BOOK DRAFT—DEVELOPING SOFTWARE TEST CASES

Monday, September 23, 2003 (tcase06equiv.draft02.doc in bookreview)

Ross Collard

rcollard@attglobal.net

Chapter 6
Equivalence

Introduction

How Equivalence Works

Equivalence is similarity or commonality. We use equivalence to group similar conditions together, and pick a representative member from each group to use as a test case. For example, let's assume that we need to test a system which prints checks. We create a test transaction, a request to print a check to a person named John Smith for $85.33. The system processes this test transaction, and as expected prints a check addressed to John Smith for $85.33. (A test transaction is just a test case in which the input takes the form of a transaction.)

Since the system worked correctly with this test transaction, do we need to investigate its behavior if we request a check for John Smith in the amount of $85.32 or $85.34? Probably not. Most of us are willing to assume that the behavior of the system is essentially the same ("equivalent") under these three different conditions. Similarly, if the first transaction fails to print a check addressed to John Smith for $85.33, can we assume that the other two test transactions will fail too, and therefore not bother to process them? Most testers would say yes.

Of course, these are only assumptions, not known facts. It could be (though we don't know this from our perspective outside the system), that John Smith has exactly $85.33 in his bank account. The first transaction for $85.33 works, but a request to print a check for John Smith for $85.34 or more will not be honored. I am assuming that we restore the original bank balance after the first test case has run, ready for the second test case. What if the system prints one check correctly but for some strange reason cannot print a second check for the same person on the same day? We would not find this bug if we assume equivalence and use only one test transaction.

Most of us instinctively use equivalence while we are testing. If one test case results in a certain behavior, whether acceptable or not, we simply assume other equivalent test cases would behave a similar way without running them. For example, if we enter a non-numeric character string into a numbers-only data entry field and the system rejects this string, we usually assume that the system will reject any other non-numeric entry too. If we say that a date of January 29, 2005 is equivalent to May 13, 2005, we are assuming that transactions processed on these dates will result in similar outcomes. We are not likely to learn anything extra by using both dates as test cases; one will suffice. Did you hear the one about the fisherman who sent the U. S. Internal Revenue Service a truckload of smelly fish? When asked why, he explained with a shrug that he owed the IRS back taxes and said: "Its equivalent."

Assumptions about Commonality versus Variance

Equivalence is based on what we think is the same, or different, from situation to situation. Therein lays the strength of the technique, and also its dangers: equivalence is only as good as the assumptions we make about what conditions are basically the same and what are different.

To see the dangers of equivalence, consider this scenario. We need to test a GUI-based feature which is supported by Windows XP. The user community has Windows XP machines which are identical in every way, except that they have either 128 MB (megabytes), 256 MB or 512 MB of semiconductor memory installed. In the test lab we have exactly one Windows XP machine available at the moment, with 128 MB of memory. We test the feature in the lab, and it works. We are willing to assume equivalence—if the GUI feature works on our test machine with 128 MB, then we assume it will also work on the users' machines with 128 MB or more memory. So the testing is complete.

Now we learn that some users have Windows XP installed on machines with only 32 MB of memory, which is less than Microsoft's minimum recommended configuration. We previously were willing to assume equivalent behavior if the amount of memory resource increases. In other words, if the feature works with 128 MB, it will work acceptably with a greater amount of memory. Can we also assume equivalence if the amount of memory is *decreased*? This is a dubious premise -- Windows does not work well with less resources. We are reasonably confident that going from 128 MB to 256 or 512 MB of memory produces equivalent behavior, but going to 32 MB does not.

Making decisions about equivalence is like taking an IQ test, where we are asked to see patterns and classify, group and compare items:

- A software bug is to a tester as a creepy crawly biological bug is to (a) a picnic party full of bug-phobic people, (b) a curious little boy who likes to surprise girls, (c) a PhD entomologist or (d) a voraciously hungry bird.

- Which item does not belong in this group: (a) horrendous bug, (b) catastrophic defect, (c) terminally embarrassing fault, or (d) honest little mistake by a well-meaning, over-worked, and under-appreciated software engineer? (With dubious humor like this, thank goodness I don't have to make my living as a comedian.)

There are two types of equivalence: intuitive ("it is obvious that this is essentially the same as that"), and specified ("the specs say that this is essentially the same as that"). Why do testers care about the type of equivalence? Tbd.

xxxxxxxxxxxxx

The domain is continuous and fully populated.

The members of the domain or range can be linearized -- mapped into points on a line.

xxxxxxxxxxxxx

Basic Equivalence Terms

An *equivalence class* is any set of testable conditions (e.g., the variations in the input data values of a transaction), which produce the same or equivalent outcomes. In other words, if the system works for one item within a class, it should work for all. We will use the abbreviation EC for an equivalence class. Note the word "class" in this context is not the same as a class in object-oriented systems.

An *equivalence test* is a test which uses only a small sample of all possible test cases, based on commonalities and usually with one test case included for each major equivalence class. The selected test cases uncover (in theory) as many defects as an exhaustive test of all conditions would have uncovered, but in practice this is unlikely because of imperfections in our equivalence classes. The key question in equivalence testing is one of test case selection: What subset of all possible test cases is likely to detect the most errors with the least effort?

An *equivalence test case* is one which is selected from an equivalence class and actually will be run. A good equivalence test case allows us to generalize about a family of failures which one underlying bug causes, and possibly also generalize from that one bug to a whole family of similar bugs. Two test cases belong to the same EC if we are reasonably confident that they will give the same pass / fail results, for example, if one fails then the other will also fail for the same reason. Testing multiple test cases from the same EC is in theory redundant, but we may still do it if are confidence of equivalence is low or the risk is high.

The *best representative value* within an equivalence class is the most appropriate one to select as the test case for that class. In theory, the members of an equivalence class are all equal and thus we can pick any one at random. In practice, some members of an equivalence class are more "equal" than others, and later we will examine how to assess their value.

Exercise 6.1: Equivalence—Data Entry Character, Part 1

(Allow 15 to 20 minutes for this exercise.)

An input data field is exactly one character in length. The valid input values for this data field include all the alphabetic characters, both upper case and lower case. The person who was assigned to test the edit logic for this input data field has developed this suite of four test cases:

Test Case Number	Input Data Value	Expected Outcome
1	B	Accept
2	g	Accept
3	%	Reject
4	Alt-F4 combination	Reject

What assumptions of equivalence has the tester made?

Answer to Exercise 6.1

We may not necessarily agree with them, but these are the main assumptions behind the suite of test cases:

- All upper-case alphabetic letters are equivalent. If the system behaves in a certain way for a "B", then it will behave in an equivalent way for an "A", "C", "D",..., "Z".
- All lower-case alphabetic letters are equivalent: "a", "b", "c", etc.
- Upper-case and lower-case versions of the same alphabetic letter are not equivalent. If the system behaves in a certain way for a "B", we are not willing to assume it will behave in an equivalent way for a "b".
- All single-character non-alpha characters are equivalent. If the system behaves in a certain way for a "%", then it will behave in an equivalent way for an "*", "3", blank, etc.
- Pressing a special or control key, such as the tab or backspace key, is equivalent to pressing any single non-alpha character key, such as the asterisk.
- All multi-character key combinations and special characters are equivalent. If the system behaves in a certain way for the "Alt-F4 combination", then it will behave in an equivalent way for a "Shift-F9", or for combinations of three or more keys pressed together.
- It does not matter whether the system uses English or another language such as German, which has the same characters plus others that are invalid in English.

An Example of Equivalence

In an earlier chapter, we saw this specification:

> *"Generate a reminder notice for each customer who has an outstanding balance due for more than 30 days. We count the 30 day period from the day when we received the customer's last payment, or from the day when the customer was last sent a bill or a reminder notice."*

The suggested set of test cases for this specification was:

> *Generate a reminder notice for a customer with an outstanding balance who has not paid for a period of: (a) 31 days, (b) 45 days, (c) 90 days, and (d) 180 days. (The customer cannot have already received a new bill or a reminder notice within the last 30 days.)*

Are these four test cases sufficient, or do we need more? Or are the four test cases overkill? We will use equivalence to understand the assumptions behind the test cases, and examine whether these are effective test cases.

For simplicity, I'll limit this discussion of equivalence to just one variable, the number of days since the customer last paid us (the "elapsed time"). The question is: What are the equivalence classes for the elapsed time since the last payment, billing or reminder date? At a first guess, there are three equivalence classes here:

Equivalence Class	Elapsed Time	Expected Behavior
EC #1	0 to 30 days	Do not generate a reminder notice
EC #2	31 days to infinity[1]	Generate a reminder notice
EC #3	Negative or non-numeric	Reject as invalid

Are these Equivalence Classes Correct?

If we have correctly defined EC #2 as one equivalence class, then we need only test case—let's say a person who is overdue by 37 days. The test strategy suggested earlier included four test cases, for accounts overdue by 31, 45, 90, and 180 days, so this appears to be three test cases too many. Are there circumstances in which we need all these test cases? Yes, if the risk is so high that taking more than one sample from the equivalence class is prudent. Or it may be, after closer examination, that EC #2 needs to be split into more than one class because there are variances in the behavior of its members.

Imagine that the processing of the reminder notices actually works as follows:

1. The customer billing system takes no action when the elapsed time is 30 days or less.
2. When the elapsed time equals or exceeds 31 days, up to 44 days, the system automatically generates a reminder notice. I am assuming that the other prerequisites have been met, such as having an outstanding balance which is high enough to qualify for a reminder notice.
3. When the elapsed time equals or exceeds 45 days, the system transfers the customer account to a delinquent accounts system. This system generates its own reminder

[1] Infinity is not a good choice of an upper limit of an equivalence class (or any range), because computers cannot work with unbounded numbers. A better choice would be to find the upper limit on delinquency which is supported by the system (let's say it is 365 days), and use this as the upper limit. In addition, there may be unintended changes in behavior, for example, when the delinquency is exactly 365 days old, or 366 days in leap years. I once saw a system where a customer account, which was delinquent by one year and one day, was treated as if it was only one day old.

notices for people who are overdue from 45 days to 89 days. Just because the standard system generates reminder notices correctly, we cannot assume that this other system for delinquent accounts does too.

4. When the elapsed time equals or exceeds 90 days, the system transfers the account again, this time to a system for severely delinquent accounts which also has its own reminder notices.

5. Finally, when the elapsed time equals or exceeds 180 days, the account is transferred again, but this time it is outsourced to the Mafia.

Under these circumstances, there are four distinctly different behaviors within class EC #2. (The first statement above, regarding the system's behavior when the elapsed time is 30 days or less, is not part of EC #2.) These changes of behavior as the balance ages violate the concept of equivalence, so the original EC #2 has to be split into multiple equivalence classes. We need four test cases, one per equivalence class, as we had originally.

Supplementary Test Cases

These four test cases are not necessarily all we need. Other aspects of the reminder notice process which need to be tested include:

- The transfer of a delinquent account from one billing system to the next as the balance ages. With the basic set of four test cases, we would never detect if a system inadvertently drops an overdue account instead of transferring it to the next system.

- The preventive controls to avoid sending a new reminder notice every day (or every reminder processing cycle). For example, consider an account which was overdue yesterday, let's say by 36 days. Then today when this same account has aged to 37 days, the system should not generate a new reminder notice.

- Situations where we should not send a reminder notice: for example, situations where the account is not overdue, or is overdue by 29 days or less. This example and the next are outside EC#2 but still need to be tested.

- Invalid test cases: e.g., where the number of days overdue is a non-numeric value or a negative value.

The message is that we cannot rely on equivalence as our only testing technique.

Combining and Splitting Equivalence Classes

I like to formulate a first draft of the equivalence classes for a test situation, and then review and adjust them. Questions I ask in reviewing the draft are:

(1) Do the members of an equivalence class exhibit different behaviors? If so, and if these are differences that we care about, we need to subdivide the class into multiple equivalence classes.

(2) Do the members of two or more equivalence classes have the same behavior? If so, we can combine them.

Consider the situation where we have these two distinct equivalence classes:

Equivalence Class	Sample Member of EC
Male customers who are 38 days overdue on balances over $10	John Smith
Female customers who are 38 days overdue on balances over $10	Mary Johnson

Let's say that the system cannot differentiate; it has no way of knowing whether a customer is male or female. In this circumstance we can combine the male and female classes into one, a customer equivalence class. If the system works in a certain way for John Smith, we are comfortable it will work the same way for Mary Johnson.

Is there any situation where we need separate male and female equivalence classes? The answer to this question is another question: where is the gender important in how the system works? For example, let's say that the wording of the billing reminder is different for females (appealing to reason and high intellect) than for males (appealing to craven fear). In this situation, we need separate equivalence classes. If instead of a customer billing system we are testing a medical information system, then differentiating between males and females is important.

Selecting the Best Representative Test Case

The best test case in a class is the one which we expect to provide the most useful information. Either this test case is the one most likely to uncover a defect, or, if the primary purpose of the test case is not to find bugs but to confirm that the system's behavior is acceptable, it is the one most likely to represent all the members of the class. If the members of the equivalence class appear to be equal in their ability to provide information, then the best representative test case is the one which requires the least effort to prepare, execute and evaluate.

Let's say that we need to test a feature which has to work on personal computers with various versions of Microsoft Windows. We are fairly confident that the set of Windows operating systems form an equivalence class, and our primary testing objective is to find bugs. We suspect that the feature will have the most trouble running under Windows 98 and therefore the bugs will

be easiest to see with this operating system. We would select Windows 98 as the best representative operating system for the purpose of finding bugs.

Incremental Equivalence Testing

Testing Incrementally

In an incremental approach, we first run one test case for an equivalence class, the best representative test case, and then examine the test results. If the first test case fails or provides inconclusive results we try other related test cases. These test cases could be within or outside the equivalence class, or both. Part of this exploration is to re-examine the assumptions behind the set of equivalence classes, and if necessary re-work these equivalence classes.

What if instead the first test case passes? This could lead to a false confidence that we have tested and finished the entire class. What else, if anything, can we do after the first test case passes? If the risk is high enough to justify the effort, we could deliberately run a second test case from the same equivalence class to double-check. This second test case in theory is unnecessary—unless we are not fully confident of our equivalence classes. Sometimes test results fall into a twilight zone where we are not sure if a test case passed. These border-line results should raise our suspicions. If we lack confidence about an equivalence class, we can pre-define the expected test results in three categories:

(1) Unambiguous pass—if the test results fall into this category, the feature needs no further testing.

(2) Ambiguous pass—the test results appear to be valid, but if they fall into this ambiguous category, we advise additional exploration.

(3) Fail—the test results appear to be invalid, and we need additional exploration.

What if we run a few equivalence test cases and obtain dubious results (a mix of passes and fails, or all passes but with results that are suspect for some reason)? What if we are unsure of exactly where the boundaries of the equivalence classes lie, and what items are within an equivalence class or not? These questions about whether items are members of a class do not occur only at the boundaries or extremes. Equivalence classes do not have to be continuous, such as a contiguous range of numbers. A system may behave differently, for example, if a number is an integer which is divisible by three, than if it is not divisible by three.

In these situations, most testers adapt their plans during the testing, based on successive iterations of feedback from the results of the test cases which were already run. This method is called successive approximation in some circles. It works like this: choose two test cases initially, one a valid member of the equivalence class and one which is not a member. Execute these two test cases and examine the results. If both test cases work as expected, the test results are not suspicious, and the situation is low risk, then we consider the equivalence testing to be complete. If both test cases do not behave as expected, or the risk is high, run additional test cases. The new test cases should be both positive (valid) and negative (invalid). We can choose

ones close to where we think the boundary of the equivalence class lies, or scatter them at random.

Applying Equivalence

This section recaps the main questions to address in applying equivalence. We need to ask:

- For what dimensions or characteristics of the test situation should we analyze equivalence?
- What are the equivalence classes for this situation?
- Taken together, do these equivalence classes cover all the possible testable conditions within the scope of this testing task, e.g., all the variations of the inputs and initial conditions?
- Can two or more equivalence classes be merged together, because there is no important difference in their behavior?
- Does any equivalence class need to be split apart, because its members do not behave in equivalent ways, i.e., have material variances in their behavior? (Could the behavior of one member of the Equivalence class fail to predict the behavior of the others?)
- What are the boundaries of these equivalence classes? Are they distinct, clear boundaries or arbitrary, fuzzy boundaries? (Fuzzy boundaries are discussed in more detail later in the book, under the heading “Boundary Value testing”.)
- How many samples (i.e., how many test cases), do we need within each equivalence class? Is one enough? Or, because of the risk and complexity, is more than one test case justified for the class?
- How do we categorize and cluster together all possible test cases into equivalence classes?
- Is any member of an equivalence class preferable to its neighbors, because of convenience and the logistics of executing the test?
- What is the best representative value within each equivalence class, to select as a test case?
- What assumptions about equivalence are we willing to make—and live with?

Exercise 6.2: Equivalence—Data Entry Character, Part 2

(Allow 30 to 45 minutes for this exercise.)

An input data field contains exactly one character. The risk associated with incorrect data being entered in this field is high, so a thorough analysis of how to test it is justified.

Valid input values for this data field include all the alphabetic characters, both upper case and lower case, and the blank character. Everything else is invalid: the numbers, non-alphanumeric characters such as "?" and "%", and multiple-key combinations which are pressed simultaneously such as Alt-F4. Control characters, such as Escape, Backspace, Delete, Left arrow and Right arrow, are enabled but, if pressed, are expected to have no effect on the data field. The Caps Lock, of course, is expected to work and to change alpha characters to upper case.

The software provides a capability to correct an input error by allowing the user to re-key the input. If a user presses a wrong key, he or she then can key in a correction which overlays the wrong value. The system does not read and edit the input character until the Enter key is pressed.

What are the equivalence classes for this input data field?

Answer to Exercise 6.2

According to the following analysis, there are seven valid and eleven invalid equivalence classes for this data entry field:

Valid Equivalence Classes

Equivalence Class (EC)	Description	Test Case (Sample Member of the EC)
A(1)	Upper case alpha character	B; <Enter>
A(2)	Lower case alpha character	G; <Enter>
A(3)	Blank	; <Enter>
A(4)	Multiple character string (not pressed simultaneously)	R?kJ*Um; <Enter>
A(5)	Non-English alpha character	Ñ; <Enter> or ç; <Enter>
A(6)	Repeated use of this data entry field	K; <Enter>; D; <Enter>
A(7)	Alternative data entry (e.g., by using a numeric keypad instead of the main keyboard, clicking on a pull-down list, scanner input or voice recognition, instead of keyboard entry)	<Click to pull down list>; <Click to select V>; <Enter>

Chapter 6
Equivalence - Answer to Exercise 6.2

Assumptions

This set of equivalence classes is based on several premises (assumptions):

Upper Case Alpha Characters

By saying that A(1) is an equivalence class, we are stating that we are willing to assume equivalence across all upper-case alpha letters. In other words, if the system works correctly (i.e., accepts the letter B), we are willing to assume that it would have worked the same way with any of the letters A, C, D, ... Z. Alternatively, if this test case (B) fails, we are willing to assume that the others would fail, and fail in the same way.

Are there any circumstances in which we are willing to make these assumptions? Consider the situation of keyboard manufacturers: they probably would not be willing to assume that if one key works, they all work. A keyboard manufacturer might decide instead that each individual letter is its own equivalence class. If the B key works, they are not willing to assume that the A, C or Z works also, and test each letter.

Lower Case Alpha Characters

By saying that A(2) is an equivalence class, we are making two assumptions:

(1) We are willing to assume equivalence across all lower-case alpha letters.

(2) We are not willing to assume equivalence across upper-case and lower-case letters. (A(1) and A(2) are two separate equivalence classes.)

Perhaps we have some basis in prior experience of being bitten by assuming that a data entry field was not case-sensitive, which later caused a problem.

Multiple Character Strings

Equivalence class A(4) assumes that the length of the input string of characters is irrelevant—if the system works correctly for any string entered before the <Enter> key is pressed, it will work for all strings regardless of their lengths. In other words, if a system works correctly for a string of two characters, e.g., *m; <Enter>, then we are willing to assume it works also correctly for a string of seven characters, such as R?kJ*Um; <Enter>, twenty characters, thirty-five characters, and so on. If we are unwilling to make this assumption, then we need to break this class down into multiple equivalence classes: (a) one EC for all strings with exactly two characters, (b) one EC for all strings with exactly three characters, and so on.

If we are not willing to assume equivalence, regardless of the length of the character string, we may still find that the difference is irrelevant. The frequency of occurrence of these different-length strings is a practical consideration which may influence our equivalence classes. Let's say, for example, that the people who are doing the data entry are highly accurate, and they press the intended key first in 98% of attempts. This means that 98% of all incoming character strings

are exactly one character long (not including the <Enter> key). In almost every remaining attempt the user corrects the error on the second keystroke, which overlays the first keystroke. Let's say that once in a blue moon (which is exactly once in every 10,000 attempts), the user flubs twice in a row, and so has to enter three data values before he or she gets it right. We might not be willing to assume equivalence regardless of the length of the character string, but strings of three or more characters happen only rarely. We are not willing to test strings of three or more, because this occurrence is too low to justify a separate test case.

Non-English Characters

Equivalence class A(5) assumes that non-English alpha characters are considered to be valid input, though the specs do not say this. This category still must be an EC—the question is whether it should be on this list of valid equivalence classes or on the next list of invalid ones. Non-English Latin-based alphabets contain qualifiers to characters which do not occur in English, such as the grave, cedilla and umlaut. Regardless of whether we consider the non-English alpha characters to be valid or invalid, do we need to test for these?

Repeated Use of the Data Entry Field

Equivalence class A(6) assumes that the data entry field will be used repetitively, e.g., after the data field has been used to enter either a value, it will be re-set and can be used again. We were not told in the specifications that this data entry field can be used repetitively, but it is highly unlikely that the field will be used only once for all time.

Alternative Data Entry

Equivalence class A(7) assumes that users can enter the data field by selecting from a list box, as well as directly from the keyboard. There could be other alternative methods of data entry, such as voice recognition. Unless we are willing to assume these alternatives are equivalent, we need to test the other data entry methods too. Can we enter the data field only from the keyboard? With today's technology, the application could be designed to allow more than one input option. Clicking on a picture could enter the character, or voice recognition. In this situation, we need to make decisions about the equivalence of the methods of data entry. For example, are we willing to assume that if a "K" is entered and accepted from the keyboard, that the system will behave the same way if the "K" was entered by voice recognition, of selected from a pull-down list?

B - Invalid Equivalence Classes

Here the term "invalid" does not mean that this is a poor equivalence class or one not worth testing; it means we are testing invalid input values.

Equivalence Class (EC)	Description	Test Case (Sample Member of the EC)

Chapter 6
Equivalence - Answer to Exercise 6.2

B(1)	Number	8; <Enter>
B(2)	Special (non-alphanumeric) character	?; <Enter>
B(3)	Multiple-key combination	Alt-F9; <Enter> pressed simultaneously
B(4)	Control character	X; Backspace; <Enter>
B(5)	Null character (no character is entered before <Enter>)	<Enter>
B(6)	Enter key after overlay	Alphabetic character; non-alpha character; <Enter>
B(7)	Tab key used as delimiter instead of Enter key	h; <Tab>
B(8)	Non-keyboard character[2]	<Enter>
B(9)	Ultra-fast data entry[3]	Multiple keys are depressed rapidly, followed by <Enter>
B(10)	Ultra-slow data entry	The pause after a key is depressed is very long, before the <Enter>
B(11)	Simultaneous entry of standard characters[4]	Two keys are pressed at exactly the same moment, e.g. Q and W, then <Enter>

Do you agree with this set of test cases? Taken together, do these equivalence classes cover all possible inputs and initial conditions? Does any equivalence class need to be split up into two or more classes, because its members do not have the same behavior? Could we reasonably

[2] This assumes it is possible to corrupt a character after it has been entered. (It happens, especially in environmentally difficult conditions such as in space or on a dirty factory floor.)

[3] Assumes that the expected behavior of the system has been pre-defined, for when ultra-fast and ultra-slow data entry happens.

[4] This equivalence class is not the same as pressing a key combination which may be considered valid, such as Alt-F9; <Enter>.

combine any two equivalence classes, because we can expect their members to work the same way?

Complications

Experienced test professionals acknowledge (often with a resigned shrug), that no matter simple a situation appears to be, there are always complications. In this "simple" situation with the one-character data entry field, complications which the tester needs to consider include:

- The "non-blank blank".
- Internal representation of alphabetic characters.
- Testing for the physically impossible.
- Recognizing and testing assumptions.

We'll discuss each of these complications below.

The "Non-Blank Blank"

What about special characters which do not print or display, or display as blanks but are not stored and represented internally as blanks? (The majority of characters fall into this category: they are not displayed or display as blanks.) On most keyboards, pressing the space bar enters a blank, but there may be other special characters or multi-key combinations which we can enter from the keyboard and show as blanks.

For any particular keyboard or data entry technique, it may be physically impossible to enter a special character which is not displayed. In this situation, we would not bother to test a physically impossible condition (because it can never happen). Nevertheless, let's instead assume that this situation is possible. The question now becomes: how do we test for special characters which do not print or display? Can we reasonably assume that these characters are part of the Blank class (number 3 on the prior list of equivalence classes)? If we make this assumption, we do not need to add any new equivalence classes and any additional test cases.

If we are not willing to assume these non-displaying characters are part of the Blank EC, can we assume they are part of the special characters EC (number 5 on the list)? If so, we do not have to add an additional test case for the non-displaying characters. If we want to be cautious (or paranoid), we can add another EC to the list (and thus we need another test case). For example, we could split EC number B(2) into three equivalence classes, as follows:

Equivalence Class (EC)	Description	Test Case (Sample Member of the EC)
B(2)	Special (non-alphanumeric)	

	characters	
B(2a)	Displayable character	?; <Enter>
B(2b)	Non-displayable character	
B(2b.1)	Ascii character[5]	; <Enter>
B(2b.2)	Non-Ascii character	Non-Ascii character (e.g., hexadecimal ”FF”)

Also, what about a null character (no character is entered at all before the Enter key is pressed)? Can we assume that the system will treat this null character the same way as is if a blank was entered, followed by the Enter key? If you are willing to make this assumption, I would like to sell you the Brooklyn Bridge, which has been sold many times in its history by people who did not own it.

It is prudent, therefore, to add another equivalence class (which is already on the prior list):

B(5)	Null character (no character is entered before <Enter>)	<Enter>

Internal Representation of Alphabetic Characters

Alphabetic characters can be stored in various different formats internally, such as ASCII (used in most personal computers and client/server networks), EBCDIC (used mostly in IBM mainframes), and Unicode (used widely in international software, such as software sold in different countries). Can we assume equivalent behavior of the input edit logic, regardless of how the system has stored the character internally?

Testing for the Physically Impossible

What if some of the actions described earlier are physically impossible, based on the design of the data entry device (which we assume is a keyboard, but could be another device instead)? For example, what if the keyboard has no mechanism to allow users to enter non-English characters, such as ü or ň ? Or what if there is a physical locking mechanism that prevents two keys being pressed simultaneously?

[5] This assumes that the system uses the ASCII character set.

The answer is that we will not test for the physically impossible condition, since it cannot happen. The danger of course, lies with the difference between the impossible and the merely improbable. We say to ourselves: "That condition could never happen," so we do not test for it and are surprised when it does in fact occur.

Testing Our Assumptions

The secret of using equivalence as a test case design technique lies in identifying the Equivalence classes first, instead of attempting to immediately identify the test cases (the input data values). Once we have defined the equivalence classes, the test cases drop out fairly easily and naturally, with one representative test case selected from each EC we are testing.

The answer we produce to this exercise depends on the assumptions we are willing to make. Therefore, it is imperative to recognize, document and critique these assumptions. (See the section in Chapter tbd, entitled: "Recognizing and Checking Assumptions".)

The Demonstrated Value of Equivalence

According to the analysis we need a total of 18 test cases for the one-character data entry field, 7 positive and 11 negative, based on the earlier set of assumptions. Let's say that you did not take the time to analyze the situation (which, after all, requires up to 45 minutes of your valuable time), but instead tested by intuition—you followed your instincts and banged away on the keyboard.

All that activity made you feel good, and you certainly think you are being productive and getting things done hands-on instead pushing pen on paper. Unfortunately, the action-oriented brute-force approach is a victory of brawn over brains. First, intuition would not give you more than an imprecise gut feel about the test coverage—you might run 8 test cases, for example, and be convinced that you have tested everything of significance. According to the analysis, though, a set of 8 test cases are less half of what we need.

Second, let's say you are absolutely determined to catch every major bug and bang away at the keyboard for 90 minutes, testing this one-character data entry field. In this time, you run 54 test cases (triple the recommended number of 18). Despite the intense effort, it is unlikely that the coverage provided by 54 intuitive or quasi-random test cases will be anywhere near as high as the coverage with the 18 test cases identified by the equivalence analysis. Unless the risk is low (in which situation we don't care very much), the 30 minutes spent on the analysis is as good investment.

We can't afford spotty, unknown coverage in high-risk situations, and we do not have the time and resources for brute-force testing. Performing the equivalence analysis is really the only way to go.