FACULDADE ENGENHARIA
Departamento de Informática

Black Box Testing#2

(adapted from lecture notes of the CSCI 3060U - Software Quality Assurance unit, J.S. Bradbury, J.R. Cordy, 2018, and lecture notes of the LU - Software Testing unit, L. Buffoni, 2017)

Testing Methods : Black Box Testing

Last Lecture

- We started with black box testing and looked at:
  - Black box method 1:
    Systematic functionality coverage testing
  - Black box method 2:
    Systematic input coverage testing

Testing Methods : Black Box Testing

Last Lecture: Functional Specifications

- Functional specifications can be formal (mathematical), or more often informal (in a natural language such as English)
- In either case, the functional specification usually contains at least three kinds of information:
    1. the intended inputs
    2. the corresponding intended actions
    3. the corresponding intended outputs
- Focusing on each one of these separately gives us three different black box systems for testing

Testing Methods : Black Box Testing

Last Lecture: Functionality Coverage Methods

- Functionality coverage partitions the functional specification into separate requirements to test
- Isolate causes by keeping test input values simple and varying one input value at a time

Testing Methods : Black Box Testing

Departamento de Informática

Last Lecture: Input Coverage Methods

- Exhaustive testing is usually impractical, but we can approximate it using input partitioning
- Shotgun testing can be added to input partitioning to give additional confidence
- Robustness testing checks for crashes on unexpected or unusual input, such as the boundaries of the input range

Testing Methods : Black Box Testing

Overview

- Today we look at the third kind of black box method, output coverage testing, and begin to consider the role of black box methods in unit and integration testing
- We'll look at:
  - Exhaustive output testing
  - Output partitioning
  - State transition testing
  - Cause-and-Effect graphing
  - Error guessing
- Testing multiple input or output streams
- Black box testing at the unit and integration levels ("gray box" testing)

FACULDADE ENGENHARIA
Departamento de Informática

Output Coverage Testing

Output Coverage

- The third kind of black box testing
- Idea: Analyze all the possible outputs specified in the functional specification (requirements), create tests to cause each one
- More difficult than input coverage: must analyze requirements to figure out what input is required to produce each output – this can be a complex and time consuming analysis
- But can be very effective in finding problems, because it requires a deep understanding of the requirements

Output Coverage Testing

Different from Input Coverage

- Output coverage testing is definitely different from input coverage

Example: suppose the requirements say: *"Output 1 if two input integers are equal, 0 otherwise"*

This specification allows two integer inputs - so if we do input partitioning, then we have the test cases:

- numbers equal
- numbers not equal
- first number zero / positive / negative
- second number zero / positive / negative

Whereas we can do exhaustive output testing with only two test cases – output 1 & output 0

FACULDADE ENGENHARIA
Departamento de Informática

Exhaustive Output Testing

More Practical than Input

- Exhaustive output testing makes one test for every possible output
- Practical more often than exhaustive input testing, because programs are often written to reduce or summarize input data (like the previous example)
- But still impractical in general - most programs have an infinite number of different possible outputs

Output Partition Testing

Output Partitioning

- Output partitioning is like input partitioning, only we analyze the possible outputs
- In a fashion similar to input partitioning, we partition all the possible outputs into a set of equivalence classes which have something in common

FACULDADE ENGENHARIA
Departamento de Informática

Output Partition Testing

Example: in our gcd spec, the output is a list of integers – so we might partition into the following cases:

	number of integers in output		
values	one	two	many
all zero	P1	P2	P3
some zero	P4	P5	P6
all positive	P7	P8	P9
all negative	P10	P11	P12
mixed	P13	P14	P15

FACULDADE ENGENHARIA
Departamento de Informática

Output Partition Testing

Designing Inputs

- Once we have the output partitions, we must design inputs to cause outputs in each class
- This is a difficult and time consuming task - the biggest drawback to output coverage testing
- Sometimes, we discover that we cannot find such an input – this implies an error or oversight in either the requirements or in the partition analysis

State Transition Testing

- State transition testing is based on the concepts of states and finite-state machines, and allows the tester to view the developing software in term of its states, transitions between states, and the inputs and events that trigger state changes.

- A state is an internal configuration of a system or component. It is defined in terms of the values assumed at a particular time for the variables that characterize the system or component.
- A finite-state machine is an abstract machine that can be represented by a state graph having a finite number of states and a finite number of transitions between states.

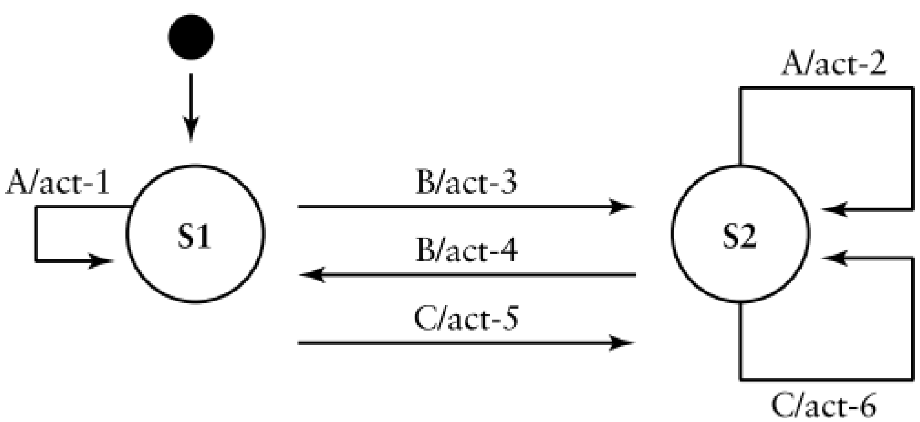
FACULDADE ENGENHARIA
Departamento de Informática

State Transition Testing

- During the specification phase a state transition graph (STG) may be generated for the system as a whole and/or specific modules. In object-oriented development the graph may be called a state chart.
- STG/state charts are commonly depicted by a set of nodes (circles, ovals, rounded rectangles) which represent states. These usually will have a name or number to identify the state.
- A set of arrows between nodes indicate what inputs or events will cause a transition or change between the two linked states.
- Outputs/actions occurring with a state transition are also depicted on a link or arrow.

State Transition Testing – Example

- The black dot represents a pointer to the initial state from outside the machine
- S1 and S2 are the two states
- Test cases: *Input A in S1, Input A in S2, Input B in S1, Input B in S2, Input C in S1, Input C in S2*
- The arrows display inputs/actions that cause the state transformations in the arrow directions. (e.g. the transition from S1 to S2 occurs with input, or event B. Action 3 occurs as part of this state transition. This is represented by the symbol "B/act3.")



FACULDADE ENGENHARIA
Departamento de Informática

State Transition Testing

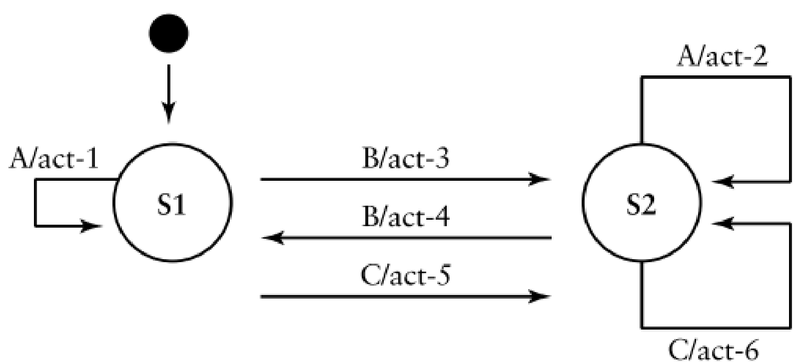
- For large systems and system components, state transition graphs can become very complex. Developers can nest them to represent different levels of abstraction.
- This approach allows the STG developer to group a set of related states together to form an encapsulated state that can be represented as a single entity on the original STG.
- The STG developer must ensure that this new state has the proper connections to the unchanged states from the original STG.
- Another way to simplify the STG is to use a state table representation which may be more concise.

FACULDADE ENGENHARIA
Departamento de Informática

State Transition Testing

- Following is presented the state table for the provided example:

Inputs	S1	S2
Input A	S1 (act-1)	S2 (act-2)
Input B	S2 (act-3)	S1 (act-4)
Input C	S2 (act-5)	S2 (act-6)



Departamento de Informática

Cause-and-Effect Graphing

- A major weakness with equivalence class partitioning is that it does not allow testers to combine conditions. (in spite of combinations can be covered in some cases by test cases generated from the classes).

- Cause-and-effect graphing is a technique that can be used to combine conditions and derive an effective set of test cases that may disclose inconsistencies in a specification.

- The specification must be transformed into a graph that resembles a digital logic circuit (so, knowledge of Boolean logic is recommended).

Cause-and-Effect Graphing

- Developing the graph, especially for a complex module with many combinations of inputs, is difficult and time consuming.

- The graph must be converted to a decision table that the tester uses to develop test cases.

- One advantage of this method is that development of the rules and the graph from the specification allows a thorough inspection of the specification.

Departamento de Informática

Cause-and-Effect Graphing – Design Test Cases

1. The tester must decompose the specification of a complex software component into lower-level units.

2. For each specification unit, the tester needs to identify causes and their effects.

   – **Cause**: is a distinct input condition or an equivalence class of input conditions.
   – **Effect:** is an output condition or a system transformation.

Putting together a table of causes and effects helps the tester to record the necessary details. The logical relationships between the causes and effects should be determined. It is useful to express these in the form of a set of rules.

FACULDADE ENGENHARIA
Departamento de Informática

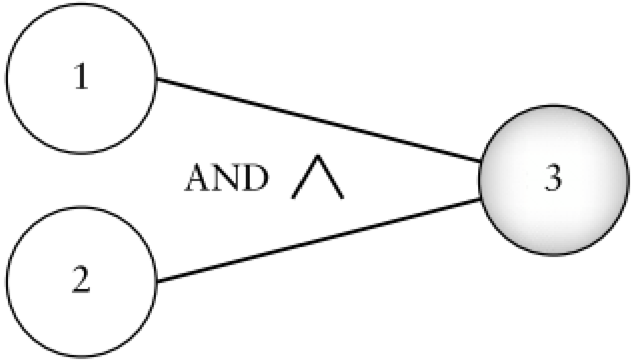
Cause-and-Effect Graphing – Design Test Cases

3. A Boolean cause-and-effect graph is created. Nodes in the graph are causes and effects. Causes are placed on the left side of the graph and effects on the right. Logical relationships are expressed using standard logical operators (AND, OR, NOT).

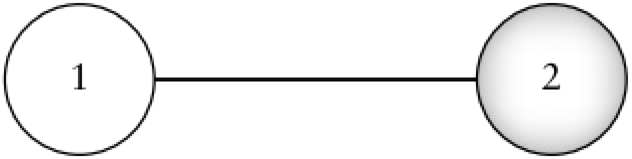
4. The graph may be annotated with constraints that describe combinations of causes and/or effects that are not possible due to environmental or syntactic constraints.
5. The graph is then converted to a decision table.
6. The columns in the decision table are transformed into test cases.

Cause-and-Effect Graphing – Design Test Cases

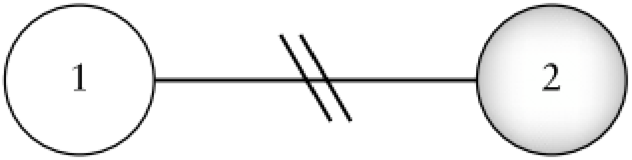
Effect **3** occurs if both causes **1** and **2** are present



Effect **2** occurs if cause **1** occurs



Effect **2** occurs if cause **1** does not occur



Cause-and-Effect Graphing – Example

- Consider the input conditions, or causes are as follows:

  C1: Positive integer from 1 to 80

  C2: Character to search for is in string

- The output conditions, or effects are:

  E1: Integer out of range

  E2: Position of character in string

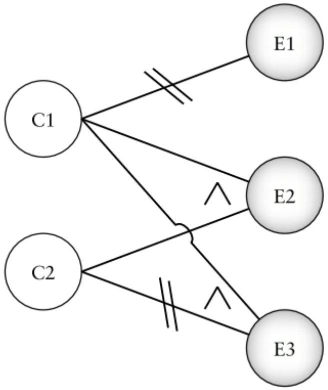
  E3: Character not found

- The rules or relationships can be described as follows:

  If C1 and C2, then E2.

  If C1 and not C2, then E3.

  If not C1, then E1.



FACULDADE ENGENHARIA
Departamento de Informática

Cause-and-Effect Graphing – Example

- Possible test cases for the string “*abcde*” are as follows:

Inputs	Length	Character to search for	Outputs
T1	5	c	3
T2	5	w	Not found
T3	90		Integer out of range

- Then, the decision table is:

	T1	T2	T3
C1	1	1	0
C2	1	0	-
E1	0	0	1
E2	1	0	0
E3	0	1	0

Error Guessing

- Designing test cases using the error guessing approach is based on the tester's/developer's past experience with code similar to the code-under test, and their intuition as to where defects may lurk in the code (e.g. a possible division by zero, a number of pointers that are manipulated, or conditions around array boundaries.

- Error guessing is an *ad hoc* approach to test design in most cases. However, if defect data for similar code or past releases of the code has been carefully recorded, the defect types classified, and failure symptoms due to the defects carefully noted, this approach can have some structure and value.

FACULDADE ENGENHARIA
Departamento de Informática

Handling Multiple Input or Output Streams

A Separation of Concerns

- For both input and output coverage methods, if there is more than one input or output stream or file, we must create separate coverage tests for each one
- Effectively, what we do is treat each separate file or stream as a pre-made input or output partition, within which we make a separate set of smaller partitions

Black Box Testing at Different Levels

Departamento de Informática

Levels of Development

- Recall that there can be many levels of testing, corresponding to the stages of software development
- In particular, black box testing of all kinds can be used at every level of software development
- At the system testing level, the level we have been looking at so far, we have seen how to create functional, input and output coverage tests for the entire program's functional specification (the requirements for the software)
- This is pure black box testing, because it does not require us to have done any development at all

“Gray” Box Testing

If We Already Have a Design …

- If we allow ourselves the luxury of waiting until we have a architectural (class level) design, or even a detailed (method level) design, then we can use black box testing at each of those levels as well

- Since we can see a part of the software (its design), black box testing at these levels is not really “pure” black box – for that reason it is sometimes called "gray box" testing

"Gray" Box Testing

If We Already Have a Design …

- At the architectural (class) design level, we can apply the same black box coverage analyses to the interface of each class, to create class level black box tests (a.k.a. interface tests)
- At the detailed (method) design level, we can apply the same black box coverage analyses to each method, using parameters and global variables as the inputs, and return values or global variable results as the outputs

“Gray” Box Testing

Unit Testing

- Because this kind of unit testing is black box, we still have the advantage that tests can be created in parallel with coding
- Moreover, black box unit testing is earlier and more precise than black box system testing - it can find errors very early *(even before the entire first version is finished)*

"Gray" Box Testing

Departamento de Informática

Unit Testing

- Black box method testing
  - test harnesses
  - role of code-level specifications (assertions)
  - automating black box unit testing
- Black box class testing ("interface testing")
  - class trace specifications
  - executable assertions

Black Box Testing of Single Method

Method Testing

- To use black box testing on code units, we need to figure out what corresponds to requirements, inputs and outputs
- If the unit we are testing is a method (procedure or function), then:
  - The "requirements" are the specification of the method
  - The "input" is the value of parameters and global environment variables used by the method
  - The "output" is the value of return values, global environment variables, and exceptions thrown (together these are referred to as the "outcome" in unit testing)
- Once we know these, test cases can be created according to any of the black box testing criteria :
  - Functionality coverage, input coverage or output coverage

Departamento de Informática

Black Box Testing of Single Method

Example:

Consider the push() method in a Stack class:

```
void push (int x) throws (stackOverflow)
```

- The input is the current state of the Stack object and the parameter value of x to push onto it
- The output is the new state of the Stack object and the exception thrown (stackOverflow)

Given these inputs and outputs, we can then apply, for example, input partitioning to create test partitions:

```
P1        Stack empty,        x = 1
P2        Stack nonempty,     x = 1
P3        Stack almost full,  x = 1
 .         .                  .
 .         .                  .
```

Test Harnesses

Executing Unit Tests

- Once we have test cases for a method, we need a way to run them
  - A method is not a program, so we cannot simply run it as one, we must make the test calls to it in some program
- A test harness is a special program written specifically to exercise a particular method, class or subsystem in unit testing
- The test harness sets up an appropriate environment to use the unit and invokes it with the test inputs, checking the test outcomes and reporting any failures
- The set of test cases are programmed in the harness as a sequence of calls or uses of the tested unit with the test inputs, with code to check the outcome values after each call

Test Harnesses

Example:

```
// A test harness to test the push() method
import Stack;

public class TestPush
{
  public static void main (String[] args)
  {
  // Create a Stack object
  Stack s = new Stack();

  // P1 – empty Stack, x=1
  s.Push(1);

  // Check result
  if (s.Depth() != 1 || s.TopValue() != 1)
   c.println ("P1 test failed");
         ...
  }
}
```

Independent Unit Testing

Factoring Out Unit Dependencies

- A problem with unit testing is that units are usually interdependent, that is, the method we are testing may cal other methods or use other classes in the software
- In the worst case, they all call one another
  - so how can we unit test them individually?
- The test harness can provide "stubs" for the other units
  - the unit being tested then uses the stub method or class instead of the real thing
  - Stub methods are programmed to give "typical“ outcomes of the real method, instead of a real outcome
  - This allows us to test independently of the real unit the stub replaces

Things Go Better with a Specification

Departamento de Informática

Assertions, Please

- When black box unit testing, explicit pre- and post- conditions for the method we are testing help a lot
- Pre and post assertions help in input and output coverage analysis, but more importantly, they help in automating the checking of outcomes – if na outcome meets the post condition, then we can normally accept it as correct
- If the pre- and post- assertions are accurate enough, we can even automate the black box testing process to a large extent

Automating Black Box Unit Testing

FACULDADE ENGENHARIA
Departamento de Informática

Testing Tools

- Tools such as JTest and C/C++Test take advantage of explicit pre-, post- and invariant assertions in code to automatically do (naive) black box unit testing

Source: Parasoft website

- JTest – https://www.parasoft.com/product/jtest/
- C/C++Test – https://www.parasoft.com/product/cpptest/

FACULDADE ENGENHARIA
Departamento de Informática

Automating Black Box Unit Testing

JTest & C/C++Test

- Given method and class interface specifications explicitly coded as pre-, post- and invariant assertions written in a special, rich formal assertion notation, these tools:
  - automatically generate a test harness for the unit
  - automatically provide stubs for any other units used by the method or class under test
  - automatically generate some (naive) input coverage test cases and the harness code to run them
- Automation is often naive, so you can add your own test cases also

FACULDADE ENGENHARIA
Departamento de Informática

Automating Black Box Unit Testing

Problems with Testing Tools

- There are some limitations with testing tools like these:
  - (1) Outcomes must be checked by hand (at least on the first run) since the tool cannot tell all of what was intended (because assertions alone are usually insufficient – if they were sufficient, then we'd already have the code!)
  - (2) The tool can't provide stubs unless it knows everything that is called or used by the unit under test – that is, unless it already has the code for the unit (hence these are not really black box tests at all)

Black Box Class Testing

Testing a Class Interface

- Naive black box class testing such as that done by the JTest and C/C++Test tools simply unit tests each method of a class separately
- In naive class testing, for each method tested,
  - the "input" is the current state of all class, object and global variables as well as the parameters to the method
  - the "output" is the new state of all class, object and global variables as well as the result values and exceptions of the method
- The class specification usually has an invariant assertion that applies to the outcome of every method, as well as the pre- and post- assertions for each method itself

Class Traces

Sequences of Method Calls

- Even very simple classes cannot be well specified, and hence well unit tested, simply by understanding them as a set of independent methods
- The real specification of a class often needs to reason about interdependent sequences of method calls, not just independent individual calls
- Assertions are not very well suited to specifying this kind of sequential dependency

Trace Specifications

Specifying Sequences

- Trace specifications are an explicit method for specifying the behaviour of sequences of method calls
- Trace specifications use trace expressions to specify the legal sequences of method calls to an object in the class
- A trace expression is a sequence of method calls with inputs and outcomes explicitly specified in the sequence

Trace Specifications

Specifying Sequences

Example:

```
s.new(), s.Empty()==true

s.new(), s.Push(Y), s.Pop()==Y,
  s.Empty()==true

s.new(), s.Push(Y), s.Push(Y),
  s.TopValue()==Y, s.Pop()==Y,
  s.Pop()==Y, s.Empty()==true
    .
    .
```

Black Box Testing Using Trace Specifications

Legal and Illegal Traces

- Trace specifications constrain the behaviour of a class using both legal traces (sequences that can or must happen), and illegal traces (sequences that must never happen)
- Trace specifications not only give us the ability to automate creation of the test harness and naive black box test cases, they also make it easy to generate black box test cases for the method call sequences as well

Implementing Assertions

FACULDADE ENGENHARIA
Departamento de Informática

Run Time Checking

- If we use pre-, post- and invariant assertions to specify our method and class interfaces, it is good practice to actually implement (check) them at run time on each method call
- Some languages automatically provide assertion checking provided that the assertions are expressed as boolean expressions in the language (which may not always be possible)

FACULDADE ENGENHARIA
Departamento de Informática

Implementing Assertions

Run Time Checking

- Checked assertions help with every kind of systematic testing, not just black box unit testing, because the assertions are checked every time the method or class is used, no matter what the test method we are using

- This finds errors earlier and pinpoints their cause more specifically than simply a bad outcome, because we can see exactly which assertion failed

- It is good practice to always liberally document your specifications, assumptions and intentions using assertions in your code – it helps in finding errors, and helps other programmers reading your code to understand your assumptions and intent

Departamento de Informática

Black Box Integration Testing

Testing Subsystems

- Black box integration testing applies the same ideas to testing subsystems as they are integrated from smaller units

- If we began with black box unit testing, we can smoothly move into black box integration testing by gradually replacing each stub unit in a unit's test harness with the corresponding real unit once each has been independently unit tested

- As stubs are replaced and the combined units (subsystems) are re-tested at each integration, we gradually build up to the black box testing of the entire integrated system

Departamento de Informática

Summary

- Output coverage methods analyze the set of possible outputs specified and create tests to cover them
- Black box testing can also be applied to unit testing of individual methods and classes given architectural or detailed design specifications
- Test harnesses are special programs written to exercise individual units under test
- Assertion and trace specifications provide interface level specifications for black box unit testing

FACULDADE ENGENHARIA

Departamento de Informática