Program Testing

Black Box Testing

Introduction 1

Equivalence Partitioning 2

Basic Idea 2

Boundary Value Analysis 4

Identifying Equivalence Classes and Boundaries 5

Range of Values 5

Specific value 6

Member of a set 6

Boolean (or a Must) 7

Split classes 7

Lists 8

Worked Examples 9

Identifying Test Cases 11

Next Date Example 13

Car Park Example 15

Conclusions 18

Seminar Problems 19

Solutions to review Questions 20

Solution to Seminar Problems 22

## Introduction

Testing is concerned with detecting bugs in our software. In order to test a program we need to have sets of test cases. A test case is a set of data values, which we use as input for the program. In the previous set of notes we established that the selection of test data requires careful thought. We showed by examples that exhaustive testing was impractical; that random testing was too much of a gamble; that a large number of test cases did not necessarily make for better testing. Further both coincidental correctness and the effects of computer arithmetic could lead to false outcomes that did not reveal bugs.

We need a method for identifying test cases that exhibits the following attributes:

* It optimises the chances of detecting bugs;
* It is economical, in the sense that the work involved is appropriate to the size of software being developed;
* It does not depend on the skills of an individual tester;
* It can be applied systematically and is supported by a good documentation standard.

The literature on testing covers a variety of methods that satisfy the above. In these notes we will focus on one approach, a black box method, that can be used a various levels of software testing, from unit testing through to acceptance testing.

In black box testing we focus on the functional requirements of the program. We do not use the code to guide us in our choice of test data, although underlying our ideas we should keep in mind the structure of programs. As with any other engineering discipline concerned with building items we should be looking for the most likely sources of faults – the weak spots. The faults detected will relate to errors of omission but not completely.

Since the methods are based on the specified behaviour the outcome of such testing will tend to establish confidence in the program. However, we should always remember that testing can show the presence of bugs not the absence.

We will look at an approach we will call ***boundary value analysis.*** It is based on two simple concepts. The first concept is ***equivalence partitioning*** which is a way of grouping related data values and so making the input domain and the output domain for a program more manageable. The second concept is really just an element of common sense. If bugs are present in a program the most likely data values to detect them are in some sense extreme values.

We are going to describe the method, provide guidelines on its application and to identify the appropriate documentation standards.

##### Review Question 1

Explain why the four attributes of a method for identifying test cases, listed above, are desirable.

## Equivalence Partitioning

Partitions occur in every aspect of our daily life. An academic course is partitioned into separate units. A building is partitioned into rooms. The act of partitioning helps to make situations more manageable. In the previous examples dividing a course obviously helps plan a timetable, as does having separate rooms. We will find that partitioning is useful in devising test cases.

First we will formally define what we mean by a partition:



This simply states that the union of all the subsets equals B and that the subsets are disjoint.

##### Review Question 2

Why are the above two properties important for testers?

### Basic Idea

The input and output domains of the program is divided into a set of equivalence classes. Each class represents a set of valid and invalid values. We consider valid values because we need to see what the program will do with such data.

##### Review Question 3

Why do we need to consider invalid inputs?

Before looking at the general situation we will consider a simple example to illustrate the idea. Assume we have a program that accepts as input an integer value representing a student’s final mark, as a percentage in the range 0 to 100, for a unit. The program outputs the following messages depending on the value of the mark:

* 0<= mark <40 **Fail**
* 40<= mark <60 **Pass**
* 60<=mark < 80 **Very Good**
* 80<=mark <=100 **Excellent**

If we only use one value to test the program we obviously have not adequately exercised the code. The test cases for this program must obviously generate each of the above messages. So we have four classes: (0,39), (40,59), (60,79), (80, 100). We have identified four valid equivalence classes.

Are these four sufficient? Do we need to consider values outside the range 0 to 100?

Suppose the programmer has coded the program as follows:

if ( mark <=39)

Fail message

else

if (( 40<= mark) && ( mark <=59))

Pass message

else

if (( 60<= mark) && ( mark <=79))

Very good message

else

Excellent message

What will happen if a values of –1 and 200 are entered? The –1 generates a fail message and the 200 an excellent message. You might feel that is not a problem. However, the program has accepted invalid values, (the mark is meant to be in the range 0 to 100.), but generated output! This represents a buggy program that will need correcting.

In addition to the four valid classes we need two invalid classes: all values less than 0 and all values greater than 100.

In order to test the example program we would need to select values from each of the valid classes and invalid classes. This means we would have at least 6 data values. Any less and we have not adequately tested the program. We are moving towards a standard approach.

##### Review Question 4

Assuming that mark is declared to be of type int in a Java program what are the range of values for the two invalid classes in the above example?

Equivalence partitioning forms a basis for our test case selection. By using the equivalence classes we hoped to ensure a good coverage of the program, to reduce the redundancy in the test set and further ensure that different testers would produce test sets that were in some sense equivalent. This last outcome is very important since it removes, to some extent, the element of chance from the process of test case generation and enables us to measure the effectiveness of the test set.

We now need to build on the concept of equivalence partitioning with a view to increasing the probability of finding bugs. The question we must address is:

*Can we identify rules that will enable us to pick the representative values from our equivalence classes that have the most chance of identifying bug in the program?*

The answer to the question is yes. Studies have shown that many common bugs occur at the boundaries of equivalence classes. Suppose that the input to a program is an integer in the range 1 to 100. The most likely values that will detect bugs are 1 and 100. The reason is that many of the errors made by programmers are of the type where < is used instead of <=. These types of mistake tend only to be detected when extreme domain values are used. These extreme values should also help detect the type of calculation bugs that internal class values would detect.

Return to the above example. Suppose the code had been written as:

if ( mark <=39)

Fail message

else

if ( mark <=59)

Pass message

else

if ( mark <=79))

Very good message

else

Excellent message

A typical programmer’s mistake would be to, say, write ( mark < 59) instead of the correct ( mark <=59). A simple finger error but it could have major implications. Any value taken from the class (40, 59) other than 59 would have failed to detect the fault. Hence we will select our values by considering the boundaries of equivalence classes.

## Boundary Value Analysis

Boundary Value Analysis, (BVA), focuses on the boundaries of the input and output space in order to generate test cases. It is based on the assumption that errors are more likely to occur near the extreme values.

There are several variations of BVA. We will develop an approach, which is based on equivalence partitioning. Whenever possible we will select test values based on boundaries. We will use the information provided by both the input specification and the output specification of the program.

We expect, in general, to generate more test cases, having a higher pay off in error detection than simple equivalence partitioning.

In our approach the values of interest are those values that are

* on the boundary
* just above the upper boundary
* just below the lower boundary

Diagramatically we have:

a

b

## Identifying Equivalence Classes and Boundaries

The key to our method of generating test cases is the identification of the equivalence relation that determine the classes and where they exist the boundary values

##### Review Question 5

Identify information that the output specification of a program could supply that would be missed if only the inputs are analysed.

The following are a set of rules to help identify equivalence classes and associated boundaries. They cover most of the situations that you will meet. Note that in each case we identify both ***valid*** and ***invalid*** situations.

### Range of Values

If the input variable consists of a range of values select ***one valid*** class covering the range and ***two invalid*** classes covering the remaining values.

Given a range bounded by a and b then the test cases should be designed with values:

* a
* b
* a - ε
* b + ε

where ε is a small increment related to the physical characteristics of the variable under consideration.

example

input value is an integer in the range 1 to 999 inclusive;

valid class values between [1,999] boundary values 1 and 999

invalid class values <1 boundary value 0

invalid class values > 999 boundary value 1000

Assuming there was only the one input variable the test cases would be 0,1,999 and 1000.

### Specific value

If there is a specific value associated with the input variable we again have ***one valid*** class and ***two invalid*** classes.

The boundaries in this situation are:

* minimum number
* maximum number
* one below the minimum
* one more than the maximum

example

input value is a 4 character string;

valid class 4 character strings

invalid class strings with less than 4 characters

invalid class strings with more than 4 characters.

The situations on the boundary are a 4 character string, a three character string, a five character string.

*example*

input specifies between 2 and 4 names

valid class 2 to 4 names

invalid class less than 2 names

invalid class more than 4 names

The boundary situations are 1 name, 2 names, 4 names, 5 names.

### Member of a set

If the input values are taken from a finite set of discrete values the should be one classes for each valid value and one invalid class formed with a value not in the set.

In this case there are no obvious boundary values.

example

The set of possible input values are bus, truck, taxi.

valid class bus

valid class truck

valid class taxi

invalid class any\_other\_value

### Boolean (or a Must)

The input must satisfy a specific condition then there will be one valid class covering the condition and one invalid class.

example

The first character of the name must be an upper case letter.

valid class name starting with an upper case letter

invalid class any name not starting with an upper case letter.

In this example we have a situation that is closely related to the range situation. The values valid class is uppercase letters. This represents a range from ‘A’ to ‘Z’ and we should select values for the first character of ‘A’ and ‘Z’. There is no boundary for the invalid class.

example

The first character of the name must be a punctuation character.

valid class name starting with a punctuation character

invalid class any name not starting with a punctuation character.

In this case there are no boundaries for either the valid nor invalid classes.

### Split classes

If there is any reason to believe that the input may be divided into groups of values that are processed differently the split the input into separate classes.

example

The input values are a range of integers with values between 1 and 100 but even numbers will be treated differently from odd numbers.

valid class even numbers in valid range

valid class odd numbers in valid class

invalid numbers greater than 100

invalid classes less than 1

The boundaries would be the smallest positive number in the range and the biggest positive number in the range, the smallest odd number in the range and the largest odd number in the range together with the two invalid boundaries of 0 and 101.

### Lists

If the input or output consists of a list of items design test cases to exercise the data structure on its boundaries.

example

the input islist with 100 entries

the test cases should

* focus on first entry
* focus on last entry

##### Output Considerations

You must consider the output from the program and design test cases based on the attributes of that output using previous guidelines.

example

Supposing the output from a program is a table giving values of temperature against pressure. You would attempt to generate test cases that:

* produced a table with the minimum number of entries
* produce a table with the maximum number of entries
* try for a test to produce a table one less than the minimum
* try for a test to produce a table one more than the maximum

##### Hierarchical Considerations

When designing test cases it may be necessary to take a hierarchical view of the inputs and outputs with each level in the hierarchy leading to its own set of test case requirements.

example

The input to a program comes from a file. The file consists a records, (structs in C++). There are two types of record in the file. Each record is made up of a number of fields.

The properties of the file of interest are:- it exists, it does not exist, it exists but is empty, it exists and has one record, it exists and is of maximum length.

The properties of the records of interest are:- type 1 is first record, type 1 is last record, there are no type 1 records, type 2 is first record, type 2 is last record, there are no type 2 records.

We can finally consider the individual fields in each record. They would be analysed as normal.

##### Review Question 6

For each of the following identify the boundary values:

1. a variable may take values in the range -1 up to 100
2. a variable may take values in the range 0.01 to 11.25
3. a variable may be a lower case letter
4. a name may consist of between 5 and 20 letters
5. a code consists of between 1 and 3 characters
6. the input may consist of up to 16 names

##### Review Question 7

A program generates a multi page report. There is no upper limit to number of pages in the report but there will always be at least one. A page will consist of a header and a body. The body of the page is a number of lines. The last page of a report need not be full.

What properties would be of interest in generating test cases?

**It is important to note that one input variable may have several attributes each generating equivalence classes and boundaries..**

### Worked Examples

###### Example 1

We start by considering the example we have looked at above :Assume we have a program that accepts as input an integer value representing a student’s final mark, as a percentage in the range 0 to 100, for a unit. The program outputs the following messages depending on the value of the mark:

* 0<= mark <40 **Fail**
* 40<= mark <60 **Pass**
* 60<=mark < 80 **Very Good**
* 80<=mark <=100 **Excellent**

The input consists of a single integer in the range 0 to 100. Thus applying the first guideline we need one valid class in the range 0 to 100 and two invalid classes: one for values less than 0 and one for values greater than 100. However, there is reason to believe the valid range consists of subclasses: namely (0,39),(40,59),(60,79),(80,100). So we have 4 valid classes. The test values we generate should be { -1, 0,39,40,59,60,79,80,100,101}. We have two invalid values and 8 valid ones.

Any individual applying this method to this examples would have the same values. We have consistency. We are not dependant upon the skills of an individual.

###### Example 2

Now we look at a more complex example. Consider a program whose input consists of three variables with the following attributes:

max : an integer in the range 10 to 20 inclusive

code : may be a F or P

names : between 1 and 4 strings, each starting with an upper case letter and followed by up to 3 digits.

The variable “max” is a simple range. The equivalence classes and boundary values can be identified directly from the above guidelines. It is also subject to a must condition. It must be of type integer.

The variable “code” takes values from a set of discrete values. The required test values can be identified using the guidelines.

The variable “names” is more complex. It has several attributes:

* The number of strings is a range.
* A Boolean condition covers the first character. It must be an upper case letter.
* The size of each string is a range
* The value of the remaining characters of the strings is a must.

We record our analysis of the inputs in a table with each class assigned a number for future reference. Note we start numbering in the valid columns and work down. When we have finished numbering the valid situations we number the invalid ones. The reason for working in this manner will become obvious later.

One interesting situation arises in the invalid boundaries for the number of names. We find we want 0 names. It may not be possible to submit such a situation to the program but we should consider the possibility.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Input Condition** | **Valid equivalence class** | **Valid Boundary Values** | **Invalid equivalence class** | **Invalid Boundaries** |
| **max**  value | 10-20 | 101  202 | <10  >20 | 914  2115 |
| type | integer3 |  | other16 |  |
| **code**  value | F,P | F4  P5 | other17 |  |
| **names**  number | 1 to 4 | 16  47 | <1  >4 | 018  519 |
| 1st character | upper case letter | A8  Z9 | other20 |  |
| size | 2 to 4 characters | 2 char10  4 char11 | <2  >4 | 1 char21  5 char22 |
| last characters | digits | 012  913 | other23 |  |

***Notice that if there isn’t a boundary value we number the equivalence class.***

## Identifying Test Cases

Once the analysis, i.e. the identification of equivalence classes and associated boundary values, has been competed we must use the outcomes to generate the actual test cases. We must identify the actual data values that we will present to the program.

We want to identify two types of test cases – ***valid test cases*** made up of values selected from the valid equivalence classes and boundary values; ***invalid test cases*** made up of values taken from the invalid classes and boundary values. There are a variety of rules we can use to combine values to make up the test cases. The ones we use are relatively simple and do not necessarily generate the “best” set of test cases. However, they are sufficient to demonstrate the method. For real-life testing more sophisticated rules should be used.

The rules we use for generating test cases from the analysis table are:

* until all **valid** situations are covered write test cases covering as many uncovered situations as possible;
* until all **invalid** situations are covered write test cases to cover one and only one uncovered invalid situation.

(by ***situation*** we mean a numbered entry in the analysis table)

When generating the test cases we should use the rules as guides but also ensure that we generate sensible cases in the context of the problem under consideration.

##### Review Question 8

Explain the reason why we only consider one invalid situation per invalid test case.

Continuing example 2 we obtain the following test cases:

|  |  |
| --- | --- |
| **Valid Test Cases** | **Situations Covered** |
| 10 F A0 | 1,3,4,6,8,10,12 |
| 20 P Z999 B123 C45 D1 | 2,3,5,79,11,13, |

|  |  |
| --- | --- |
| **Invalid Test Cases** | **Situations Covered** |
| 9 F A0 | 14 |
| 21 F A0 | 15 |
| & F A0 | 16 |
| 10 Z A0 | 17 |
| 20 P | 18 |
| 20 P Z999 B123 C45 D1 E1 | 19 |
| 20 P a1 | 20 |
| 20 P B | 21 |
| 20 F C1234 | 22 |
| 10 F AA | 23 |

Any tester using our method, Boundary Value Analysis, for the scenario of example 2 would produce test cases that exhibit certain essential commonality. There would be

* Exactly 2 valid test cases, no more and no less;
* Exactly 10 invalid test cases, no more and no less;
* In the valid tests
* The values 10 and 20 would appear;
* F and P would be used;
* A name would start with an ‘A’ and one with a ‘Z’;
* A name would contain the digit ‘1’
* A name would contain the digit ‘9’;
* There would be 1 name;
* There would be 4 names;
* There would be a 2 character name;
* There would be a 4 character name;
* The invalid tests would similarly be contstrained.

In a sense there would be a “correct” but not unique set of test cases. This reduces the influence of the individual tester. It provides a basis for analysing the effectiveness of the tests from the point of view of code coverage and redundancy.

**Note we are not saying that the data generated is perfect. Nor that we have identified all the possible situations we might want to cover. In testing there is no such thing as a perfect test case generation method!**

## Next Date Example

The program accepts as input a date in the form of three integers represnting the day, the month, and the year. The year is constrained to lie in the range 1812 to 2012. The output from the program is the date of the next day in numeric format.

A simple view of the problem leads to the following set of equivalence classes:

Variable Equivalence Classes

Day 1<=day<=31

Month 1 <= month <= 12

Year 1812<= year <= 2012

Which leads to the following table

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Input Condition** | **Valid equivalence class** | **Valid Boundary Values** | **Invalid equivalence class** | **Invalid Boundaries** |
| **Day**  value | 1 - 31 | 11  312 | <1  >31 | 010  3211 |
| Type | Integer3 |  | Other12 |  |
| **Month**  value | 1-12 | 14  125 | <1  >12 | 013  1314 |
| type | Integer6 |  | Other15 |  |
| **Year**  value | 1812- 2012 | 18127  20128 | <1812  >2012 | 181116  201317 |
| type | Integer9 |  | Other18 |  |

Application of the rules lead to the following test cases:

Valid

1 1 1812

31 12 2012

Invalid

0 1 1812

32 12 2012

x 1 1812

1 0 1812

31 13 2012

31 x 2012

1 1 1811

31 12 2013

31 12 x

These obviously form an inadequate set of test cases. There are many situations that are not covered. The problem arises from a poor analysis of the problem inputs.

A better and more realistic set of equivalence classes would be:

Variable Equivalence Classes

Day 1<=day <=27

28,29,30,31

Month month has 30 days

month has 31 days

Feb

Year year is leap year

year is common year

Now have very much more complex set of classes and boundaries which are more likely to reflect structure of program. This leads to the following analysis table:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Input Condition** | **Valid equivalence class** | **Valid Boundary Values** | **Invalid equivalence class** | **Invalid Boundaries** |
| **Day**  value | 1 – 27  28, 29, 30, 31 | 11  272  283  294  305  316 | <1  >31 | 026  3227 |
| Type | Integer7 |  | Other28 |  |
| **Month**  30 day month | ,48,69,910,1111 |  | <1  >12 | 029  1330 |
| 31 day month | 112,313, 514, 715, 816,1017, 1218 |  |  |  |
| Feb | 219 |  |  |  |
| type | Integer20 |  | Other31 |  |
| **Year**  value |  |  | <1812  >2012 | 181132  201333 |
| leap years | Set of leap years | 181221  201222 |  |  |
| common years | Set of common years | 181323  201124 |  |  |
| type | Integer25 |  | Other34 |  |

Some of the entries in the table need a little explanation. Why are all the months enumerated but not all the years? There is no way of calculating the number of days in a month. It is just a given fact. A programmer would need to write code that identified each month. However, there is a formula that is used to determine if a given year is a leap year or a common year. Hence we can treat the two sets of years as ranges. (The above table is slightly incorrect, since although in general a leap year is one that is divisible exactly by 4, there is a different rule for centuries which mean that 1900 was not a leap year but 2000 was).

The next step would be to select the test cases. In doing this we should use common sense. Select combinations of values from the table that have some meaning.

## Car Park Example

Parking charges at the local car park are based on three categories of user. Cars are charged in the following manner: 50p per hour for the first 2 hours, 30p per hour for the next three hours and 10p per hour for any subsequent hours. Larger vehicles, (which we will term trucks), are charged at the following rates: 150p per hour for the first 2 hours, 100p per hour for the next three hours and 50p per hour for any subsequent hours. Senior citizens can park with no charge.

A program has been implemented to calculate the cost of parking and we are going to design test cases for the program.

The input to the program consists of the following:

User type : C,T,S

Time in minutes: an integer in range 1 to 1440

(for this example we assume no vehicle can park for more than 24 hours)

The output from the program will be one of:

The parking cost: a real number in pounds

The error message: *invalid code*

##### Some initial thoughts on the input

The user-type variable’s equivalence class is generated using the member of set guidelines.

The time variable requires more careful thought. At first glance it looks like a simple range. However, the range needs to be split into sub ranges - 1 to 120, 121 to 300, and 301 to 1440.

From this analysis we can produce the following table:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Input condition** | **Valid equivalence class** | **Valid boundary values** | **Invalid equivalence class** | **Invalid boundaries** |
| **User\_type**  value | C, T, S | C1  T2  S3 | other11 |  |
| **Time**  value | 1-120 | 14  1205 | <1 | 012 |
|  | 121-300 | 1216  3007 |  |  |
|  | 301-1440 | 3018  14409 | >1440 | 144113 |
| type | integer10 |  | other14 |  |

Before we attempt to generate the test cases we need to revisit the program specification. It would seem obvious that different parts of the program will calculate the charges for different user types. If we apply the simple rule about covering classes and boundaries with the minimum number of valid tests we will not have an adequate test set. In this example we must use our common sense and pair the category cars with each time subrange and similarly trucks.

##### The test cases based on input:

|  |  |
| --- | --- |
| **Valid** | **Situations Covered** |
| C 1 | 1,4,10 |
| C 120 | 1,5,10 |
| C 121 | 1,6,10 |
| C 300 | 1,7,10 |
| C 301 | 1,8,10 |
| C 1440 | 1,9,10 |
| T 1 | 2,4,10 |
| T 120 | 2,5,10 |
| T 121 | 2,6,10 |
| T 300 | 2,7,10 |
| T 301 | 2,8,10 |
| T 1440 | 2,9,10 |
| S 100 | 2,10 |

|  |  |
| --- | --- |
| **Invalid** | **Situations Covered** |
| Z 1 | 11 |
| C 0 | 12 |
| C 1441 | 13 |
| T 0 | 12 |
| T 1441 | 13 |
| C & | 14 |

The reason for using invalid boundaries 12 and 13 twice is because we expect different parts of the code to be executed dependant on the user-type value.

##### Analysis of output

There are two types of output - the cost or an error message. We can calculate the minimum parking cost and given the previous assumption about the maximum time period we have a maximum cost of £15.50.

Thus we have:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Output condition** | **Valid equivalence class** | **Valid boundary values** | **Invalid equivalence class** | **Invalid boundaries** |
| **Parking Cost**  Value | 0.0 -15.5 | 0.01  15.52 | <0.0  >15.5 | -0.54  16.05 |
| **Error Message** | “invalid code”3 |  | none |  |

From this table we can identify the need for test cases to generate the five outputs identified in the table. This gives

|  |  |
| --- | --- |
| **Valid** |  |
| S 100 | 1 |
| T 1440 | 2 |
| A 100 | 3 |

Each of these test cases has an equivalent in the test cases based on the input analysis.

It is not possible to devise a case to cause a negative output for the parking cost. A cost of £16.00 would be generated by the test case T 1441 and that is already covered.

## Conclusions

We have now completed our discussion of equivalence and boundary value analysis. The BVA method enables us to systematically generate test cases through an analysis of the program’s input and output requirements. The method will give good results and should be used as part of any software testing.

However, BVA is not perfect. No currently used method of test case generation is perfect. In order to obtain better test coverage it is essential that several alternative methods are used to arrive at a test set. There are several alternative black box methods. These could be used to support our efforts but since they will tend to be based on the program specification they may not add much to the test set. Rather it is better to supplement black box testing with white box testing. In the next set of notes we will look at such techniques.

Finally it is worth re-emphasising that the objective of testing is to detect bugs. When applying any method of test case generation always use common sense. Try and identify any relationships between variables. If after all the test cases have been generated it seems to you that certain obvious values have not been used - then use them. A simple illustration of this could be the use of 0 as the input to a program that performs some arithmetic when the input variable is in the range -100 to 100.

**Solutions to review Questions**

**1.** The purpose of testing is to detect bugs soany method should be designed to maximise that aim. If the method is designed just to show the program works then we could be releasing buggy software. optimises the chances of detecting bugs;

The production of test cases and the analysis of results takes time and effort. In real world programming this means money. Obviously the testing effort should be appropriate to the nature of the application.

In companies there is always a staff turnover. Quality can not depend on a single person. It must be something that can be guaranteed irrespective of individuals.

By having a method, which is well understood its weaknesses and strengths can be identified and accommodated. The life of commercial software goes far beyond the initial development stage. Software must be maintained. In the maintenance phase good documentation is essential.

**2.** The first property guarantees that every element of B is in some subset while the second guarantees that no element of B is in two subsets. These properties yield important assurances: completeness and non-redundancy.

Two potential disadvantages of black box testing are gaps and redundancies: some things remain untested while others are repeatedly tested.

**3.** We need to consider invalid inputs because we are looking for bugs in the program. If the program can accept garbage and apparently produce results we may need to modify the program to make it more robust.

**4.** In Java the range of value for type intis defined to be -2147483648 to 2147483647. So the two classes are (-2147483648, -1) and (101, 2147483647)

**5.** Error conditions; subcases

**6.**

a) -2,-1,100,101

b) 0.00,0.01,11.25,11.26

c) a,z - no invalid boundaries

d) 4,5,20,21 letters

e) 0,1,3,4 characters

1. 0,1,16,17 names

**7.** The properties of the report of interest would be:

* an empty report
* one line report
* single page report with page full
* full page plus one line report

The properties of the lines of interest would be:

* one character line
* full line
* full line plus one char

**8.** Because we do not want to mask the effect of bad data. If we input two bad values how do we know which caused the resulting behaviour?