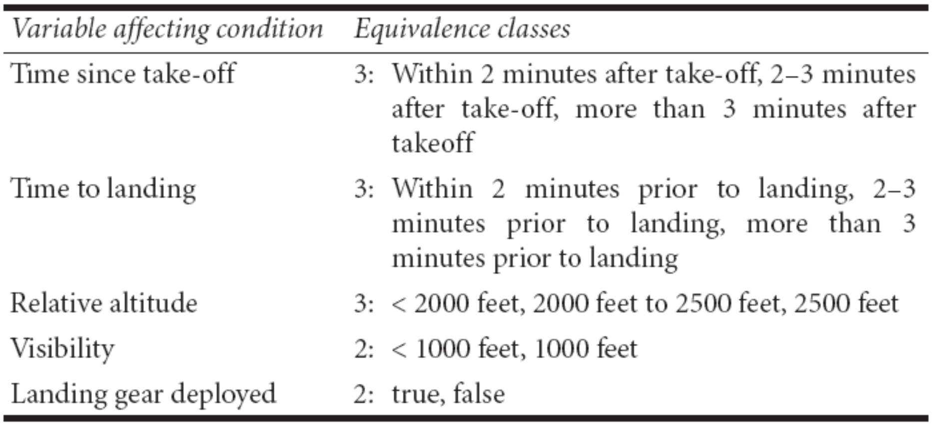
10.3 Defects in Ordinary Algorithms

Incorrect logical conditions

- *Defect*:
  - —The logical conditions that govern looping and if-then-else statements are wrongly formulated.
- *Testing strategy*:
  - —Use equivalence class and boundary testing.
  - —Consider as an input each variable used in a rule or logical condition.

Example of incorrect logical conditions defect

- The landing gear must be deployed whenever the plane is within 2 minutes from landing or takeoff, or within 2000 feet from the ground. If visibility is less than 1000 feet, then the landing gear must be deployed whenever the plane is within 3 minutes from landing or lower than 2500 feet
  - —Total number of system equivalence classes: 108

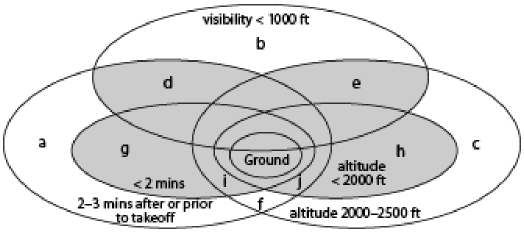


Variable affecting condition	*Equivalence classes*
Time since take-off	3: Within 2 minutes after take-off, 2–3 minutes after take-off, more than 3 minutes after takeoff
Time to landing	3: Within 2 minutes prior to landing, 2–3 minutes prior to landing, more than 3 minutes prior to landing
Relative altitude	3: < 2000 feet, 2000 feet to 2500 feet, 2500 feet
Visibility	2: < 1000 feet, 1000 feet
Landing gear deployed	2: true, false

Example of incorrect logical conditions defect

What is the hard-to-find defect in the following code?

```
if(!landingGearDeployed &&
   (min(now-takeoffTime,estLandTime-now))<
     (visibility < 1000 ? 180 :120) ||
   relativeAltitude <
     (visibility < 1000 ? 2500 :2000)
  )
{
 throw
   new LandingGearException(
}
```



Defects in Ordinary Algorithms

Performing a calculation in the wrong part of a control construct

- ***Defect***:
  - —The program performs an action when it should not, or does not perform an action when it should.
  - —Typically caused by inappropriately excluding or including the action from a loop or a if construct.
- ***Testing strategies***:
  - —Design tests that execute each loop zero times, exactly once, and more than once.
  - —Anything that could happen while looping is made to occur on the first, an intermediate, and the last iteration.

Example of performing a calculation in the wrong part of a control construct

```
while(j<maximum)
{
    k=someOperation(j);
    j++;
}
if(k==-1) signalAnError();

if (j<maximum)
    doSomething();
if (debug) printDebugMessage();
else doSomethingElse();
```

Defects in Ordinary Algorithms

Not terminating a loop or recursion

- *Defect*:
  - A loop or a recursion does not always terminate, i.e. it is ‘infinite’.
- *Testing strategies*:
  - Analyze what causes a repetitive action to be stopped.
  - Run test cases that you anticipate might not be handled correctly.

Defects in Ordinary Algorithms

Not setting up the correct preconditions for an algorithm

- *Defect*:
  - *Preconditions* state what must be true before the algorithm should be executed.
  - A defect would exist if a program proceeds to do its work, even when the preconditions are not satisfied.
- *Testing strategy*:
  - Run test cases in which each precondition is not satisfied.

Defects in Ordinary Algorithms

Not handling null conditions

- *Defect*:
  - A *null condition* is a situation where there normally are one or more data items to process, but sometimes there are none.
  - It is a defect when a program behaves abnormally when a null condition is encountered.

- *Testing strategy*:
  - Brainstorm to determine unusual conditions and run appropriate tests.

Defects in Ordinary Algorithms

Not handling singleton or non-singleton conditions

- *Defect*:
  - A *singleton condition* occurs when there is normally *more than one* of something, but sometimes there is only one.
  - A *non-singleton condition* is the inverse.
  - Defects occur when the unusual case is not properly handled.

- *Testing strategy*:
  - Brainstorm to determine unusual conditions and run appropriate tests.

Defects in Ordinary Algorithms

Off-by-one errors

- *Defect*:
  - —A program inappropriately adds or subtracts one.
  - —Or loops one too many times or one too few times.
  - —This is a particularly common type of defect.

- *Testing strategy*:
  - —Develop tests in which you verify that the program:
    - computes the correct numerical answer.
    - performs the correct number of iterations.

Example of off-by-one defect

```
for (i=1; i<arrayname.length; i++)
{
     /* do something */
}
```

Use `Iterators` to help eliminate these defects

```
while (iterator.hasNext())
{
    anOperation(++val);
}
```

Defects in Ordinary Algorithms

Operator precedence errors

- *Defect*:
  - —An operator precedence error occurs when a programmer omits needed parentheses, or puts parentheses in the wrong place.
  - —Operator precedence errors are often extremely obvious...
    - but can occasionally lie hidden until special conditions arise.
  - —E.g. If x*y+z should be x*(y+z) this would be hidden if z was normally zero.
- *Testing*:
  - —In software that computes formulae, run tests that anticipate such defects.

Defects in Ordinary Algorithms

Use of inappropriate standard algorithms

- *Defect*:
  - —An inappropriate standard algorithm is one that is unnecessarily inefficient or has some other property that is widely recognized as being bad.
- *Testing strategies*:
  - —The tester has to know the properties of algorithms and design tests that will determine whether any undesirable algorithms have been implemented.

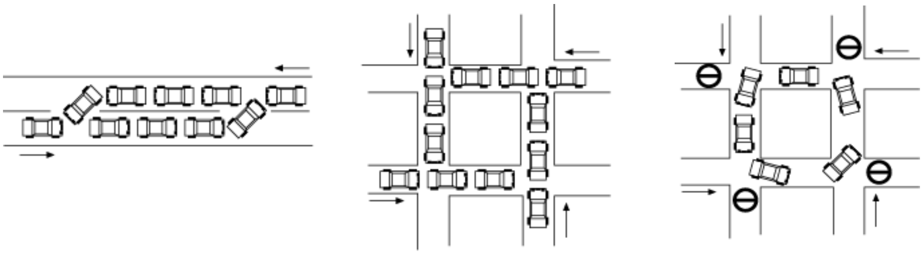
Example of inappropriate standard algorithms

- An inefficient sort algorithm
  - —The most classical ‘bad’ choice of algorithm is sorting using a so-called ‘bubble sort’
- An inefficient search algorithm
  - —Ensure that the search time does not increase unacceptably as the list gets longer
  - —Check that the position of the searched item does not have a noticeable impact on search time.
- A non-stable sort
- A search or sort that is case sensitive when it should not be, or vice versa

10.5 Defects in Timing and Co-ordination

Deadlock and livelock

- *Defects*:
  - —A deadlock is a situation where two or more threads are stopped, waiting for each other to do something.
    - The system is hung
  - —Livelock is similar, but now the system can do some computations, but can never get out of some states.

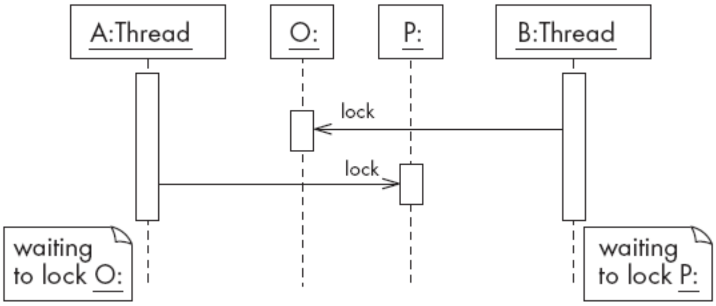


Defects in Timing and Co-ordination

Deadlock and livelock

- ***Testing strategies***:
  - Deadlocks and livelocks occur due to unusual combinations of conditions that are hard to anticipate or reproduce.
  - It is often most effective to use *inspection* to detect such defects, rather than testing alone.
  - However, when testing:
    - Vary the time consumption of different threads.
    - Run a large number of threads concurrently.
    - Deliberately deny resources to one or more threads.

Example of deadlock

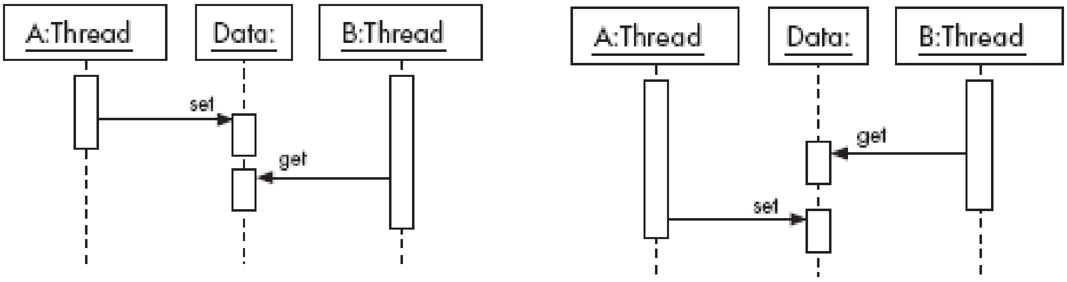


Defects in Timing and Co-ordination

Critical races

- *Defects*:
    - One thread experiences a failure because another thread interferes with the 'normal' sequence of events.
- *Testing strategies*:
    - It is particularly hard to test for critical races using black box testing alone.
    - One possible, although invasive, strategy is to deliberately slow down one of the threads.
    - Use inspection.

Example of critical race



a) Normal

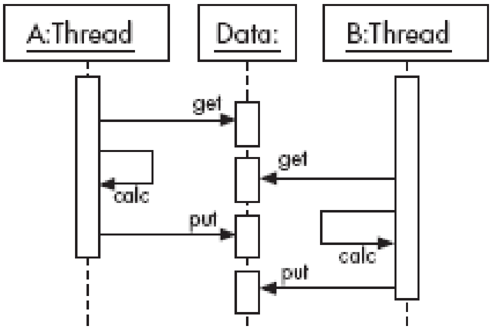
b) Abnormal due to delay in thread A

Semaphore and synchronization

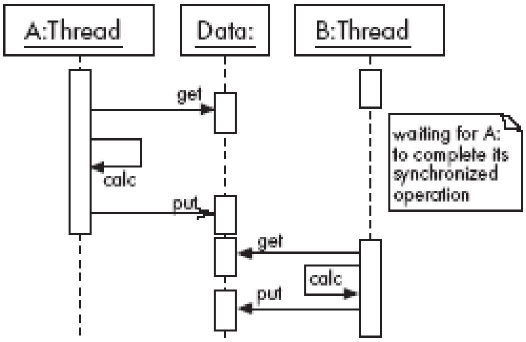
Critical races can be prevented by *locking* data so that they cannot be accessed by other threads when they are not ready

- One widely used locking mechanism is called a *semaphore*.
- In Java, the `synchronized` keyword can be used.
    - —It ensures that no other thread can access an object until the synchronized method terminates.

Example of a synchronized method



a) Abnormal: The value put by thread A is immediately overwritten by the value put by thread B.



b) The problem has been solved by accessing the data using synchronized methods

10.6 Defects in Handling Stress and Unusual Situations

Insufficient throughput or response time on minimal configurations

- *Defect*:
  - On a minimal configuration, the system's throughput or response time fail to meet requirements.
- *Testing strategy*:
  - Perform testing using minimally configured platforms.

Defects in Handling Stress and Unusual Situations

Incompatibility with specific configurations of hardware or software

- *Defect*:
  - The system fails if it is run using particular configurations of hardware, operating systems and external libraries.
- *Testing strategy*:
  - Extensively execute the system with all possible configurations that might be encountered by users.

Defects in Handling Stress and Unusual Situations

Defects in handling peak loads or missing resources

- *Defects*:
  - —The system does not gracefully handle resource shortage.
  - —Resources that might be in short supply include:
    - memory, disk space or network bandwidth, permission.
  - —The program being tested should report the problem in a way the user will understand.
- *Testing strategies*:
  - —Devise a method of denying the resources.
  - —Run a very large number of copies of the program being tested, all at the same time.

Defects in Handling Stress and Unusual Situations

Inappropriate management of resources

- *Defect*:
  - —A program uses certain resources but does not make them available when it no longer needs them.
- *Testing strategy*:
  - —Run the program intensively in such a way that it uses many resources, relinquishes them and then uses them again repeatedly.

Defects in Handling Stress and Unusual Situations

Defects in the process of recovering from a crash

- *Defects*:
  - —Any system will undergo a sudden failure if its hardware fails, or if its power is turned off.
  - —It is a defect if the system is left in an unstable state and hence is unable to fully recover.
  - —It is also a defect if a system does not correctly deal with the crashes of related systems.
- *Testing strategies*:
  - —Kill a program at various times during execution.
  - —Try turning the power off, however operating systems themselves are often intolerant of doing that.

Test plans

A *test plan* is a document that contains a complete set of test cases for a system

—Along with other information about the testing process.

- The test plan is one of the standard forms of documentation.
- If a project does not have a test plan:
  - —Testing will inevitably be done in an ad-hoc manner.
  - —Leading to poor quality software.
- The test plan should be written long before the testing starts.
- You can start to develop the test plan once you have developed the requirements.

Information to include in a formal test case

A. Identification and classification:
  - — Each test case should have a number, and may also be given a descriptive title.
  - — The system, subsystem or module being tested should also be clearly indicated.
  - — The importance of the test case should be indicated.

B. Instructions:
  - — Tell the tester exactly what to do.
  - — The tester should not normally have to refer to any documentation in order to execute the instructions.

C. Expected result:
  - — Tells the tester what the system should do in response to the instructions.
  - — The tester reports a failure if the expected result is not encountered.

D. Cleanup (when needed):
  - — Tells the tester how to make the system go ‘back to normal’ or shut down after the test.

Levels of importance of test cases

- Level 1:
  - — First pass critical test cases.
  - — Designed to verify the system runs and is safe.
  - — No further testing is possible.
- Level 2:
  - — General test cases.
  - — Verify that day-to-day functions correctly.
  - — Still permit testing of other aspects of the system.
- Level 3:
  - — Detailed test cases.
  - — Test requirements that are of lesser importance.
  - — The system functions most of the time but has not yet met quality objectives.

10.9 Strategies for Testing Large Systems

Big bang testing versus integration testing

- In *big bang* testing, you take the entire system and test it as a unit
- A better strategy in most cases is *incremental testing*:
  - You test each individual subsystem in isolation
  - Continue testing as you add more and more subsystems to the final product
  - Incremental testing can be performed *horizontally* or *vertically*, depending on the architecture
    - Horizontal testing can be used when the system is divided into separate sub-applications

Top down testing

- Start by testing just the user interface.
- The underlying functionality are simulated by *stubs*.
  - Pieces of code that have the same interface as the lower level functionality.
  - Do not perform any real computations or manipulate any real data.
- Then you work downwards, integrating lower and lower layers.
- The big drawback to top down testing is the cost of writing the stubs.

Bottom-up testing

- Start by testing the very lowest levels of the software.
- You needs *drivers* to test the lower layers of software.
  - —Drivers are simple programs designed specifically for testing that make calls to the lower layers.
- Drivers in bottom-up testing have a similar role to stubs in top-down testing, and are time-consuming to write.

Regression testing

- It tends to be far too expensive to re-run every single test case every time a change is made to software.
- Hence only a subset of the previously-successful test cases is actually re-run.
- This process is called *regression testing*.
  - —The tests that are re-run are called regression tests.
- Regression test cases are carefully selected to cover as much of the system as possible.

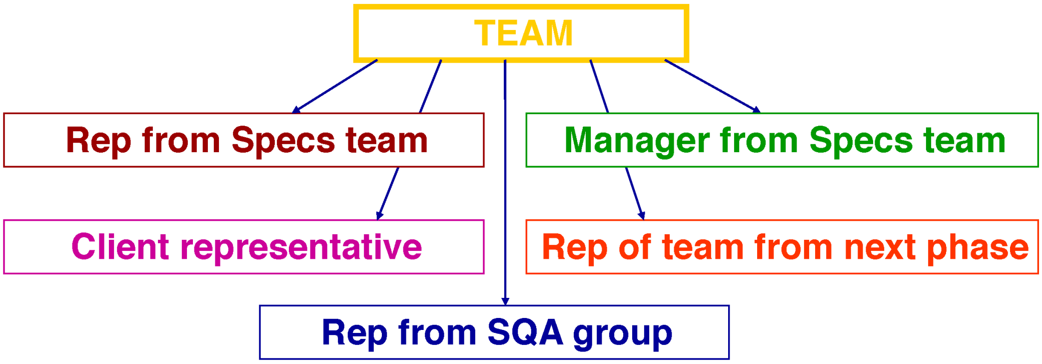
The “law of conservation of bugs”:

- ***The number of bugs remaining in a large system is proportional to the number of bugs already fixed***

Nonexecution-Based Testing

- Person creating a product should not be the only one responsible for reviewing it.
- A document is is checked by a team of software professionals with a range of skills.
- Increases chances of finding a fault.
- Types of reviews
  - Walkthroughs
  - Inspections

Walkthroughs



TEAM

Rep from Specs team

Manager from Specs team

Client representative

Rep of team from next phase

Rep from SQA group

- Reviewer prepares two lists:
  - Items that the reviewer does not understand
  - Items that the reviewer believes are incorrect

Managing walkthroughs

- Distribute material for walkthrough in advance.
- Includes senior technical staff.
- Chaired by the SQA representative.
- Task is to record fault for <u>later</u> correction
  - Not much time to fix it during walkthrough
  - Other individuals are trained to fix it better
  - Cost of 1 <u>team</u> vs cost of 1 <u>person</u>
  - Not all faults need to be fixed as not all “faults” flagged are incorrect

---

Managing walkthroughs (contd.)

- Two ways of doing walkthroughs
  - Participant driven
    - Present lists of unclear items and incorrect items
    - Rep from specs team responds to each query
  - Document driven
    - Person responsible for document walks the participants through the document
    - Reviewers interrupt with prepared comments or comments triggered by the presentation
- Interactive process
- Not to be used for the evaluation of participants

Inspections

- Proposed by Fagan for testing
  - designs
  - code
- An inspection goes beyond a walkthrough
- Five formal stages
- Stage 1 - Overview
  - Overview document (specs/design/code/ plan) to be prepared by person responsible for producing the product.
  - Document is distributed to participants.

Inspections (contd.)

- Stage 2 - Preparation
  - Understand the document in detail.
  - List of fault types found in inspections ranked by frequency used for concentrating efforts.
- Stage 3 - Inspection
  - Walk through the document and ensure that
    - Each item is covered
    - Every branch is taken at least once
    - Find faults and document them (don't correct)
    - Leader (moderator) produces a written report

Inspections (contd.)

- Stage 4 - Rework
  - Resolve all faults and problems
- Stage 5 - Follow-up
  - Moderator must ensure that every issue has been resolved in some way
- Team
  - Moderator-manager, leader of inspection team
  - Designer - team responsible for current phase
  - Implementer - team responsible for next phase
  - Tester - preferably from SQA team

What to look for in inspections

- Is each item in a specs doc adequately and correctly addressed?
- Do actual and formal parameters match?
- Error handling mechanisms identified?
- Design compatible with hardware resources?
- What about with software resources?

What to record in inspections

- Record fault statistics
- Categorize by severity, fault type
- Compare # faults with average # faults in same stage of development
- Find disproportionate # in some modules, then begin checking other modules
- Too many faults => redesign the module
- Information on fault types will help in code inspection in the same module

---

Pros and cons of inspection

- High number of faults found even before testing (design and code inspections)
- Higher programmer productivity, less time on module testing
- Fewer faults found in product that was inspected before
- If faults are detected early in the process there is a huge savings
- What if walkthroughs are used for performance appraisal?

Testing or inspecting, which comes first?

- It is important to inspect software *before* extensively testing it.
- The reason for this is that inspecting allows you to quickly get rid of many defects.
- If you test first, and inspectors recommend that redesign is needed, the testing work has been wasted.
  - There is a growing consensus that it is most efficient to inspect software befor*e a*ny testing is done.
- Even before developer testing

Inspect Code 1 of 5: Classes Overall

C1. Is its (the class') name appropriate?
  - consistent with the requirements and/or the design?
  - sufficiently specialized / general?

C2. Could it be abstract (to be used only as a base)?

C3. Does its header describe its purpose?

C4. Does its header reference the requirements and/or design element to which it corresponds?

C5. Does it state the package to which it belongs?

C6. Is it as private as it can be?

C7. Should it be final (Java)

C8. Have the documentation standards been applied?

Adapted from *Software Engineering: An Object-Oriented Perspective* by Eric J. Braude (Wiley 2001), with permission.

Inspect Code 2 of 5 : Attributes

A1. Is it (the attribute) necessary?

A2. Could it be static?

- Does every instance really need its own variable?

A3. Should it be final?

- Does its value really change?
  - Would a "getter" method alone be preferable (see section tbd)

A4. Are the naming conventions properly applied?

A5. Is it as private as possible?

A6. Are the attributes as independent as possible?

A7. Is there a comprehensive initialization strategy?

- at declaration-time?
- with constructor(s)?
- using static{}?
- Mix the above? How?

Adapted from *Software Engineering: An Object-Oriented Perspective* by Eric J. Braude (Wiley 2001), with permission.

Inspect Code 3 of 5 : Constructors

CO1. Is it (the constructor) necessary?

- Would a factory method be preferable?
  - More flexible
  - Extra function call per construction

CO2. Does it leverage existing constructors?

(a Java-only capability)

CO3. Does it initialize of all the attributes?

CO4. Is it as private as possible?

CO5. Does it execute the inherited constructor(s) where necessary?

Adapted from *Software Engineering: An Object-Oriented Perspective* by Eric J. Braude (Wiley 2001), with permission.

Inspect Code 4 of 5: Method Headers

MH1. Is the method appropriately named?

- method name consistent with requirements &/or design?

MH2. Is it as private as possible?

MH3. Could it be static?

MH4. Should it be be final?

MH5. Does the header describe method's purpose?

MH6. Does the method header reference the requirements and/or design section that it satisfies?

MH7. Does it state all necessary invariants?

MH8. Does it state all pre-conditions?

MH9. Does it state all post-conditions?

MH10. Does it apply documentation standards?

Adapted from *Software Engineering: An Object-Oriented Perspective* by Eric J. Braude (Wiley 2001), with permission.

Inspect Code 5 of 5: Method Bodies

MB1. Is the algorithm consistent with the detailed design pseudocode and/or flowchart?

MB2. Does the code assume no more than the stated preconditions?

MB3. Does the code produce every one of the postconditions?

MB4. Does the code respect the required invariant?

MB5. Does every loop terminate?

MB6. Are required notational standards observed?

MB7. Has every line been thoroughly checked?

MB8. Are all braces balanced?

MB9. Are illegal parameters considered? (see section tbd)

MB10. Does the code return the correct type?

Adapted from *Software Engineering: An Object-Oriented Perspective* by Eric J. Braude (Wiley 2001), with permission.