Teaching Specification-Based Testing

Sau-Fun Tang[1]
Faculty of Information
and Communication Technologies
Swinburne University of Technology
Hawthorn 3122, Australia
sftang@hkbu.edu.hk

Pak-Lok Poon
School of Accounting
and Finance
The Hong Kong Polytechnic University
Hung Hom, Kowloon, Hong Kong
afplpoon@inet.polyu.edu.hk

T.Y. Chen
Faculty of Information and Communication Technologies
Swinburne University of Technology
Hawthorn 3122, Australia
tychen@it.swin.edu.au

Abstract

Historically, relatively less emphasis has been placed on software testing in comparison with other activities, such as systems analysis and design, of the software life cycle in an undergraduate computer science or software engineering curriculum. Testing, however, is a common and important technique used to detect program faults. Thus, testing must be taught rigorously to the students. This paper reports our experience of teaching the classification-tree method as a black-box testing technique at The University of Melbourne and Swinburne University of Technology.

Keywords Black-box testing, category-partition method, classification-tree method, specification-based testing, test case selection

1. Introduction

It is reported that about 75% of all large software systems delivered to the customer are failures that are either not used at all, or do not meet the customer's requirements [11]. The high failure rate indicates that software quality remains a critical issue that confronts most software practitioners and users. Because *testing* is a common and important technique in achieving software quality, there is a strong demand for universities to teach testing rigorously to students studying computer science or software engineering.

In general, any testing method can be classified into either the *white-box* or the *black-box* approach. The former approach involves the generation of test cases based on the source code of the program.

[1] Also with the Department of Finance and Decision Sciences, Hong Kong Baptist University, Hong Kong.

Examples are control-flow testing [8], data-flow testing [8, 13], and domain testing [7, 18]. The latter approach deals with the generation of test cases from information derived from the specification, without requiring the knowledge of the internal structure of the program. Examples are random testing [10], cause-effect graphing [14], the category-partition method (CPM) [3, 6, 12], and the classification-tree method (CTM) [1, 3, 5, 9, 16].

This paper focuses on the pedagogy for black-box testing, and it is based on our experience of teaching undergraduates and postgraduates who studied computer science or software engineering in two Australian universities, namely The University of Melbourne (UM) and Swinburne University of Technology (SUT) [2]. We believe that the effectiveness of teaching software testing depends on two important success factors:

(a) The "testing methods" we select for teaching, and

(b) The "ways" we teach the selected methods.

Consider success factor (a). Five black-box testing methods or approaches (namely, random testing [10], cause-effect graphing [14], CPM [3, 6, 12], CTM [1, 3, 5, 9, 16], and decision-table approach [17]) have been selected and evaluated for their appropriateness in teaching, with respect to the four evaluation criteria listed in Table 1 (see the leftmost column in the table) [2]. Based on the evaluation results shown in Table 1, CTM is selected for teaching because it is the only method that fulfils all the criteria (to be discussed in detail later). Additionally, random testing is selected for teaching although it can only fulfill some of the evaluation criteria. The rationale is that we do not want the students to learn only one testing method. Moreover, teaching both CTM and random testing allows the students to compare the effectiveness of the two methods themselves.

	Random Testing	Cause-Effect Graphing	Category-Partition Method (CPM)	Classification-Tree Method (CTM)	Decision-Table Approach
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	√	×	√	√	×

Table 1: Comparison of Different Black-Box Testing Methods or Approach

2. Overview of CTM

Before discussing how we teach CTM (that is, success factor (b) mentioned in Section 1), let us first outline the main concept of the method. Basically, CTM aims to identify all the feasible combinations of classes as "test cases", and at the same time suppress the combination of incompatible classes as far as possible. CTM consists of the following steps:

(1) Decompose the specification into functional units $\mathcal{U}$s that can be tested independently. For each $\mathcal{U}$ selected for testing, repeat steps (2)–(5) below.

(2) Identify a set of classifications and classes for the selected $\mathcal{U}$. *Classifications* are defined as the parameters and environment conditions (that is, the states of the system during execution) of $\mathcal{U}$ that will affect its execution behavior. *Classes* are defined as the non-overlapping subsets of the values of the corresponding category.

(3) Construct a classification tree $\mathcal{T}$ from the identified classifications and classes.

(4) Construct the test case table from $\mathcal{T}$.

(5) Identify all the feasible combinations of classes from the test case table. Each combination of classes thus represents a test case.

Figure 1 shows a sample classification tree $\mathcal{T}_{\text{TRADE}}$, constructed for a specification S_{TRADE} related to the credit sales of goods by a wholesaler to retail customer. Readers may refer to [2] for details.

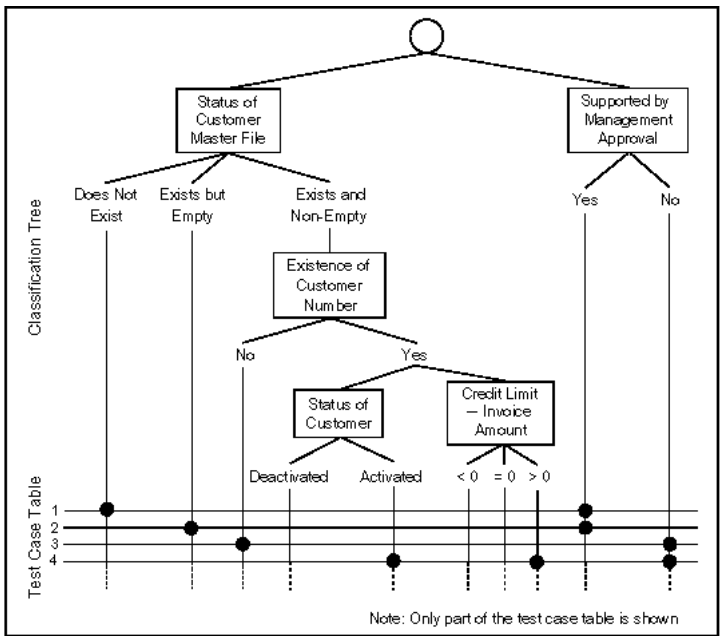

Figure 1: Classification Tree $\mathcal{T}_{\text{TRADE}}$ for S_{TRADE}

Having outlined the main steps of CTM, the following presents the rationale for teaching the method as a black-box testing technique, with respect to the evaluation criteria listed in Table 1:

- *Well-defined steps and deliverables*. CTM consists of well-defined steps with clearly defined deliverables, thus teaching the method is easy.
- *Less reliance on automated tools*. CTM can be applied without the aid of automated tools, as long as the specification is not too complex. This will ease the potential problem in obtaining access to the associated automated tool.

- *No restriction on the type of specification*. CTM can be easily applied to an "informal" specification. Therefore, the method can be taught to those students who do not have any training in formal specifications.
- *Less reliance on the experience of large software development*. CTM focuses on the possible combinations of data (or classes) in the input domain of the software. Thus, compared to other black-box testing methods, past experience in large software development has less impact on the successful application of CTM.

3. Approach to Teaching CTM

Now, we are ready to discuss success factor (b) mentioned in Section 1. We first taught CTM in an advanced software engineering course (referred to as "course 342") in UM. This course starts with white-box testing, followed by black-box testing (which covers both CTM and random testing). The coverage of both white-box and black-box testing in course 342 reflects our view that neither approach is sufficient; each is complementary to the other.

Based on our observation of the students' performance and the results of a project on CTM done by the students, we have refined our teaching approach in UM and then reapplied it in SUT. The results of the students' projects in SUT are better than that in UM, thus confirming the effectiveness of our refined teaching approach. This teaching approach consists of the following steps:

(1) Prepare a formal lecture to discuss the concept of the CTM. Teaching of the method should be supported by related literature such as [1, 3, 5, 9, 16]. According to our teaching experience in both UM and SUT, we note that many students will make the following two mistakes:

- The occurrence of problematic classifications and classes, such as irrelevant classifications, invalid classes, combinable classes, and composite classes. We have formally defined and discussed these problematic classifications and classes in [3].
- Construction of $\mathcal{T}$ based on control flow information. Figure 2 shows an (incorrect) $\mathcal{T}$ constructed for S_{TRADE} by some students using such an approach, resulting in the omission of some valid test cases [2].

To avoid repeating these mistakes, before the start of the project in step (3) below, we recommend discussing with students about the various types of problematic classification and class using the article [3], and elaborating 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.

(2) Develop a tutorial using various examples to reinforce the concept of the method and illustrate the mistakes in (1) above.

(3) Use a project to reinforce the lecture and tutorial contents through practice. In the project, we recommend using some business-oriented specifications as examples. Thus, the students can practice CTM with specifications typically found in the commercial sector.

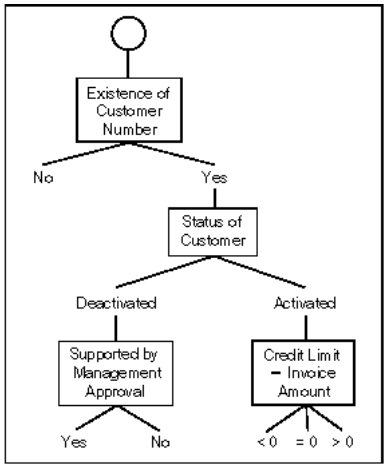

Figure 2: A Classification Tree for S_{TRADE} Based on Control Flow

(4) 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. 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. The phenomenon does happen in the students' projects in UM and SUT. We suggest to select some of these $\mathcal{T}$s constructed by the students and discuss their structural properties. Criteria should be provided to the students to help them determine which $\mathcal{T}$ is better. Examples of these criteria include the number of classifications and classes, the ease of understanding $\mathcal{T}$, and the effectiveness of T in the generation of valid test cases.

In the follow-up tutorial, we should also discuss the problem of applying CTM to increasingly complex specifications. In this case, the complex specification should be decomposed into smaller $\mathcal{U}$s that can be independently tested.

In addition, explanations should be given to students that software quality does not rely solely on testing, because testing is only one of many verification and validation techniques [15]. In many situations, life-cycle quality approaches, such as reviews and inspections, quality metrics, and software processes, should be used with testing. For example, inspections should be used as often as possible to prevent requirements 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 catch and remove these faults. Readers may note that we have studied applying CTM to requirements inspection [4]. Thus, students should also be taught with such an application.

4. Summary and Conclusion

In this paper, we have explained why CTM should be selected for teaching software testing, and how it should be taught. The recommended approach in Section 3 is based on our teaching experience of CTM in UM and SUT. Feedback from students in these two universities has confirmed that the teaching of CTM is viable and effective. We also observe that students found CTM easy to understand and use.

Intuitively, it would be better if students are provided with programs to test by using CTM. Hence, if possible, we suggest integrating the teaching of CTM with some other programming course. This approach would allow students to use CTM to find faults in programs written by other classmates.

Acknowledgment

This study is supported in part by a grant of the Research Grants Council of Hong Kong (Project No. PolyU5177/04E).

References

[1] A. Cain, T.Y. Chen, D. Grant, P.-L. Poon, S.-F. Tang, and T.H. Tse, "An automated test data generation system based on the integrated classification-tree methodology", in *Software Engineering Research and Applications* (*Lecture Notes in Computer Science*), vol. 3026, Berlin: Springer, 2004, pp. 225–238.

[2] T.Y. Chen and P.-L. Poon, "Experience with teaching black-box testing in a computer science/software engineering curriculum", *IEEE Transactions on Education*, vol. 47, no. 1, pp. 42–50, 2004.

[3] T.Y. Chen, P.-L. Poon, S.-F. Tang, and T.H. Tse, "On the identification of categories and choices for specification-based test case generation", *Information and Software Technology*, vol. 46, no. 13, pp. 887–898, 2004.

[4] T.Y. Chen, P.-L. Poon, S.-F. Tang, T.H. Tse, and Y.T. Yu, "Towards a problem-driven approach to perspective-based reading", in *Proceedings of the 7th IEEE International Symposium on High Assurance Systems Engineering* (*HASE 2002*), Tokyo, Japan, October 2002, pp. 221–229.

[5] T.Y. Chen, P.-L. Poon, and T.H. Tse, "An integrated classification-tree methodology for test case generation", *International Journal of Software Engineering and Knowledge Engineering*, vol. 10, no. 6, pp. 647–679, 2000.

[6] T.Y. Chen, P.-L. Poon, and T.H. Tse, "A choice relation framework for supporting category-partition test case generation", *IEEE Transactions on Software Engineering*, vol. 29, no. 7, pp. 577–593, 2003.

[7] L.A. Clarke, J. Hassell, and D.J. Richardson, "A closer look at domain testing", *IEEE Transactions on Software Engineering*, vol. SE-8, no. 4, pp. 380–390, 1982.

[8] L.M. Foreman and S.H. Zweben, "A study of the effectiveness of control and data flow testing strategies", *Journal of Systems and Software*, vol. 21, no. 3, pp. 215–228, 1993.

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

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

[11] D. Mullet, “The software crisis”, *Benchmarks Online*, vol. 2, no. 7, July 1999.

[12] T.J. Ostrand and M.J. Balcer, “The category-partition method for specifying and generating functional tests”, *Communications of the ACM*, vol. 31, no. 6, pp. 676–686, 1988.

[13] S. Rapps and E.J. Weyuker, “Selecting software test data using data flow information”, *IEEE Transactions on Software Engineering*, vol. SE-11, no. 4, pp. 367–375, 1985.

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

[15] T. Shepard, M. Lamb, and D. Kelly, “More testing should be taught”, *Communications of the ACM*, vol. 44, no. 6, pp. 103–108, 2001.

[16] H. Singh, M. Conrad, and S. Sadeghipour, “Test case design based on Z and the classification-tree method”, in *Proceedings of the 1st International Conference on Formal Engineering Methods*, Hiroshima, Japan, November 1997, pp. 81–90.

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

[18] L.J. White and E.I. Cohen, “A domain strategy for computer program testing”, *IEEE Transactions on Software Engineering*, vol. SE-6, no. 3, pp. 247–257, 1980.