SQA Chapter 6 Supplementary note

Compiled by Roshan Chitrakar

Source: Ron Patton, Software Testing

Testing Specifications

- Highlights of this chapter include
  - What is black-box and white-box testing
  - How static and dynamic testing differ
  - What high-level techniques can be used for reviewing a product specification

A tester uses the specification document to find bugs before the first line of code is written

Black-Box and White-Box Testing

- In black-box testing, the tester only knows what the software is supposed to do—he can’t look in the box to see how it operates.
- In white-box testing (sometimes called clear-box testing), the software tester has access to the program’s code and can examine it for clues to help him with his testing—he can see inside the box.
- Based on what he sees, the tester may determine that certain numbers are more or less likely to fail and can tailor his testing based on that information.

Static and Dynamic Testing

- Static testing refers to testing something that’s not running—examining and reviewing it.
- Dynamic testing is what you would normally think of as testing—running and using the software

Static Black-Box Testing: Testing the Specification

- Testing the specification is static black-box testing.
- You can take that document, perform static black-box testing, and carefully examine it for bugs.
- You can even test an unwritten specification by questioning the people who are designing and writing the software.

High-Level Review of the Specification

- The first step is to stand back and view it from a high level. Examine the spec for large fundamental problems, oversights, and omissions.
- Pretend to Be the Customer
- Research Existing Standards and Guidelines
- Review and Test Similar Software

Research Existing Standards and Guidelines

- ***Corporate Terminology and Conventions.*** If this software is tailored for a specific company, it should adopt the common terms and conventions used by the employees of

that company.

- ***Industry Requirements.*** The medical, pharmaceutical, industrial, and financial industries have very strict standards that their software must follow.
- ***Government Standards.*** The government, especially the military, has strict standards.
- ***Graphical User Interface (GUI).*** If your software runs under Microsoft Windows or Apple Macintosh operating systems, there are published standards and guidelines for how the software should look and feel to a user.
- ***Hardware and Networking Standards.*** Low-level software and hardware interface standards must be adhered to, to assure compatibility across systems.

Review and Test Similar Software

- ***Scale***. Will your software be smaller or larger? Will that size make a difference in your testing?
- ***Complexity***. Will your software be more or less complex? Will this impact your testing?
- ***Testability***. Will you have the resources, time, and expertise to test software such as this?
- ***Quality/Reliability***. Is this software representative of the overall quality planned for your software? Will your software be more or less reliable?

Low-Level Specification Test Techniques

- Specification Attributes Checklist
- Specification Terminology Checklist

Specification Attributes Checklist

- ***Complete***. Is anything missing or forgotten? Is it thorough? Does it include everything necessary to make it stand alone?
- ***Accurate***. Is the proposed solution correct? Does it properly define the goal? Are there any errors?
- ***Precise, Unambiguous, and Clear***. Is the description exact and not vague? Is there a single interpretation? Is it easy to read and understandable?
- ***Consistent***. Is the description of the feature written so that it doesn't conflict with itself or other items in the specification?
- ***Relevant***. Is the statement necessary to specify the feature? Is it extra information that should be left out? Is the feature traceable to an original customer need?
- ***Feasible***. Can the feature be implemented with the available personnel, tools, and resources within the specified budget and schedule?
- ***Code-free***. Does the specification stick with defining the product and not the underlying software design, architecture, and code?
- ***Testable***. Can the feature be tested? Is enough information provided that a tester could create tests to verify its operation?

Specification Terminology Checklist

- ***Always, Every, All, None, Never.*** If you see words such as these that denote something as certain or absolute, make sure that it is, indeed, certain. Put on your tester's hat and think of cases that violate them.
- ***Certainly, Therefore, Clearly, Obviously, Evidently***. These words tend to persuade you into accepting something as a given. Don't fall into the trap.
- ***Some, Sometimes, Often, Usually, Ordinarily, Customarily, Most, Mostly***. These words are too vague. It's impossible to test a feature that operates "sometimes."
- ***Etc., And So Forth, And So On, Such As***. Lists that finish with words such as these aren't testable. Lists need to be absolute or explained so that there's no confusion as to how the series is generated and what appears next in the list.
- ***Good, Fast, Cheap, Efficient, Small, Stable***. These are unquantifiable terms. They aren't testable. If they appear in a specification, they must be further defined to explain exactly what they mean.
- ***Handled, Processed, Rejected, Skipped, Eliminated***. These terms can hide large amounts of functionality that need to be specified.
- ***If…Then…(but missing Else)***. Look for statements that have "If…Then" clauses but don't have a matching "Else." Ask yourself what will happen if the "if" doesn't happen.

Dynamic White-Box Testing

Dynamic white-box testing is using information you gain from seeing what the code does and how it works to determine.

Also called “Structural Testing”

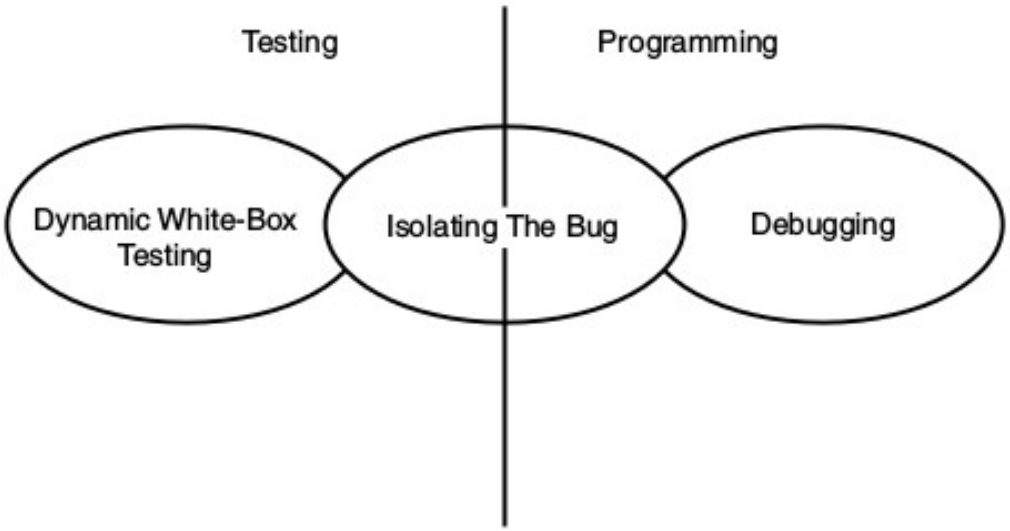
Dynamic white-box testing isn’t limited just to seeing what the code does. It also can involve directly testing and controlling the software viz.

1. Directly testing low-level functions, procedures, subroutines, or libraries
2. Adjusting your test cases based on what you know
3. Gaining access to read variables and state information
4. Measuring how much of the code and specifically what code you “hit”

Dynamic White-Box Testing versus Debugging

The goal of dynamic white-box testing is to find bugs. The goal of debugging is to fix them.

If it's white-box testing, that could even include information about what lines of code look suspicious. The programmer who does the debugging picks the process up from there, determines exactly what is causing the bug, and attempts to fix it.



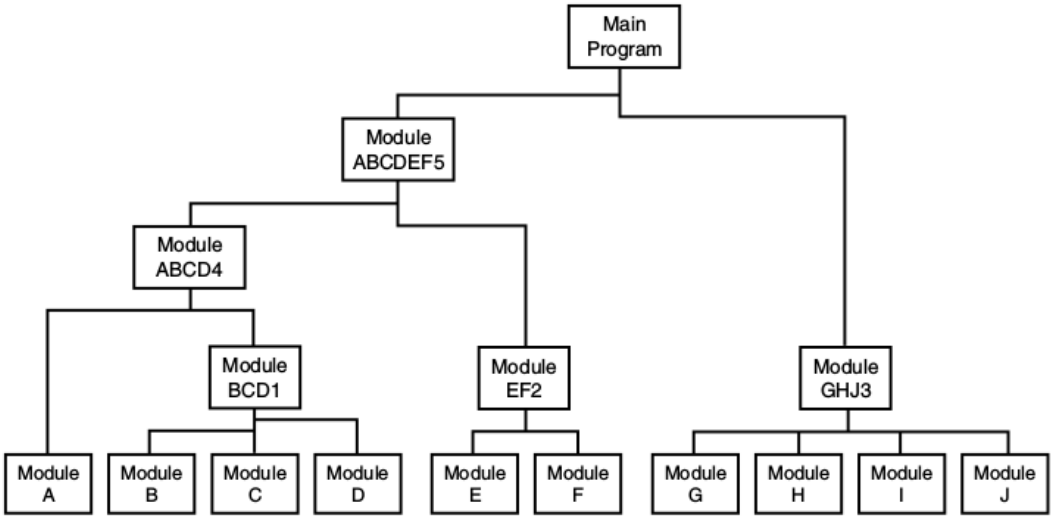
- You will use many of the same tools that programmers use.
- You can use the same compiler but possibly with different settings to enable better error detection.
- You may also write your own programs to test separate code modules

Unit and Integration Testing

Testing that occurs at the lowest level is called unit testing or module testing.

Integration testing is performed against groups of modules until the entire product is tested at once in a process called system testing.

When a problem is found at the unit level, the problem must be in that unit. If a bug is found when multiple units are integrated, it must be related to how the modules interact.

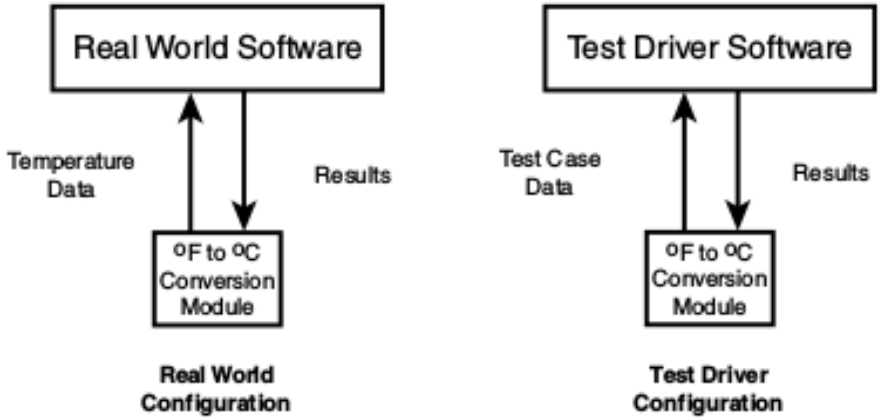


Top-down and Bottom-up testing

There are two approaches to incremental testing: bottom-up and top-down.

In bottom-up testing, you write your own modules, called test drivers.

These drivers send test-case data to the modules under test, read back the results, and verify that they're correct.

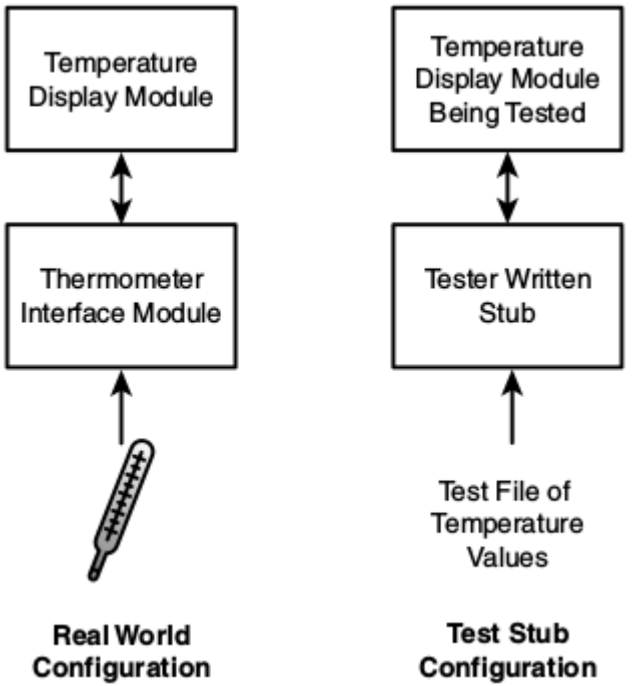


Top-down and Bottom-up testing
(contd.)

Top-down testing may sound like big-bang testing on a smaller scale.

you could write a small piece of code called a stub that acts just like the interface module.

With this test stub configuration, you could quickly run through numerous test values.



Data Coverage

Divide the code just as you did in black-box testing —into its data and its states (or program flow)

you can easily map the white-box information you gain to the black-box cases you've already written.

Consider the data first. Data includes all the variables, constants, arrays, data structures, keyboard and mouse input, files and screen input and output, and I/O to other devices such as modems, networks, and so on.

Data Flow

Data flow coverage involves tracking a piece of data completely through the software.

At the unit test level, this would just be through an individual module or function. The same tracking could be done through several integrated modules.

you would use a debugger and watch variables to view the data as the program runs.

With dynamic white-box testing, you could also check intermediate values during program execution.

Sub-Boundaries

Every piece of software will have its own unique sub-boundaries e.g.

- A module that computes taxes might switch from using a data table to using a formula at a certain financial cut-off point.
- An operating system running low on RAM may start moving data to temporary storage on the hard drive.
- A complex numerical analysis program may switch to a different equation for solving the problem depending on the size of the number.

If you perform white-box testing, you need to examine the code carefully to look for sub-boundary conditions.

Formulas and Equations

A good black-box tester would hopefully choose a test case of n=0, but a white-box tester, after seeing the formula in the code, would know to try n=0 because that would cause the formula to blow up with a divide-by-zero error.

But, what if n was the result of another computation?

Error Forcing

If you’re running the software that you’re testing in a debugger, you don’t just have the ability to watch variables and see what values they hold —you can also force them to specific values.

You could use your debugger to force it to zero.

If you take care in selecting your error forcing scenarios, error forcing can be an effective tool.

Code Coverage

You must test the program's states and the program's flow among them; must attempt to enter and exit every module, execute every line of code, and follow every logic and decision path. Examining the software at this level of detail is called code-coverage analysis.

For very small programs or individual modules, using a debugger is often sufficient. However, performing code coverage on most software requires a specialized tool known as a code coverage analyzer.

Code-coverage analyzers hook into the software you're testing and run transparently in the background while you run your test cases.

You can then obtain statistics that identify which portions of the software were executed and which portions weren't.

You will also have a general feel for the quality of the software.

Program Statement and Line Coverage

Your goal is to make sure that you execute every statement in the program at least once.

You could run your tests and add test cases until every statement in the program is touched.

It can tell you if every statement is executed, but it can't tell you if you've taken all the paths through the software.

LISTING 7.1 It's Very Easy to Test Every Line of This Simple Program

```
PRINT "Hello World"
PRINT "The date is: "; Date$
PRINT "The time is: "; Time$
END
```

Branch Coverage

The simplest form of path testing is called branch coverage testing.

In the example below, ensuring 100 percent statement coverage requires only a single test case with the Date$ variable set to January 1, 2000.

But you still need to try a test case for a date that's not January 1, 2000, which would execute the other path through the program.

LISTING 7.2 The IF Statement Creates Another Branch Through the Code

```
PRINT "Hello World"
IF Date$ = "01-01-2000" THEN
    PRINT "Happy New Year"
    END IF
PRING "The date is: "; Date$
PRINT "The time is: "; Time$
END
```

Condition Coverage

Condition coverage testing takes the extra conditions on the branch statements into account.

```
PRINT "Hello World"
IF Date$ = "01-01-2000" AND Time$ = "00:00:00" THEN
    PRINT "Happy New Year"
    END IF
PRINT "The date is: "; Date$
PRINT "The time is: "; Time$
END
```

The following cases assure that each possibility in the IF statement are covered. If you were concerned only with branch coverage, the first three conditions would be redundant and could be equivalence partitioned into a single test case. But, with condition coverage testing, all four cases are important because they exercise different conditions of the IF statement in line 4.

Date$	*Time$*	*Line # Execution*
01-01-0000	11:11:11	1,2,5,6,7
01-01-0000	00:00:00	1,2,5,6,7
01-01-2000	11:11:11	1,2,5,6,7
01-01-2000	00:00:00	1,2,3,4,5,6,7