Software Testing

CEN 5035
Software Engineering

Prepared by
Stephen M. Thebaut, Ph.D.
University of Florida
Topics

- Basic concepts
- Black Box Testing Techniques
- White Box Testing Techniques
- Integration and Higher-Level Testing
Definitions of “TESTING”

- Beizer: The act of executing tests to demonstrate the correspondence between an element and its specification.
- Myers: The process of executing a program with the intent of finding errors.
Definitions of “TESTING” (cont’d)

- IEEE: The process of exercising or evaluating a system or system component by manual or automated means to verify that it satisfies specified requirements or to identify differences between expected and actual results.
Fisherman’s Dilemma

• You have 3 days for fishing and 2 lakes to choose from. Day 1 at lake X nets 8 fish. Day 2 at lake Y nets 32 fish. Which lake do you return to for day 3?

• Does your answer depend on any assumptions?
Di Lemma

- In general, the probability of the existence of more errors in a section of a program is directly related to the number of errors already found in that section.
Invalid and Unexpected Inputs

- Test cases must be written for INVALID and UNEXPECTED, as well as valid and expected, input conditions.
- In many systems, MOST of the code is concerned with input error checking and handling.
Anatomy of a Test Case

- What are the parts of a test case?
  1. a description of input condition(s)
  2. a description of expected results
- Where do “expected results” come from?
Who Should Test Your Program?

- Most people are inclined to defend what they produce – not find fault with it.
- Thus, programmers should avoid testing their own programs.
- But what if this is not possible?
When Might Testing Guarantee an Error-Free Program?

a. When branch, condition, and loop coverage are achieved
b. When dataflow testing is utilized
c. When path and compound condition coverage are achieved
d. When all combinations of all possible input and state variable values are covered
e. (None of the above.)
Exhaustive Testing is Exhausting!

- **Situation:**
  - A module has 2 input parameters.
  - Word size is 32 bits.
  - Testing is completely automated: 100 nanoseconds are required for each test case.

- **Question:** How long would it take to test this module *exhaustively*, i.e., covering every possible combination of input values?
Exhaustive Testing is Exhausting (cont’d)

- Short Answer:
- Long Answer:
Exhaustive Testing is Exhausting (cont’d)

• Short Answer:

• Long Answer:

• Since we can’t generally test everything (i.e., test *exhaustively*), we need to weigh COST and RISK.
Testing Techniques

- **Black-Box:** Testing based solely on analysis of requirements (unit/component specification, user documentation, etc.). Also know as *functional testing*.

- **White-Box:** Testing based on analysis of internal logic (design, code, etc.). (But *expected* results still come from requirements.) Also known as *structural testing*. 
Levels or Phases of Testing

- **Unit**: testing of the smallest programmer work assignments that can reasonably be planned and tracked (e.g., function, procedure, module, object class, etc.)

- **Component**: testing a collection of units that make up a component (e.g., program, package, task, interacting object classes, etc.)

(cont’d)
Levels or Phases of Testing (cont’d)

- **Product**: testing a collection of components that make up a product (e.g., subsystem, application, etc.)
- **System**: testing a collection of products that make up a deliverable system

(cont’d)
Levels or Phases of Testing (cont’d)

- Testing usually:
  - begins with *functional* (black-box) tests,
  - is supplemented by *structural* (white-box) tests, and
  - progresses from the unit level toward the system level with one or more integration steps.
Other Types of Testing

- **Integration**: testing which takes place as sub-elements are combined (i.e., *integrated*) to form higher-level elements

- **Regression**: re-testing to detect problems caused by the adverse effects of program change

- **Acceptance**: formal testing conducted to enable the customer to determine whether or not to accept the system (acceptance criteria may be defined in a contract)

(cont’d)
Other Types of Testing (cont’d)

- **Alpha**: actual end-user testing performed within the development environment
- **Beta**: end-user testing performed within the user environment prior to general release
Waterfall Model of Testing Process

Test Planning
Test Design
Test Implementation
Test Execution
Execution Analysis
Result Documentation
Final Reporting

our focus
What Does Testing Cost?

- About 50% of the total life-cycle effort is spent on testing.
- About 50% of the total life-cycle time is spent on testing.
Costs of Errors Over Life Cycle

- The sooner an error can be found and corrected, the lower the cost.
- Costs can increase *exponentially* with time between injection and discovery.
V&V for Software Engineers

- V&V techniques have evolved considerably and require specialized knowledge, disciplined creativity, and ingenuity.
- Software engineers should be familiar with all V&V techniques, and should be able to employ (and assess the effectiveness of) those techniques appropriate to their responsibilities.
Testing-Related Vehicles for Continuous Process Improvement

- **Post-Test Analysis:** reviewing the results of a testing activity with the intent to improve its effectiveness

- **Causal Analysis:** identifying the causes of errors and approaches to eliminate future occurrences

(cont’d)
And More Generally…

- **Benchmarking**: general practice of recording and comparing indices of performance, quality, cost, etc., to help identify “best practices”
Black-Box Testing Techniques

- Partition Testing
- Combinatorial Approaches
- Boundary Value Analysis
- Intuition and Experience
Definition of Black-Box Testing

- Testing based solely on analysis of requirements (specification, user documentation, etc.).
- Also know as *functional* testing.
- Black-box testing concerns techniques for *designing* tests; it is *not* a "level" of testing.
- Black-box techniques apply to *all* levels of testing (e.g., unit, component, product, and system).
Partition Testing

• Can be thought of as “exhaustive testing Las Vegas style…”

• Idea is to partition the input space into a small number of equivalence classes such that, according to the specification, every element of a given class is “handled” (i.e., mapped to an output) “in the same manner.”

(cont’d)
Partition Testing (cont’d)

- **If** the program is implemented in such a way that being “handled in the same manner” means that either
  - every element of the class would be mapped to a correct output, or
  - every element of the class would be mapped to an incorrect output,

  **then** testing the program with just one element from each equivalence class would be tantamount to *exhaustive testing*.

  (cont’d)
Partition Testing (cont’d)

- Two types of classes are identified: *valid* (corresponding to inputs deemed valid from the specification) and *invalid* (corresponding to inputs deemed erroneous from the specification).
- Technique is also known as *input space partitioning* and *equivalence partitioning*. 
Partition Testing Example

- Program Specification:
  An ordered pair of numbers, (x, y), are input and a message is output stating whether they are in ascending order, descending order, or equal. If the input is other than an ordered pair of numbers, an error message is output.
Partition Testing Example (cont’d)

- Equivalence Classes:

  \{ (x, y) \mid x \leq y \} (V)

  \{ (x, y) \mid x > y \} (V) \quad \text{Valid classes}

  \{ (x, y) \mid x = y \} (V)

  \{ \text{input is other than an ordered pair of numbers} \} (I) \quad \text{Invalid class}
Valid \((x, y)\) Input Space

\[ x < y \quad \text{and} \quad x = y \quad \text{and} \quad x > y \]
Sample Program Design

• Conceptually, would the underlying assumption of partition testing hold for these classes if the following program design was employed?

```java
if (input is other than an ordered pair of numbers)
    then output(“invalid input”)
else
    if x<y then output(“ascending order”)
    else
        if x>y then output(“ascending order”)
        else
            output(“equal”)
```
Identifying Test Cases

- When partition testing yields a set of mutually exclusive classes that partition the input space, templates of test case inputs that would provide the desired coverage can easily be identified.

- A test case COVERAGE MATRIX is generally utilized to document this.
### A Test Case Coverage Matrix

<table>
<thead>
<tr>
<th>EQUIVALENCE CLASSES</th>
<th>TEST CASES</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1</td>
</tr>
<tr>
<td>{ (x, y) \mid x&gt;y } (V)</td>
<td>V</td>
</tr>
<tr>
<td>{ (x, y) \mid x&lt;y } (V)</td>
<td></td>
</tr>
<tr>
<td>{ (x, y) \mid x=y } (V)</td>
<td></td>
</tr>
<tr>
<td>{ other } (I)</td>
<td></td>
</tr>
</tbody>
</table>
Dealing with Complex Multiple-Input Situations

• Note that in the example above, the PAIR of $x, y$ inputs were considered as a unit, yielding a set of mutually exclusive classes that partition the joint (two-dimensional) $x, y$ input space.

(cont’d)
Dealing with Complex Multiple-Input Situations (cont’d)

- For more complex specifications, equivalence classes are often identified for INDIVIDUAL input variables, or even INDIVIDUAL ATTRIBUTES of individual input variables, yielding disjoint sets of overlapping classes.

(cont’d)
Dealing with Complex Multiple-Input Situations (cont’d)

- In such cases, a strategy for identifying appropriate COMBINATIONS of equivalence classes must be employed.
- One such strategy is known as “Cause-Effect Analysis.”
Cause-Effect Analysis

• Cause-Effect Analysis is a combinatorial approach that can be viewed as a logical extension of partition testing.

• It is a systematic means for generating test cases to cover different combinations of input “Causes” resulting in output “Effects.”
Causes and Effects

- A **CAUSE** may be thought of as a distinct input condition, or an “equivalence class” of input conditions.
- An **EFFECT** may be thought of as a distinct output condition or change in program state.

(cont’d)
Causes and Effects (cont’d)

- Causes and Effects are represented as Boolean variables.
- The logical relationships among them CAN (but need not) be represented as one or more Boolean graphs.
C-E Analysis Process Steps

1. Identify Causes and Effects
2. Deduce Logical Relationships and Constraints
3. Identify an appropriate Test Case Selection Strategy
4. Construct a Test Case Coverage Matrix
Illustration of C-E Analysis

City Tax Specification

The first input is a yes/no response to the question “Do you reside within the city?” The second input is gross pay for the year in question.

A non-resident will pay 1% of the gross pay in city tax.

Residents pay on the following scale:
- If gross pay is no more than $30,000, the tax is 1%.
- If gross pay is more than $30,000, but no more than $50,000, the tax is 5%.
- If gross pay is more than $50,000, the tax is 15%.
Boolean Graph Representation

Non-Res(1) \lor ((11) 1\% \text{ tax})

[0,30K](3)

(30K,50K](4) \land ((12) 5\% \text{ tax})

Res(2) \land ((13) 15\% \text{ tax})

>50K(5)
Boolean Graph Representation

Non-Res (1) \lor (11) \ 1\% \ tax
\ [0,30K] (3) \land (12) \ 5\% \ tax
\ (30K,50K] (4) \land (13) \ 15\% \ tax
Res (2)\land (12) \ 5\% \ tax
>50K (5)
A Test Case Selection Strategy

**REPEAT**

Select the next (initially, the first) Effect.

Tracing back through the graph (right to left), find **all feasible combinations of connected Cause values that result in the Effect being True.**

For each **new** such combination found:

- Determine values of all other Effects, and
- Enter values for each Cause and Effect in a new column of the test case coverage matrix.

**UNTIL** each Effect has been selected.
### Resulting Coverage Matrix

<table>
<thead>
<tr>
<th>TEST CASES</th>
<th>CAUSES</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Non-Resident (1)</td>
<td>T</td>
<td>T</td>
<td>F</td>
<td>F</td>
<td>F</td>
</tr>
<tr>
<td></td>
<td>Resident (2)</td>
<td>F</td>
<td>F</td>
<td>T</td>
<td>T</td>
<td>T</td>
</tr>
<tr>
<td></td>
<td>$0 \leq \text{Gross Pay} \leq $30K (3)</td>
<td>T</td>
<td>F</td>
<td>T</td>
<td>F</td>
<td>F</td>
</tr>
<tr>
<td></td>
<td>$30K &lt; \text{Gross Pay} \leq $50K (4)</td>
<td>F</td>
<td>♠</td>
<td>F</td>
<td>T</td>
<td>F</td>
</tr>
<tr>
<td></td>
<td>Gross Pay &gt; $50K (5)</td>
<td>F</td>
<td>♠</td>
<td>F</td>
<td>F</td>
<td>T</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>EFFECTS</th>
<th>1% tax (11)</th>
<th>5% tax (12)</th>
<th>15% tax (13)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>T</td>
<td>F</td>
<td>F</td>
</tr>
<tr>
<td></td>
<td>T</td>
<td>F</td>
<td>F</td>
</tr>
<tr>
<td></td>
<td>T</td>
<td>F</td>
<td>F</td>
</tr>
</tbody>
</table>

♠ don’t care, subject to Cause constraint B
Boundary Value Analysis

- A technique based on identifying, and generating test cases to explore boundary conditions.
- Boundary conditions are an extremely rich source of errors.
- Natural language based specifications of boundaries are often ambiguous, as in “for input values of X between 0 and 40,...”
Guidelines for Identifying Boundary Values

- “K will range in value from 0.0 to 4.0”
  Identify values at the endpoints of the range and just beyond.
- “The file will contain 1-25 records”
  Identify the minimum, the maximum, and values just below the minimum and above the maximum.
Test Case Design Based on Intuition and Experience

- Also known as *Error Guessing, Ad Hoc Testing, Artistic Testing*, etc.
- Testers utilize intuition and experience to identify potential errors and design test cases to reveal them.
- Can be extremely effective.
- Test plans should reflect the explicit allocation of resources for this activity.
Guidelines for identifying test cases

- Design tests for reasonable but *incorrect assumptions* that may have been made by developers.
- Design tests to detect errors in handling *special situations or cases*.
- Design tests to explore unexpected or unusual program use or environmental scenarios.
White-Box Testing Techniques

- Logic Coverage
- Path Conditions
- Program Instrumentation
Definition of White-Box Testing

- Testing based on analysis of internal logic (design, code, etc.). (But expected results still come from requirements.)
- Also know as structural testing.
- White-box testing concerns techniques for designing tests; it is not a “level” of testing.
- White-box testing techniques apply primarily to lower levels of testing (e.g., unit and component).
input(Y)
if (Y<=0) then
  \( Y := -Y \)
end_if
while (Y>0) do
  input(X)
  \( Y := Y-1 \)
end_while
Logic Coverage

• **Statement Coverage:**
  - Requires that each statement will have been executed at least once.
  - Also known as *Node Coverage*.

• **Branch Coverage:**
  - Requires that each branch will have been traversed, and that every program entry point will have been taken, at least once.
  - Also known as *Edge Coverage*. 
Logic Coverage (cont’d)

• What is the relationship between Statement and Branch Coverage?

• **Possibilities:**
  1. None.
  2. Statement Coverage *subsumes* Branch Coverage (“statement => branch”).
  4. Both (2) and (3) (i.e., they are *equivalent*)
“statement => branch”

Min. number of cases required for Statement Coverage?

Min. number of cases required for Branch Coverage?

Therefore, Statement Coverage does **NOT** subsume Branch Coverage.
“branch => statement”

- Normally, **YES.**
Logic Coverage (cont’d)

- A branch predicate may have more than one condition.

```plaintext
input(X,Y)
if (Y<=0) or (X=0) then
  Y := -Y
end_if
while (Y>0) and (not EOF) do
  input(X)
  Y := Y-1
end_while
```
Logic Coverage (cont’d)

- **Condition Coverage:**
  - Requires that each condition will have been True at least once and False at least once.

- **What is the relationship between Branch and Condition Coverage?**
if A or B then
  s1
else
  s2
end_if_then_else

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
<th>Branch</th>
</tr>
</thead>
<tbody>
<tr>
<td>test 1</td>
<td>T</td>
<td>F</td>
<td>?</td>
</tr>
<tr>
<td>test 2</td>
<td>F</td>
<td>F</td>
<td>?</td>
</tr>
</tbody>
</table>
if A or B then
  s1
else
  s2
end_if_then_else

<table>
<thead>
<tr>
<th>test 1</th>
<th>A</th>
<th>B</th>
<th>Branch</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>T</td>
<td>F</td>
<td>true</td>
</tr>
<tr>
<td>test 2</td>
<td>F</td>
<td>F</td>
<td>false</td>
</tr>
</tbody>
</table>

Branch Coverage \(\Rightarrow\) Condition Coverage
Logic Coverage (cont’d)

if A or B then
  s1
else
  s2
end_if_then_else

<table>
<thead>
<tr>
<th>test 3</th>
<th>A</th>
<th>B</th>
<th>Branch</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>T</td>
<td>F</td>
<td>?</td>
</tr>
<tr>
<td>test 4</td>
<td>F</td>
<td>T</td>
<td>?</td>
</tr>
</tbody>
</table>
Logic Coverage (cont’d)

if \( A \) or \( B \) then
  \( s1 \)
else
  \( s2 \)
end_if_then_else

|\begin{tabular}{|c|c|c|}
\hline
| \textbf{test 3} & \textbf{T} & \textbf{F} & \textbf{true} \\
\textbf{test 4} & \textbf{F} & \textbf{T} & \textbf{true} \\
\hline
\end{tabular}|  

\textbf{Condition Coverage} \n\n\n\n\n\textbf{Branch Coverage}
Logic Coverage (cont’d)

• **Branch/Condition Coverage:**
  - Requires that both Branch AND Condition Coverage will have been achieved.

• Therefore, Branch/Condition Coverage subsumes both Branch Coverage and Condition Coverage.
Logic Coverage (cont’d)

- The evaluation of conditions may be masked during testing.
- For example,

  \[ \text{if (A) or (y/x=5) then...} \]

may be compiled in such a way that if A is found to be true, 
\[ y/x=5 \] will not be evaluated.
Logic Coverage (cont’d)

• **Compound Condition Coverage:**
  - Requires that all *combinations* of condition values at every branch statement will have been covered, and that every entry point will have been taken, at least once.
  - Also know as *Multiple Condition Coverage*.
  - Subsumes Branch/Condition Coverage, regardless of the order in which conditions are evaluated.
Logic Coverage (cont’d)

Combinations of condition values: TT, TF, FT, FF

input(X,Y)
if \( Y \leq 0 \) or \( X = 0 \) then
    Y := -Y
end_if
Logic Coverage (cont’d)

• **Path Coverage:**
  - Requires that all program paths will have been traversed at least once.
  - Often described as the “strongest” form of logic coverage. (Is it stronger than Compound Condition Coverage?)
  - Usually impossible when loops are present.
Logic Coverage (cont’d)

repeat 29 times

$3 \times 3 \times \ldots \times 3 = 3^{30}$ paths
Logic Coverage (cont’d)

- Various strategies have been developed for identifying useful subsets of paths for testing when Path Coverage is impractical:
  - Loop Coverage,
  - Basis Paths Coverage, and
  - Dataflow Coverage.
Summary of Logic Coverage Relationships

- Compound Condition
- Branch / Condition
- Condition
- Branch
- Statement
- Path
Path Conditions

• With a little luck, at least some white-box coverage goals will have been met by executing test cases designed using black-box strategies.

• Designing additional test cases for this purpose involves identifying inputs that will cause given program paths to be executed. This can be difficult...

  (cont’d)
Path Conditions (cont’d)

• To cause a path to be executed requires that the test case satisfy the path condition.

• For a given path, the PATH CONDITION is the conjunction of branch predicates that are required to hold for all the branches along the path to be taken.
Consider an example…

(1) input(A,B)
    if (A>0) then
(2)    Z := A
    else
(3)    Z := 0
    end_if_else
    if (B>0) then
(4)    Z := Z+B
    end_if
(5) output(Z)

What is the path condition for path <1,2,5>?
Consider ANOTHER example…

(1) input(A, B)
    if \( A > B \) then
(2) \( B := B \times B \)
    end_if
    if \( B < 0 \) then
(3) \( Z := A \)
    else
(4) \( Z := B \)
    end_if_else
(5) output(Z)

What is the path condition for path \( <1, 2, 3, 5> \)?
Path Conditions (cont’d)

- To be useful, path conditions should be expressed in terms that reflect relevant state changes along the path.
- A path is **INFEASIBLE** if its path condition reduces to FALSE.
Program Instrumentation

- Allows for the measurement of white-box coverage during program execution.
- Code is inserted into a program to record the cumulative execution of statements, branches, du-paths, etc.
- Execution takes longer and program timing may be altered.
Integration and Higher-Level Testing

- Context
- Integration Testing
- Higher-Level Testing Issues
• Higher-level testing begins with the integration of (already unit-tested) modules to form higher-level program entities (e.g., components).

• The primary objective of integration testing is to discover interface errors among the elements being integrated.
Once the elements have been successfully integrated (i.e., once they are able to function together), the functional and non-functional characteristics of the higher-level element can be tested thoroughly (via component, product, or system testing).
Integration Testing

• Integration testing is carried out when integrating (i.e., combining):
  – Units or modules to form a component
  – Components to form a product
  – Products to form a system

• The strategy employed can significantly affect the time and effort required to yield a working, higher-level element.
Integration Testing Strategies

- An *incremental* integration strategy is employed since it can significantly reduce error localization and correction time.
- The *optimum* incremental approach is inherently dependent on the individual project and the pros and cons of the various alternatives.
Incremental Strategies

- Suppose we are integrating a group of modules to form a component, the control structure of which will form a “calling hierarchy” as shown.
Incremental Strategies (cont’d)

- The order in which modules may be integrated include:
  - From the top (“root”) module toward the bottom (“top-down”)
  - From bottom (“leaf”) modules toward the top (“bottom-up”)
  - By function
  - Critical or high-risk modules first
  - By availability

(cont’d)
Incremental Strategies (cont’d)

- *Scaffolding* (in the form of *drivers* and/or *stubs* to exercise the modules, and *oracles* to inspect test results) will be required.
Top-Down Strategy
Top-Down Strategy (cont’d)

driver

A

B
stub

C
stub

stub
Top-Down Strategy (cont’d)

driver

A

B stub
C stub
D stub

stub

E F G H I J K L
Top-Down Strategy (cont’d)
Top-Down Strategy (cont’d)
Top-Down Strategy (cont’d)

driver

A

B

C

D

stub

stub

stub

stub

E

F

G

H

I

J

K

L

A

B

C

D

E

F

G

H

I

J

K

L
Top-Down Strategy (cont’d)

The diagram illustrates a top-down strategy with a driver node at the top, labeled 'A'. From 'A', branches lead to 'B', 'C', and 'D'. Further branching occurs with 'E', 'F', 'G', and 'H'. Each node is labeled with a letter representing a component or sub-component of the strategy. The diagram also includes stubs, indicating placeholders for additional components. The layout suggests a hierarchical structure, with the driver node at the root and subsequent levels of the strategy represented by the branches.
Top-Down Strategy (cont’d)
Top-Down Strategy (cont’d)

driver

A

B

C

D

E

F

G

H

I

J

stub

stub

A

B

C

D

E

F

G

H

I

J

K

L
Top-Down Strategy (cont’d)

driver
A
B
C
D
E
F
G
H
I
J
K
stub

A
B
C
D
E
F
G
H
I
J
K
L
Top-Down Strategy (cont’d)

driver

A

B
E
J

C
F

D
G
H
K

I

J

A
B
C
D
E
F
G
H
I
J
K
L

K
L
Bottom-Up Strategy

driver

F
J

A
B
C
D
E
F
G
H
I
J
K
L
Bottom-Up Strategy (cont’d)

driver

E  F  J
Bottom-Up Strategy (cont’d)

driver

```
   B
  / \  
 E   F
    /   
   J
```

```
A
 / 
B---C
 |   |
 |   D
 |   |
 |   G
 |   |
 |   H
 |   |
 |   I
```

E F G H I J K L
Bottom-Up Strategy (cont’d)
Bottom-Up Strategy (cont’d)
Bottom-Up Strategy (cont’d)
Bottom-Up Strategy (cont’d)
Bottom-Up Strategy (cont’d)
Bottom-Up Strategy (cont’d)

Driver

B

E

F

J

C

G

K

D

H

I

L

A

B

C

D

E

F

G

H

I

J

K

L
Bottom-Up Strategy (cont’d)
Bottom-Up Strategy (cont’d)
Bottom-Up Strategy (cont’d)
Bottom-Up Strategy (cont’d)
Bottom-Up Strategy (cont’d)

driver

A

B

C

D

E

F

G

H

I

J

K

L

A

B

C

D

E

F

G

H

I

J

K

L
Higher-Level Testing

- Higher-level tests focus on the core functionality specified for higher level elements, and on certain emergent properties that become more observable as testing progresses toward the system level.
- The black-box testing strategies already considered (e.g., partition and combinatorial approaches) apply to functional testing at any level.

(cont’d)
Higher-Level Testing (cont’d)

• Higher-level testing related to emergent system properties may include:
  – Usability test
  – Installability test
  – Serviceability test
  – Performance test
  – Stress test
  – Security test
  – Software compatibility test
  – Device and configuration test
  – Recovery test
  – Reliability test

• We briefly consider just three of these.
Installability Test

• Focus is functional and non-functional requirements related to the installation of the product/system.

• Coverage includes:
  – Media correctness and fidelity
  – Relevant documentation (including examples)
  – Installation processes and supporting system functions.

(cont’d)
Installability Test (cont’d)

- Functions, procedures, documentation, etc., associated with product/system decommissioning must also be tested.
Stress Test

- Focus is system behavior at or near overload conditions (i.e., “pushing the system to failure”).
- Often undertaken with performance testing.
- In general, products should exhibit “graceful” failures and non-abrupt performance degradation.
Reliability Test

• Requirements may be expressed as:
  – the probability of no failure in a specified time interval, or as
  – the expected mean time to failure.

• Appropriate interpretations for failure and time are critical.

• Utilizes Statistical testing based on an operational profile.
Topics

- Basic concepts
- Black Box Testing Techniques
- White Box Testing Techniques
- Integration and Higher-Level Testing
Software Testing

CEN 5035

Software Engineering

Prepared by
Stephen M. Thebaut, Ph.D.

University of Florida