Testing the Software with Blinders on

[Reading assignment: Chapter 5, pp. 63-79]

Dynamic black-box testing

- Dynamic black-box testing is testing without having an insight into the details of the underlying code.
  - Dynamic, because the program is running
  - Black-box, because testing is done without knowledge of how the program is implemented.
- Sometimes referred to as *behavioral testing*.
- Requires an executable program and a specification (or at least a user manual).
- Test cases are formulated as a set of pairs
  - E.g., (input, expected output)

Test Data and Test Cases

- ***Test data:*** Inputs which have been devised to test the system.
- ***Test cases:*** Inputs to test the system and the predicted outputs from these inputs if the system operates according to its specification.

Test-to-pass and test-to-fail

- Test-to-pass:
  - assures that the software minimally works,
  - does not push the capabilities of the software,
  - applies simple and straightforward test cases,
  - does not try to “break” the program.
- Test-to-fail:
  - designing and running test cases with the sole purpose of breaking the software.
  - strategically chosen test cases to probe for common weaknesses in the software.

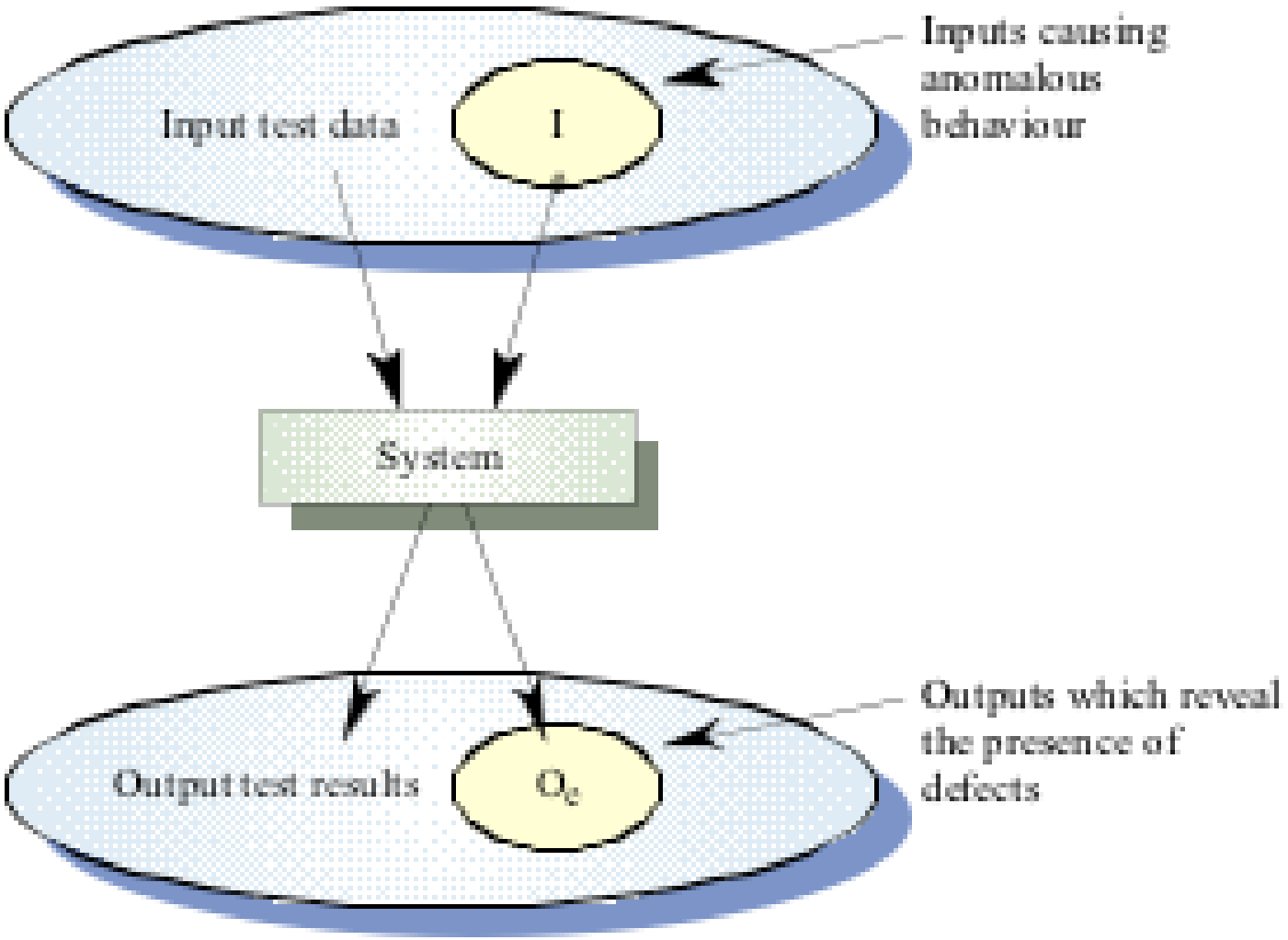
Discussion …

- Why should a tester always start with a test-to-pass approach?
- Isn’t this a waste of time?
- What assurance does test-to-pass give us?
- Shouldn’t the programmers (i.e., not the testers) do test-to-fail?

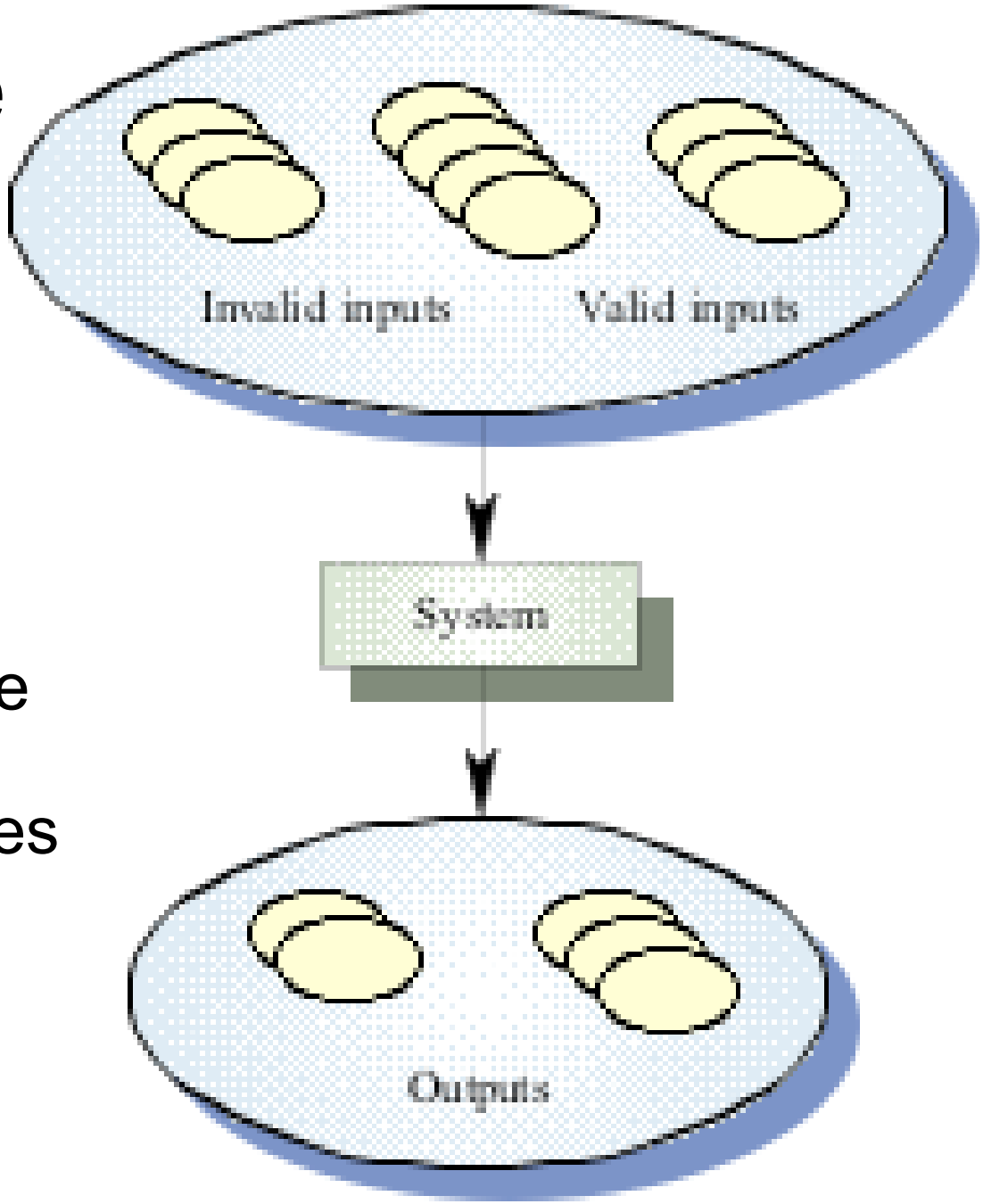
Black-box testing

- Characteristics of Black-box testing:
  - Program is treated as a black box.
  - Implementation details do not matter.
  - Requires an end-user perspective.
  - Criteria are not precise.
  - Test planning can begin early.

Black-box testing



Equivalence Partitioning



- Equivalence partitioning is the process of methodically reducing the huge (or infinite) set of possible test cases into a small, but equally effective, set of test cases.

Search routine specification

procedure Search (Key : INTEGER ; T: array 1..N of INTEGER;
Found : BOOLEAN; L: 1..N) ;

Pre-condition

-- the array has at least one element
1 <= N

Post-condition

-- the element is found and is referenced by L
(Found and T (L) = Key)
or
-- the element is not in the array
(**not** Found **and**
not (**exists** i, 1 >= i >= N, T (i) = Key))

Search routine input partitions

- Inputs which conform to the pre-conditions.
- Inputs where a pre-condition does not hold.
- Inputs where the key element is a member of the array.
- Inputs where the key element is not a member of the array.

Search routine input partitions

Array	**Element**
Single value	In array
Single value	Not in array
More than 1 value	First element in array
More than 1 value	Last element in array
More than 1 value	Middle element in array
More than 1 value	Not in array

Search routine test cases

Input array (T)	**Key** (Key)	**Output** (Found, L)
17	17	true, 1
17	0	false, ??
17, 29, 21, 23	17	true, 1
41, 18, 9, 31, 30, 16, 45	45	true, 6
17, 18, 21, 23, 29, 41, 38	23	true, 4
21, 23, 29, 33, 38	25	false, ??

Data Testing

- If you think of a program as a function, the input of the program is its domain.
- Examples of program data are:
  - words typed into MS Word
  - numbers entered into Excel
  - picture displayed in Photoshop
  - the number of shots remaining in an arcade game
  - …

Boundary input data

- Boundary conditions are situations at the edge of the planned operational limits of the software.
  - E.g., negative to zero to positive numbers, exceeding the input field length of a form, etc.
- Choose input data that lie on the boundary when formulating equivalence partitions.
  - Test the valid data just inside the boundary
  - Test the last possible valid data
  - Test the invalid data just outside the boundary
- Security flaws such as buffer overflow attacks exploit boundaries of array buffers.

Example of Data Testing: Syntax Testing

- System inputs must be validated. Internal and external inputs conform to formats:
  - Textual format of data input from users.
  - File formats.
  - Database schemata.
- Data formats can be mechanically converted into many input data validation tests.
- Such a conversion is easy when the input is expressed in a formal notation such as BNF (Backus-Naur Form).

Garbage-In Garbage-Out

- “Garbage-In equals Garbage-Out” is one of the worst cop-outs ever invented by the computer industry.
- GI-GO does not explain anything except our failure to:
  - install good validation checks
  - test the system’s tolerance for bad data.
- Systems that interface with the public must be especially robust and consequently must have prolific input-validation checks.

Million Monkey Phenomenon

- A million monkeys sit at a million typewriters for a million years and eventually one of them will type Hamlet!
- Input validation is the first line of defense against a hostile world.

Input-Tolerance Testing

- Good user interface designers design their systems so that it just doesn't accept garbage.
- Good testers subject systems to the most creative "garbage" possible.
- Input-tolerance testing is usually done as part of system testing and usually by independent testers.

Syntax Testing Steps

- Identify the target language or format.
- Define the syntax of the language, formally, in a notation such as BNF.
- Test and Debug the syntax:
  - Test the “normal” conditions by covering the BNF syntax graph of the input language. (minimum requirement)
  - Test the “garbage” conditions by testing the system against invalid data. (high payoff)

Automation is Necessary

- Test execution automation is essential for syntax testing because this method produces a large number of tests.

How to Find the Syntax

- Every input has a syntax.
- The syntax may be:
  - formally specified
  - undocumented
  - just understood
- … but it does exist!
- Testers need a formal specification to test the syntax and create useful “garbage”.

BNF

- Syntax is defined in BNF as a set of definitions. Each definition may in-turn refer to other definitions or to itself.
- The LHS of a definition is the name given to the collection of objects on the RHS.
  - ::= means “is defined as”.
  - | means “or”.
  - * means “zero or more occurrences”.
  - + means “one or more occurrences”.
  - A^n means “n repetitions of A”.

BNF Example

special_digit ::= 0 | 1 | 2 | 5
other_digit ::= 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
ordinary_digit ::= special_digit | other_digit
exchange_part ::= *other _ digit*^{2} ordinary_digit
number_part ::= *ordinary _ digit*^{4}
phone_number ::= exchange_part number_part

- Correct phone numbers:
  - 3469900, 9904567, 3300000
- Incorrect phone numbers:
  - 0551212, 123, 8, ABCDEFG

Why BNF?

- Using a BNF specification is an easy way to design format-validation test cases.
- It is also an easy way for designers to organize their work.
- You should not begin to design tests until you are able to distinguish incorrect data from correct data.

Test Case Generation

- There are three possible kinds of incorrect actions:
  - Recognizer does not recognize a good string.
  - Recognizer accepts a bad string.
  - Recognizer crashes during attempt to recognize a string.
- Even small BNF specifications lead to many good strings and far more bad strings.
- There is neither time nor need to test all strings.

Testing Strategy

- Create one error at a time, while keeping all other components of the input string correct.
- Once a complete set of tests has been specified for single errors, do the same for double errors, then triple, errors, ...
- Focus on one level at a time and keep the level above and below as correct as you can.

Example: Telephone Number (Level 1)

- **phone_number ::= exchange_part number_part**
  - Empty string.
  - An *exchange_part* by itself.
  - Two from *exchange_part*.
  - Two from *number_part*.
  - One from *exchange_part* and two from *number_part*.
  - Two from *exchange_part* and one from *number_part*.
  - ...

Example: Telephone Number (Level 2)

- Bad **exchange_part**:
- **exchange_part ::= other_digit^2 ordinary_digit**
  - Empty string.
  - No *other_digit* part.
  - Two from *ordinary_digit*.
  - Three from *ordinary_digit*.
  - ...

Example: Telephone Number (Level 2)

- Bad **number_part**:
- **number_part ::= ordinary_digit**^{4}
  - Not enough from *ordinary_digit*.
  - Too many from *ordinary_digit*.
  - ...

Example: Telephone Number (Level 3)

- **ordinary_digit ::= special_digit | other_digit**
  - Not a digit - alphabetic.
  - Not a digit - control character.
  - Not a digit - delimiter.
  - ...

Example: Telephone Number (Level 4)

- Bad **other_digit**:
  - *other_digit* ::= 2 | 3 | 4 | 5 6 | 7 | 8 | 9
- Bad **special_digit**:
  - *special_digit* ::= 0| 1 | 2 | 5
- ...

Delimiter Errors

- Delimiters are characters or strings placed between two fields to denote where one ends and the other begins.
- **Delimiter Problems:**
  - Missing delimiter. *e.g.,* (x+y
  - Wrong delimiter. *e.g.,* (x+y]
  - Not a delimiter. *e.g.,* (x+y 1
  - Poorly matched delimiters. *e.g.,* (x+y))

Sources of Syntax

- Designer-Tester Cooperation
- Manuals
- Help Screens
- Design Documents
- Prototypes
- Programmer Interviews
- Experimental (hacking)

Dangers of Syntax Test Design

- It's easy to forget the "normal" cases.
- Don't go overboard with combinations:
  - Syntax testing is easy compared to structural testing.
  - Don't ignore structural testing because you are thorough in syntax testing.
  - Knowing a program's design may help you eliminate cases without sacrificing the thoroughness of the testing process.

Syntax Testing Drivers

- Build a driver program that automatically sequences through a set of test cases usually stored as data.
- Do not try to build the “garbage” strings automatically because you will be going down a diverging infinite sequence of syntax testing.

Design Automation: Primitive Method

- Use a word processor to specify a covering set of correct input strings.
- Using search/replace, replace correct sub-strings with incorrect ones.
- Using the syntax definition graph as a guide, generate all single-error cases.
- Do same for double errors, triple errors, …

Design Automation: Random String Generators

- Easy to do, but useless.
- Random strings get recognized as invalid too soon.
- The probability of hitting vulnerable points is too low because there are simply too many “garbage” strings.

Productivity, Training, Effectiveness

- Syntax testing is a great confidence builder for people who have never designed tests.
- A testing trainee can easily produce 20-30 test cases per hour after a bit of training.
- Syntax testing is an excellent way of convincing a novice tester that:
  - Testing is often an infinite process.
  - A tester's problem is knowing which tests to ignore.

Ad-lib Testing

- Ad-lib testing is futile and doesn't prove anything.
- Most of the ad-lib tests will be input strings with format violations.
- Ad-lib testers will try good strings that they think are bad ones!
- If ad-lib tests are able to prove something, then the system is so buggy that it deserves to be thrown out!

Summary

- Express the syntax of the input in a formal language such as BNF.
- Simplify the syntax definition graph before you design the test cases.
- Design syntax tests level by level from top to bottom making only one error at a time, one level at a time, leaving everything else correct.

Summary

- Test the valid test cases by “covering” the syntax definition graph.
- Consider delimiters.
- Automate the testing process by using drivers.
- Give ad-lib testers the attention they crave, but remember that they can probably be replaced by a random string generator.

You now know …

- … test-to-pass test-to-fail testing
- … black-box testing
- … equivalence partitions
- … data testing
- … syntax testing as a special case of data testing