Experience With Teaching Black-Box Testing in a Computer Science/Software Engineering Curriculum

T. Y. Chen, *Member, IEEE,* and Pak-Lok Poon, *Member, IEEE*

***Abstract*—Software testing is a popular and important technique for improving software quality. There is a strong need for universities to teach testing rigorously to students studying computer science or software engineering. This paper reports the experience of teaching the classification-tree method as a black-box testing technique at the University of Melbourne, Melbourne, Australia, and Swinburne University of Technology, Melbourne, Australia. It aims to foster discussion of appropriate teaching methods of software testing.**

***Index Terms*—Black-box testing, classification-tree method (CTM), software testing, specification-based testing, test-case selection.**

I. Introduction

ANY SOFTWARE testing method can be classified into either the white-box or the black-box approach. The *white-box* approach concerns the construction of test cases from the source code of the program. Many white-box testing methods require the coverage of certain features of the program's structure. Examples are control-flow testing [2] and data-flow testing [2], [3]. The basic rationale is that unless a given feature of the program's structure is executed, there is no way to reveal the possible faults associated with it. On the other hand, the *black-box* approach involves the construction of test cases that is based on the specification of the program. It treats the program as a "black box," the internal structure of which is unknown. Examples are random testing [4] and cause–effect graphing [5].

Generally speaking, software developers tend to spend 40–50% of predelivery development costs on testing in order to achieve reasonable quality levels [6], [7]. In addition, studies undertaken by IBM and others showed that to correct a fault after coding is at least ten times as costly as before and 100 times as costly to correct a production fault [8]. There is no doubt that testing remains a popular and important technique for improving software quality, by uncovering the faults in the program as early as possible, so that these faults will not be propagated through to the final production software, where the cost of removal is far greater.

Because of the importance of testing in the assurance of software quality [9], [10], there is a strong need for universities to teach testing rigorously to students studying computer science or software engineering [11]–[13]. Still, many people consider that a brief introduction to testing in a software engineering text is sufficient. The authors argue that this approach to teaching software testing is inadequate, mainly because the students would not be able to acquire any hands-on testing experience. In fact, Shepard *et al.* [7] have also pointed out that the current level and coverage of teaching testing in most computer science/software engineering curricula is too low. This problem motivates the investigation of an effective way to teach software testing, particularly black-box testing.

The remainder of this paper is structured as follows. Section II discusses some major constraints in teaching black-box testing. Section III outlines and compares some common black-box testing methods. Section IV introduces the classification-tree method (CTM). Section V discusses the rationale for teaching this method. Section VI describes the authors' approach to teaching CTM at the University of Melbourne (UM), Melbourne, Australia. Section VII discusses some common types of incorrect classification. Section VIII discusses experience and observations of teaching CTM in the UM. Section IX discusses the effectiveness of this teaching approach, by applying it at Swinburne University of Technology (SUT), Melbourne, Australia. Section X describes some related work. Finally, Section XI concludes the paper.

II. Constraints of Teaching Black-Box Testing

In most universities, both white-box testing and black-box testing will be taught to students studying computer science or software engineering. Neither testing approach is sufficient; each is complementary to the other.

The following are some of the major constraints found in teaching black-box testing.

a) *Small variety and less systematic approach of black-box testing methods*. Compared with white-box testing, there are significantly fewer black-box testing methods. In addition, most black-box testing methods rely heavily on human intuition and, hence, are less systematic. As a result, difficulties are encountered when choosing which black-box testing methods to teach. Regardless of these difficulties, teaching black-box testing cannot be excluded, because black-box testing and white-box

Manuscript received November 16, 1999; revised December 10, 2002. This work was supported in part by the Research Grants Council of Hong Kong under Project HKU 7029/01E and CityU 1048/01E. A preliminary version of this paper [1] was presented at the 1998 International Conference on Software Engineering: Education and Practice, Dunedin, New Zealand, in January 1998.

T. Y. Chen was with the University of Melbourne, Melbourne, Australia. He is now with the School of Information Technology, Swinburne University of Technology, Hawthorn 3122, Australia.

P.-L. Poon is with the School of Accounting and Finance, The Hong Kong Polytechnic University, Kowloon, Hong Kong (e-mail: afplpoon@inet.polyu.edu.hk).

Digital Object Identifier 10.1109/TE.2003.817617

testing are complementary to each other. Furthermore, black-box testing is extensively used in the industry.

b) *Difficulty in obtaining access to the associated automated tools*. Some black-box testing methods, such as the category-partition method [14], [15], require automated tools to support teaching. This requirement creates further difficulty in teaching because these automated tools are not easily accessible.

c) *Lack of knowledge of formal specifications*. Any specification can be classified into either a *formal* or an *informal* specification. The former documents customer requirements for software using a mathematical notation. The latter primarily uses natural language as the medium of specification. Intuitively, teaching black-box testing would be easier for formal specifications. However, because there is no consensus on the order of teaching formal methods and testing among different universities, students in some universities may not have learned formal specifications when they learned black-box testing.[1]

d) *Inexperience in software applications*. Construction of test cases from informal specifications is usually performed in an ad hoc manner. In addition, many black-box testing methods are *function oriented*; that is, test cases are constructed based on the functions of the software. Hence, the effective use of these testing methods by the students depends largely on their expertise and experience in the application areas of the software. However, most students (especially undergraduates) have not yet acquired an adequate level of expertise and experience, mainly because of the lack of understanding of requirement specifications in these application areas.

III. Overview of Common Black-Box Testing Methods

With respect to the constraints discussed in the preceding section, the authors have considered the feasibility of teaching several common black-box testing methods. They include the following.

a) *Random testing*. This method is unarguably the simplest; it requires test cases to be selected at random from the input domain [4]. It makes no claims to cover anything (for example, program statements, as in control-flow testing), except insofar as chance dictates. Therefore, random testing is not expected to compete with other testing methods in its ability to find faults [4].

b) *Cause–effect graphing*. In this method, *causes* (inputs that elicit a response from the program) and *effects* (outputs or any observable responses of the program) are identified from the specification, and their relationships are described using a boolean graph [5]. This method has the advantage of exercising combinations of inputs that might not otherwise have been tried. Its major drawback is the creation of the boolean graph, which can be tedious when many causes and effects are involved.

[1]One may argue that constraint (c) is not applicable to those universities where formal specifications are taught prior to testing. The authors note, however, that teaching black-box testing for informal specifications is still needed because formal methods may not be used in some commercial organizations.

c) *Category-partition method (CPM)*. This method involves the identification of *categories* (the major properties that categorize the input parameters or environmental conditions of a program) and their associated *choices* (the different kinds of values or value ranges that are possible for a category) [14]. The constraints among the identified choices are determined and represented in a textual form. An associated generator tool is then used to generate valid combinations of choices as test cases [15].

d) *Classification-tree method (CTM)*. This method is similar to CPM in many ways [16], [17]. The major difference between them is that CTM captures the constraints among classifications (which are equivalent to categories in CPM) using a tree structure, while CPM captures the constraints among choices in a textual form.

e) *Decision-table method*. In this method, relevant conditions and actions are identified from the specification, then a set of rules that links each combination of conditions and the corresponding combination of actions is determined. From each of these rules, a test case is generated [18]. This method is considered to be ineffective compared with CPM and CTM because it does not make use of the constraints among conditions to avoid invalid combinations.

In view of the constraints mentioned in Section II, these methods have been evaluated to determine which ones are appropriate for teaching. Ideally, the selected methods should be relatively systematic and, therefore, easy to teach. In addition, they should not rely heavily on automated tools and should not require knowledge of formal specifications and substantial experience of large software development. Based on these selection criteria, the authors consider that CTM is the most appropriate method for teaching. The rationale for selecting CTM for teaching will be discussed in Section V.

IV. Overview of the Classification-Tree Method

Before explaining the reasons for choosing CTM for teaching, the authors first present an overview of this method. The basic concept of CTM is to generate all possible test cases based on the input domain of a program, via the construction of a classification tree [16]. CTM is comprised of the following steps.

a) The first step is the identification of all the classifications and their associated classes from the specification. *Classifications* are defined as the different criteria for partitioning the input domain of the program to be tested; *classes* are defined as the disjoint subsets of values for each classification. A classification is either a p-type or an e-type classification. A *p-type classification* is defined for any parameter, that is, an explicit input to a program; otherwise, a classification is defined as an *e-type classification*, which normally represents an environmental condition.

Consider, for instance, a program $\mathrm{P_{SQRT}}$ that reads an input file F containing two real numbers m and n and outputs the value of $\sqrt{m+n}$. Here, "Status of F" is an

A goods trading system P_{TRADE} shall be developed for the Best Wholesale Shop. For each credit purchase, P_{TRADE} accepts the customer and purchase details to determine whether this purchase should be approved. The following describes the functions of P_{TRADE}.

1) Accept the customer number.
2) Check the customer number against the Customer Master File. If the customer number exists, go to step (3). Otherwise, reject the transaction and stop.
3) Check the status of the customer. If it is deactivated, go to step (4). Otherwise, go to step (5).
4) Check whether the transaction is supported by special management approval. If yes, accept the transaction and stop. Otherwise, reject the transaction and stop.
5) Compare the credit limit and the invoice amount. If the credit limit is less than the invoice amount, reject the transaction. Otherwise, accept the transaction.

It should be noted that details of customer number, invoice amount, and management approval are manually entered by users. Other details, such as customer's status and credit limit, are automatically retrieved from the Customer Master File by P_{TRADE}.

Fig. 1. Specification S_{TRADE} for the program P_{TRADE}.

e-type classification and "$m + n$" is a p-type classification. The first classification "Status of F" has three associated classes, namely "Status of $F =$ Does Not Exist," "Status of $F =$ Exists but Empty," and "Status of $F =$ Exists and Non-Empty."[2] On the other hand, the second classification "$m+n$" has "$m+n < 0$" and "$m+n \geq 0$" as its associated classes.

b) The second step is the construction of the classification tree $\mathcal{T}$ from the identified classifications and classes.
c) The third step is the construction of the test-case table from $\mathcal{T}$.
d) The final step is the identification of all the feasible combinations of classes from the test-case table. Each combination of classes thus represents one test case.

Example 1 presented hereafter illustrates the above concept.

Example 1 (Trading Example): Suppose a program P_{TRADE}, with the specification denoted by S_{TRADE}, is to be tested, as in Fig. 1.

[2]When there is no ambiguity, these classes can be simply referred to as "Does Not Exist," "Exists but Empty," and "Exists and Non-Empty."

Suppose that all the classifications and classes for S_{TRADE} are identified, as in Table I. In this table, only "Status of Customer Master File" is an e-type classification; whereas, all the remaining ones are p-type classifications. With Table I, an obvious and simple approach (referred to as the *select-and-combine* approach) is to select and combine one class from each classification so that each combination represents one test case. Using this approach, the total number of test cases constructed is $72 (= 3 \times 2 \times 2 \times 3 \times 2)$.

Given any test case tc, it is *valid* only if: 1) tc does not contain a group of contradictory classes with respect to the specification and 2) tc contains a sufficient number of classes from which a stand-alone input can be formed. Otherwise, tc is *invalid*. Consider the select-and-combine approach again. Despite its simplicity, numerous invalid test cases may be generated because of combining contradictory classes. For example, the class "Existence of Customer Number = No" should not be combined with any class of the classification "Status of Customer" to form part of a test case. The reason is that the absence of a customer number in the Customer Master File would mean that the corresponding customer record does not exist in the file; hence, one need not consider such a scenario. By combining these contradictory classes, a total of 36 invalid test cases will be generated. It should be noted, however, that additional invalid test cases will be generated because of the combination of other contradictory classes, such as "Status of Customer Master File = Does Not Exist" and "Status of Customer = Activated."

CTM was first developed by Grochtmann and Grimm [16] and enhanced by Chen and Poon [19] to reduce the number of invalid test cases. In this method, the reduction is achieved by capturing the constraints among classifications in a classification tree. Fig. 2 shows a classification tree $\mathcal{T}_{TRADE}$ with part of the test-case table for S_{TRADE}, constructed from the classifications and classes in Table I. In this figure, there are several observations: 1) classifications are enclosed in boxes, while classes are not; 2) each column of the test-case table corresponds to a terminal class of $\mathcal{T}_{TRADE}$; and 3) each row of the test-case table corresponds to one test case (for example, row 4 of the table represents a test case tc = {Status of Customer Master File = Exists and Non-Empty, Existence of Customer Number = Yes, Status of Customer = Activated, Credit Limit− Invoice Amount > 0, Supported by Management Approval = No}).

A circle at the top of a classification tree $\mathcal{T}$, as shown in Fig. 2, is the *general root node*. It represents the whole input domain. The classifications directly under the general root node, such as "Status of Customer Master File" and "Supported by Management Approval" in Fig. 2, are called *top-level classifications*. In general, a classification X may have one or more classes directly under it. X is known as the *parent classification*, and each of these classes under it is known as a *child class*. In Fig. 2, for example, "Status of Customer" is the parent classification of "Deactivated" and "Activated," whereas "Deactivated" and "Activated" are the child classes of "Status of Customer." Similarly, a class x may have one or more classifications directly under it. Then, x is known as the *parent class*, and each of these classifications under it is known as a *child classification*. In Fig. 2, for example, "Exists and Non-Empty" is the parent class of "Existence of Customer Number," whereas "Existence

TABLE I
POSSIBLE CLASSIFICATIONS AND CLASSES FOR S_{TRADE}

Classifications	Associated Classes
Status of Customer Master File	Does Not Exist, Exists but Empty, Exists and Non-Empty
Existence of Customer Number	Yes, No
Status of Customer	Deactivated, Activated
Credit Limit – Invoice Amount	< 0, = 0, > 0
Supported by Management Approval	Yes, No

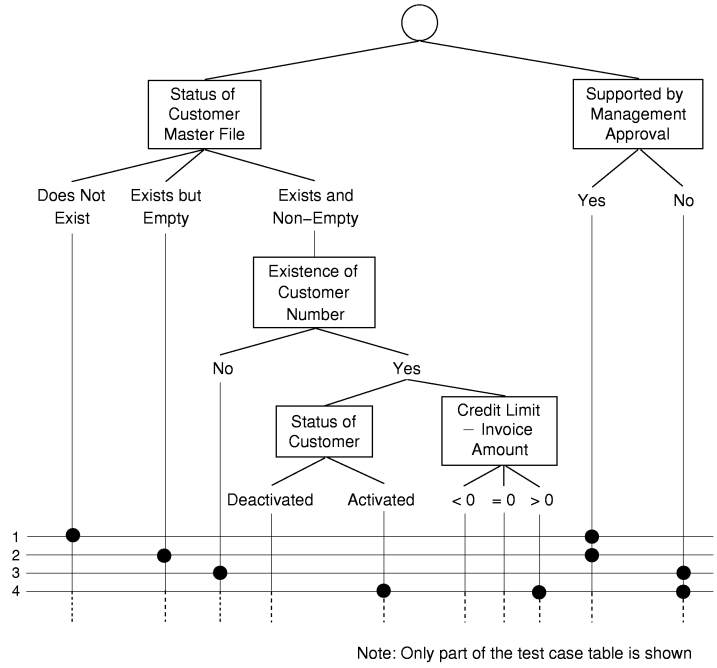

Fig. 2. Classification tree $\mathcal{T}_{TRADE}$ for S_{TRADE}.

of Customer Number" is a child classification of "Exists and Non-Empty."

From $\mathcal{T}$, test cases can be expressed in the test-case table using the following steps.

1) Draw the grids of the test-case table under $\mathcal{T}$. The columns of the table correspond to the terminal nodes of $\mathcal{T}$. The rows correspond to test cases.
2) Construct a test case in the test-case table by selecting a combination of classes in $\mathcal{T}$ as follows.
   a) Select one and only one child class of each top-level classification.
   b) For every child classification of each selected class, recursively select one and only one child class.

By using these steps, a total of 18 test cases can be constructed from $\mathcal{T}_{TRADE}$. Some of these test cases are shown in the part of the test-case table in Fig. 2. For example, in the first test case in the table, dots associated with "Status of Customer Master File = Does Not Exist" and "Supported by Management Approval = Yes" represent the selection of such classes.

V. RATIONALE FOR TEACHING CTM

Having introduced the concept of CTM in Section IV, the authors are now ready to present the rationale for teaching it as a black-box testing technique, with respect to the selection criteria stated in Section III.

a) *Well-defined steps and deliverables*. Obviously, for any black-box testing method that relies heavily on human intuition, the experience of the software tester will have a significant impact on the effectiveness of applying this method. Unfortunately, experience is difficult or even impossible to teach. However, CTM consists of four well-defined steps with clearly defined deliverables, namely: 1) the identification of classifications and their associated classes; 2) the construction of a classification tree $\mathcal{T}$; 3) the construction of the test-case table; and 4) the construction of test cases. With well-defined steps and deliverables, teaching CTM is easy.

b) *Less reliance on automated tools*. CTM can still be applied without the aid of automated tools, as long as the specification is not too complex.

c) *No restriction on the type of specification*. As illustrated in Example 1, CTM can be easily applied to an informal specification. This method is equally applicable to a formal specification [17]. Therefore, CTM can be taught to those students who do not have any training in formal specifications.

d) *Less reliance on the experience of large software development*. Unlike most black-box testing methods, CTM

TABLE II
COMPARISON OF DIFFERENT BLACK-BOX TESTING METHODS

	Random Testing	Cause-Effect Graphing	Category-Partition Method (CPM)	Classification-Tree Method (CTM)	Decision-Table Method
Well defined steps & deliverables	×	√	√	√	√
Less reliance on automated tools	×	×	×	√	√
No restriction on the type of specification	√	√	√	√	√
Less reliance on the experience of large software development	√	×	√	√	×

focuses on the possible combinations of data (or classes) in the input domain (as opposed to the functions) of the software. Thus, relatively speaking, past experience in large software development has less impact on the successful application of CTM.

Table II summarizes the appropriateness of the black-box testing methods (as listed in Section III) for the purpose of teaching with respect to selection criteria (a)–(d) above. This table shows that, with the exception of CTM, other black-box testing methods cannot fulfill all the selection criteria. For example, cause–effect graphing and CPM require automated tools to support teaching. This requirement also applies to random testing when nonnumeric inputs are involved. In addition, both cause–effect graphing and the decision-table method require substantial experience of software development in order to apply them effectively.

VI. Approach to Teaching the Classification-Tree Method

In UM, students studying for computer science or software engineering may elect an advanced software engineering course that lasts for one semester (its code is 342; this code will be used to refer to the course for the remainder of the paper). The course 342 involves 26 lecture hours and 12 tutorial hours. Roughly speaking, one-half of the course is devoted to white-box and black-box testing. The other half covers topics such as programming language issues, program portability and performance, computer-aided software engineering (CASE), user interfaces, and software configuration management.

Course 342 starts with white-box testing first, followed by black-box testing. For white-box testing, the instructors teach control-flow coverage criteria [2], data-flow coverage criteria [2], [3], and domain-testing strategies [20], [21]. For black-box testing, CTM [16], [17] and random testing [4] are taught. (CTM is selected for teaching because all the constraints described in Section II are applicable to UM.) One advantage of teaching random testing in addition to CTM is that students are not restricted to learning only one black-box testing method. The students can then compare CTM and random testing themselves. The coverage of both white-box testing and black-box testing in course 342 reflects the authors' view that neither approach is sufficient; each is complementary to the other.

In course 342, a one-hour lecture was devoted to the introduction of CTM. Teaching of the method was supported by related literature such as [16], [17], [19], [22]. The lecture was reinforced by a one-hour tutorial with various examples (including the one used in [16], which involves a program used to count the incidence of an element within a list), and a project which required students to construct $\mathcal{T}$s for two informal specifications. For each specification, students were asked to identify possible classifications and classes and to construct the corresponding $\mathcal{T}$. In the project, two business-oriented specifications were deliberately used. Thus, the students practiced CTM with specifications typically found in the commercial sector. In fact, $\mathrm{S_{TRADE}}$ in Example 1 was one of the specifications used in the project. The other specification in the project, denoted by $\mathrm{S_{CARD}}$, was related to the use of credit cards for the purchase of various goods. To further improve the students' understanding of the method, a tutorial was organized after the students had submitted their projects. In this tutorial, some common mistakes made by the students in the project (these mistakes will be discussed in Section VIII) were discussed. Through the discussion, the students learned why these mistakes were made and how to avoid repeating them in the future.

Normally, more than one correct $\mathcal{T}$ may be constructed from the same specification because a different set of classifications and classes may be defined for the same specification at the tester's will. For example, when testing resource constraints are tight, the software tester may define classifications and classes at a more abstract level. In this way, fewer classifications and classes will be defined, resulting in a simpler $\mathcal{T}$. Obviously, the simpler the $\mathcal{T}$, the smaller will be the number of generated test cases. In fact, this situation of having different but correct $\mathcal{T}$s for the same specification occurred in the project used in 342. Hence, the authors selected some of these $\mathcal{T}$s constructed by the students and discussed their structural properties. They also provided some criteria for determining which $\mathcal{T}$ was better. Examples of these criteria include the number of classifications and classes, the ease of understanding $\mathcal{T}$, and the effectiveness of $\mathcal{T}$ in the generation of valid test cases.

In the tutorial also discussed was the problem of applying CTM to increasingly complex specifications. The students learned that, when this situation arises, the complex specification should be decomposed into smaller functional units that can be "independently" tested before the application of CTM [16]. The constraints that were found in teaching black-box testing (as discussed in Section II) and the mistakes made by the students in the project reinforced the authors' earlier argument that software testing should be taught rigorously.

VII. Common Types of Incorrect Classification

Before discussing observations in Section VIII, some common types of incorrect classification, which will facilitate subsequent discussion, are presented. These incorrect classifications are usually caused by a partial understanding of the problem domain of the specification or the concepts of classifications and classes.

a) *Classification with overlapping classes (OLCA)*. Classes are said to be *overlapping* if they have common elements or values. For example, suppose that the classification "Credit Limit − Invoice Amount" is defined for S_{TRADE} in Example 1 (Fig. 1), with "< 0," "$= 0$," and "≥ 0" as its three associated classes. In this case, "$= 0$" and "≥ 0" are overlapping classes, and "Credit Limit − Invoice Amount" is an OLCA.

b) *Classification with composite classes (CMCA)*. A *composite class* is one which should be replaced by two or more different classes; otherwise, some important test cases would be missing. For example, suppose that the classification "Credit Limit − Invoice Amount," with "< 0" and "≥ 0" as its two associated classes, is defined for S_{TRADE} in Example 1. In this case, either (Credit Limit = Invoice Amount) or (Credit Limit $>$ Invoice Amount) may not be tested. Hence, "≥ 0" is a composite class, and "Credit Limit − Invoice Amount" is a CMCA.

c) *Biased (BACA) and unbiased (UBCA) classifications with composite classes*. Any CMCA is a *biased* classification with composite classes (BACA) if its semantic meaning leads to the definition of composite classes. Otherwise, it is an *unbiased* classification with composite classes (UBCA). Consider the CMCA "Credit Limit − Invoice Amount" with the classes "< 0" and "≥ 0" in (b). The introduction of the composite class "≥ 0" is not forced by the semantic meaning of "Credit Limit − Invoice Amount." Hence, "Credit Limit − Invoice Amount" is a UBCA. Instead of defining the classification "Credit Limit − Invoice Amount," suppose that a new classification "Credit Limit $>$ Invoice Amount" is defined. This new classification forces the classes "Yes" and "No" to be defined. In this case, either (Credit Limit $<$ Invoice Amount) or (Credit Limit = Invoice Amount) may not be tested. Hence, "No" is a composite class. As a consequence, "Credit Limit $>$ Invoice Amount" is a BACA.

d) *Composite classification (CMCF)*. CMCF is defined as a classification that should be replaced by two or more distinct classifications; otherwise, some important test cases would not be constructed. For example, suppose that the following classifications and classes are defined for S_{CARD}:

- the classification "Credit Limit" (denoted by CL) with "$CL = \$2000$" (for a classic credit card) and "$CL = \$4000$" (for a gold credit card) as its two associated classes;
- the classification "Purchase Amount" (denoted by PA) with "$\$0.00 \leq PA \leq \$2000.00$," "$\$2000.00 < PA \leq \$4000.00$," and "$PA > \$4000.00$" as its three associated classes.

By varying the combination of the classes of these two classifications, four test cases can be constructed to cover the following four distinct situations:

- an accepted purchase using a gold credit card, by combining "$CL = \$4000$," "$\$2000.00 < PA \leq \$4000.00$," and possibly classes from other classifications;
- a rejected purchase using a gold credit card, by combining "$CL = \$4000$," "$PA > \$4000.00$," and possibly classes from other classifications;
- an accepted purchase using a classic credit card, by combining "$CL = \$2000$," "$\$0.00 \leq PA \leq 2000.00$," and possibly classes from other classifications; and
- a rejected purchase using a classic credit card, by combining "$CL = \$2000$," "$\$2000.00 < PA \leq \$4000.00$," and possibly classes from other classifications.

Suppose that both classifications "Credit Limit" and "Purchase Amount" are replaced by another classification "Credit Limit − Purchase Amount," with "< 0" and "≥ 0" as its two associated classes. Obviously, these two classes cannot be used to construct sufficient test cases that fully cover the above four situations. Therefore, "Credit Limit − Purchase Amount" is a CMCF.

VIII. Observations and Suggestions

A. Observations

In the academic year 1998, there were 48 students studying in course 342. The authors observed that articles [16] and [22] were particularly useful for introducing the basic concept of CTM. The teaching of articles [17] and [19] was only required when students wanted to learn CTM in depth. When reviewing students' project work, there were several observations made.

a) *Identification of incorrect classifications*. Many students have only been able to identify p-type classifications. This observation suggests that, when teaching CTM, the students should be reminded not to miss the identification of e-type classifications. In addition, the incorrect classifications made by the students have been analyzed. The most common types of incorrect classification were: 1) classifications with overlapping classes (OLCA); 2) biased classifications with composite classes (BACA); 3) unbiased classifications with composite classes (UBCA); and 4) composite classifications (CMCF).

Table III shows the percentages of students' submissions containing at least one OLCA, BACA, UBCA, or CMCF.[3] These percentages are quite high, with averages of 24.0% and 52.6% for S_{TRADE} and S_{CARD}, respectively. Table IV shows the percentages of occurrences of OLCA, BACA, UBCA, or CMCF in all students' submissions, with respect to the total number of identified classifications. The percentages of incorrect classifications shown in Table IV are not negligible (with averages of 4.2% and 6.6% for S_{TRADE} and S_{CARD}, respectively). In summary, Tables III and IV suggest that students are likely to define incorrect classifications from specifications.

[3]Meanwhile, in Tables III and IV, only the percentages that are *not* enclosed in brackets are considered. These percentages are derived from data obtained at UM. Those percentages enclosed in brackets that are derived from data obtained in another university will be discussed in Section IX.

TABLE III
PERCENTAGES OF STUDENTS' SUBMISSIONS CONTAINING OLCA, BACA, UBCA, OR CMCF

Specification	Percentage of Students' Submissions Containing at Least One				Average Percentage
	OLCA	BACA	UBCA	CMCF	
S_{TRADE}	12.5 (4.5)	35.4 (13.6)	35.4 (18.2)	12.5 (4.5)	24.0 (10.2)
S_{CARD}	52.1 (22.7)	35.4 (9.1)	37.5 (31.8)	85.4 (63.6)	52.6 (31.8)

TABLE IV
PERCENTAGES OF OLCA, BACA, UBCA, OR CMCF IN ALL STUDENTS' SUBMISSIONS

Specification	Percentage of Occurrences in All Students' Submissions				Average Percentage
	OLCA	BACA	UBCA	CMCF	
S_{TRADE}	2.2 (0.7)	6.2 (2.1)	6.2 (2.7)	2.2 (0.7)	4.2 (1.6)
S_{CARD}	5.3 (2.2)	4.5 (0.9)	4.1 (3.1)	12.6 (7.6)	6.6 (3.5)

b) *Construction of $\mathcal{T}$ based on control-flow information*. The fundamental purpose of using $\mathcal{T}$ is to facilitate the identification of all possible combinations of classes in the input domain of the software [16]. However, more than 80% of students have assembled the classifications and classes into a tree based on the *control flow* of the software. Fig. 3 shows a $\mathcal{T}$ constructed for S_{TRADE} by some of these students using such an approach. Compared with Fig. 2, Fig. 3 has excluded some valid test cases. A typical example of missing valid combinations of classes is "Status of Customer = Activated" and "Supported by Management Approval = Yes." In addition, constructing $\mathcal{T}$ based on control flow would make CTM no longer a genuine black-box method, since knowledge of control flow cannot be obtained without some information about the program itself.[4]

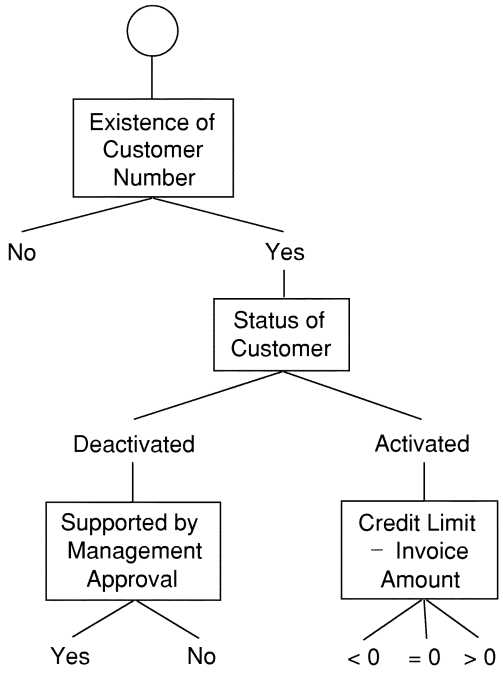

Fig. 3. One classification tree for S_{TRADE} based on control flow.

B. Suggestions

The authors have two main suggestions regarding the teaching of CTM. First, observation (b) in Section VIII-A suggests that two points should be emphasized in teaching CTM:

1) State clearly and repeatedly that all possible combinations of classes in the input domain are the main focus of this method.
2) Elaborate on the differences between a $\mathcal{T}$ and a control-flow diagram by reminding students that a $\mathcal{T}$ is data oriented rather than control oriented.

Second, although CTM is relatively more systematic and formalized than most black-box testing methods, it still has two ad hoc components, namely: 1) the identification of classifications and classes and 2) the assembly of classifications and classes into $\mathcal{T}$. Hence, experience recommends that the teaching of CTM should consist of the following steps.

- Prepare a formal lecture to discuss the concept of the method (primarily based on the articles [16] and [22]) and elaborate in detail the difference between a $\mathcal{T}$ and a control-flow diagram.
- Develop a tutorial using various examples to reinforce the concept of the method. In this tutorial, students should be reminded that there may be e-type classifications in addition to p-type classifications, and examples should be given to illustrate common types of incorrect classification.
- Use a project to reinforce the lecture and tutorial contents through practice.
- Prepare a follow-up tutorial, after the students have submitted their projects, to explain their mistakes, and give some guidelines on how these mistakes can be avoided.

In addition, explanations have been given to students that software quality should not rely solely on testing, because testing is only one of many verification and validation techniques [7]. In many situations, life-cycle quality approaches, such as reviews and inspections, quality metrics, and soft-

[4]It may be argued that if the "Status of Customer" is "Activated," then "Supported by Management Approval" is never checked. This argument raises a pedagogic question about the use of a specification to generate test cases without taking into account "implied" control flow. In the project, the students never saw the code; they worked from the specifications. Further investigation of this issue is worthwhile but beyond the scope of this paper.

ware processes, should be used with testing. For example, inspections should be used as often as possible to prevent requirement defects from propagating to subsequent phases of the software development life cycle. Otherwise, program faults may occur, resulting in schedule delays and additional costs, even if software developers can eventually detect and remove these faults.

IX. Effectiveness of Recommended Approach

In order to measure the effectiveness of their recommended teaching approach mentioned in Section VIII, the authors have applied it to a class of 22 undergraduates in the School of Information Technology, SUT, by using the same project of the course 342 in UM. These 22 students are currently enrolled in the Bachelor of Applied Science (Computer Science and Software Engineering) program or the Bachelor of Software Engineering program, and they are in their third or fourth year of study. The authors observed that all the teaching constraints described in Section II are also applicable here. The effectiveness of their recommended approach is analyzed as follows.

(a) *Identification of incorrect classifications*. Most of the students in SUT were able to identify e-type classifications. Only three out of these 22 students did not identify any e-type classification. This result represents a significant improvement compared with the students in UM, where almost all were unable to identify any e-type classification. In addition, consider the percentages enclosed in brackets in Tables III and IV, which are applicable to SUT. In Table III, the average percentage of students' submissions containing at least one OLCA, BACA, UBCA, or CMCF is reduced from 24.0% (for UM) to 10.2% (for SUT) for S_{TRADE} and from 52.6% (for UM) to 31.8% (for SUT) for S_{CARD}. This result represents an improvement of 57.5% for S_{TRADE} and 39.5% for S_{CARD}. Similarly, in Table IV, the average percentage of occurrences of OLCA, BACA, UBCA, or CMCF in all students' submissions is reduced from 4.2% (for UM) to 1.6% (for SUT) for S_{TRADE} and from 6.6% (for UM) to 3.5% (for SUT) for S_{CARD}. The improvements are 61.9% and 47.0% for S_{TRADE} and S_{CARD}, respectively. These figures confirm the recommendations about the importance of the use of tutorials.

(b) *Construction of $\mathcal{T}$ based on control-flow information*. Before distributing the project to the students, a formal lecture was given on the difference between a $\mathcal{T}$ and a control-flow diagram by using a simple example. Nevertheless, many students still confused the two. A plausible reason for this confusion is that students with some programming experience tend to use control-flow information unconsciously for constructing test cases. Readers may note the pedagogic question about the use of "implied" control flow to generate test cases from a specification as mentioned in footnote 4. To solve this problem, the authors recommend that prior to starting their projects, students should have been given several specifications (preferably at least three) to practice the construction of $\mathcal{T}$s, control-flow diagrams, and sets of test cases.

One may argue that (a) above may not actually reflect the effectiveness of the recommended approach because the surveyed students in SUT were brighter and grasped the concepts faster, or perhaps the instruction was different in that setting. With regard to the difference in caliber between the students at UM and SUT, one should note that these two universities are located in the Melbourne City and that UM is generally regarded as the most established university in the city. Statistics also show that the average entrance score of UM is higher than that of SUT. This finding rules out the possibility that the better performance of the SUT students was a result of their higher caliber. In addition, efforts were made to ensure that the study in UM and that in SUT are the same, in terms of the instructor, the teaching materials, and the specifications used in the studies.

In SUT, questionnaires were used to obtain students' feedback on CTM. Out of the 22 students in the class, 18 completed and returned the questionnaires. Of these 18 students, 72% consider that CTM is easy to understand, and 89% indicate that it is easy to use, compared with random testing involving nonnumeric inputs.[5] In addition, 94% indicate that they will recommend CTM to others. After teaching CTM, the instructors observed that the students understood three important concepts: 1) comprehensive testing is only possible with the use of a systematic method; 2) the effective application of any testing method requires practice; and (3) test cases can be prepared without source code.

X. Related Work

There are some published works on the teaching of software testing. In their paper [11], Hoffman *et al.* describe a tool-based approach to teaching automated testing to both undergraduates and postgraduates. They also make extensive use of automated tools in testing and grading students' exercises for C modules and C++ class libraries. These tools reduce grading time, allowing graders to focus on issues other than execution behavior (such as code style). Carrington [12] describes some approaches to teaching black-box testing to second-year undergraduate classes over several years, by using Z specifications. In addition, he describes an undergraduate course that covers both software design and testing [13]. Ramakrishan and Sajeev [23] suggest and develop an Internet-based, interactive, multimedia environment for students to learn software testing in an object-oriented (O-O) system. Their Internet-based environment uses a case study approach with the use of various UML (Unified Modeling Language) diagrams. The environment also provides facilities for animating changes to a program undergoing testing and simulating test cases of varying complexity. The authors' teaching approach primarily makes use of informal specifications to derive test cases and does not rely heavily on automated tools. This approach differs from the work in [11] and [12] in that the former involves the extensive use of automated tools, while the latter focuses exclusively on formal specifications. The authors' approach is largely based on a project and differs from the work in [13] which uses a case-study approach to give students experience with multiple problems and multiple-solution techniques. Furthermore, the

[5]As in UM, random testing was taught to the students in SUT.

approach presented here is more general, compared with the approach suggested in [23], which is restricted to O-O testing.

XI. Conclusion

In this paper, the authors have described some major constraints of teaching black-box testing. They have also explained in detail why CTM should be chosen for teaching with respect to these constraints. Furthermore, they have discussed the main concept of CTM and how to teach it as a black-box technique in UM and SUT. Their experience has confirmed that the teaching of CTM is viable and effective. Based on the students' feedback, they have found that CTM is easy to understand and use and that students have found CTM useful. Since CTM is a specification-based testing technique, students have appreciated that specifications play a significant role in software quality and that more emphasis should be given to verification and validation in teaching where the role of specifications is important [7].

The authors note that it would be better if students were provided with programs to test by using CTM. Hence, the authors are studying the feasibility of integrating their course with some other programming courses. This approach would allow students to use CTM to find faults in programs written by other classmates. Moreover, the graphical layout of a classification tree greatly facilitates the students in understanding the important aspects and their interrelationships, as stated in the specification. This understanding would in turn help the students to implement programs according to the specification. Thus, the students will benefit if teaching of CTM is incorporated into other programming courses. Since the most pragmatic way to evaluate the usefulness of their course is to know to what extent the students would continue to use CTM after their graduation, the authors intend to conduct a follow-up survey.

Acknowledgment

Part of this research was performed when the authors were with the Department of Computer Science and Software Engineering at UM. They are grateful to the students at UM and SUT who were involved in this study for their efforts and feedback on CTM. They are also grateful to the anonymous reviewers of this paper for their constructive comments.

References

[1] T. Y. Chen and P. L. Poon, "Teaching black box testing," in *Proc. 1998 Int. Conf. Software Engineering: Education and Practice*, Dunedin, New Zealand, Jan. 1998, pp. 324–329.

[2] L. M. Foreman and S. H. Zweben, "A study of the effectiveness of control and data flow testing strategies," *J. Syst. Softw.*, vol. 21, no. 3, pp. 215–228, June 1993.

[3] S. Rapps and E. J. Weyuker, "Selecting software test data using data flow information," *IEEE Trans. Softw. Eng.*, vol. 11, pp. 367–375, Apr. 1985.

[4] R. Hamlet, "Random testing," in *Encyclopedia of Software Engineering*. New York: Wiley, 1994, pp. 970–978.

[5] M. Roper, *Software Testing*. London: McGraw-Hill, 1994, pp. 77–83.

[6] J. Sanders and E. Curran, *Software Quality: A Framework for Success in Software Development and Support*. Wokingham: Addison-Wesley, 1994, pp. 3–12.

[7] T. Shepard, M. Lamb, and D. Kelly, "More testing should be taught," *Commun. ACM*, vol. 44, no. 6, pp. 103–108, June 2001.

[8] W. Perry, *Effective Methods for Software Testing*. New York: Wiley, 1995, pp. 13–29.

[9] D. Gelperin and B. Hetzel, "The growth of software testing," *Commun. ACM*, vol. 31, no. 6, pp. 687–695, June 1988.

[10] R. Ferguson and B. Korel, "The chaining approach for software test data generation," *ACM Trans. Softw. Eng. Methodology*, vol. 5, no. 1, pp. 63–86, Jan. 1996.

[11] D. Hoffman, P. Strooper, and P. Walsh, "Teaching and testing," in *Proc. 9th Conf. Software Engineering Education*, Daytona Beach, FL, Apr. 1996, pp. 248–258.

[12] D. Carrington, "Teaching software testing," in *Proc. 2nd Australasian Computer Science Education Conf.*, Melbourne, Australia, June 1997, pp. 59–64.

[13] ——, "Teaching software design and testing," in *Proc. 28th Annu. Frontiers in Education Conf.*, Tempe, AZ, Nov. 1998, pp. 547–550.

[14] T. J. Ostrand and M. J. Balcer, "The category-partition method for specifying and generating functional tests," *Commun. ACM*, vol. 31, no. 6, pp. 676–686, June 1988.

[15] M. J. Balcer, W. M. Hasling, and T. J. Ostrand, "Automatic generation of test scripts from formal test specifications," in *Proc. 3rd ACM Annu. Symp. Software Testing, Analysis, and Verification*, Key West, FL, Dec. 1989, pp. 210–218.

[16] M. Grochtmann and K. Grimm, "Classification trees for partition testing," *Softw. Testing, Verification Reliab.*, vol. 3, no. 2, pp. 63–82, 1993.

[17] H. Singh, M. Conrad, and S. Sadeghipour, "Test case design based on Z and the classification-tree method," in *Proc. 1st Int. Conf. Formal Engineering Methods*, Hiroshima, Japan, Nov. 1997, pp. 81–90.

[18] R. Weber, *Information Systems Control and Audit*. Upper Saddle River, NJ: Prentice-Hall, 1999, pp. 724–727.

[19] T. Y. Chen and P. L. Poon, "Construction of classification trees via the classification-hierarchy table," *Inf. Softw. Tech.*, vol. 39, no. 13, pp. 889–896, 1997.

[20] L. J. White and E. I. Cohen, "A domain strategy for computer program testing," *IEEE Trans. Softw. Eng.*, vol. 6, pp. 247–257, May 1980.

[21] S. J. Zeil, F. H. Afifi, and L. J. White, "Detection of linear errors via domain testing," *ACM Trans. Softw. Eng. Methodology*, vol. 1, no. 4, pp. 422–451, Oct. 1992.

[22] T. Y. Chen, P. L. Poon, and S. F. Tang, "A systematic method for auditing user acceptance tests," *Inf. Syst. Audit Control J.*, vol. 5, pp. 31–36, Sept.–Oct. 1998.

[23] S. Ramakrishan and A. S. M. Sajeev, "An Internet environment for learning software testing processes," in *Proc. Int. Conf. Software Engineering and Applications*, Hyderabad, India, Dec. 1997, pp. 33–40.

T. Y. Chen (M'03) received the B.Sc. and M.Phil. degrees from the University of Hong Kong in 1971 and 1974, respectively, the M.Sc. and DIC degrees from the Imperial College of Science and Technology, London, U.K., both in 1976, and the Ph.D. degree from the University of Melbourne, Melbourne, Australia, in 1986.

He is currently the Professor of Software Engineering in the School of Information Technology, Swinburne University of Technology (SUT), Melbourne, Australia. He is a Member of the Editorial Board of *Software Testing, Verification and Reliability*. His research interests include software testing, debugging, software maintenance, and software design.

Pak-Lok Poon (M'01) received the Master's of Business degree in information technology from the Royal Melbourne Institute of Technology, Melbourne, Australia, in 1983 and the Ph.D. degree in software engineering from the University of Melbourne, Melbourne, Australia, in 2001.

He is currently an Assistant Professor in the School of Accounting and Finance, The Hong Kong Polytechnic University, Kowloon, Hong Kong, where he teaches courses on information systems. He is on the editorial committee of the *Information Systems Control Journal*. His research interests include software testing, requirement inspection, computer audit and control, electronic commerce, and computers in education.