Queen Mary
University of Lo

ECS505U
SOFTWARE ENGINEERING

MUSTAFA BOZKURT & LORENZO JAMONE

LECTURER IN SOFTWARE ENGINEERING

Queen Mary
University of London

WEEK 9

SOFTWARE TESTING

Queen Mary
University of London

LESSON OBJECTIVES

- Understand why software testing is important
- Learn what a test case is
- Learn about a range of commonly used software testing techniques and strategies

SOFTWARE TESTING: DEFINITION

Queen Mary University of London

Software Testing is:

The execution of software for purposes of validation and verification. (ISTQB)

The process of executing a program or system with the intent of finding errors. (Myers)

Any activity aimed at evaluating an attribute or capability of a program or system and determining that it meets its required results (Hetzel)

An investigation conducted to provide stakeholders with information about the quality of the product or service under test. (Kaner)

SOFTWARE TESTING: BUG, DEFECT, ERROR & FAILURE

Queen Mary University of London

A **failure** is an unacceptable behavior exhibited by a system.

A **defect** (a.k.a. **bug**) is a flaw in any aspect of the system including the requirements, the design and the code, that contributes, or may potentially contribute, to the occurrence of one or more failures.

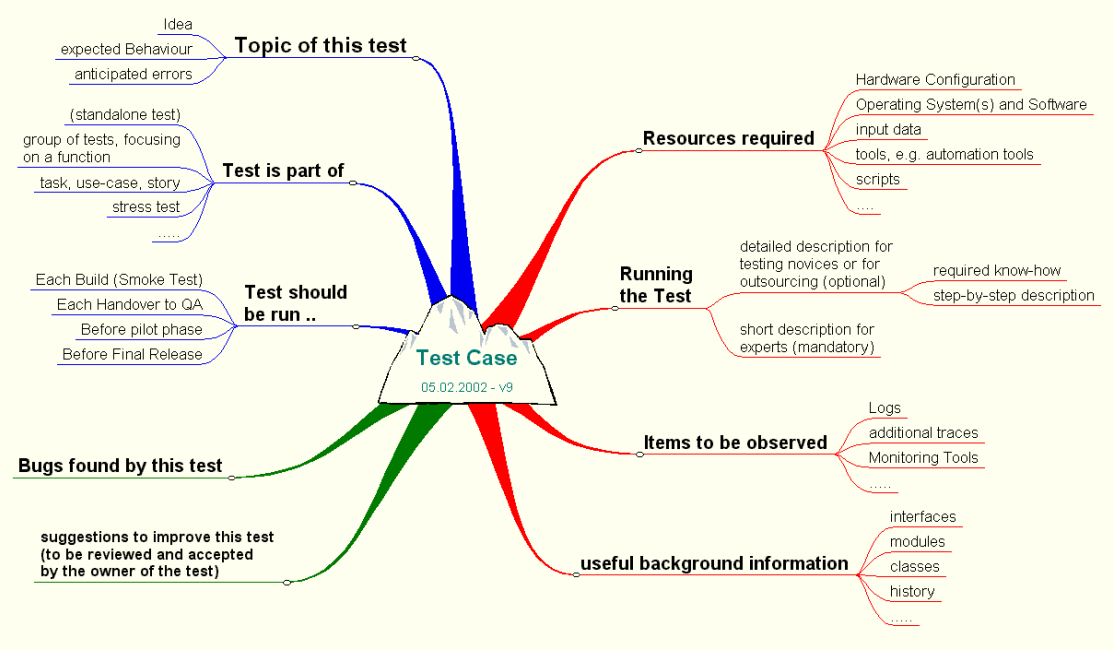
An **error** (a.k.a. **mistake**) is a slip-up or inappropriate decision by a software developer that leads to the introduction of a defect into the system

SOFTWARE TESTING: TEST CASE

Queen Mary University of London

Requires test data (inputs) and knowledge of the expected output.

A test case is the tuple (i, o),where i is the test data and o=S(i) for S the Software Under Test (SUT).



http://2.bp.blogspot.com/_vdqOsYKAf0Y/SxKr7eSIdbI/AAAAAAAAAp0/TshJWzz-5V0/s1600/TestCase_new.gif

Queen Mary University of London

TEST CASES EXAMPLE

Specifications of software to test:

Capitalise a given word (no spaces allowed)

Input (i)	Expected Output (o)	Test Case
test	TEST	(test, TEST)
Test me	error	(Test me, error)
TesT	TEST	(TesT , TEST),
t3st	T3ST	(t3st, T3ST),
NULL (input)	error	(NULL, error)

SOME REALLY BORING OBSERVATIONS

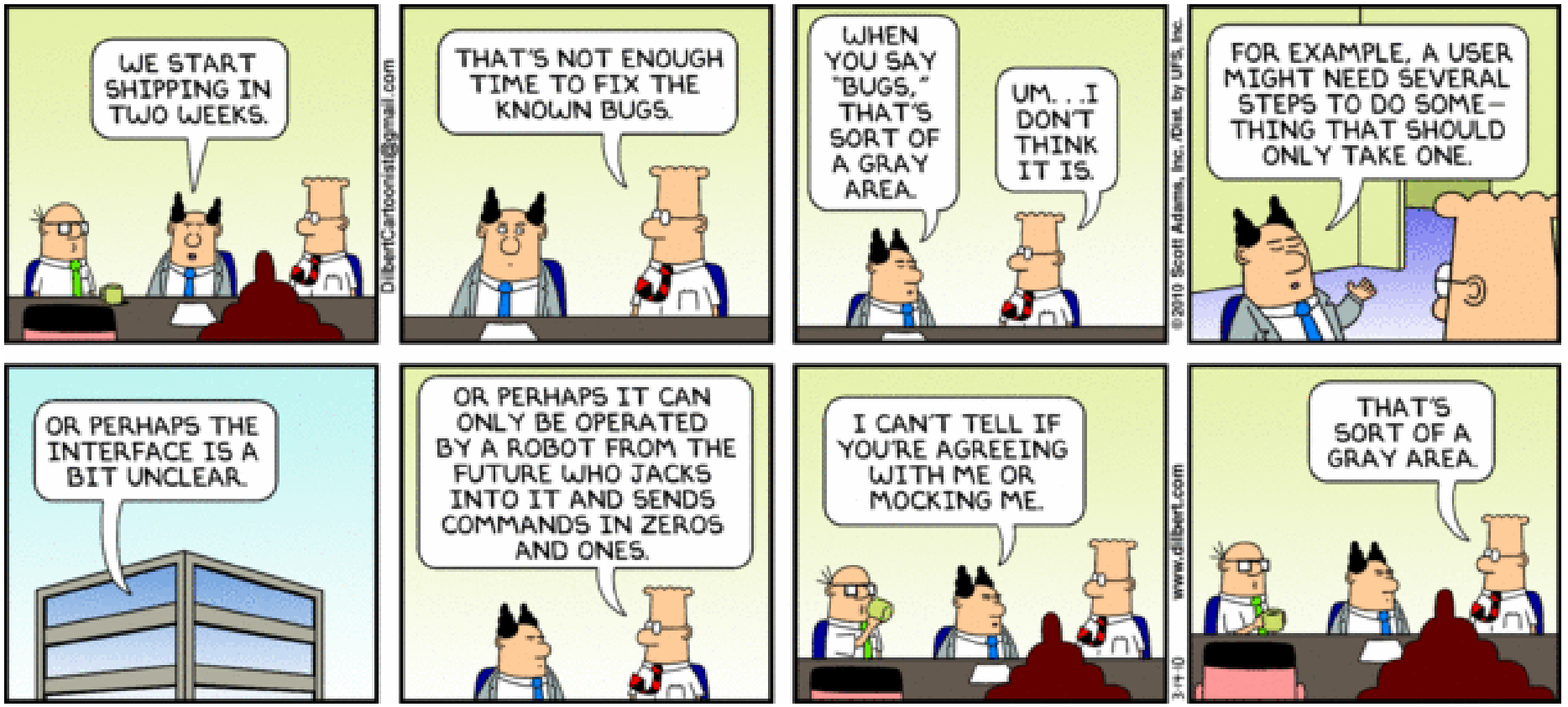
Queen Mary University of London

It is impossible to completely test any nontrivial module or any system.

- Halting problem
- Cost and effort

“Testing can only show the presence of bugs, not their absence” (Dijkstra)

- Except fault-based testing
  - Testing is fault-based when it seeks to demonstrate that prescribed faults are not in a program. (A Theory of Fault-Based Testing, L. J. Morell, IEEE Transactions on Software Engineering, 1990)


WE START SHIPPING IN TWO WEEKS.
THAT'S NOT ENOUGH TIME TO FIX THE KNOWN BUGS.
WHEN YOU SAY "BUGS," THAT'S SORT OF A GRAY AREA.
UM. . .I DON'T THINK IT IS.
FOR EXAMPLE, A USER MIGHT NEED SEVERAL STEPS TO DO SOME-THING THAT SHOULD ONLY TAKE ONE.
OR PERHAPS THE INTERFACE IS A BIT UNCLEAR.
OR PERHAPS IT CAN ONLY BE OPERATED BY A ROBOT FROM THE FUTURE WHO JACKS INTO IT AND SENDS COMMANDS IN ZEROS AND ONES.
I CAN'T TELL IF YOU'RE AGREEING WITH ME OR MOCKING ME.
THAT'S SORT OF A GRAY AREA.
DilbertCartoonist@gmail.com
©2010 Scott Adams, Inc./Dist. by UFS, Inc.
www.dilbert.com
3-1-10

TESTING: THE 'STANDARD' APPROACH

Queen Mary University of London

1. Write some code
2. Get scared enough about some aspect of it to feel the need to 'try it out'
3. Write some kind of driver that invokes the bit of code. Add a few print statements to verify it's doing what you think it should
4. Run the test, eyeball the output and then delete (or comment out) the print statements
5. Go back to step 1

- Andy Hunt and Dave Thomas, IEEE Software 19(1), 2002

GOOD TESTING TAKES CREATIVITY

Queen Mary University of London

- Testing requires great skill.
- Testing is done best by independent testers
- Programmers often stick to the data set that makes the program work
- A program often does not work when tried by somebody else.

TESTING

- Functional Testing
- Non-functional Testing
  - Performance testing
  - Stress/Load testing
  - Usability testing
  - Security testing

TESTING IN DEVELOPMENT

- Unit or module testing
- Smoke testing (build verification/acceptance testing)
- Integration testing
- System testing

TESTING AFTER DEVELOPMENT

Queen Mary University of London

- Alpha test
- Beta test
- Acceptance testing
- Operation (during use)
- Regression Testing (also performed during development)

EFFECTIVE AND EFFICIENT TESTING

Queen Mary University of London

- To test effectively, you must use a strategy that uncovers as many defects as possible.
- To test efficiently, you must find the largest possible number of defects using the fewest possible tests.
- An effective and efficient testing strategy is often called a high-yield strategy.

TESTING STRATEGIES

Bottom-up

- Test Driver needed (Unit testing is example)

Top-down

- Test stub needed

Queen Mary
University of London

TESTING METHODS

- Dynamic testing vs Static testing (static analysis)
- Black Box Testing vs White Box Testing
- Statistical Testing

WHITE-BOX TESTING

White-box testing (also known as clear box testing, glass box testing, transparent box testing, and **structural testing**) is a method of testing software that tests internal structures or workings of an application, as opposed to its functionality (i.e. black-box testing).

Wikipedia

Determining test cases from a knowledge of the internal logic of the software.

Queen Mary
University of London

CODE COVERAGE

Code coverage analysis is the process of:

- Finding areas of a program not exercised by a set of test cases
- Creating additional test cases to increase coverage
- Determining a quantitative measure of code coverage, which is an indirect measure of quality

An optional aspect of code coverage analysis is:

- Identifying redundant test cases that do not increase coverage

WHITE-BOX TESTING TECHNIQUES

Queen Mary University of London

- Statement coverage
- Branch coverage
- Condition coverage
- Modified condition/decision coverage (MC/DC)
- Path coverage

Queen Mary University of London

STATEMENT COVERAGE

- The statement coverage is also known as line coverage or segment coverage.
- The statement coverage covers only the **true conditions**.
- We can identify the statements executed and where the code is not executed because of blockage.
- Each and every line of code needs to be checked and executed

$$coverage = \frac{number\ of\ statements\ executed}{Total\ number\ of\ statements}\ x\ 100\%$$

Queen Mary
University of London

STATEMENT COVERAGE

Advantages:

- It verifies what the written code is expected to do and not to do
- It checks the flow of different paths in the program and it also ensure that whether those path are tested or not.

Disadvantages:

- It cannot test the false conditions.
- It does not report that whether the loop reaches its termination condition.
- It does not understand the logical operators.

Queen Mary University of London

STATEMENT COVERAGE

```
if (x > 10 && y > 100 && z <= -11) {
    doSomething();
}
x += z;
if (x < 0) {
    doSomethingElse();
}
```

TC: x = 11 y=101 z=-12
Statement coverage: 100%

```
if (x < 10) {
    doSomething();
}
x *= 2;
if (x >= 20) {
    doSomethingElse();
}
```

TC1: x = 9, TC2: x=10
Statement coverage: 100%

Queen Mary University of London

BRANCH COVERAGE

- It covers both the true and false conditions unlikely the statement coverage.
- A branch is the outcome of a decision, so branch coverage simply measures which decision outcomes have been tested.

A decision is an IF statement, a loop control statement (e.g. DO-WHILE or REPEAT-UNTIL), or a CASE statement, where there are two or more outcomes from the statement.

$$coverage = \frac{number\ of\ branches\ executed}{Total\ number\ of\ branches}\ x\ 100\%$$

Queen Mary
University of London

BRANCH COVERAGE

Advantages:

- To validate that all the branches in the code are reached
- To ensure that no branches lead to any abnormality of the program's operation
- It eliminate problems that occur with statement coverage testing

Disadvantages:

- This metric ignores branches within boolean expressions which occur due to short-circuit operators.

BRANCH COVERAGE

```
if (x < 10) {
    doSomething();
}
x *= 2;
if (x >= 20) {
    doSomethingElse();
}
```

TC1: x = 9, TC2: x=10
Branch coverage: 100%

```
if (x > 10 && y > 100 && z <= -11) {
    doSomething();
}
x += z;
if (x < 0) {
    doSomethingElse();
}
```

TC: x = 11 y=101 z=-12
Branch coverage: 50%

TC1: x = 11 y=101 z=-12
TC2: x = 15 y=101 z=-10
Branch coverage: 100%

Queen Mary University of London

BRANCH COVERAGE

```
String s = null;
if (x > 10) {
    s = "te.st";
}
String[] reg = s.split(".");
```

What's the difference between performing statement and branch coverage of this code?

Queen Mary University of London

PATH COVERAGE

Path coverage aims to execute each of the possible paths in each function.

A path represents the flow of execution from the start of a method to its exit.

A method with N decisions has 2^N possible paths, and if the method contains a loop, it may have an infinite number of paths.

$$coverage = \frac{number\ of\ paths\ executed}{Total\ number\ of\ paths}\ x\ 100\%$$

PATH COVERAGE

```
public int foo(int x) {
    if (x >= 10) {
        x += 1;
    }
    if (x >= 5) {
        x /= 2;
    }
    if (x >= 2) {
        x *= x;
    }
    return x;
}
```

X value	Path executed
x ≥ 10	TTT
9 ≥ x ≥ 5	FTT
4 ≥ x ≥ 2	FFT
1 ≥ x	FFF

Number of paths?

$2^3 = 8$

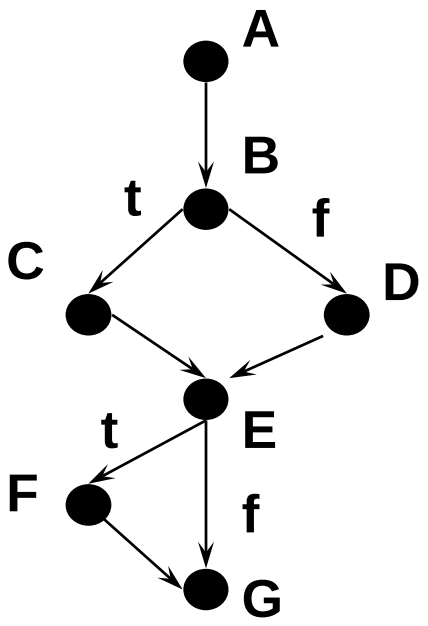
All paths?

TTT, TTF, TFF, TFT,
FTT, FTF, FFT, FFF

Can we cover all paths?

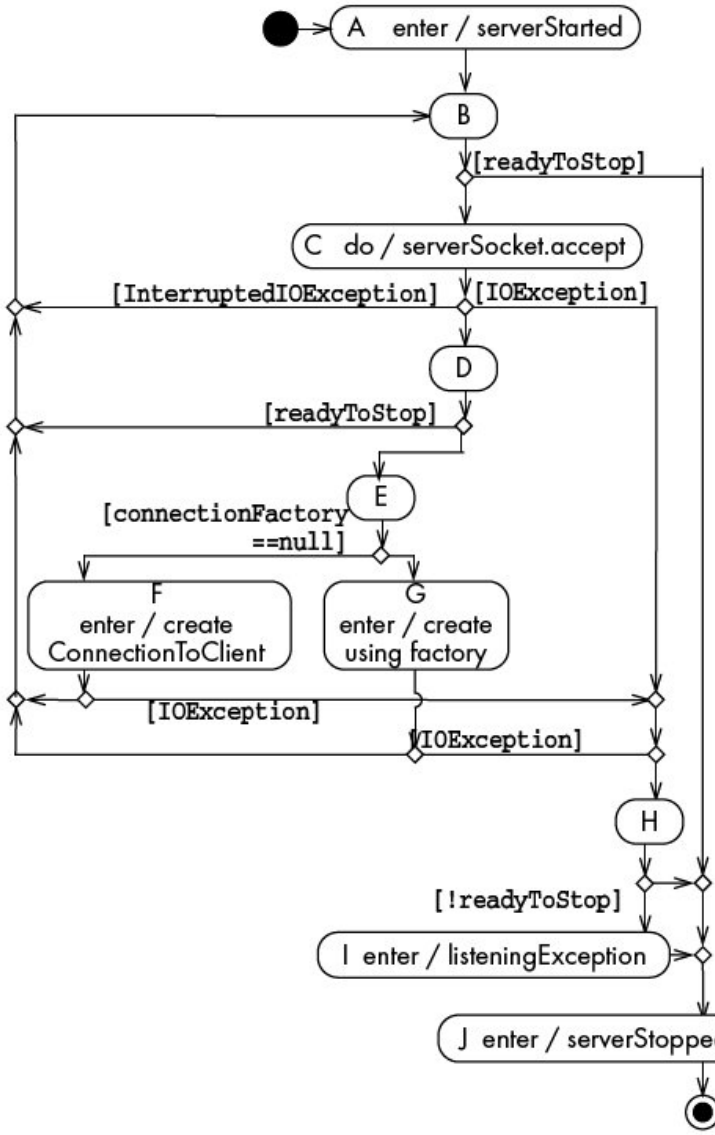
INFEASIBLE PATHS

```
A   input(score);
B   if score < 40
C        then print (‘fail’)
D        else print (`pass’);
E   if score >= 70
F        then print (‘with distinction’);
G   end
```



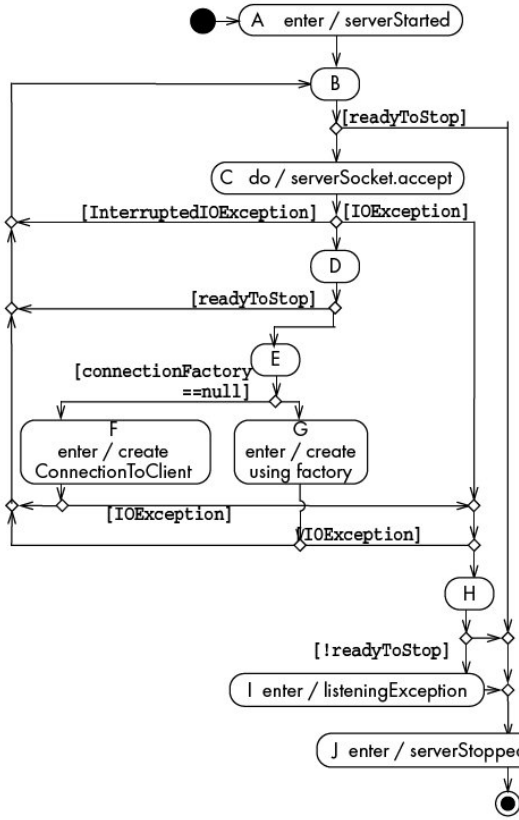
Can you find an infeasible path here?

DETERMINING THE PATHS




```
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();
}
```

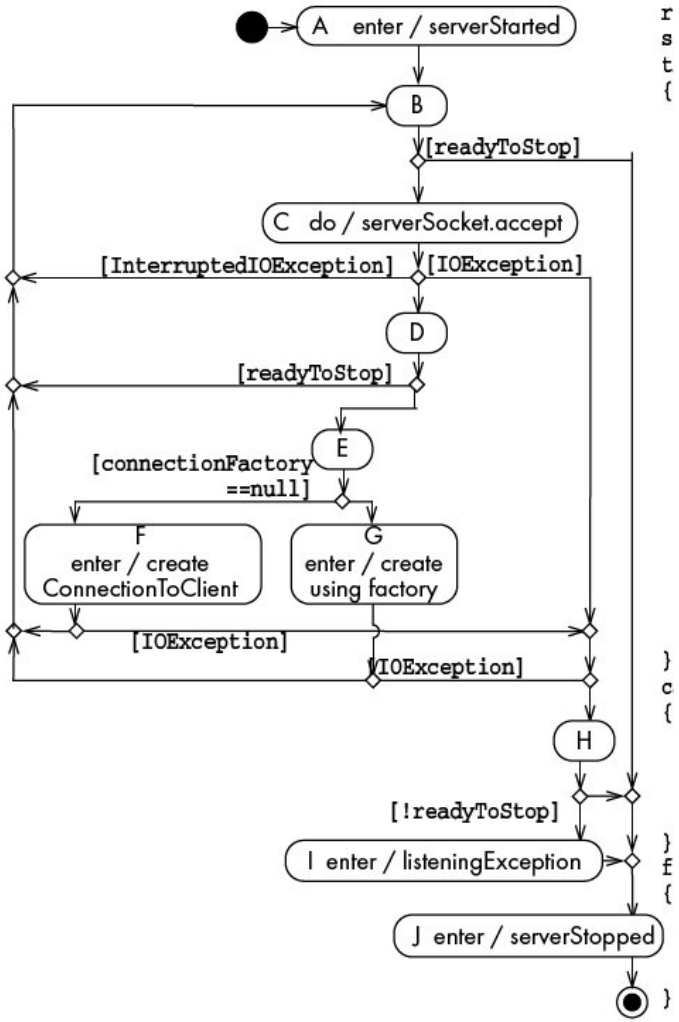
CHOOSING THE RIGHT STRATEGY




```
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();
}
```

- Cover all possible paths.
- Cover all possible edges/branches
- Cover all nodes

TEST CASES

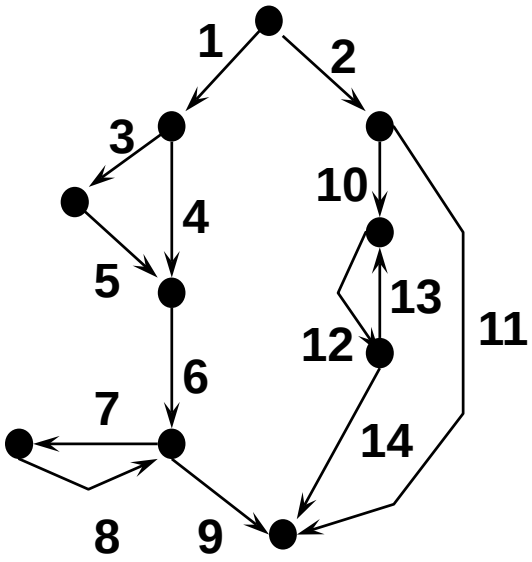


1. A-B-C-B-C-…-B-C-D-E-F-B-C-…-B-C-B-J: run the server, connect a client, and send the stop command.
2. A-B-C-B-C-…-B-C-D-E-G-B-C-…-B-C-B-J: repeat the same sequence, but this time use a connection factory.
3. A-B-C-B-C-…-B-C-H-I-J: same as 2, but force an I/O exception to occur while accepting the connection.
4. A-B-C-B-C-…-B-C-D-E-G-H-J: same as 2, but force an exception to occur during creation of the connection class
5. A-B-C-B-C-…-B-C-D-E-F-H-J: same as 4, but with a regular ConnectionToClient.
6. A-B-C-B-C-…-B-C-D-B-J: same as 1, except send the stop command just after the connection is accepted.

To cover all nodes just need 1,3 and 4.

STRUCTURAL TESTING: EXAMPLES

Queen Mary University of London



All paths (infinite number)
<2 11>, <2 10 12 14>, <2 10 12 (13 12)^n 14>, <1 3 5 6 9>, <1 4 6 9>, <1 3 5 6 (7 8)^n 9>, <1 4 6 (7 8)^n 9> (any n>0)

Visit-each-loop (7)
<2 11>, <2 10 12 14>, <2 10 12 13 12 14>, <1 3 5 6 9>, <1 3 5 6 7 8 9>, <1 4 6 9>, <1 4 6 7 8 9>

Simple paths (6)
<2 11>, <2 10 12 14>, <1 3 5 6 9>, <1 3 5 6 7 8 9>, <1 4 6 9>, <1 4 6 7 8 9>

Basis paths (6)
<2 11>, <2 10 12 13 12 14>, <1 3 5 6 9>, <1 3 5 6 7 8 9>, <1 4 6 9>, <1 4 6 7 8 9>

Branch (4)
<2 11>, <2 10 12 13 12 14>, <1 3 5 6 9>, <1 4 6 7 8 9>

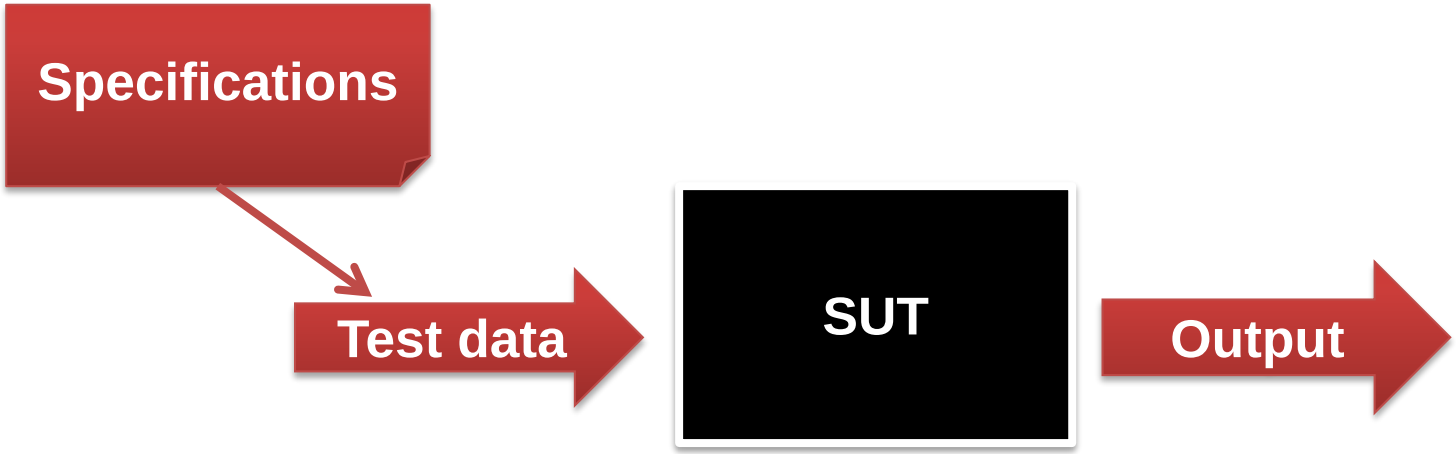
Statement (2)
<2 10 12 14>, <1 3 5 6 7 8 9>

BLACK-BOX TESTING

Black-box (also functional) testing is a method of software testing that examines the functionality of an application without peering into its internal structures or workings.

Wikipedia

Determining test cases without the knowledge of the internal logic of the software



Queen Mary University of London

BLACK-BOX TESTING

- Equivalence partitioning
- Boundary value analysis
- Statistical testing
- Cause-effect graphing
- Error guessing
- Random testing
- Syntax directed testing
- State transition testing

EQUIVALENCE PARTITIONING

```
public boolean checkValidMonth(int month) {
    if (month >= 1 && month <= 12) {
        return true;
    } else {
        return false;
    }
}
```

Brute force: all int values (-2^{31} to $2^{31}-1$)

Equivalence classes for month validation, where the input value is a Java int

Equivalence class	*Range of values*
Invalid – larger	13 to $2^{31}-1$ (can be loosely expressed as 13 to $\propto$)
Valid	1 to 12
Invalid – smaller	-2^{31} to 0 (can be loosely expressed as $-\propto$ to 0)

BOUNDARY VALUE ANALYSIS

Choose test cases directly on, above and beneath the edges of the input equivalence class and output equivalence classes

```
public boolean checkValidMonth(int month) {
    if (month >= 1 && month <= 12) {
        return true;
    } else {
        return false;
    }
}
```

Equivalence classes for month validation, where the input value is a Java **int**

Equivalence class	*Range of values*
Invalid – larger	13 to $2^{31}-1$ (can be loosely expressed as 13 to $\propto$)
Valid	1 to 12
Invalid – smaller	-2^{31} to 0 (can be loosely expressed as $-\propto$ to 0)

Test for:

Partition 1: 12,13, 2^{31-1}, 2^{31}

Partition 2: 0,1,12,13

Partition 3: 1,0, -2^{31} ,-2^{31-1}

STATISTICAL USAGE TESTING

Function	Usage Frequency	Distribution Interval
Update (U)	32%	00-31
Delete (D)	14%	32-45
Query (Q)	46%	46-91
Print (P)	8%	92-99

Operational profile

Test number	Random Numbers	Test Cases
1	29 11 47 52 26 94	U U Q Q U P
2	62 98 39 78 82 65	Q P D Q Q Q
3	83 32 58 41 36 17	Q D Q D D U
4	36 49 96 82 20 77	D Q P Q U Q

24 test cases: 5 Update, 11 Query, 5 Delete, 3 Print

STATISTICAL USAGE TESTING

Function	Usage Frequency	Distribution Interval
Update (U)	32%	00-31
Delete (D)	14%	32-45
Query (Q)	46%	46-91
Print (P)	8%	92-99

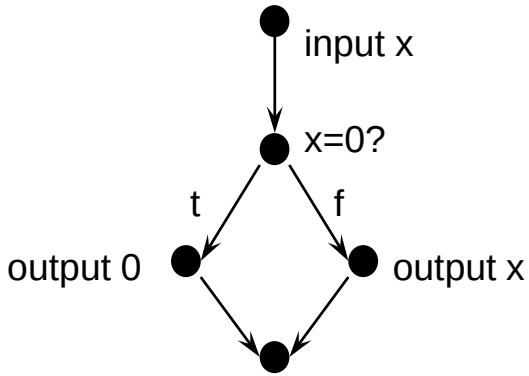
Operational profile

Test number	Random Numbers	Test Cases
1	29 11 47 52 26 94	U U Q Q U P
2	62 98 39 78 82 65	Q P D Q Q Q
3	83 32 58 41 15 17	Q D Q D U U
4	36 49 07 82 20 77	D Q U Q U Q

BLACK-BOX AND WHITE-BOX TESTING

Queen Mary University of London

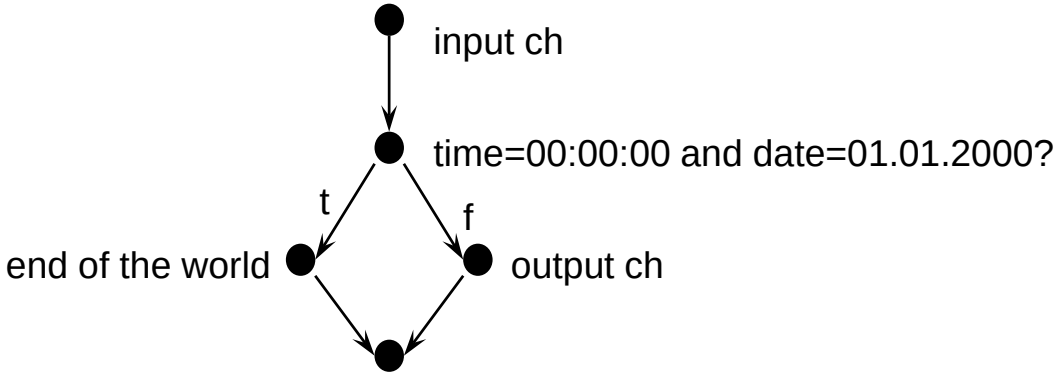
```
public int square(int x){
    if(x == 0)
        return 0;
    else
        return x;
}
```



x = 0 and 1 achieves 100% branch coverage

Easy to do full path testing without finding obvious fault

```
public void echo(String ch){
    // date representing 00:00:00 01.01.2000
    Calendar y2k = new GregorianCalendar();
    if(new Date().compareTo(y2k.getTime()) == 0){
        endOfTheWorld();
    } else{
        System.out.print(ch);
    }
}
```



Extensive black box testing would not find the obvious fault here

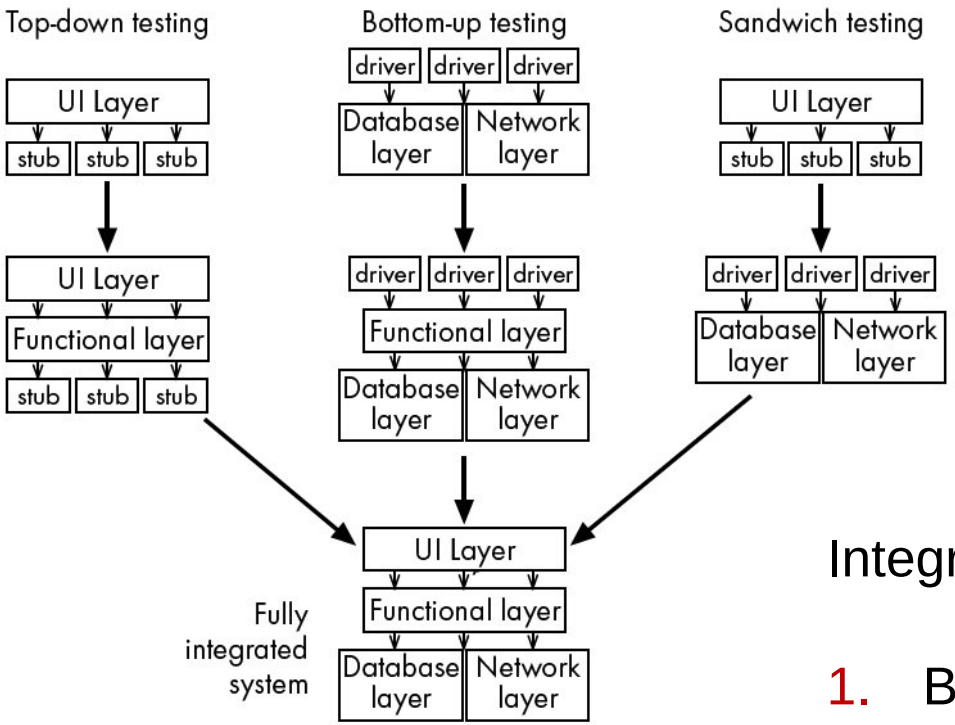
Queen Mary
University of London

INTEGRATION TESTING

Integration testing tests integration or interfaces between components, interactions to different parts of the system such as an operating system, file system and hardware or interfaces between systems.

It occurs after unit testing and before validation testing. Integration testing takes as its input modules that have been unit tested, groups them in larger aggregates, applies tests defined in an integration test plan to those aggregates, and delivers as its output the integrated system ready for system testing.

INTEGRATION TESTING



Integration testing types:

1. Big-bang
2. Bottom-up
3. Top-down
4. Sandwich

Queen Mary
University of London

INTEGRATION TESTING

1. **Big Bang integration testing**:

In Big Bang all components or modules are integrated simultaneously, after which everything is tested as a whole.

Advantages: Everything is finished before integration testing starts.

Disadvantage: The major disadvantage is that in general it is time consuming (more complicated) and difficult to trace the cause of failures because of late integration.

Queen Mary University of London

INTEGRATION TESTING

2. Bottom-up integration testing:

Testing takes place from the bottom of the control flow upwards.
Components or systems are substituted by drivers.

Advantages:

- Easier to perform as development and testing can be done together.
- Easier to detect bugs.

Disadvantages:

- Interface defects harder to detect at the end of cycle.
- Required test drivers for modules might be require a lot of effort.

Queen Mary University of London

INTEGRATION TESTING

3. Top-down integration testing:

In top-down, testing takes place from top to bottom, following the control flow or architectural structure (e.g. starting from the GUI or main menu). Incomplete components or systems are substituted by stubs.

Advantages: Stubs often require less effort compared to the drivers.

Disadvantages:

- Business logic is tested at the end of cycle so bugs might be discovered late
- Harder to detect errors in business logic due to late testing of it.

INTEGRATION TESTING

Sandwich Testing is an approach to combine top down testing with bottom up testing.

Risky - hardest integration testing is performed starting with the risky and more complex software module(s) first.

Queen Mary
University of London

INTEGRATION TESTING

In most cases integration is very hard, it is referred to as "integration hell“ due to many issues faces during performing it.

Continuous integration (CI) is the practice of merging all developer working copies to a shared mainline frequently.

Extreme programming relies heavily on CI and integrating several times a day is suggested. Use of smoke testing is common.

Queen Mary
University of London

REGRESSION TESTING

How do we test our code during maintenance?

REGRESSION TESTING

Retesting to ensure previously working code does not fail as a result of changes

Should be applied irrespective of testing strategy or method

REGRESSION TESTING

Retest all

It is expensive as it needs to re-run all the cases, it ensures that there are no errors because of the modified code.

Regression test selection

Selecting the part of test suite that tests modified parts of the code.

Test case prioritization

Prioritize the test cases so as to increase a test suite's rate of fault detection. Test case prioritization techniques schedule test cases so that the test cases that are higher in priority are executed before the test cases that have a lesser priority.

DEFECTS IN ORDINARY ALGORITHMS

Queen Mary University of London

Incorrect logical conditions:

The logical conditions that govern looping and if-then-else statements are wrongly formulated.

Use equivalence class and boundary testing.

```
public String triangle(int sideA, int sideB, int sideC){
    if(sideA == sideB || sideB == sideC || sideC == sideA){
        return "Isosceles";
    } else if(sideA == sideB && sideB == sideC){
        return "Equilateral";
    } else
        return "Scalene";
}
```

DEFECTS IN ORDINARY ALGORITHMS

Queen Mary University of London

Performing a calculation in the wrong part of a control construct:

The program performs an action when it should not, or does not perform an action when it should. These are typically caused by inappropriately excluding the action from, or including the action in, a loop or if-then-else construct.

Design tests that execute each loop zero times, exactly once, and more than once.

```
public void checkGreater(int number1, int number2) {
    while (number1 > number2) {
        System.out.print("Number 1 is greater");
        number2++;
    }
    if(number2 == number1) doSomething();

}
```

DEFECTS IN ORDINARY ALGORITHMS

Queen Mary University of London

Not terminating a loop or recursion:

A loop or a recursion does not always terminate, that is, it is ‘infinite’.

Although the programmer should have analyzed all loops or recursions to ensure they reach a terminating case, a tester should nevertheless assume that the programmer has made an error.

```
public void checkGreater(int number1, int number2) {
    while (number1 > number2) {
        System.out.print("Number 1 is greater");
        if(number2 == number1) doSomething();
        //number2++;
    }


}
```

DEFECTS IN ORDINARY ALGORITHMS

Queen Mary University of London

Not setting up the correct preconditions for an algorithm:

One specifies preconditions that state what must be true before the algorithm should be executed..

Run test cases in which each precondition is not satisfied. Preferably its input values are just beyond what the algorithm can accept.

```
// sideA, sideB and sideC must be positive integers
// combined legnth of two sides must be greater than the third side
// e.g. sideA + sideB > sideC
public String triangle(int sideA, int sideB, int sideC) {
    if (sideA == sideB && sideB == sideC) {
        return "Equilateral";
    } else if (sideA == sideB || sideB == sideC || sideC == sideA) {
        return "Isosceles";
    } else {
        return "Scalene";
    }
}
```

DEFECTS IN ORDINARY ALGORITHMS

Queen Mary University of London

Not handling null conditions:

It is a defect when a program behaves abnormally when a null condition is encountered. In these situations, the program should 'do nothing, gracefully'.

Determine all possible null conditions and run test cases that would highlight any inappropriate behavior.

```
public int returnListLength(List l){
    if(l== null)
        return -1;
    else
        return l.size();
}
```

DEFECTS IN ORDINARY ALGORITHMS

Queen Mary University of London

Operator precedence errors:

An operator precedence error occurs when a programmer omits needed parentheses, or puts parentheses in the wrong place.

In software that computes formulae, run tests that anticipate defects such as those described in the example below.

```
public int doSomething(int number) {
    // increase number by one and multiply with 11
    return number + 1 * 11;
}
```

DEFECTS IN ORDINARY ALGORITHMS

Queen Mary University of London

Use of inappropriate standard algorithms:

An inappropriate standard algorithm is one that is unnecessarily inefficient or has some other property that is widely recognized as being bad.

- An inefficient sort algorithm
- An inefficient search algorithm
- A non-stable sort
- A search or sort that is case sensitive
- when it should not be, or vice versa

```
public void sortValues(int[] l){
    java.util.Arrays.sort(l);
    bubbleSort(l);
}

public static void bubbleSort(int[] x) {
    int n = x.length;
    for (int pass = 1; pass < n; pass++) {
        for (int i = 0; i < n - pass; i++) {
            if (x[i] > x[i + 1]) {
                int temp = x[i];
                x[i] = x[i + 1];
                x[i + 1] = temp;
            }
        }
    }
}
```

DEFECTS IN ORDINARY ALGORITHMS

Queen Mary University of London

Not using enough bits or digits to store maximum values:

A system does not use variables that are capable of representing the largest possible values that could be stored. When the capacity of a data type is exceeded, an unexpected exception might be thrown, or else the data may be stored incorrectly.

Test using very large numbers to ensure that the system has a wide enough margin of error.

```
public int changeSign(int number){
    if(number < 0){
        number = number * -1;
    }
    assert number > 0;
    return number;
}
```

DEFECTS IN ORDINARY ALGORITHMS

Queen Mary University of London

Assuming a floating-point value will be exactly equal to some other value:

If you perform an arithmetic calculation on a floating-point value, then the result will very rarely be computed exactly. It is therefore a defect to test if a floating-point value is exactly equal to some other value. You should instead test if it is within a small range around that value.

Standard boundary testing should detect this type of defect.

```
public static float floatPower(float number, int power) {
    if (power > 1) {
        return number * floatPower(number, power - 1);
    } else {
        return number;
    }
}

public static void main(String[] args) {
    for (int i = 0; i < 10; i++) {
        System.out.println("output " + i + ": " + floatPower(13.234561233f, 5));
    }
}
```

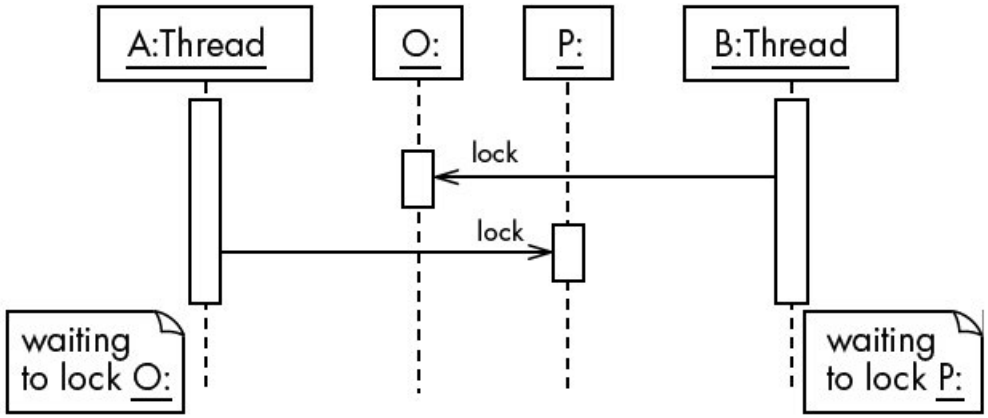

```
output 0: 406020.25
output 1: 406020.25
output 2: 406020.25
output 3: 406020.25
output 4: 406020.25
output 5: 406020.25
output 6: 406020.25
output 7: 406020.25
output 8: 406020.25
output 9: 406020.25
BUILD SUCCESSFUL (total time: 0 seconds)
```

DEFECTS IN ORDINARY ALGORITHMS

Queen Mary University of London

Deadlock and livelock:

A deadlock is a situation where two or more threads or processes are stopped, waiting for each other to do something before either can proceed.



DEFECTS IN ORDINARY ALGORITHMS

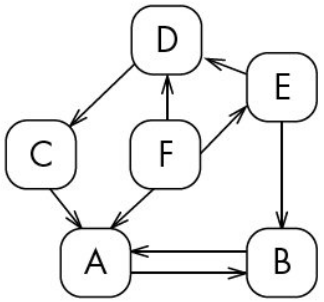
Queen Mary University of London

Deadlock and livelock:

Livelock is similar to deadlock, in the sense that the system is stuck in a particular behavior that it cannot get out of.

The difference is as follows:

- In deadlock the system is normally hung, with nothing going on
- In livelock, the system can do some computations, but it can never get out of a limited set of states.



DEFECTS IN ORDINARY ALGORITHMS

Queen Mary University of London

Deadlock and livelock:

Deadlocks and livelocks tend to occur as a result of 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.

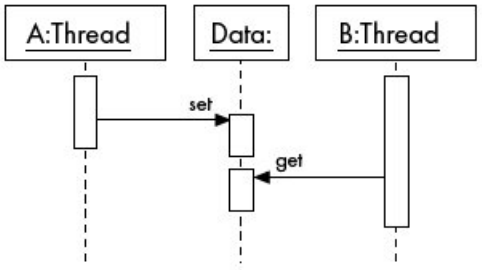
Best detected by formal verification approaches such as model checking.

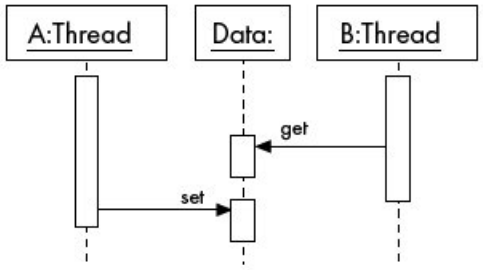
DEFECTS IN ORDINARY ALGORITHMS

Queen Mary University of London

Critical races:

A critical race is a defect in which one thread or process can sometimes experience a failure because another thread or process interferes with the ‘normal’ sequence of events. The defect is not that the other thread tries to do something, but that the system allows interference to occur.





STRATEGIES FOR TESTING LARGE SYSTEMS

Queen Mary University of London

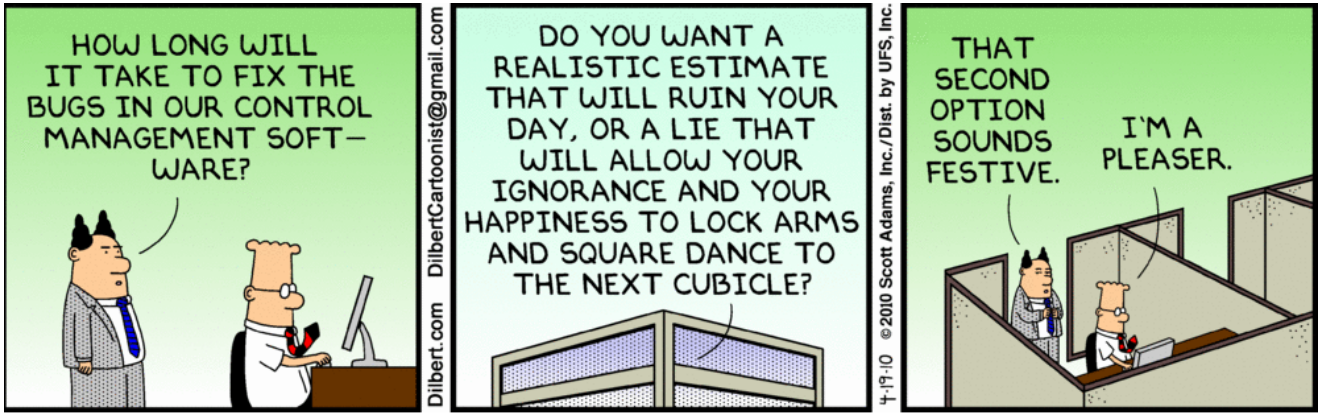
Integration testing:

Testing if the parts of a system or subsystem work together as expected is called integration testing.

The simplest approach to integration testing is big bang testing.

- In this approach, you take the entire integrated system and test it all at once.

In incremental testing, first test each individual subsystem in isolation, and then continue testing as you integrate more and more subsystems.


HOW LONG WILL IT TAKE TO FIX THE BUGS IN OUR CONTROL MANAGEMENT SOFT-WARE?
DO YOU WANT A REALISTIC ESTIMATE THAT WILL RUIN YOUR DAY, OR A LIE THAT WILL ALLOW YOUR IGNORANCE AND YOUR HAPPINESS TO LOCK ARMS AND SQUARE DANCE TO THE NEXT CUBICLE?
THAT SECOND OPTION SOUNDS FESTIVE.
I'M A PLEASER.
Dilbert.com DilbertCartoonist@gmail.com
4-19-10 ©2010 Scott Adams, Inc./Dist. by UFS, Inc.


I SPENT THE WEEK WRITING A TEST SCRIPT FOR OUR PRODUCT.
AND I WROTE A TEST SCRIPT TO TEST DILBERT'S TEST SCRIPT.
YOUR SCRIPT WAS ALMOST PERFECT. KEEP UP THE GOOD WORK, BUDDY.
Dilbert.com DilbertCartoonist@gmail.com
3-24-11 ©2011 Scott Adams, Inc./Dist. by UFS, Inc.


CAN YOU TEST THE SOFTWARE TODAY?
NO, I'M MAKING MAJOR CHANGES TUESDAY.
YOU COULD TEST THE CURRENT VERSION.
I WISH PEOPLE WOULDN'T SLAP THEIR FOREHEADS AND SAY "AYE-YI-YI-YI" EVERY TIME I TALK.
www.dilbert.com scottadams@aol.com
© 2000 United Feature Syndicate, Inc.

Queen Mary University of London

LESSON SUMMARY

- Testing aims to find faults
- Testing must be properly planned
- Effective testing necessarily involves a range of methods
- Testing is still a black art, but many rules and heuristics are available
- Unit testing is known to be one of the most cost-effective QA procedures. You can do unit testing with full automated support using JUnit.