White Box Testing

Sharon Robson

August 2009

Introduction

White box testing is tough! Often our backgrounds prevent us from understanding the value of white box testing, as most testers don't have programming experience, training and skills. Persisting and learning the techniques of white box testing can pay off in better test coverage and effective test design due to better understanding of the System Under Test (SUT) and the techniques that may be used to test it efficiently.

What is it?

White Box testing is using our skills to look into the structure of the System Under Test (SUT), either at the detailed design level or at the code level, to analyse and develop understanding, then to use this understanding to help us design and develop test cases. These test cases would focus on any potential issues we see in the detail of the SUT layout. These may be more and different risk areas or issues that can be identified than simply using the Black Box test techniques.

We are basically using the structure of the system, in a similar way that we use the requirements, to tell us what tests to write and run. We can use the design and the code to achieve greater coverage of the system, and more specific tests focussed on high risk areas of the SUT.

Why do we do it?

Testing is all about telling people about the quality of the system under test. That is our role. We focus our testing on various test levels or test for specific quality risks and tell people about the defects and behaviour of the system to ensure that the project knows what the system can and cannot do, and allow the key Stakeholders to make decisions about the quality of the system i.e. is it “good enough”.

We use Black Box techniques to prove that the system does what it should do, and doesn't do what it shouldn't do, but using Black Box techniques only we don't know what else has been put into the structure of the system that is outside the defined requirements that may impact on the behaviour of the system. Sometimes Black Box techniques do not allow us to know enough about the system, nor do they provide the level of detail required for testing of higher risk systems.

One of the key approaches to White Box testing is to have access to the detailed design and code and the tools to read the code and assess coverage, or the assistance from the developers in accessing code and assessing the code coverage. We would struggle to do comprehensive System Testing without these White Box approaches. For example, if we had a system that was a Front End Application, which generates a file of input that is then loaded into the database, which then triggers a stored procedure that updates other database tables. Without a White Box test design approach we would not be able to accurately test:

- the contents of the file
- the transfer of the file
- the receipt of the file
- the parsing of the data into the data base
- ensuring that the data is put into the right fields in the right tables in the right formats; and

- that the stored procedure was kicked off at the right time and did the right activities.

This would be a group of very common system level tests. To do it we need the greater visibility provided by understanding the structure of the system, so that we can make sure that all the Components and Interfaces are in place prior to us designing and kicking off our testing. We also need to know the exact nature of the changes that are happening, and what will happen if these changes do not occur so that we can design our positive and negative test cases as required.

Another key reason for White Box testing is to confirm the "coverage" of our testing. Coverage is defined as "the degree, expressed as a percentage, to which a specified coverage item has been exercised by a test suite." (***ISTQB Glossary***). We use coverage metrics to tell ourselves and the stakeholders how much of the system we have tested.

Coverage assessment can be done on both Black Box and White Box test techniques as coverage metrics can be used to assess if additional testing is required to exercise more of the code, or different aspects of the code. The degree of coverage that may be required varies from application to application, and can also depend on what the application is to be used for. Generally the more safety or life critical the application, the more rigour needs to be applied to the test coverage. The need for proving coverage is very common in safety critical software, often supported by government legislation such as in the medical software domain.

If we are going to look at and report on coverage of the system we need to be able to identify how many statements, decisions, conditions etc. there are, so we can then assess our coverage. We need to understand the structure of the system to know this. Without this "inside" knowledge we would not be able to tell what pieces of code we have tested and which we have not. From Wikipedia we can look at the different types of White Box Test coverage criteria:

- ***Function coverage*** *- Has each function in the program been executed?*
- ***Statement coverage*** *- Has each line of the source code been executed?*
- ***Decision coverage*** *(also known as Branch coverage) - Has each control structure (such as an "if" statement) evaluated both to true and false?*
- ***Condition coverage*** *- Has each boolean sub-expression evaluated both to true and false (this does not necessarily imply decision coverage)?*
- **Path coverage** - Has every possible route through a given part of the code been executed?
- **Entry/exit coverage** - Has every possible call and return of the function been executed?

Safety-critical applications are often required to demonstrate that testing achieves 100% of some form of code coverage. http://en.wikipedia.org/wiki/Code_coverage

Sometimes we have to give information about the degree to which the software development approach adhered to certain standards. For example RTCA DO-178B/ED-12B (***ISTQB Advanced Syllabus***) stipulates that White Box test techniques must be used when testing avionics (see table below). To adhere to this standard you must be able to prove that all source code has been executed and tested. To design these tests you must know what the source code is, how to execute it, and how to know that you have complete coverage.

Criticality	Potential Failure Impact	Required Coverage
Level A: Catastrophic	Software failure can result in a catastrophic failure of the system.	Modified Condition/ Decision, Decision, and Statement
Level B: Hazardous/ Severe	Software failure can result in a hazardous or severe/major failure of the system.	Decision and Statement
Level C: Major	Software failure can result in a major failure of the system.	Statement
Level D: Minor	Software failure can result in a minor failure of the system.	None
Level E: No effect	Software failure cannot have an effect on the system.	None

FAA-DO 178B: Mandated Coverage

When thinking about the "why" of White Box techniques it is also useful to think about the types of defects we would be looking for with White Box test design techniques. This can relate to the various test levels also but can focus specifically the Unit and Integration test levels. It is very difficult to design tests at these levels without White Box test design techniques. At Unit or Component test level we are trying to assess that the basic units of code do what they are meant to do, which means we need to know what they are meant to do. We can usually only derive this from the code or the detailed design.

For Integration testing we need to know how the Units or Components fit together, how their relationships work, and what information is in the interfaces that they use to develop these relationships. We need to make sure that the interfaces contain the correct contents, and are dealt with correctly. We cannot do this without a detailed Interface description and tests that tell us that the interfaces are structured and handled correctly. Both of Unit and Integration testing approaches rely on knowledge of the structure of the system to give us the information to derive our tests, and thus are White Box Techniques.

Unit and Integration testing could be mainly done by developers, but if we needed to ensure independence of testing, and make sure that the tests were structured and potentially recorded correctly for future use, the test team may need to be involved in the development and understanding of these tests. Often when working with safety or life critical systems it is important to have as many tests done by different teams as possible.

With the advent of the more "agile" development practices, testers may also be exposed more often to the source code and designs of the systems that they are working on. Agile practices rely on testers working closely with the developers to build simple and robust systems. Testing at the unit level, using White Box techniques can also play a role in this development approach.

The ISTQB Foundation Syllabus indicates that testers who wish to become certified at the Foundation level need to know about White Box testing; when, where, and how to do it. The syllabus defines levels of knowledge (K levels) that people need to achieve for certification. K2 indicates that the tester needs to understand, explain, give reasons, compare, classify, categorise, give examples, and or summarise the

concept. K3 means that the tester must apply and or use the concept. The following are the Learning Objectives (LO) from the Foundation Syllabus about White Box Testing.

> ***4.2 Categories of test design techniques (K2)***
> *LO-4.2.1 Recall reasons that both specification-based (black-box) and structure-based (white-box)approaches to test case design are useful, and list the common techniques for each. (K1)*
> *LO-4.2.2 Explain the characteristics and differences between specification-based testing, structure-based testing and experience-based testing. (K2)*
>
> ***4.4 Structure-based or white-box techniques (K3)***
> *LO-4.4.1 Describe the concept and importance of code coverage. (K2)*
> *LO-4.4.2 Explain the concepts of statement and decision coverage, and understand that these concepts can also be used at other test levels than component testing (e.g. on business procedures at system level). (K2)*
> *LO-4.4.3 Write test cases from given control flows using the following test design techniques:*
> *o statement testing;*
> *o decision testing. (K3)*
> *LO-4.4.4 Assess statement and decision coverage for completeness. (K3)*
>
> ***4.6 Choosing test techniques (K2)***
> *LO-4.6.1 List the factors that influence the selection of the appropriate test design technique for a particular kind of problem, such as the type of system, risk, customer requirements, models for use case modeling, requirements models or tester knowledge. (K2)*
> ***ISTQB Foundation Syllabus (v2007)***

Understanding the "Why" we do White Box testing goes a long way towards meeting the Foundation Syllabus requirements.

How do we do it?

Analysis of the test basis is how we develop our specific tests i.e. we are looking to prove that a specific requirement is met. We use a variety of test design techniques to generate expected results, which we then incorporate into test cases which tell us how much we will know about the system.

With Black Box techniques can we look at the requirements to tell us how we expect the system to behave. We design tests that prove that the system does what it is meant to do, or that it does not do what it is not meant to do. *White Box test design techniques are exactly the same*. We just use a different test basis. Instead of requirements, we use a design document or the code itself to design tests that prove that the code does what we expected it to do.

For example: In a design document there may be a diagram that shows the system architecture.

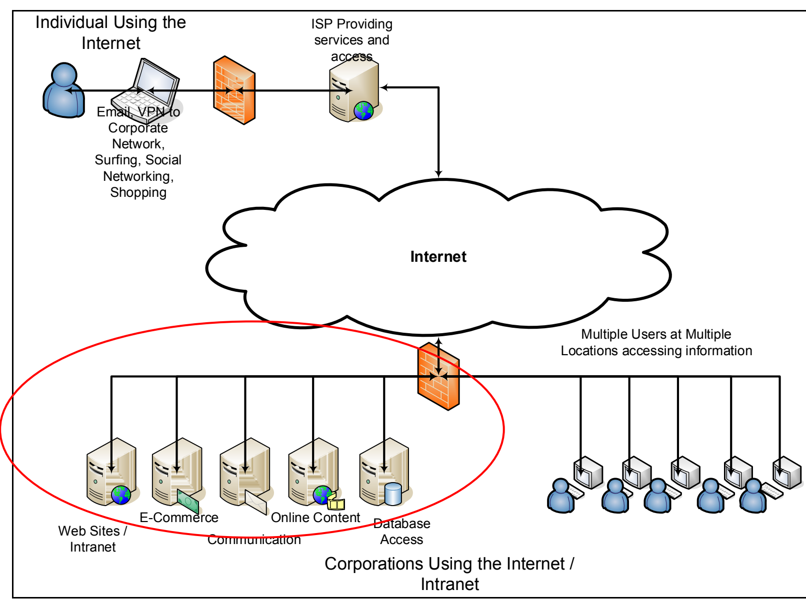

Figure 1 - Example Architecture Diagram

We could design white box tests that uses a “ping” tool to prove that our server can access all the other required servers.

Test Cases and Expected Outcomes

Test Case	Test Condition: Only the appropriate Servers should communicate directly with each other Y = Yes they should communicate N = No access should be available	Web Site	E-commerce	Communication	Content	DB Access
1	Web Site	Y	Y	N	N	N
2	E-Commerce	Y	NA	N	N	Y
3	Communication	N	N	NA	N	N
4	Content	Y	Y	N	NA	Y
5	DB Access	N	Y	N	N	NA

Without analysing the architecture diagram we would not have know the structure of the system, the number of servers and we would not have know what our expected results were. We can now run 5 tests using “ping” tools and technology to determine that only the right servers talk to each other.

Then we can look at a Code analysis example:

HTML: making sure our (Softed) web site is structured correctly:

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd"><html
xmlns="http://www.w3.org/1999/xhtml"><head><link type="text/css" rel="Stylesheet"
href="css/Main.css" /><link type="text/css" rel="Stylesheet" href="css/Menu.css"
/><link type="text/css" rel="Stylesheet" href="css/Nav.css" />
<script language="javascript" type="text/javascript"
src="/js/Miscellaneous.js"></script>
<title>Software Education Group</title><meta name="description" content="Training,
conferences and consulting addressing all aspects of software systems, from
```

concept to commissioning." /><link type="text/css" rel="Stylesheet" href="css/Home.css" /><link type="text/css" rel="Stylesheet" href="css/Cover.css" />

We could design tests based on this to make sure that the information contained in the web site is correct; the information is spelt correctly, and that the appropriate language and techniques have been used. This is particularly important if we are testing for such thing as portability and maintainability from ISO9126.

In the above example you can see that we are using Javascript – which means that we are able to design tests that ensure that the java works as expected on all platforms. We are also able to see that there is a reference to a Doc Type in the first section. We can check that this is the correct document type for what we are trying to develop, and that it does adhere to the W3C standards that are described in the header (see highlighted sections).

What Specific Techniques and Coverage Criteria do we need to know?
The objective at Foundation level is to understand the concept of coverage, i.e. what makes 100% coverage. How to figure out how many tests may need to be run to give maximum coverage with minimum tests; as well as the data to be used in those tests is also important. From the Foundation Syllabus there are specific Learning Objectives:

> *LO-4.4.3 Write test cases from given control flows using the following test design techniques:*
> *o statement testing;*
> *o decision testing. (K3)*
> *LO-4.4.4 Assess statement and decision coverage for completeness. (K3)*

Let’s consider the following example of a pretty simple calculation:

BMI is determined by your weight in kg
divided by your (height in metres)^2.
It is designed for men and women over the age of 18.
A healthy BMI is between 20 and 25.
A result below 20 indicates that you may be underweight;
a figure above 25 indicates that you may be overweight.

Statement Testing
With Statement testing you are exercising the code to ensure that each line of code in the code has been executed. So you are trying to provide information that indicates that the statement, or the “direction” that is being taken by the code, has been taken as expected. These statements can also be control statements such as “if”, “case”, “for” or “while” or some other piece of code that directs the code to follow a different path that the one the program was originally on. The object of Statement testing is to check that each executable line (or statement) in the code is executed. Usually we are trying to figure out the minimum number of tests we need to execute to get 100% statement coverage i.e. every line of is executed.

1. *Read Weight (w)*
2. *Read Height (h)*
3. *IF w > 400 THEN*
4. *Print "invalid weight"*
5. *ENDIF*
6. *If h > 3 THEN*
7. *Print "invalid height"*
8. *ENDIF*
9. *BMI=w/(h*h)*
10. *Print ("BMI = " BMI)*
11. *End*

(example 1 – Statement Testing)

In this example we can see that there are 11 statements in the code, including two IF statements. They are highlighted. They are control statements as they change the direction that the code is going based on the result of the "question" being asked by the code.

If we look at this from a testing point of view (or an exam point of view) we are usually looking for the **Minimum number of tests to get Maximum coverage** so we need to analyse the code to see what test data we can put into a test case to give us the maximum coverage with minimum tests.

There is a "rule of thumb" of doing this – count the "else"s in the code and then add one ("Else" + 1). For example, in the following code snippet, there are no "elses" therefore we should only need 1 test case for 100% statement coverage.

Let's check that. In this case we would need **1 Test (1 set of test data)** to execute these IF statements as well as every other line in the code…..we would need to put in values, for example, where w = 550 and h = 4. We would have the question "IF w > 400" answered true, which means that "Print "invalid weight" would be executed, as would the line "Print "invalid height". The BMI would be calculated and the output "BMI = 34.38" would be displayed. All the lines of code have been executed.

Our data and coverage can be summarised:

Data			Statements	Statement	
Weight	Height	BMI	Executed	Coverage	Comments
550	4	34.38	11	100%	

If we put in w = 350 and h = 3.5, we would only exercise line 9 onwards (If h > 3 statement), this would not give us the minimum number of tests as we would have to run another test case to cover lines 6, 7 and 8 or we would not have 100% statement coverage. Often this level of analysis is required to confirm Statement Coverage.

We could formalise this into a test case such as follows:

Test Case	Data	Expected Outcome
1	w = 550, h = 4	"invalid weight" is printed "invalid height" is printed "BMI = 34.38" is printed

If we expand our BMI program to include weight categories, the program gets more convoluted and we introduce some "ELSE"s.

```
1.  Read Weight (w)
2.  Read Height (h)
3.  IF w > 400 THEN
4.  Print “invalid weight”
5.  ENDIF
6.  If h > 3 THEN
7.  Print “invalid height”
8.  ENDIF
9.  BMI=w/(h*h)
10. Print ( “BMI = “ BMI)
11. IF BMI < 20 THEN
12. Print “underweight”
13. ELSE
14. IF BMI >= 20 AND BMI <=25 THEN
15. Print “ideal weight”
16. ELSE
17. IF BMI > 25 THEN – note this statement is now redundant
18. Print “overweight”
19. ENDIF
```

(example 2 – Statement Testing)

The thing to notice here is that the “else” is also a statement…..it is asking the code to take a different direction. Remember – we are just trying to figure out the minimum number of test cases needed to exercise all these lines of code (statements). Using our rule of thumb (Else + 1) we can estimate that we should need 3 test cases as a minimum to test the statements here. Why? Because we need to exercise the “else” statement also. Logically the data we put in to execute the first side of the IF cannot be used to exercise the ELSE also.

So we need **1 test case** where we have BMI < 20. The IF Statement is executed which comes out “true”. This means that line 12 is executed. But we will need **another test case**, where BMI is between 20 and 25 (inclusive) to execute the lines 15. And then another test case where BMI is more than 25 to exercise line 18. Three test cases are enough for 100% Statement Coverage if we use the right data. All the lines in the code have been covered.

Data			Statements Executed	Unique Statements	Statement Coverage	Lines Executed
Weight	Height	BMI				
550	4	34.38	13	2	33%	1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 17, 18, 19
500	6	13.89	13	2	33%	1, 2, 3, 4, 5, 6, 7, 8, 9, 10 11, 12, 19
600	5	24.00	13	2	33%	1, 2, 3, 4, 5, 6, 7, 8, 9, 10 14, 15, 19
					100%	

Remember though, the objective is just to <u>ensure that each statement is executed</u> – with Statement Testing we would structure our tests to ensure that we “trigger” the statement that we need to prove has been triggered.

We could formalise this into test cases as follows:

Test Case	Data	Expected Outcome
1	w = 550, h = 4	w > 400 is true “invalid weight” is printed h > 3 is true “invalid height” is printed “BMI = 34.38” is printed BMI < 20 is false, move through to line 14 BMI >= 20 AND BMI <=25 is also false, move through to line 17 BMI > 25 is true, “overweight” is printed End program
2	w = 500, h = 6	w > 400 is true “invalid weight” is printed h > 3 is true “invalid height” is printed “BMI = 13.89” is printed BMI < 20 is true “underweight” it printed BMI >= 20 AND BMI <=25 is false, move through to line 17 BMI > 25 is false End program
3	w = 600, h = 5	w > 400 is true “invalid weight” is printed h > 3 is true “invalid height” is printed “BMI = 24.00” is printed BMI < 20 is false BMI >= 20 AND BMI <=25 is true “ideal weight” is printed BMI > 25 is false End program

Branch / Decision Testing

Branch Testing or Decision Testing however, has the objective to test every option (true and false) on every control statement, including compound decisions. A compound decision is where the second decision depends on the previous decision.

Let's use our previous examples, but this time we are looking for Branch or Decision Coverage, we need to isolate the decisions.

1. *Read Weight (w)*
2. *Read Height (h)*
3. *IF w > 400 THEN*
4. *Print "invalid weight"*
5. *ENDIF*
6. *If h > 3 THEN*
7. *Print "invalid height"*
8. *ENDIF*
9. *BMI=w/(h*h)*
10. *Print ("BMI = " BMI)*
11. *IF BMI < 20 THEN*
12. *Print "underweight"*
13. *ELSE*
14. *IF BMI >= 20 AND BMI <=25 THEN*
15. *Print "ideal weight"*
16. *ELSE*
17. *IF BMI > 25 THEN – note this statement is now redundant*
18. *Print "overweight"*
19. *ENDIF*

(example 3 – Branch/Decision Testing)

In this case we have five decisions, line 3, line 6, line 11, line 14 and line 17. In Statement testing all we had to do was execute the line. In looking for Branch/Decision testing Coverage we have to execute BOTH options of the statement. Our Statement Test coverage tells us the values to use to make these decisions true. But that is only half of the Decisions….what if they weren't true?? We need to test what the program does when the statements are executed in both ways.

The potential number of test cases can also be derived by a rule of thumb…you need to identify the "end points" of the code, i.e. when does it do something and then finish?

In this case there are six print statements (underlined). However, only three of them are "end points" – where nothing else happens after the decision is completed. This tells me that I would need at least three test cases for 100% Branch/Decision Coverage, one for each of the potential end points.

So for 100% Branch/Decision Testing coverage we need to look for test cases to exercise both sides of the decision and produce each of the end points. So we need weights less than 400 and greater than 401. We also need heights less than 3 and greater than 3.1. These however can be combined into test cases that stimulate each of our BMI outputs (our end points).

My Test Cases could be as follows:

Test Case	Data	Expected Outcome
1	w = 550, h = 4	w > 400 is true “invalid weight” is printed h > 3 is true “invalid height” is printed “BMI = 34.38” is printed BMI < 20 is false, move through to line 14 BMI >= 20 AND BMI <=25 is also false, move through to line 17 BMI > 25 is true, “overweight” is printed End program
2	w = 60, h = 1.7	w > 400 is false h > 3 is false “BMI = 20.76” is printed BMI < 20 is false BMI >= 20 AND BMI <=25 is true “ideal weight” is printed BMI > 25 is false End program
3	w = 300, h = 6	w > 400 is false h > 3 is true “invalid height” is printed “BMI = 8.33” is printed BMI < 20 is true “underweight” it printed BMI >= 20 AND BMI <=25 is false, move through to line 17 BMI > 25 is false End program

So three test cases exercise both sides of each of the control statements and would test that these decisions have been made correctly.

Another way to look at Branch/Decision testing is to draw a diagram that shows how the code is flowing (similar to a control or data flow diagram) and then count the number of tests you would need. The diagram from the code snippet above could look like this:

A diagram of these tests is shown below:

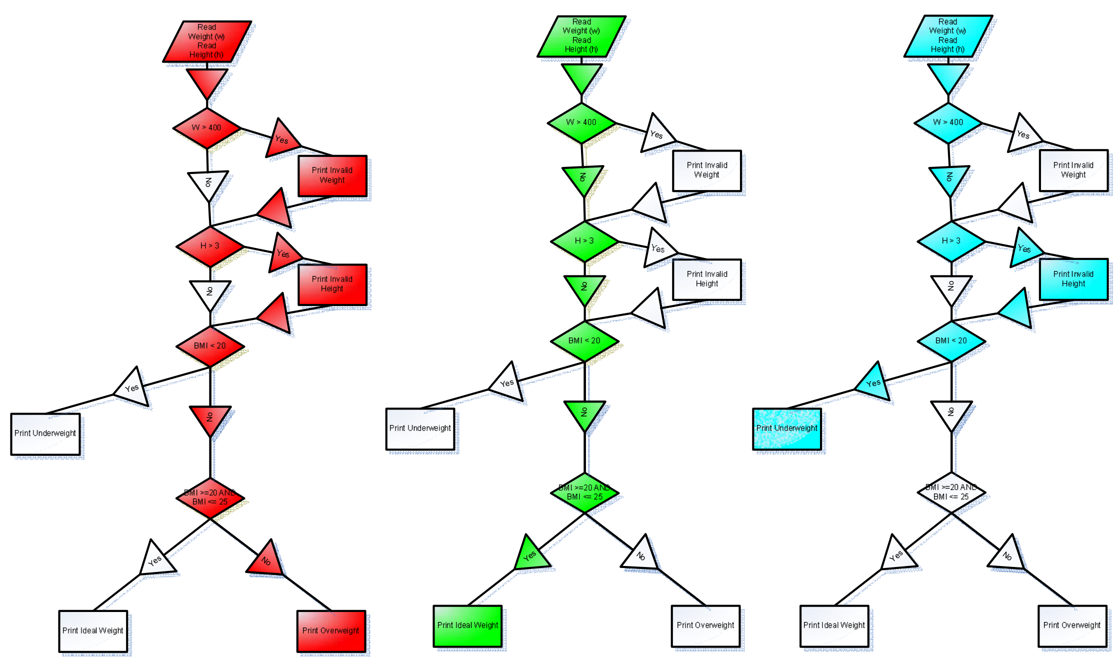

Figure 2 - Branch/Decision Testing

If we consider the above example, we can see a Conditional Decision. We cannot get to one decision without going through another. We cannot print “ideal weight” or “overweight” without first determining the value is greater than 20.

We could also put this into a table if we wanted to not draw a diagram. We can look at this using Boolean values to generate a table which we populate with all possible combinations of True and False. We see that Case 4 actually could not happen.

Case	BMI < 20	BMI > 25	Outcome
1	True	False	“underweight”
2	False	True	“overweight”
3	False	False	“ideal weight”
4 - NA	True	True	Cannot happen

So we know we need 3 test cases for 100% Branch/Decision Coverage, with the minimum number of test cases.

If we extend this to include the weight and height decisions also, we can see that we may have missed a test case – this would be Condition Combination Coverage – which is beyond the scope of the foundation syllabus, but handy to know – as you can see, test case 4 has not been covered by our testing.

Case	Weight > 400	Height > 3	Outcome
1	True	True	“invalid weight” “invalid height”
2	False	False	No invalids
3	False	True	“invalid height” only
4	True	False	“invalid weight” only

This could be seen in the diagram as a missed test case, and could be picked up by completing data flow testing. The diagram above could be modified to show the potential data flows and would allow us to assess the number of test cases for coverage of these. This is a stronger coverage than Branch/Decision Testing.

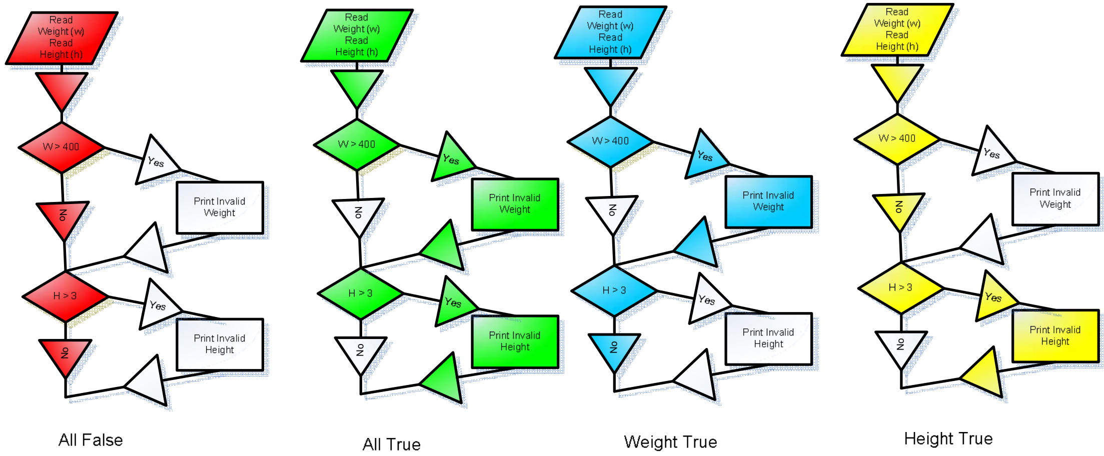

Figure 3 - Data Flow Analysis

Loop Testing

Control Statements can also be "while" loops or "for" loops. Which means we would have to have the right number of test cases to trigger the loop and action the lines inside the loop.

1. Input Number_of_People
2. Total = 0
3. While Number_of_People > 0
4. Input weight
5. Total weight = Total + weight
6. Number_of_people = Number_of_people – 1
7. End Loop
8. Print "Total Weigh is" & Total.
(example 4 –Loop Testing)

In this case we are being asked to see if the Number of people is greater than 0 to enter the loop and access the lines of code inside the loop. So we can generate 1 test case that will execute the loop statement and all of the other statements inside the loop. For example our test case could be Number_of_people = 3. This means that the loop will be executed 3 times. But we only need 1 test case to do this. Once we are inside the loop, the number of people self decrements so that eventually we just fall out of the loop and execute the remaining lines of code.

We could formalise this into a test case such as TC1:

Test Case	Data	Expected Outcome
1	Number of people = 3 Weight = 54, 63, 78 kgs	While 3 > 0 (which it is) Weight = 54 Total weight = 54 Number of people = number of people (3) – 1 = 2 While 2 > 0 (which it is) Weight = 63 Total weight = 54+63 = 117 Number of people = number of people (2) – 1 = 1 While 1 > 0 (which it is) Weight = 78 Total weight = 117+78 = 195 Number of people = number of people (1) – 1 = 0 While 0 > 0 (which it isn't thus do not enter the loop) Total Weight is 195 is printed

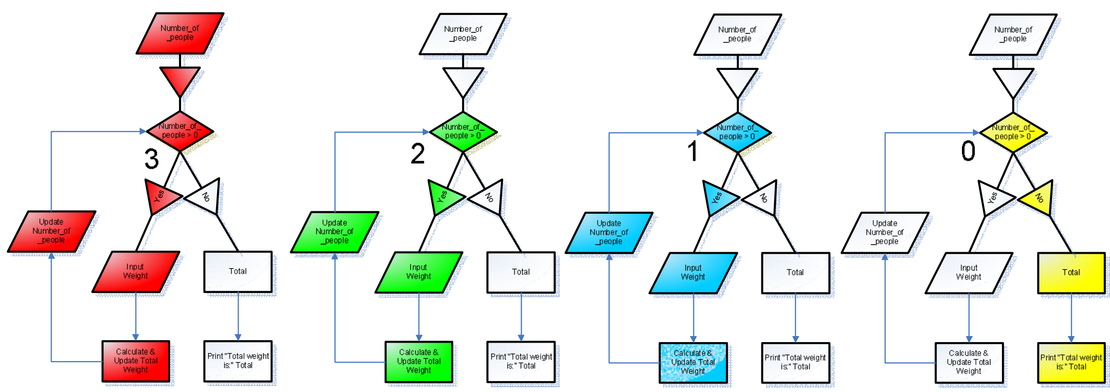

Figure 4 - Loop Coverage

Conclusion

As testers we work on all sorts of systems, including safety or life critical systems. It is important that we understand the depth that we may need to go to ensure adequate testing is done. The advent of newer lifecycles and testing approaches, and the increased access to tools that allow us to view the code allows us to understand the systems we test in more depth. To assist in enhancing the quality of software, White Box testing may be the next step we need to take. Testers need to understand the White Box techniques that are available to make educated decisions about their use for the specific systems we are currently, and in future, will be testing.

Bibliography

Everett, McLeod; "Software Testing – Testing Across the Entire Software Development Life Cycle" Wiley, 2007
Graham, Van Veenendaal, Evans and Black; "Foundations of Software Testing" Thomson (2007)
ISTQB Advanced Syllabus (v2007)
ISTQB Foundation Syllabus (v2007)
ISTQB Standard Glossary of terms used in Software Testing v2.0 (2007)
Kaner, Bach, Pettichord: "Lessons Learned in Software Testing" Wiley, 2002
Nagler "Learning C++" a Hands-On Approach"; PWS Publishing Company, 1997

Web Sites

http://en.wikipedia.org/wiki/DO-178B
http://www.scribd.com/doc/13059040/Adapting-Legacy-Systems-for-DO178B-Certification
http://en.wikipedia.org/wiki/Code_coverage