CSC301: Introduction to Software Engineering

Lecture 8

Wael Aboulsaadat

Software Testing

Basic definitions

- **A *failure*** is an unacceptable behaviour exhibited by a system
  - The frequency of failures measures the *reliability*
  - An important design objective is to achieve a very low failure rate and hence high reliability.
  - A failure can result from a violation of an *explicit* or *implicit* requirement

- **A *defect*** is a flaw in any aspect of the system that contributes, or may potentially contribute, to the occurrence of one or more failures
  - could be in the requirements, the design and the code
  - It might take several defects to cause a particular failure

- **An *error*** is a slip-up or inappropriate decision by a software developer that leads to the introduction of a defect

Effective and Efficient Testing

- 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 <u>using the fewest possible tests</u>
  - Testing is like detective work:
    - The tester must try to understand how programmers and designers think, so as to better find defects.
    - The tester must not leave anything uncovered, and must be suspicious of everything.
    - It does not pay to take an excessive amount of time; tester has to be *efficient*.

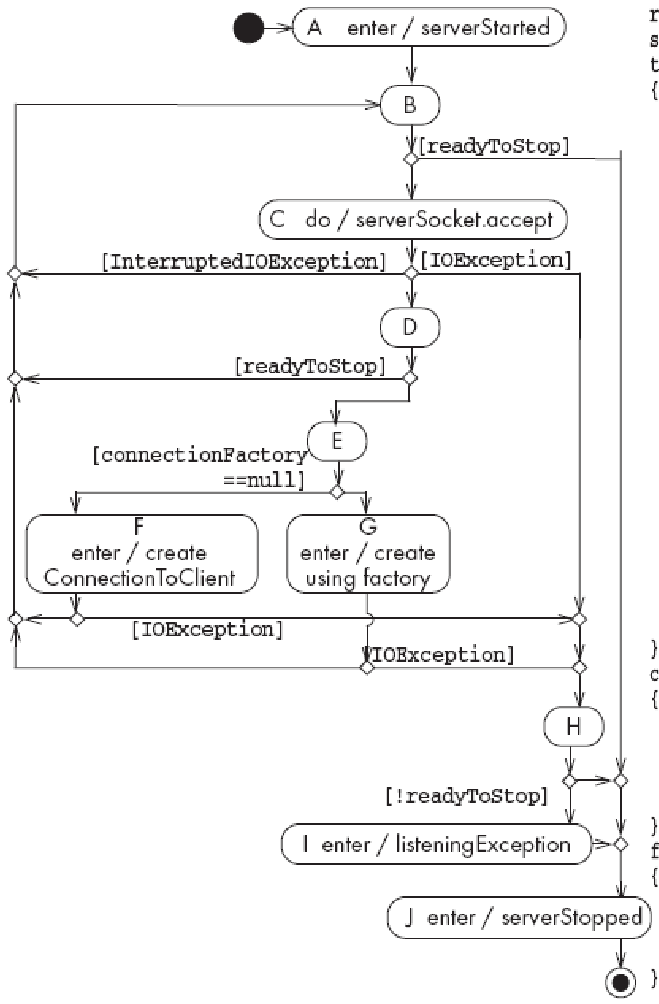
Glass-box testing

- Also called ‘white-box’ or ‘structural’ testing
- Testers have access to the system design
  - They can
    - Examine the design documents
    - View the code
    - Observe at run time the steps taken by algorithms and their internal data
  - Individual programmers often informally employ glass-box testing to verify their own code

Flow graph for glass-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 glass-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 classes

- It is inappropriate to test by *brute force*, using *every possible* input value
  - Takes a huge amount of time
  - Is impractical
  - Is pointless!

- You should divide the possible inputs into groups which you believe will be treated similarly by all algorithms.
  - Such groups are called *equivalence classes*.
  - A tester needs only to run one test per equivalence class
  - The tester has to
    - understand the required input,
    - appreciate how the software may have been designed

Examples of equivalence classes

- Valid input is a month number (1-12)
  - Equivalence classes are: [-∞..0], [1..12], [13.. ∞]

- Valid input is one of ten strings representing a type of fuel
  - Equivalence classes are
    - 10 classes, one for each string
    - A class representing all other strings

Combinations of equivalence classes

- Combinatorial explosion means that you cannot realistically test every possible system-wide equivalence class.
  - If there are 4 inputs with 5 possible values there are 5^4 (i.e. 625) possible system-wide equivalence classes.
- You should first make sure that at least one test is run with every equivalence class of every individual input.
- You should also test all combinations where an input is likely to *affect the interpretation* of another.
- You should test a few other random combinations of equivalence classes.

Example equivalence class combinations

- One valid input is either ‘Metric’ or ‘US/Imperial’
  - Equivalence classes are:
    - Metric, US/Imperial, Other
- Another valid input is maximum speed: 1 to 750 km/h or 1 to 500 mph
  - Validity depends on whether metric or US/imperial
  - Equivalence classes are:
    - [-∞..0], [1..500], [501..750], [751.. ∞]
- Some test combinations
    - Metric, [1..500]
    - US/Imperial, [1..500]
    - Metric, [501..750]

Testing at boundaries of equivalence classes

- More errors in software occur at the boundaries of equivalence classes
- The idea of equivalence class testing should be expanded to specifically test values at the extremes of each equivalence class
  - E.g. The number 0 often causes problems

- *E.g.*: If the valid input is a month number (1-12)
  - Test equivalence classes as before
  - Test 0, 1, 12 and 13 as well as very large positive and negative values

Detecting specific categories of defects

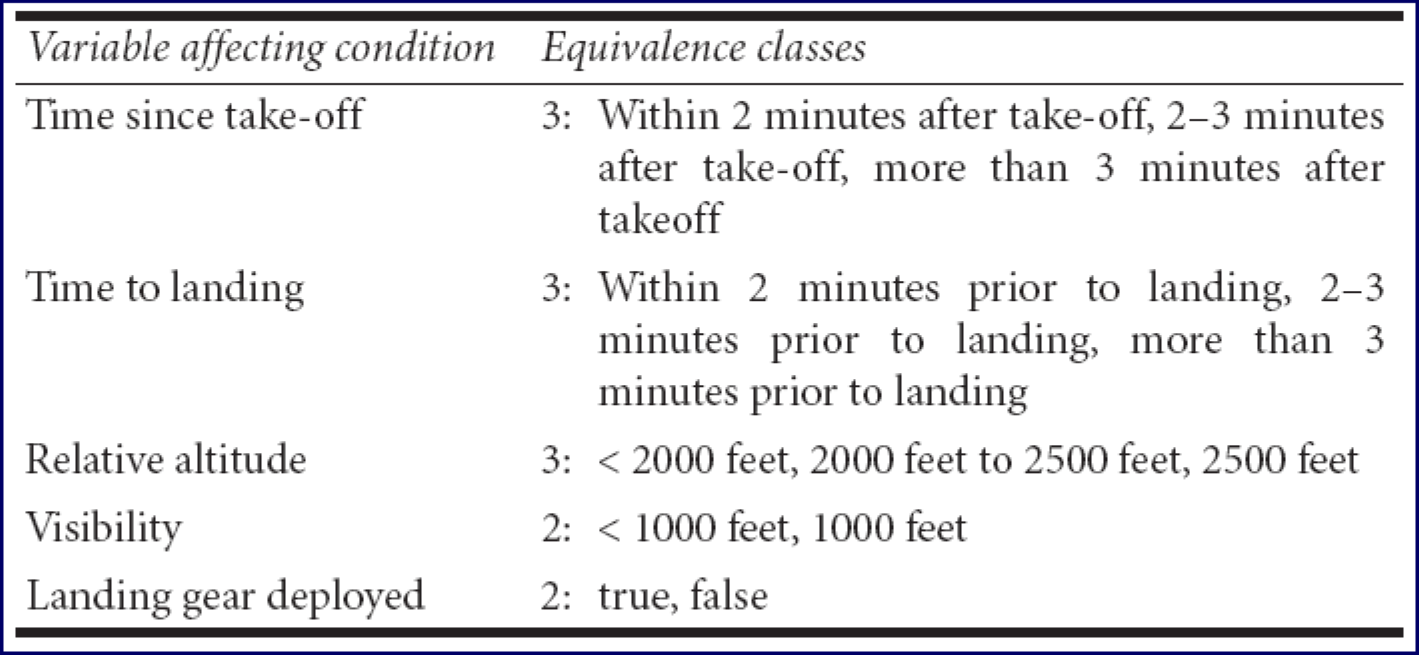
- A tester must try to uncover any defects the other software engineers might have introduced.
  - This means designing tests that explicitly try to catch a range of specific types of defects that commonly occur

Defects in Ordinary Algorithms

1) 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 logical conditions

– 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



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

Defects in Ordinary Algorithms

2) 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.

Defects in Ordinary Algorithms

3) Not terminating a loop or recursion

- *Defect*:
  - A loop or a recursion does not always terminate, i.e. it is ‘infinite’.

■

- *Testing strategies*:
  - Analyse what causes a repetitive action to be stopped.
  - Run test cases that you anticipate might not be handled correctly.

Defects in Ordinary Algorithms

4) 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

5) 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

6) 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

7) 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.

Defects in Ordinary Algorithms

8) 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

9) 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 search or sort that is case sensitive when it should not be, or vice versa

Defects in Numerical Algorithms

10) Not using enough bits or digits

– *Defect*:
  - A system does not use variables capable of representing the largest values that could be stored.
  - When the capacity is exceeded, an unexpected exception is thrown, or the data stored is incorrect.

■

– *Testing strategies*:
  - Test using very large numbers to ensure the system has a wide enough margin of error.

Defects in Numerical Algorithms

11) Not using enough places after the decimal point or significant figures

- *Defects*:
  - A floating point value might not have the capacity to store enough significant figures.
  - A fixed point value might not store enough places after the decimal point.
  - A typical manifestation is excessive rounding.
- *Testing strategies*:
  - Perform calculations that involve many significant figures, and large differences in magnitude.
  - Verify that the calculated results are correct.

Defects in Numerical Algorithms

12) Ordering operations poorly so errors build up

- *Defect*:
  - A large number does not store enough significant figures to be able to accurately represent the result.
- *Testing strategies*:
  - Make sure the program works with inputs that have large positive and negative exponents.
  - Have the program work with numbers that vary a lot in magnitude.
    - Make sure computations are still accurately performed.

Defects in Numerical Algorithms

13) Assuming a floating point value will be exactly equal to some other value

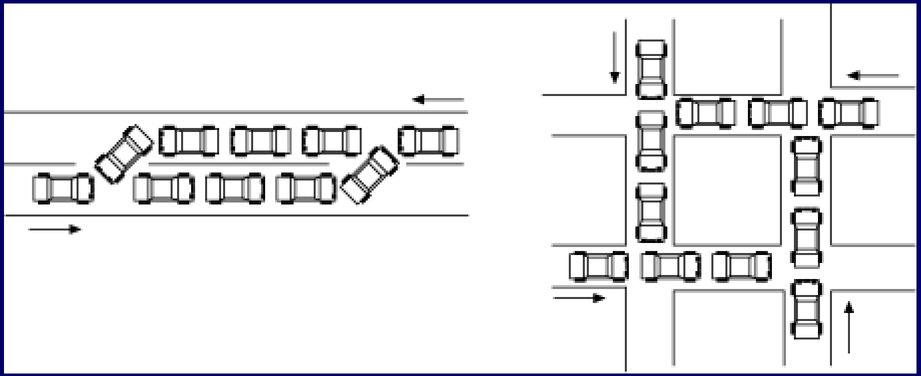
- *Defect*:
  - If you perform an arithmetic calculation on a floating point value, then the result will very rarely be computed exactly.
  - To test equality, you should always test if it is within a small range around that value.
- *Testing strategies*:
  - Standard boundary testing should detect this type of defect.

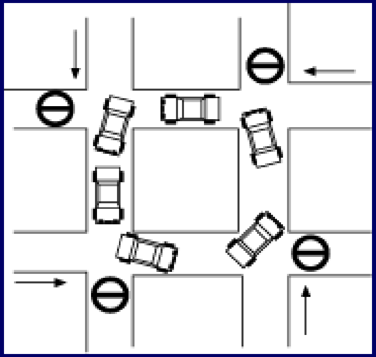
Defects in Timing and Co-ordination

1) 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.

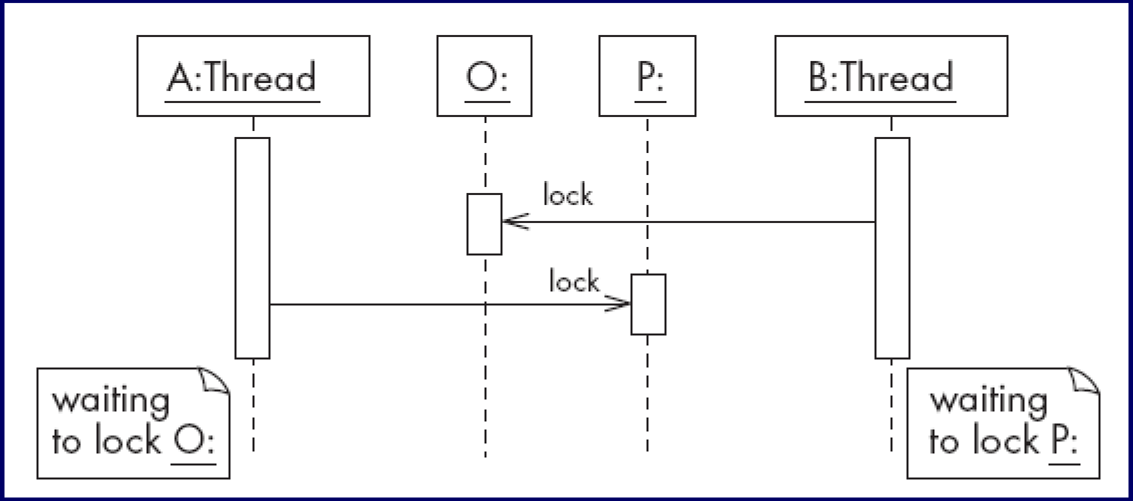




Defects in Timing and Co-ordination

- 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

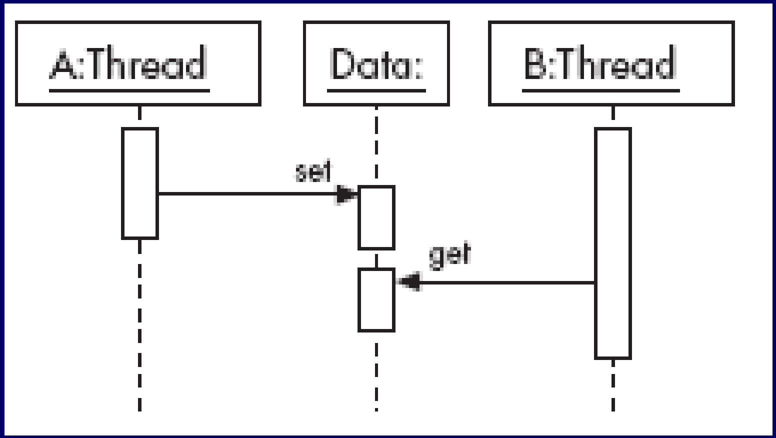


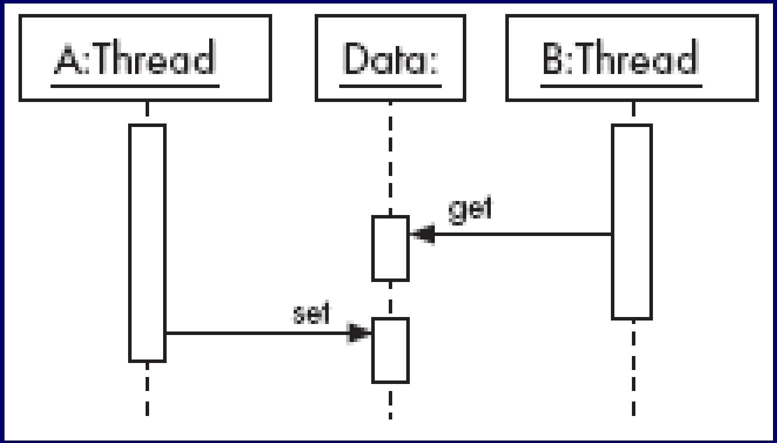
Defects in Timing and Co-ordination

2) 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





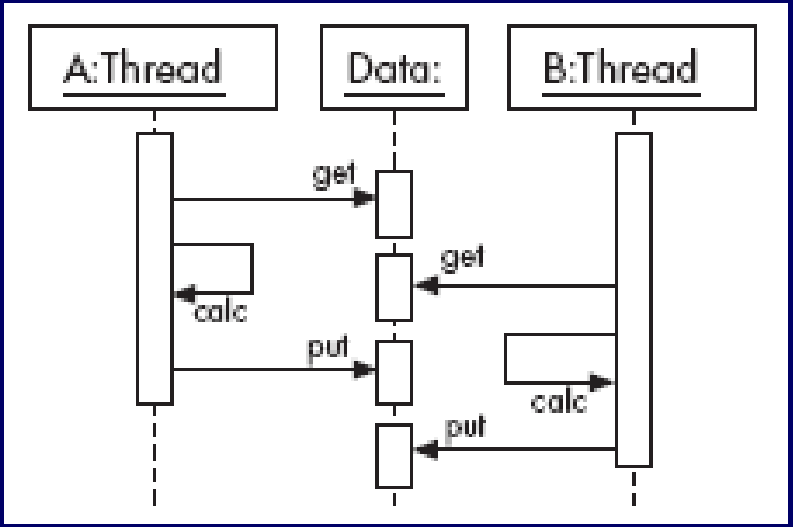
a) Normal

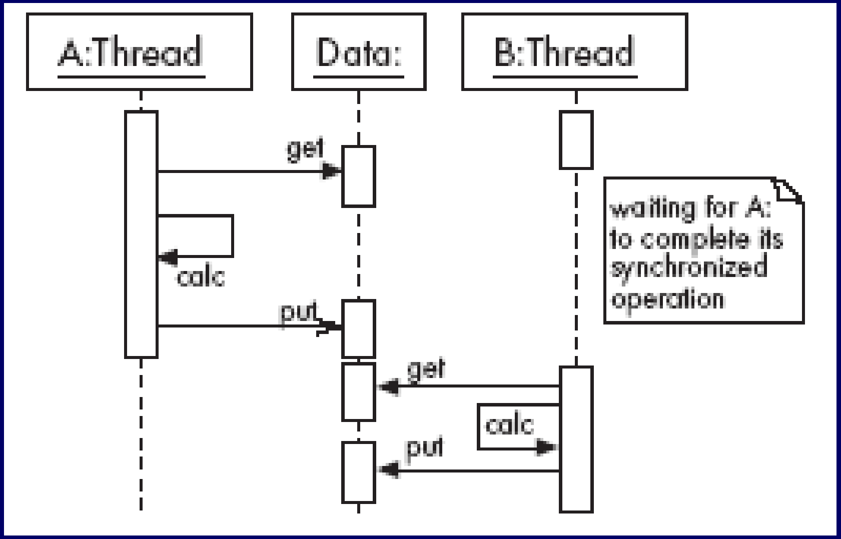
b) Abnormal due to delay in thread A

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

Defects in Handling Stress and Unusual Situations

1) 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

2) 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

3) 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

4) 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

5) 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.

Documentation defects

- ***Defect***:
  - The software has a defect if the user manual, reference manual or on-line help:
    - gives incorrect information
    - fails to give information relevant to a problem.
- ***Testing strategy***:
  - Examine all the end-user documentation, making sure it is correct.
  - Work through the use cases, making sure that each of them is adequately explained to the user.

Writing Formal Test Cases and Test Plans

- A *test case* is an explicit set of instructions designed to detect a particular class of defect in a software system.
  - A test case can give rise to many tests.
  - Each test is a particular running of the test case on a particular version of the system.

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.

Detailed Example: Test cases for Phase 2 of the SimpleChat

General Setup for Test Cases in the 2000 Series
System: SimpleChat/OCSF   Phase: 2

Instructions:

1. Install Java, minimum release 1.2.0, on Windows 95, 98 or ME.
2. Install Java, minimum release 1.2.0, on Windows NT or 2000.
3. Install Java, minimum release 1.2.0, on a Solaris system.
4. Install the SimpleChat - Phase 2 on each of the above platforms.

Test cases for Phase 2 of the SimpleChat

Test Case 2001
System: SimpleChat		Phase: 2
Server startup check with default arguments
Severity: 1

Instructions:
1. At the console, enter: java EchoServer.

Expected result:
1. The server reports that it is listening for clients by displaying the following message:

```
Server listening for clients on port 5555
```

2. The server console waits for user input.

Cleanup:
1. Hit CTRL+C to kill the server.

Test cases for Phase 2 of the SimpleChat

Test Case 2002
System: SimpleChat Phase: 2
Client startup check without a login
Severity: 1

Instructions:
1. At the console, enter: java ClientConsole.

Expected result:
1. The client reports it cannot connect without a login by displaying:
   ERROR - No login ID specified. Connection aborted.
2. The client terminates.

Cleanup: (if client is still active)
1. Hit CTRL+C to kill the client.

Test cases for Phase 2 of the SimpleChat

Test Case 2003
System: SimpleChat Phase: 2
Client startup check with a login and without a server
Severity: 1

Instructions:
1. At the console, enter: java ClientConsole <loginID>
 where <loginID> is the name you wish to be identified by.

Expected result:
1. The client reports it cannot connect to a server by displaying:
 Cannot open connection. Awaiting command.
2. The client waits for user input

Cleanup: (if client is still active)
1. Hit CTRL+C to kill the client.

Test cases for Phase 2 of the SimpleChat

Test Case 2007
System: SimpleChat Phase: 2
Server termination command check
Severity: 2

Instructions:
1. Start a server (Test Case 2001 instruction 1) using default arguments.
2. Type #quit into the server's console.

Expected result:
1. The server quits.

Cleanup (If the server is still active):
1. Hit CTRL+C to kill the server.

Test cases for Phase 2 of the SimpleChat

Test Case 2013
System: SimpleChat Phase: 2
Client host and port setup commands check
Severity: 2

Instructions:
1. Start a client without a server (Test Case 2003).
2. At the client's console, type #sethost <newhost> where <newhost> is the name of a computer on the network
3. At the client's console, type #setport 1234.

Expected result:
1. The client displays
Host set to: <newhost>
Port set to: 1234.

Cleanup:
1. Type #quit to kill the client.

Test cases for Phase 2 of the SimpleChat

Test Case 2019
System: SimpleChat Phase: 2
Different platform tests
Severity: 3

Instructions:
1. Repeat test cases 2001 to 2018 on Windows 95, 98, NT or 2000, and Solaris

Expected results:
1. The same as before.

Determining test cases by enumerating attributes

- It is important that the test cases test every aspect of the requirements.
  - Each detail in the requirements is called an *attribute*.
    - An attribute can be thought of as something that is testable.
    - A good first step when creating a set of test cases is to *enumerate* the attributes.
    - A way to enumerate attributes is to circle all the important points in the requirements document.
  - However there are often many attributes that are *implicit*.

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.

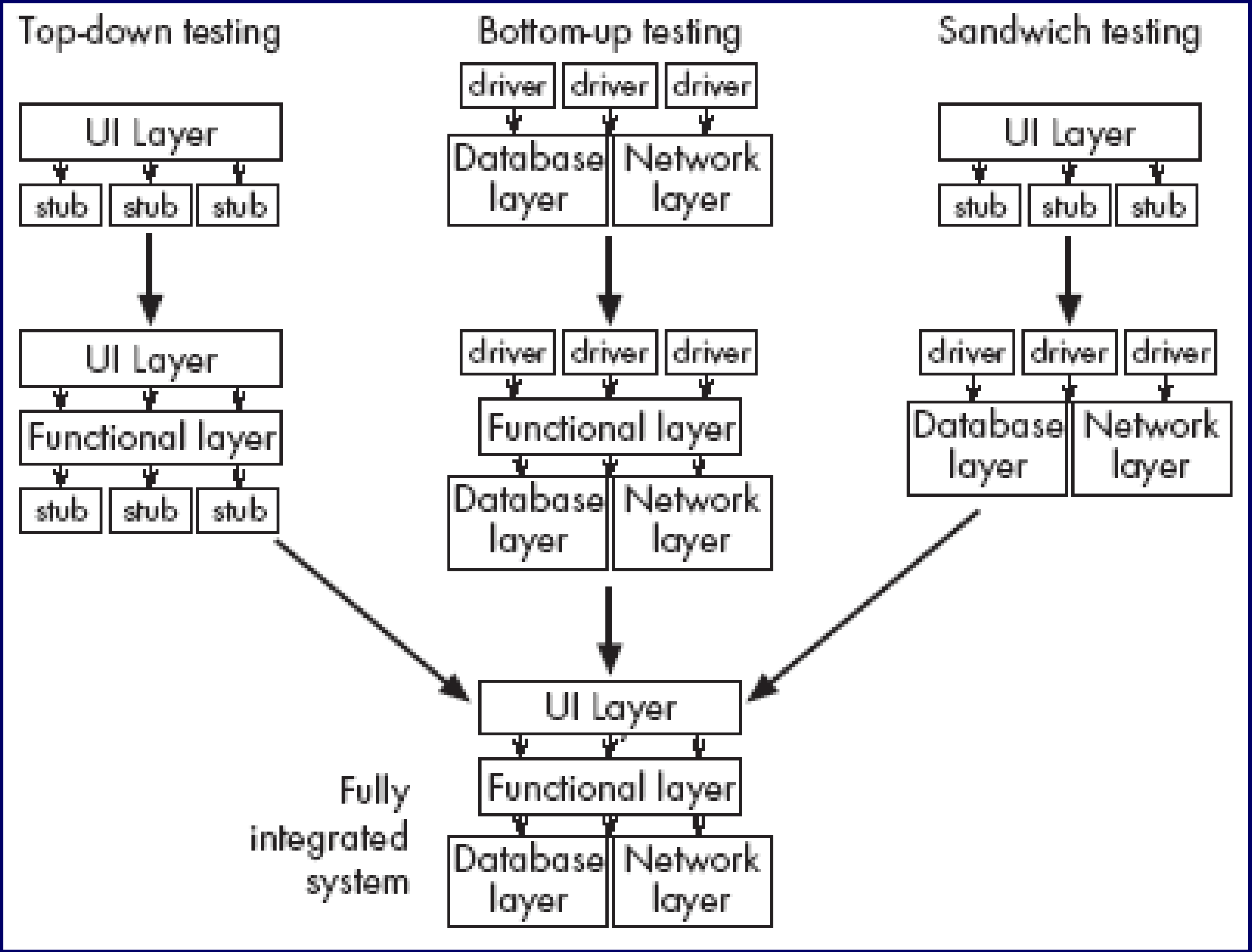
Bottom-up testing

- 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.

Sandwich testing

- Sandwich testing is a hybrid between bottom-up and top down testing.
- Test the user interface in isolation, using stubs.
- Test the very lowest level functions, using drivers.
- When the complete system is integrated, only the middle layer remains on which to perform the final set of tests.

Vertical strategies for incremental integration testing



The test-fix-test cycle

- When a failure occurs during testing:
  - Each failure report is entered into a failure tracking system.
  - It is then screened and assigned a priority.
  - Low-priority failures might be put on a *known bugs list* that is included with the software’s *release notes*.
  - Some failure reports might be merged if they appear to result from the same defects.
  - Somebody is assigned to investigate a failure.
  - That person tracks down the defect and fixes it.
  - Finally a new version of the system is created, ready to be tested again.

The ripple effect

- There is a high probability that the efforts to remove the defects may have actually added new defects
  - The maintainer tries to fix problems without fully understanding the ramifications of the changes
  - The maintainer makes ordinary human errors
  - The system *regresses* into a more and more failure-prone state

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***

Deciding when to stop testing

- All of the level 1 test cases must have been successfully executed.
- Certain pre-defined percentages of level 2 and level 3 test cases must have been executed successfully.
- The targets must have been achieved and are maintained for at least two cycles of 'builds'.
    - A *build* involves compiling and integrating all the components.
    - Failure rates can fluctuate from build to build as:
        - Different sets of regression tests are run.
        - New defects are introduced.

The roles of people involved in testing

- The first pass of unit and integration testing is called *developer testing*
  - Preliminary testing performed by the software developers who do the design.

- *Independent testing* is performed by a separate group.
  - They do not have a vested interest in seeing as many test cases pass as possible.
  - They develop specific expertise in how to do good testing, and how to use testing tools.

Testing performed by users and clients

- *Alpha testing*
  - Performed by the user or client, but under the supervision of the software development team.
- *Beta testing*
  - Performed by the user or client in a normal work environment.
  - Recruited from the potential user population.
  - An *open beta release* is the release of low-quality software to the general population.
- *Acceptance testing*
  - Performed by users and customers.
  - However, the customers do it on their own initiative.

Inspections

- An inspection is an activity in which one or more people systematically
  - Examine source code or documentation, looking for defects.
  - Normally, inspection involves a meeting...
    - Although participants can also inspect alone at their desks.

Roles on inspection teams

- The *author*
- The *moderator*.
  - Calls and runs the meeting.
  - Makes sure that the general principles of inspection are adhered to.
- The *secretary*.
  - Responsible for recording the defects when they are found.
  - Must have a thorough knowledge of software engineering.
- *Paraphrasers*.
  - Step through the document explaining it in their own words.

Principles of inspecting

1) Inspect the most important documents of all types
   - code, design documents, test plans and requirements
2) Choose an effective and efficient inspection team
   - between two and five people
   - Including experienced software engineers
3) Require that participants prepare for inspections
   - They should study the documents prior to the meeting and come prepared with a list of defects
4) Only inspect documents that are ready
   - Attempting to inspect a very poor document will result in defects being missed

Principles of inspecting

5) Avoid discussing how to fix defects
  - Fixing defects can be left to the author

6) Avoid discussing style issues
  - Issues like are important, but should be discussed separately

7) Do not rush the inspection process
  - A good speed to inspect is
    - 200 lines of code per hour (including comments)
    - or ten pages of text per hour

Principles of inspecting

8) Avoid making participants tired
   - It is best not to inspect for more than two hours at a time, or for more than four hours a day

9) Keep and use logs of inspections
   - You can also use the logs to track the quality of the design process

10) Re-inspect when changes are made
   - You should re-inspect any document or code that is changed more than 20%

A peer-review process

- Managers are normally not involved
  - This allows the participants to express their criticisms more openly, not fearing repercussions
  - The members of an inspection team should feel they are all working together to create a better document
  - Nobody should be blamed