System and Acceptance Tests

CITS 1220 Software Engineering
Lecture Overview

- White and black box testing
- Types of test: unit, integration, system, acceptance
- Formal test plans and test cases
- Supplement: defects and how to find them
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 using the fewest possible tests

  - 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*. 
Testing Approaches

- Black Box
- White Box
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
Black box testing

- takes an external perspective of the test object to derive test cases.
- Tests can be functional or non-functional, though usually functional.
- Test designer selects valid and invalid input and determines the correct output.
- There is no knowledge of the test object's internal structure.
Black box techniques

- Equivalence partitioning
- Boundary value analysis
- State transition tables
- Use case testing
White Box 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
- Also called glass-box or structured testing
Code coverage (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)
try
  
  while(!readyToStop) // B
  
  try
    
    synchronized(this)
    
    Socket clientSocket = serverSocket.accept(); // C
    
    if (!readyToStop) // D
      
      if (connectionFactory == null) { // E
        new ConnectionToClient( // F
          this.clientThreadGroup, clientSocket, this);
      } else {
        connectionFactory.createConnection( // G
          this.clientThreadGroup, clientSocket, this);
      }
    
    catch (InterruptedException exception) { }
  
  catch (IOException exception)
  
    if (!readyToStop) // H
      
      listeningException(exception); // I
  
  finally
    
    readyToStop = true; // J
    connectionListener = null;
    serverStopped();

I enter / serverStopped
Types of Test

- Unit Tests
- Integration Tests
- System Tests
- Acceptance Tests
Unit Testing

- **unit testing** is a procedure used to validate that individual units of **source code** are working properly. A unit is the smallest testable part of an application. In **object-oriented programming**, the smallest unit is a **method**; which may belong to a base/super class, abstract class or derived/child class.

- Ideally, each **test case** is independent from the others; **mock objects** and **test harnesses** can be used to assist testing a module in isolation. Unit testing is typically done by **developers** and not by **end-users**.

- JUnit is a widely used tool for automating unit testing (see lecture 4 and workshop 4)
Integration Testing

- **Integration testing** is the phase of software testing in which individual software modules are combined and tested as a group.
- It follows **unit testing** and precedes **system 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**.
Purpose of Integration Testing

- is to verify functional, performance and reliability requirements placed on major design items.
- These "design items" (or groups of units), are exercised through their interfaces using black box testing
- The overall idea is a "building block" approach, in which verified assemblages are added to a verified base which is then used to support the integration testing of further assemblages.
System Tests

- are tests conducted on a complete, integrated system to evaluate the system's compliance with its specified requirements.

- System testing falls within the scope of black box testing: requires no knowledge of the inner design of the code or logic.
Acceptance Testing

- is black-box testing performed on a system prior to its delivery.

- Also known as functional testing, black-box testing, release acceptance, QA testing, application testing, confidence testing, final testing, validation testing, usability testing, or factory acceptance testing.

- Acceptance testing performed by the customer is known as beta testing, user acceptance testing (UAT), end user testing, site (acceptance) testing, or field (acceptance) testing.
User Acceptance Testing (UAT)

- A process to obtain confirmation by a Subject Matter Expert (SME), preferably the owner or client of the object under test, through trial or review, that the modification or addition meets mutually agreed-upon requirements.

- In software development, UAT is one of the final stages of a project and often occurs before a client or customer accepts the new system.
UATs (cont)

- Users of the system perform UATs, which developers derive from the client's contract or the user requirements specification.
- The results of UATs give confidence to the clients as to how the system will perform in production.
- UATs may also be a legal or contractual requirement for acceptance of the system.
Agile Acceptance Testing

- is the functional testing of a user story by the software development team during the implementation phase.
- The customer specifies scenarios to test when a user story has been correctly implemented.
- A story can have one or many acceptance tests, whatever it takes to ensure the functionality works. Acceptance tests are black box system tests.
- Each acceptance test represents some expected result from the system.
AAT (cont)

- Customers are responsible for verifying the correctness of the acceptance tests and reviewing test scores to decide which failed tests are of highest priority.
- Acceptance tests are also used as regression tests prior to a production release.
- A user story is not considered complete until it has passed its acceptance tests.
- This means that new acceptance tests must be created each iteration or the development team will report zero progress.
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.
Formal test case information

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.
Lecture Supplement

- detecting specific categories of defects
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

- 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.
e.g. 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

<table>
<thead>
<tr>
<th>Variable affecting condition</th>
<th>Equivalence classes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Time since take-off</td>
<td>3: Within 2 minutes after take-off, 2–3 minutes after take-off, more than 3 minutes after takeoff</td>
</tr>
<tr>
<td>Time to landing</td>
<td>3: Within 2 minutes prior to landing, 2–3 minutes prior to landing, more than 3 minutes prior to landing</td>
</tr>
<tr>
<td>Relative altitude</td>
<td>3: &lt; 2000 feet, 2000 feet to 2500 feet, 2500 feet</td>
</tr>
<tr>
<td>Visibility</td>
<td>2: &lt; 1000 feet, 1000 feet</td>
</tr>
<tr>
<td>Landing gear deployed</td>
<td>2: true, false</td>
</tr>
</tbody>
</table>
Example of incorrect logical conditions defect

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

```java
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

```c
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**: Analyse 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

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

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

Use **Iterators** to help eliminate these defects
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
Defects in Numerical Algorithms

- 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

- 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

- Ordering ops 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

- 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.
Example of defect in testing floating value equality

Bad:

```java
for (double d = 0.0; d != 10.0; d+=2.0) {...}
```

Better:

```java
for (double d = 0.0; d < 10.0; d+=2.0) {...}
```
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.