World Transactions on Engineering and Technology Education
Vol.1, No.2, 2002

A teaching approach for software testing

Andrew McAllister† & Sudip Misra‡

University of New Brunswick, Fredericton, Canada†
Carleton University, Ottawa, Canada‡

ABSTRACT: Teaching students how to test software is complicated by the absence of a simple, integrated approach for generating test plans. No single testing technique fulfils these needs and teaching only a collection of disparate techniques makes it difficult to assign work for students. This article provides an integrated approach for test plan generation that can be used by students in programming and software engineering courses. The approach provides simple guidelines that prompt the discovery of sets of test cases that are typically more complete than students produce on an ad hoc basis. A technique is introduced that ensures all program statements are executed during testing and that loops are tested in a rigorous manner. Experience shows that this technique tends to be simpler to use than existing techniques that identify independent paths through programs. All of the guidelines presented in this article can be applied without automated tools. The primary strength of the approach is in demonstrating to students how rigorous generation of test plans can identify test cases that might otherwise not occur to the tester, and how multiple techniques can be combined to complement one another.

INTRODUCTION

Despite the explosion of research in the field of software engineering, the pedagogy of software testing has received relatively little attention. This article defines an approach for generating software test plans intended for use by students in programming and software engineering courses. The primary goal is to provide an approach that helps to convey to students the need for rigour in testing, and how this rigour can help to make their test plans more complete.

There is no limit on the number and type of possible defects in software. This uncertainty explains the impracticality of testing for all possible bugs in a piece of software. Various techniques have been developed for testing software. Software testing texts (eg [1-5]), software engineering texts (eg [6][7]) and survey papers (eg [8-11]) discuss a variety of techniques. Some techniques are based on the execution of all branches and controls within a program [6][12]. Others are based on the execution of all program segments [12]. Others focus on the testing of boundary conditions [6].

To address the testing of specific features used in object-oriented constructs (eg testing inheritance, polymorphism, dynamic binding and interaction between classes), several testing techniques are proposed in the literature (eg [13-19]).

Various techniques utilise different criteria for software testing and no single technique provides coverage of all possible types of errors. Different existing techniques offer distinct advantages and disadvantages. For example, the path-analysis based techniques may encounter an infinite number of paths or may not detect all paths. Some techniques are very complex to apply and require considerable skill on the part of the software tester. Moreover, many testing techniques are labour-intensive.

From the standpoint of teaching software testing to university students, such an unorganised collection of separate techniques can be difficult to use. While one can cover a disparate set of topics during lectures, assigning work for students can be greatly facilitated by having a testing approach that is:

- Usable by hand so that the steps involved are simple enough to apply with a reasonable amount of time and effort.
- Organised as a list of guidelines that can be applied in order, unlike most of the existing approaches in the literature. The hope is that this will help to increase the ease with which students can apply the approach.
- Quite obvious as to how the technique is to be applied in virtually all cases. The amount of complex decision-making should be minimised.

Teaching the existence and purpose of the different testing approaches is important, but difficulties arise when students are assigned the task of producing a test plan for a given program in a short period of time. Trying to test their software with a disparate set of techniques as covered in class lectures becomes cumbersome. It can be difficult to decide which techniques should be used and how they should be applied. There is a need for a simple approach that can be applied in most situations.

This article discusses an approach that can be used to aid in teaching how to generate software test plans. In this context, it is not necessary to provide a technique to detect all kinds of bugs. For teaching purposes, it is sufficient to have an approach that helps students learn the process of testing, including the importance of rigour in testing as opposed to an ad hoc approach.

The remainder of the article is organised as follows. The proposed test plan generation approach is first described using illustrative examples at each step. This is followed by an evaluation of the proposed approach and the authors' experience in using it. The final section provides conclusions and directions for future research.

THE TEACHING APPROACH

As part of the authors' software engineering courses, the topic of software testing is introduced to students by providing an overview of fundamental testing concepts. The topics covered are similar to those described in a variety of software engineering textbooks, eg [6][7]. These topics include:

- The necessity of testing.
- Testing objectives.
- The relationship of testing to other software quality assurance activities, such as code reviews, quality metrics, project standards, etc.
- Testing in the software lifecycle and types of testing (unit, integration, functional, acceptance).
- Testing activities (plan, execute test cases, evaluate results, debug/fix, measure).
- Typical testing environments and tools; drivers and stubs.
- Categories of testing techniques: black box testing (based on functional specifications) versus white box testing (based on knowledge of the source code), etc.
- Management of testing activities. In particular, recognition of the need to allocate sufficient time in project plans for all testing activities.

This concept overview sets the stage for what the authors consider to be the critical question that students should address, namely: *when faced with the task of testing a particular software (sub) system, how should one proceed?* In class discussion, students are invariably successful in identifying test case planning as one of the most potentially problematic tasks. If one were able to produce magically a high-quality set of test cases for a given program, then running the test cases and determining which cases identify problems in the code is conceptually straightforward by comparison. Debugging is also problematic, of course, but students improve their skill in this task by completing programming assignments in a variety of courses within the computer science curriculum. Therefore, the focus of teaching how to test (and the primary contribution of this article) is in providing students with practical guidelines they can use to identify test cases.

The authors' approach is based on the following insights:

- There is a need to teach students that software should be tested as exhaustively as possible, while completing the task within reasonable time limits. However, this need does not imply that students should be required to generate exhaustive sets of test cases when completing course assignments. Once students learn to apply a quality technique, repeating this technique to generate large numbers of test cases tends to induce boredom rather than additional learning. For course assignments, this issue can be handled by requiring students to either (a) test exhaustively only a modest portion of their code, or (b) produce a modest quantity of test cases, providing practice with each applicable technique (as discussed below).
- Exhaustive coverage of all published software testing techniques is impractical and unnecessary. Students can learn the need for rigour in testing by applying a relatively small set of carefully selected, complementary techniques.
- Automated tools are available to aid in generating test cases. However, manual test case generation is still commonly performed in many software development projects. In addition, not all education programmes have access to such tools. The experiences of some educators show that automation of test plan generation is of limited use since it is unreasonably time consuming to automate and involves difficulties not encountered with manual testing [17]. Manual application of testing techniques can force students to develop more insight into how the techniques work than might be necessary to run an automated tool. For these reasons, it is required that students learn to generate test cases manually.

Fundamentals of the Testing Approach

The following fundamentals form the basis for the proposed approach to software testing:

Fundamental #1 relates to testing every identifiable part of the system at least once. This concept applies both in the context of black box testing (eg test every screen, every input field) as well as white box testing (eg execute every line of code at least once, test every loop condition). This does not imply that the program should be tested in every conceivable way in which it could be used, which is impossible for the vast majority of systems. Students are provided with checklists of guidelines they can use to identify candidate system components for testing.

Fundamental #2 is to push the system hard; try to break it. Many of the guidelines included in the approach are derived from the concept of boundary testing, which is based on the theory that a significant percentage of errors occur in extreme situations (eg using largest/smallest possible values, largest/smallest quantity of data, largest/smallest possible differences between values, etc).

Fundamental #3 covers the use of several techniques to generate test cases. Any redundant test cases that result should be eliminated. Using the black box approach, a tester might note that a particular report allows several lines of output and may devise a test case to do so. Later, when examining the code that produces this report, the tester might devise a test case to run through a loop several times (thus producing several lines of output). The two test cases involve input that is equivalent (for testing purposes) and produce equivalent output, ie they test the same functionality. One should be eliminated from the test plan. This does not represent wasted effort. Different techniques identify different sets of test cases. The fact that these sets typically overlap is natural and is less important than the more complete coverage that results from using multiple techniques.

Fundamental #4 relates to documenting all test cases in a test plan. The end result of test case generation is a document that defines all of the test cases. Simply running the test cases without first documenting them is unacceptable. After finding and fixing errors, the entire set of test cases should be executed again. (This process is repeated until the entire test plan can be executed with no errors. Repeating only the test cases that fail is risky, since new errors are often introduced when fixing bugs.) Repeated execution is impossible if the test cases are not documented. In addition, the test plan becomes part of the documentation delivered with the completed system.

A table format is utilised to document test cases: one test case per row, with the following five columns:

1. Test case number: Test cases are numbered consecutively in the table, starting with 1.
2. Purpose: What aspect of the program's behaviour is this test case intended to exercise? In some cases this information might be obvious by examining the input values (column 3), but this is not always true.
3. Input: What input values are to be used when executing this test case? This can include any type of input allowed by the program, such as mouse clicks, audio input, scanned input, specific data files to be used, etc.
4. Expected Result: Based on the program's specification, how should the system respond to the input? (If the system responds otherwise, this represents an error.)
5. Observed Result: Did the system behave as expected or was an error detected? This column is left blank during test case generation; it is used only during the execution of test cases. To speed up test case execution, each cell in this column can contain check boxes for *As Expected*, *Error* and *Fixed*. Since this column is completed each time the test plan is executed, a separate copy of the plan can be printed for each execution. (An alternative is to have multiple *Observed Result* columns in the table, one for each execution of the test plan.)

A critical concept in completing such a table is understanding the nature of the information that should be included in the *Input* and *Expected Result* columns. Input should be defined using specific values rather than general classes of values. For example, if a numeric value must be entered into a field as part of the execution of a test case, it is insufficient to specify that *a negative value* should be entered. Instead, a specific negative value must be selected by the test planner and included in the *Input* column, for example *-5*. Without such specificity, the test plan is not repeatable.

Similarly, *Expected Result* entries should be as specific as possible. For example, rather than simply saying that *an error message should appear*, the test plan should specify exactly what message should appear. A common student mistake is to enter a phrase such as *The operation should complete properly* in the *Expected Result* column, without specifying what constitutes correct completion (a more specific result might be stated as, for example: *A customer record is added to the database and the user is returned to the main menu.*)

The Approach Part I: Black Box Testing

In the *black box* approach, test cases are generated based on a functional specification of the software. That is, assuming a specification is available that defines how the software should behave, test cases are generated to determine whether the software's behaviour is consistent with the specification. The following guidelines are consistent with the fundamentals described above and provide students with checklists they can use to think of possible use cases.

For every user operation defined by the specification, the following should be used to identify test cases based on system behaviour (Note: this becomes *For every use case* for object-oriented development):

- Execute every operation/scenario identified in the specification.
- Include *typical* and *exceptional* scenarios: all types of exceptions that can be considered, even those not documented in the specification. NOTE: Students will inevitably miss some types of exceptions at this point, but at least this guideline gets them thinking in the correct direction. Other testing guidelines in subsequent steps prompt students to think of further exceptions).
- When documenting a test case for an exception, it is possible that the desired system behaviour might not be documented in the specification. For example, if a user enters no input for a particular prompt, it might be reasonable for the system to either (a) inform the user that input is required and then cancel the operation, or (b) keep displaying the prompt until the user either clicks the *Cancel* button or enters some input. If the specification simply states something like *Input is mandatory for this prompt* then (on a real project) the tester must consult with someone in authority (eg the system architect) to determine how to fill out the *Expected Result* column for this test case. For course assignments, it is advised that students note all such cases, make a choice of desired behaviour on their own and state any assumptions.

In order to identify test cases based on system input, the following should be undertaken to test the usage/execution of every user interface component:

- Pop up every screen/window and dialogue box, including all error dialogues. The exception is that some error dialogues might not be testable since they are designed to appear only if the software is faulty.
- Click on every menu item and button (and complete the resultant operation). Note: By this point in the process, students will undoubtedly start generating use cases that are redundant with those based on system behaviour. This is by design and redundant test cases are simply omitted from the test plan).
- Test every setting for all check boxes and radio buttons. (This means more than just clicking to toggle each one on and off). For example, for a given check box, click it *on* and execute whatever operation depends on the setting of this check box. Ensure that the *on* setting took effect as it should. Then click the check box *off* and execute the operation again to ensure that the appropriate behaviour occurs. Testing the settings of multiple check boxes, radio buttons, etc, with a single test case is acceptable.
- Select the first and last selections, as well as one near the middle, for each drop-down selection list. If such a list has variable content, then this should be tested with no items, one item, a few items and the maximum number of items. Test that each list defaults to the appropriate selection.
- For each field in which a user can type a value, enter:
    - No value.
    - Minimum allowable, maximum and medium values. For example, if a field allows entry of a number between 1 and 100, define a test case for 1, another for 100 and a third test case for 50.
    - Illegal values (eg out of range, or alphanumeric where numeric only is required).

- Varying quantity of values:
- The minimum allowable quantity (and 1 less if possible).
- The maximum allowable quantity (fill the field). Also attempt to overfill the field).
- A quantity somewhere in the middle.

- Test all mouse click, resize, drag/drop, etc, operations.
- Test alternative inputs:

  - Shortcuts to invoke operations (eg Ctrl+C for Copy).
  - Keyboard alternatives to mouse operations (eg Enter instead of clicking OK, tab to move from one GUI field to another).

- Test (the full range of) any other forms of input allowed:

  - Data files.
  - Real-time data streams.
  - Voice, etc.

To identifying test cases based on system output, the following should be undertaken. The functional specification should define the range of variability that is required/acceptable for all required system outputs. Attempts should be made to produce each type of output, pushing the bounds of these ranges as follows:

- Produce each report with:

  - A null/empty/minimum amount of data.
  - A moderate quantity of data, then a maximum/large quantity of data.
  - Invalid results (may not be possible).
  - Small, medium and large values.

- Produce the full range of screen displays, sounds, etc.
- Expected results can include checking the contents of any data files created during test case execution.

The Approach Part II: White Box Testing

By definition, black box testing focuses on the user-perceivable aspects of a system; namely: system input, output and behaviour. On the other hand, white box testing is based on how the system is constructed, which includes using knowledge of the program source code. The guidelines suggested in this section are designed so that students focus somewhat on what comes between system input and output; in other words, on the program statements that execute the operations and on the variables and data structures that store values.

The following describes identification of test cases based on Boolean conditions. For every condition that compares two values *a* and *b*, test cases can be defined where:

- a = b.
- a < b.
- a > b.
- a and b are almost equal.
- a and b are vastly different.

It should be noted that when a condition appears in the middle of some processing operation, there is a need to work out what system inputs will result in the desired values of *a* and *b* at this point in the process.

With regard to the identification of test cases based on data storage (for selected variables that store data values), attempts should be made to assign:

- Minimum, medium and maximum values.
- Both typical and unusual values (eg null string).
- Invalid values. Note: Students must use some judgement here so that the number of test cases does not become overly large. It is suggested that focus is placed on variables that store significant intermediate results and the results of significant calculations).

For each data structure (eg array, list, tree, etc) attempts should seek to create:

- Null content.
- Minimum allowable content.
- Structure of typical size or shape.
- Extreme or odd shapes (eg a one-sided binary tree).
- Invalid structure (eg a root-less tree).
- Very large structure (this is important for testing memory capacity).

The following should be considered when identifying test cases based on control flow and loops. An important goal is to ensure that every statement in the source code is executed at least once during testing (it is assumed that every statement in the program to be tested *can* be executed; modern compilers tend to complain about unreachable sections of code.) Basic path-testing techniques (eg [6]) accomplish this goal but can be complex to apply. A simpler technique is proposed that provides the additional benefit of testing loop execution.

The flow of execution through the statements of a program is controlled by *if* and loop conditions. During the execution of all test cases taken together, when each condition is evaluated to true at least once, and evaluates to be false at least once, then every possible *direction* is taken from each decision point in the program and every statement must be executed at least once. To ensure that this happens, tables of the form shown in Tables 1 and 2 can be used. The conditions in these tables are based on the Java code example in Figure 1, the desired behaviour of which is described by the specification in Figure 2.

Table 1: Ensuring that each *if* condition evaluates to both true and false.

If condition #	Test case where condition evaluates to be:	
	True	False
1 - (grade >= 85.0)	1	1
2 - (grade <= 100.0)	1	2
3 - (grade >= 85.0) && (grade <= 100.0)	1	1
4 - (grade >= 70.0)	1	1
5 - (grade >= 55.0)	1	3
6 - (count > 0)	1	4

Each *if* condition is identified in a separate row of Table 1. The simplest way to do this seems to be to scan through the source code and number the conditions as they are encountered.

Students typically print their source code and number the conditions by writing on the printed program. In the sample Java source code in Figure 1, comments are used to denote condition numbers. Also, for the sake of clarity, the conditions themselves are included in the leftmost column of Table 1. Only the condition numbers are normally required in this column.

Table 2: Ensuring that each loop is executed with a varying number of iterations.

Loop condition #	Test case number where the condition terminates the loop after this many iterations			
	zero	one	multiple	maximum
1 - (grade >= 0.0)	4	2	1	n/a

```
System.out.println("Enter one numeric grade per line "
                        + "(end with negative number):");
double grade = console.readDouble();
double total = 0.0;
int    count = 1;
String letterGrade = "";
while (grade >= 0.0)  {              // Loop condition #1
   count++;
   total += grade;
   if (   (grade >= 85.0)            // If condition #1
         && (grade <= 100.0))  // If condition #2
        letterGrade = "A";     // Entire compound condition is #3
   else if (grade >= 70.0)         // If condition #4
        letterGrade = "B";
   else if (grade >= 55.0)         // If condition #5
        letterGrade = "C";
   else
        letterGrade = "F";
   System.out.println("Letter grade: " + letterGrade);
   // Get next grade
   grade = console.readDouble();
}
if (count > 0)                       // If condition #6
   System.out.println("Average: " + (total/(double)count));
```

Figure 1: Sample Java source code.

The program should accept a series of numeric grades (between 0.0 and 100.0) entered by the user. The user enters a negative sentinel value to indicate the end of the input. Invalid grades are to be rejected by the program. After each valid grade is entered, the program must print the corresponding letter grade, according to the following conversion:

85 to 100 (inclusive):	A
At least 70, but less than 85:	B
At least 55, but less than 70:	C
Less than 55:	F

The program must also count and sum the numeric grades, then calculate and display a numeric average after the sentinel value is entered. The sentinel value does not count as a grade; in other words, it does not affect the calculation of the average.

Figure 2: Program specification.

For compound conditions (ie those involving *and*, *or*) each sub-condition should be numbered separately (and placed in its own row in Table 1), as well as the compound condition as a whole. For example, conditions 1 and 2 in Table 1 are subparts of condition 3.

Strictly speaking, to ensure that all statements in a program are executed at least once, only the compound condition as a whole is required in Table 1, since the condition as a whole determines the flow of control through the program.

However, an *and* condition typically has more than one way of becoming false, and an *or* condition can be true for more than one reason. For example, condition 3 in Table 1 can be false for two reasons, namely: (a) a grade is below 85, or (b) a grade is above 100. Including the sub-conditions in separate rows of the table forces each of these cases to be tested. In other words, this technique goes beyond simply ensuring execution of all program statements.

Loop conditions are also numbered. Each one occupies a row in a separate table, as shown in Table 2. It does not matter whether *if* and *loop* conditions are numbered separately or as a single series. The only issue is to identify each condition in some unambiguous manner.

Each cell in Tables 1 and 2 is filled with a single test case number (or *n/a* meaning *not applicable*). If students apply this technique late in the process of generating test cases (that is, if the other guidelines listed previously are used first), then it is possible that the tables can be partially or wholly filled based on test cases that already exist. However, for this example, it is assumed that no test cases exist upon starting. If this is so, then the choice of test case 1 is relatively unimportant; it is suggested that students define inputs for what they consider to be a typical execution of the program.

Test case 1 in Table 3 is such a case. It involves the entry of a few typical grades, followed by a valid sentinel value. Once this test case is defined (or when an existing test case is being considered), the tester must determine which cells in Tables 1 and 2 are satisfied by this test case. In this example, test case 1 satisfies ten cells in Tables 1 and 2. For instance, if conditions 1, 2, 3 and 6 are true immediately following entry of the value 90; if condition 4 is true and if conditions 1 and 3 are false after entry of 75, and so on. The while loop is executed more than once during this test case, so test case 1 is entered in the *multiple* column in Table 2.

The *maximum* column in Table 2 is used when there is some upper boundary on the number of possible or allowable iterations for a given loop. This is not the case in this example, so *n/a* is entered.

Additional test cases can be considered for any cells that remain blank in Tables 1 and 2. For example, a value greater than 100 must be entered so that if condition 2 will evaluate to false.

Test case 2 is then added to Table 3 to accomplish this and the *false* cell for if condition 2 in Table 1 is filled in accordingly. In addition, since test case 2 involves entry of only a single grade (other than the sentinel), then this test case also accomplishes the goal of executing the while loop exactly once. Therefore, test case 2 is entered in the *one* column of Table 2 for loop condition 1.

Table 3: Test cases generated for the Java code in Figure 1.

Test Case #	Purpose	Input	Expected Result	Observed Result
1	A *typical* program execution - A series of different grades	90, 75, 60, -1	Letter grade: A Letter grade: B Letter grade: C Average: 75.0	As expected ☐ Error found ☐ Error fixed ☐
2	An invalid grade is entered.	101, -1	Messages: Invalid grade. No grades entered. *(assumption)*	As expected ☐ Error found ☐ Error fixed ☐
3	A failing grade is entered.	30, -1	Letter grade: F Average: 30.0	As expected ☐ Error found ☐ Error fixed ☐
4	Sentinel value is entered immediately, with no valid grades	-100	Message: No grades entered. *(assumption)*	As expected ☐ Error found ☐ Error fixed ☐

The *false* cell for *if* condition 5 in Table 1 prompts the definition of test case 3 in Table 3 (entry of a grade less than 55). This test case is similar to test case 2 in that both result in exactly one iteration of the loop. However, there is no need to add test case 3 to the *one* column in Table 2. In general, once a given test case satisfies a specific cell in Tables 1 or 2, there is no need to make note of any other test cases that happen to do the same.

To complete the example, the *zero* cell for loop condition 1 leads us to define test case 4, in which no grades are entered prior to the sentinel. This is the case for which *if* condition 6 is included in the program: to avoid division by zero when no grades are entered. A student completing Table 1 might reasonably enter test case 4 in the *false* cell for *if* condition 6 (as has been done here), expecting this to be so. The execution of test case 4 will then result in an error being found: the count is improperly initialised to 1 instead of 0 (zero). This error will also be detected by test case 1 since an incorrect average will be calculated as a result.

An alternative and equally likely scenario for the completion of Table 1 is as follows. In attempting to predict the expected result of test case 4, the student might notice the improper initialisation of the count variable. Finding errors like this during test case generation is not an uncommon event, and two actions are possible. It is recommended that the student should fix the error immediately and then continue with testing (the number 4 remains in the *false* cell for *if* condition 6). This is usually simple to do in a learning environment since the programmer and the tester are typically the same person.

Alternatively, *n/a* can be entered in the *false* cell for *if* condition 6, since for this version of the program, *if* condition 6 cannot become false because count starts at 1 and can never decrease. After test case execution, all known errors are fixed before updating the test cases (including replacement of this *n/a* entry with test case 4) and testing again. This alternative approach might make more sense in a commercial setting *if* batches of known errors are passed to programmers for fixing between rounds of testing.

An important point to make regarding the use of this technique (and for white box testing in general) is that even though test cases are identified based on the program source code, the expected results are determined based on the program specification. The expected results documented in Table 3 illustrate a common problem in both classroom and commercial settings: expected results can be difficult to define when a specification is incomplete.

As in this example, one of the most common areas for this type of difficulty is in error handling. In this case the specification in Figure 2 states that invalid grades should be rejected, but makes no mention of how this should be accomplished. When completing the *Expected Result* cell for test case 2 in Table 3, the fictitious student decided (quite reasonably) that the program should display a message to the effect that 101 is an invalid grade. This is an assumption and is noted as such. Test cases 2 and 4 also involve the assumption that a message should be displayed when an average cannot be calculated because no grades have been entered. Even if the tester wrote the program, it is not uncommon for the testing exercise to force students to reconsider lapses or invalid assumptions made during programming. In this way, effective teaching of software testing can also improve programming skills.

The execution of such test cases shows that the example program does not display the messages noted in test cases 2 and 4. Moreover, test case 2 shows that invalid grades are included in the calculation of the average and that an average is calculated and displayed even when no valid grades are entered.

To further illustrate the use of this technique, consider a loop that searches through the elements of an array and is coded as follows: while ((i < arraySize) && notFound))

As with an *if* condition, the sub-conditions are listed separately from the compound condition in Table 4. Note that a Boolean variable is treated the same as any other condition.

Table 4: Handling a compound loop condition.

Loop condition #	Test case number where the condition terminates the loop after this many iterations			
	zero	one	multiple	maximum
1 - (i < arraySize)	1	2	3	3
2 – notFound	n/a	4	6	7
3 - (i < arraySize) && notFound	1	2	3	3

The following list describes seven test cases that can be used to populate the cells in Table 4:

1. The arraySize is zero.
2. The search item is not present in an array with one element.
3. The search item is not present in an array with multiple elements.
4. The search item is the only element in the array.
5. The search item is located in the first element of an array with multiple elements.
6. The search item is located in a middle element of an array with multiple elements.
7. The search item is located in the last element of an array with multiple elements.

Condition 1 terminates the loop when the entire array has been searched, which happens when the search item is not found. Therefore, the cells for condition 1 are satisfied by defining test cases where a search item is not found in arrays of various sizes.

Condition 2 terminates the loop when the search item is found. This cannot take place in an empty array, so the *zero* cell does not apply to condition 2. Condition 2 terminates the loop after a single iteration when the search item is found in the first element of an array. This can happen either in an array of size one or in an array with multiple elements (test cases 4 and 5).

Strictly speaking, the proposed testing technique requires that only one of these two test cases be included (to populate the *one* cell for condition 2) but if both test cases occur to the tester, then it makes sense to include both in the test plan. Either of the two test case numbers can be entered in Table 4.

The maximum possible number of loop iterations is achieved when an entire array (with multiple elements) is searched, which is true for test cases 3 and 7.

The entries in Table 4 for condition 3 (the compound condition) illustrate the value of listing sub-conditions separately. The compound condition can be made to terminate the loop a varying number of times using only test cases 1 to 3, without ever finding a search item. Separate treatment of sub-conditions prompts the tester to think of a more complete set of test cases.

Figure 3 illustrates a final point to be made about this technique. The code in Figure 3 includes two loop conditions and one *if* condition. However, none of these three conditions can be affected by program input. This code executes in exactly the same manner every time the program is run, and can be tested adequately by any single test case that executes this portion of the program (which is guaranteed to happen, since the technique ensures that all program statements are executed). This type of invariant behaviour is common, for example, in loops that initialise arrays and other data structures. Such conditions are omitted when numbering conditions for inclusion in Tables 1 and 2.

EVALUATION OF THE APPROACH

The authors' literature survey has revealed no other work that is specifically oriented towards tackling improvements in teaching software testing to students. The approach presented in this article represents two innovations in software testing, namely:

- The approach integrates concepts borrowed from existing disparate techniques so that students have a single reference to guide their work.
- A new table-based technique is presented that helps in identifying test cases that exercise *if* and loop conditions in a rigorous manner.

```
// Display a checkerboard pattern of alternating
// white and black squares
for (int row = 1; row <= 8; row++)
{    for (int column = 1; column <= 8; column++)
     {   if ((row%2)==(column%2))
              // Display a white square
         else
              // Display a black square
     }
}
```

Figure 3: Conditions that are independent of program input.

The approach is applicable to virtually any programming language and is not dependent on the availability of automated testing tools or environments.

A wide variety of published techniques address program characteristics not tested using this approach. For instance, one current research issue is the generation of test cases for programs that involve dynamic binding and polymorphic features [18]. In such situations, the object that will process a message is not pre-determined but is decided dynamically during execution. The actual flow of control is not determined statically beforehand but is decided dynamically at run-time, and therefore cannot be predicted [20]. This limits the applicability of our technique, a part of which involves static examination of source code to determine flow of control.

The authors consider this not to be a large problem for the following reasons:

- The authors' approach is intended for use in a learning environment where such types of programs can be controlled.
- The approach is not intended to provide comprehensive testing for all types of programs. Rather, it is designed to demonstrate to students the need for comprehensive testing, as well as to provide examples of techniques that can help to address this need.
- Even programs that include problematic characteristics can be tested with this approach. Some aspects of the programs may be less rigorously tested than others, but students and instructors can add new techniques and guidelines to the approach in order to fit the needs of specific courses.

The approach described in this paper has been taught for several years in second- and third-year software engineering courses as part of the undergraduate Computer Science curriculum at the University of New Brunswick, Fredericton, Canada. This experience has been used to evolve the approach to its current state. Students report consistently that the approach is easy to apply and that the resultant test cases are simple to use (in guiding test case execution).

The most gratifying feedback comes from students with work experience, which includes both mature students and those in the University's computer science co-op programme (students receive academic credit for four- and eight-month industry work terms). Several such students have discussed their involvement in commercial projects where software developers have been left to their own devices to perform testing as they see fit, and where test cases have been generated on an ad hoc, black box basis only, simply using *as many test cases as they can think of*. It should be noted that in the experience of the authors, this situation is all too common, even in organisations that specialise in software development.

Even though the approach in this paper was not designed for use in commercial settings, several students have reported that it is an improvement on the testing practices they have used in commercial projects. Furthermore, they believe it will aid them in subsequent projects.

CONCLUSIONS AND FUTURE WORK

The authors' experience indicates that the proposed test plan generation approach is easier to teach than the medley of existing techniques typically presented by software engineering textbooks. Perhaps more importantly, the proposed approach provides a usable reference that eases significantly the task of developing software testing assignments, while increasing the degree of rigour that students can use in completing these assignments.

There are several potential areas where further research is possible. First, although the approach has been designed for pedagogical purposes, one could very well examine its effectiveness by applying it within a commercial context. One could study if there would be major changes in ideas the students would have to undergo in the transition from school to work. A second possibility is to attempt to measure the effectiveness of the test cases generated using the approach. Do students identify more bugs in their software using this approach as opposed to unguided, ad hoc testing efforts? It might also be possible to design other testing approaches for teaching purposes. Experience may show that it is possible to further simplify the steps of the approach, or to test more aspects of the software (ie to incorporate additional testing techniques). Finally, one can investigate ways to accomplish other testing activities in a teaching environment, such as test case execution and evaluation, debugging and error fixing, as well as error rate evaluation.

REFERENCES

1. Beizer, B., *Black Box Testing: Techniques for Functional Testing of Software and Systems*. New York: Wiley (1995).
2. Beizer, B., *Software Testing Techniques*. New York: Van Nostrand Reinhold (1990).
3. DeMillo, R.A., *Software Testing and Evaluation*. Redwood City: Benjamin/Cummings Publishing Co. (1987).
4. Hetzel, B., *The Complete Guide to Software Testing*. Massachusetts: QED Information Sciences (1988).
5. Marick, B. *The Craft of Software Testing*. Englewood Cliffs: Prentice Hall (1995).
6. Pressman, R.S., *Software Engineering: A Practitioner's Approach* (5th edn). New York: McGraw-Hill (2001).
7. Bruegge, B. and Dutoit, A.H., *Object-Oriented Software Engineering: Conquering Complex and Changing Systems*. Englewood Cliffs: Prentice Hall (2000).
8. Binder, R.V., Testing object-oriented software: a survey. *J. of Software Testing, Verification and Reliability*, 6, **3-4**, 125-252 (1996).
9. Offutt, A.J., An experimental evaluation of data flow and mutation testing. *Software - Practice and Experience*, 26, **2**, 165-176 (1996).
10. Schach, S.R., Testing principles and practice. *ACM Computing Reviews*, 28, **1**, 277-279 (1996).
11. Weyuker, E.J., Comparison of program testing strategies. *Proc. 4th Symp. on Software Testing, Analysis and Verification*. Victoria, Canada, 1-10 (1991).
12. Abbott, J., *Software Testing Techniques*. Manchester: NCC Publications (1986).
13. Cheatham, T.J. and Mellinger, L., Testing object-oriented software systems. *Proc. ACM Computer Science Conf.*, New York, USA, 161-165 (1990).
14. Dibachi, R., Techniques for testing java applications. *Proc. 6th Inter. Conf. on Software Testing, Analysis and Review*. San Jose, USA, 481-494 (1997).
15. Fiedler, S.P., Object-oriented unit testing. *Hewlett-Packard J.*, 40, **2**, 69-74 (1989).
16. Frankl, P., A Framework for Testing Object-oriented Programs. Technical Report, Department of Electrical Engineering and Computer Science, Polytechnic University, New York (1989).
17. Hoffman, D.M. and Strooper, P.S., A case study in class testing. *Proc. CASCON '93*, Toronto, Canada, 472-482 (1993).
18. Labiche, Y., Thevenod-Fosse, P., Waeselynck, H. and Durand, M.H., Testing levels for object-oriented software. *Proc. 22nd IEEE Inter. Conf. on Software Engng. (ICSE).* Limerick, Ireland, 136-145 (2000).
19. Murphy, G.C., Townsend, P. and Wong, P.S., Experiences with cluster and class testing. *CACM*, 37, **9**, 39-47 (1994).
20. Graham, D., Testing o-o systems. *Proc. Object Expo. and Java Expo*. London, England, UK, 309-318 (1996).