HKU CS Tech Report TR-2006-14

To appear in *Information Systems Control Journal* 6 (2006)

Applying Testing to Requirements Inspection for Software Quality Assurance

T.Y. Chen, Pak-Lok Poon, Sau-Fun Tang, T.H. Tse, Yuen Tak Yu

Developing software systems involves a series of activities where there are many opportunities to make errors. Such errors may occur at an early stage of the development process where user requirements are incorrectly or incompletely specified, and also in subsequent stages when design and programming faults are introduced. Thus, software development should always be accompanied by quality assurance (QA) activities. Two common QA activities are *requirements inspection* and *software testing* (otherwise simply known as *testing*), which are often used in different phases of the software development life cycle (SDLC)[14].

Traditionally, requirements inspection is performed at an early stage of SDLC to reveal defects in a requirements specification (thereafter simply referred to as a specification). On the other hand, testing is commonly done at a later stage of SDLC to look for program faults after coding. Because their purposes are different, requirements inspection and testing are often treated as "separate" and "unrelated" tasks by software practitioners.

In recent years, many researchers have proposed to apply testing techniques to requirements inspection at an initial phase of the SDLC.[4, 15] The idea is that generating test cases from a specification may uncover requirements defects well before programming starts[5]. Thus, the possibility of inadvertently developing software based on an incorrect specification can be reduced. The benefits of such proposals are particularly prominent for large-scale projects[3] where the specifications are complicated and may easily contain many requirements defects, and the costs of repairing these defects at the late stages of the SDLC are typically tens or hundreds of times greater than if the defects are corrected early.[3, 6, 14]

To support this proposal, we shall describe a method of applying testing techniques to requirements inspection, with a view to improving the quality of the specification before software design commences.[15] But before the discussion, we first outline the basic concepts of requirements inspection and testing.

REQUIREMENTS INSPECTION

A requirements specification is an important document normally produced at the initial (problem definition) stage of SDLC. It defines the functionality, scope, and constraints of the software. It also serves as a basis for contracts as well as communication between the software developer and the user. The importance of the specification cannot be over-emphasized. If the development is based on an incomplete and incorrect specification, then even with well written code, the software will still be unsatisfactory and unable to fulfil user requirements. In addition, defects in a vague specification will be propagated to subsequent phases of the SDLC. At best, developers will eventually catch these defects, but at the expense of schedule delays and additional costs. At worst, the defects will remain undetected, resulting in the delivery of a faulty software system to users.

Michael E. Fagan of IBM is credited with introducing the use of inspections in software development.[1] Although many programmers use informal peer reviews of their code, Fagan made formal inspection an integral part of the development process, for locating defects in code or in other documentation such as requirements specifications and designs. Inspections are team activities; the basic idea is to formally inspect the item by one or more reviewers, typical in a meeting after individual preparations. Other members in the inspection team include the producer of the item to be inspected and a moderator who facilitates the inspection process.

Various forms of inspections are now adopted in the software industry. The cost-effectiveness of inspections in revealing defects has been extensively reported. For example, Doolan observed a 30-fold return on investment for every hour spent on inspecting specifications.[2] Russell also reported a similar return of 33 hours of software maintenance saved for every hour of inspection.[3]

A common practice to do software inspection is to use a checklist, which provides reviewers with a list of items to check. However, well written checklists are not generally available. In addition, most checklists do not help reviewers focus on particular aspects of a specification, implicitly treating all the information in the specification as equally important. As a result, reviewers are left with an ill-defined responsibility of detecting all defects in the entire specification.

To address this issue, Basili et al. have developed a technique known as *perspective-based reading* (*PBR*), which operates under the premise that different information in a specification has different levels of importance for different uses of the document.[4, 5] More specifically, PBR focuses on the point of view of the people who will make use of the specifications. One reviewer may read from the perspective of the software designer, another from the perspective of the software tester, and yet another from the perspective of the end user of the software. Each of these reviewers then produces a model that can be analyzed to answer questions based on the perspective. For example, Reviewer 1 reading from the designer's perspective would consider questions related to high-level design. Similarly, Reviewer 2 reading from the tester's perspective would consider questions arising from activities related to test case generation, and Reviewer 3 representing the end user would consider questions related to the completeness and correctness of the requirements with regard to system functionality. The premise is that these perspectives together will provide a more comprehensive coverage of the specification. As each reviewer is responsible for a relatively narrowly focused view of the specification, the reading should lead to more in-depth analysis of any potential defects in the specification. Empirical evidences are that the premise holds, as reported in experiments involving the inspection of NASA documents.[4]

SOFTWARE TESTING

Testing is commonly regarded as a predominant software QA activity. It is a major means to detect software faults and to prevent them from propagating through to the final production system, where the cost of defect removal is far greater. Unlike inspections, testing can evaluate how well a software system actually performs its function in its intended or simulated environment. Statistics have shown that the cost and time spent on testing in software development projects are significant.[6]

There are various forms of testing for discovering different types of faults and effects in software. For instance, testing may be done by the end-user to evaluate the usability or human-interaction issues of the software.[14] Also, performance testing is often done to assess the behaviour of software under special usage scenarios such as when the system is under heavy stress of data loading. Although the evaluation of “non-functional” quality of software is important, ultimately, the software must be functionally correct in order for it to serve its designated purpose and be accepted by the acquirer. [11] In the latter part of this article, we shall use a “functional” testing technique to illustrate how it can be applied to requirements inspection. However, the principle of applying testing to requirements inspection is rather general, and it can be properly adapted to the use of other systematic testing techniques.

Typically, testing consists of a series of tasks, namely (a) defining testing objectives, (b) designing and generating test cases, (c) executing the software with the generated test cases, and (d) analyzing the result by comparing the actual and the expected outputs. In particular, task (b) will significantly affect the scope of testing and, hence, the chance of detecting software faults.

Two main approaches for test case generation exist: white-box and black-box. The *white-box* approach generates test cases according to the information derived from the source code of the software under test. Examples are control flow testing[7] and data flow testing[8]. The *black-box* approach, on the other hand, generates test cases from information derived from the specification, without requiring the knowledge of the internal structure of the software. Examples are the choice relation framework[9] and the classification-tree method (CTM)[10, 11, 12, 13]. In general, the black-box approach is extensively used in the commercial software.[10, 12] Nevertheless, neither the white-box nor the black-box approach is sufficient; they complement each other.

The white-box and black-box approaches are general and applicable across all environments, architectures, and applications, but unique guidelines and approaches to testing are sometimes warranted, such as when testing graphical user interfaces (GUIs), client/server architectures, and real-time systems. Also, although our discussions below focus on the techniques for functional correctness testing, other testing techniques may in principle also be applied. Readers may refer to Pressman’s book for more details of techniques for other forms of testing, such as usability and performance testing.[14]

APPLYING TESTING TO REQUIREMENTS INSPECTION

The PBR technique from the tester's perspective is particularly useful and advantageous because (a) generating test cases as soon as the requirements specification is ready at the early stage of SDLC provides the software tester an opportunity to detect any *anomalies* ("potential" defects that could turn out to be "genuine" defects), and (b) the generated test cases can be used for testing at a later stage.[5]

Recall that test cases can be generated using a white-box or black-box approach. However, the white-box approach cannot be used for PBR because, at the time of requirements definition, the software system has not yet been developed. Among the black-box test case generation methods, CTM suits well for PBR under the tester's perspective, particularly for software which processes a large number of possible combinations of inputs.[15] CTM has been studied in depth in the literature[10, 11, 12, 13], to which the reader may refer for the details of this method. The following outlines the main steps of CTM:

Outline of the Classification-Tree Method (CTM):

(1) **Decomposing the specification.** Decompose the specification into functional units that can be tested independently. For each of these units, perform steps (2) to (5) below.

(2) **Identifying classifications and classes.** Identify classifications and their associated classes from the specification. A c*lassification* is defined as a "major property or characteristic of an input parameter or an environment condition of the software system that affects its execution behaviour". Each classification is partitioned into a set of *classes*, which represent "all the different kinds of values that are possible for the classification".[9, 10, 13]

(3) **Constructing a classification tree.** Construct a *classification tree* by assembling the classifications and classes in a hierarchical structure. The structure captures the "constraints" that govern whether two classes can be combined as part of some test frames.

(4) **Constructing a test case table.** Construct a *test case table* (also known as a *combination table*) from the classification tree.

(5) **Generating test cases.** Select valid combinations of classes from the test case table as *test frames* using some predefined selection rules (see the literature[10, 11, 13] for the details of these rules). Then, for each test frame, select an element from every class in it to form a test case.

Figures 1 and 2 illustrate the steps of CTM by examples. Note that, in Figures 2–4, the dotted lines serve to indicate that only part of the classification tree is shown due to space limitations. For example, in Figure 2, the dotted lines near the top indicates that there are other classifications (not shown in the figure) at the same level of classifications "Class of Ticket" and "Number of Vegetarian Meals" in a "complete" classification tree.

Suppose, in the specification, we have the following description:

> … An aircraft has three different types of cabin, corresponding to three classes of ticket: first, business, and economy. The business class cabin is further categorized into two subtypes depending on the location of the cabin (upper and lower decks). … The number of vegetarian meals is required for determining the number of this type of meals to be prepared and loaded onto the aircraft …

Examples of classifications and classes:

Three possible classifications are “Class of Ticket”, “Location of Seat”, and “Number of Vegetarian Meals”. The first classification has three associated classes: “First”, “Business”, and “Economy”. The second classification has two associated classes: “Upper Deck” and “Lower Deck”. The third classification has two associated classes: “= 0” and “≥ 0”.

An example of constraints between classes:

The classes “Upper Deck” and “Lower Deck” in the classification “Location of Seat” cannot be combined with the class “First” or “Economy” in the classification “Class of Ticket” to form part of any test frame.

Examples of a classification tree and a combination table:

See Figure 2. The upper part shows part of a classification tree. The grid at the bottom part is a partial test case table. Note the classification “Location of Seat” and its associated classes “Upper Deck” and “Lower Deck” occur under the class “Business” (of the classification “Class of Ticket”), but not the classes “First” and “Economy”, in the classification tree. This tree structure essentially captures the constraint that the classes “Upper Deck” and “Lower Deck” cannot be combined with the class “First” or “Economy” to form part of any test frame.

Examples of a test frame and a test case:

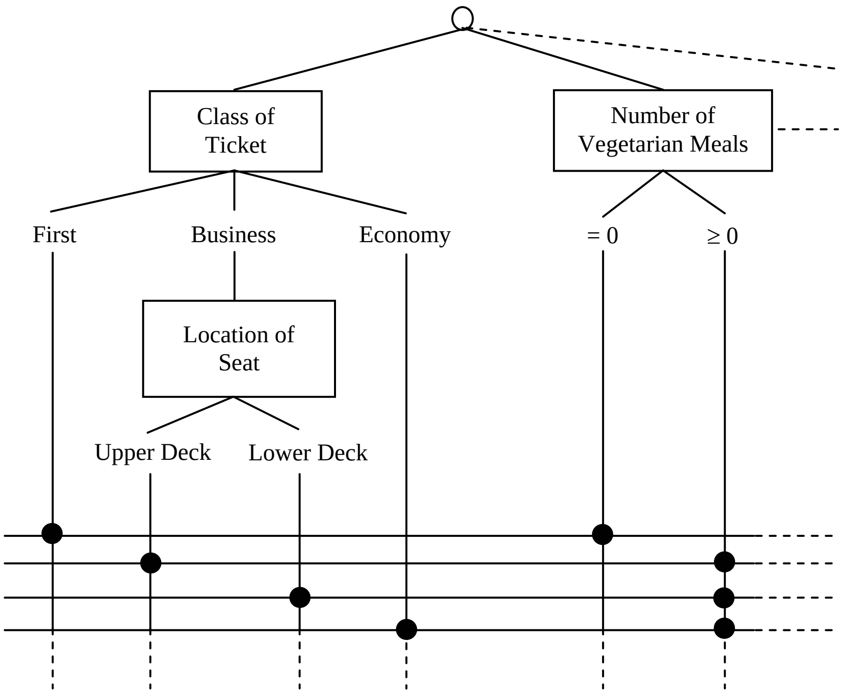
See Figure 2. The second row of the test case table corresponds to the following test frame (which is a set of classes):

{ Business (Class of Ticket), Upper Deck (Location of Seat), ≥ 0 (Number of Vegetarian Meals), … }

We then randomly select an element from every class in the above test frame to form a test case. An example of a test case is:

Class of Ticket = Business, Location of Seat = Upper Deck, Number of Vegetarian Meals = 142, …

Figure 1. Examples of CTM



The grid below the classification tree is the test case table. Consider the second row of the test case table. The dots indicate that the classes "Business", "Upper Deck", and "≥ 0", and some other classes not shown in the figure, are combined together to form a test frame, from which a corresponding test case can be selected.

Figure 2. A Partial Classification Tree and Test Case Table

Our method, PBR_{CTM}, applies CTM to PBR under the tester's perspective.[15] In PBR_{CTM}, the reviewer attempts to generate test cases using CTM, and while doing this, answers a list of questions specifically tailored to each step of CTM. The rationale is that, if the reviewer cannot answer the questions with respect to a particular requirement in the specification, then there is an anomaly associated with that requirement, which may be due to some defect. In this way, the reviewer can identify the defects, if any, which are to be corrected by the author after the inspection so that the requirements will better support the later phases of the SDLC.

The steps of PBR_{CTM} are as follows:

Steps of PBR_{CTM}:

Decompose the requirements specification into several "requirements units", each of which can be processed independently for the purpose of testing. For each requirements unit, perform the following tasks for test case generation. During the test case generation process, answer the questions below and record any anomalies thus identified in the requirements unit.

Phase I: Preliminary Checking of Requirements Unit

Answer the following questions:

Q.I.1. Does the requirements unit make sense from what you know about the problem domain?

Q.I.2. Does the requirements unit provide sufficient information about the execution behaviour of the software?

Q.I.3. Based on the requirements stated in the specification and your knowledge of the problem domain, have any relevant and important aspects been missing from the requirements unit?

Phase II: Test Case Generation

(1) **Identifying classifications and classes.** For every requirements unit, identify its associated classifications and classes from the important aspects that affect the software's execution behaviour.

(2) **Constructing a classification tree.** From the information given in the requirements unit, determine the constraints among the identified classifications and classes. In accordance with these constraints, assemble the classifications and classes into a hierarchical structure, which is known as a classification tree.

(3) **Constructing a test case table.** Construct the corresponding test case table from the classification tree to facilitate the generation of test cases in the next step.

(4) **Generating test cases.** Select valid combinations of classes from the test case table as test frames. From each test frame, select one element from each of its classes to form a test case.

In steps (1) – (4) above, answer the following questions:

Q.II.1. Do you have sufficient information to identify classifications and their associated classes?

Q.II.2. Based on the information given in the requirements unit, have classes been identified so that no element in the input domain may appear in more than one class within the same classification?

Q.II.3. When assembling the classifications and classes to form a classification tree, does the requirements unit provide sufficient information for determining the constraints among the classifications and classes?

Q.II.4. For every test frame, do you have sufficient information to select an element from each of its classes to form part of a test case?

Q.II.5. Are there other interpretations of the requirements that the software developer may make on the basis of the description given? If so, will this affect the test cases you generate?

Phase III: Test Plan Preparation

For each generated test case, record the corresponding expected behavior of the software. In other words, how do you expect the software to respond to this test case you have just generated? In this phase, the reviewer should ask the key question of whether the resulting behavior could be specified appropriately and without ambiguity.

Note that PBR_{CTM} may be fine-tuned to fit in a particular application domain according to: (a) the specific environment and context, and (b) the level of domain knowledge of the reviewer who applies it. Consider, for example, question Q.I.2 in Phase I of PBR_{CTM}. Suppose the problem domain is related to the airline catering industry, and the reviewer knows that the type of aircraft (for example, Boeing 747 and Airbus 340) is a relevant and important aspect affecting the system's execution behaviour. In this case, question Q.I.2 should be elaborated in more details by explicitly reminding the reviewer to check for the information of the aircraft type in the specification.

EXPERIENCE IN APPLYING PBR$_{CTM}$

We applied PBR_{CTM} to a real-life airline catering specification. This specification was produced for a meal ordering system (MOS) in an international company providing catering services for several airlines. The catering company prefers to remain anonymous and in this paper it is referred to as AIR-FOOD. The main function of MOS is to help AIR-FOOD determine the number and types of meals to be prepared and loaded onto the aircraft served by AIR-FOOD. To facilitate our discussions, the specification is referred to as S_{MOS}.

Two participants, whom we call X and Y respectively, worked on the project. Both of them have several years of experience working in the airline industry. Furthermore, Participant X has some practical experience using CTM for software testing, whereas Participant Y has been involved in MOS development and implementation. We asked Participant X to use PBR_{CTM} to inspect S_{MOS} and record all the potential defects, which were then verified by Participant Y to determine whether these potential defects were genuine.

At the beginning of our project, Participant X first decomposed S_{MOS} into several requirements units to be inspected or tested independently. The rest of this paper focuses on one of these requirements units, denoted by U_{meal}, related to the generation of daily meal schedules.

Phase I: Preliminary Checking of Requirements Unit

Participant X performs a preliminary check of U_{meal}. The objective is to determine whether U_{meal} makes sense in general, and whether there is any relevant and important aspect (related to the daily meal schedule generation) omitted from U_{meal}. Participant X did not detect any anomaly in this phase. The result does not surprise us because MOS has been released for production use in AIR-FOOD for several years and, hence, any obvious and trivial omission from U_{meal} should have been detected and corrected before we used PBR_{CTM}.

Phase II: Test Case Generation

(1) Identifying classifications and classes

While performing this task, Participant X detected 13 potential defects, which were subsequently verified by Participant Y to be genuine defects. Below we use an example to show how these defects are detected. Consider the following description in U_{meal}:

> … Airlines can change aircraft type, ETD [Estimated Time of Departure] and/or flight sector of a flight on a particular date even after the master flight schedule [MFS] is announced. The system keeps an exceptional flight schedule [EFS] in addition to the Master [MFS]. This exceptional flight schedule has higher priority over the Master [MFS] when creating [the] daily meal schedule …

From the above paragraph, Participant X identified the classification "Number of EFS" to indicate the number of EFSs that an MFS has. It is obvious that "= 0" is a possible class for this classification, which represents the case when the MFS is not associated with any EFS. When Participant X tried to identify other classes for the classification, he found that U_{meal} did not contain sufficient information for determining which of the following alternatives should apply:

(a) Identify "= 1" as the only extra class, or
(b) Identify two more extra classes: "= 1" and "≥ 2".

The above confusion occurs because U_{meal} does not state the maximum number of EFSs to be associated with an MFS whatsoever. If it had been stated clearly in U_{meal} that each MFS can have at most one EFS, then we would know that alternative (a) should apply. On the other hand, if U_{meal} had stated clearly that each MFS can have more than one EFS, then alternative (b) should apply. In the latter case, Participant X further observed that U_{meal} does not provide information about how MOS should respond to these EFSs. For example, it was not clear whether MOS would "only" consider the "last" EFS and ignore the remaining EFSs when generating the daily meal schedule, or the system should consider "all" the EFSs in the generation process.

(2) Constructing a classification tree

Here, Participant X identified the constraints among the classifications and classes based on U_{meal} in order to construct a classification tree. In addition to the 13 defects detected in step (1), four more defects were detected in this step and confirmed by Participant Y to be genuine. Below we use an example to illustrate how such defects were detected. Consider the following description in U_{meal}:

> … Aircraft configuration, such as the number of crewmembers, is retrieved from the Menu Planning System (MPS). However, this information can be overridden for flight schedules with additional crewmembers. The system keeps an exceptional crew configuration [ECC] table for this purpose. Exceptional crew configuration has higher priority over aircraft configuration in MPS when creating [the] daily meal schedule … An exceptional crew configuration record includes the airline, flight number, number of crewmembers …

From his domain knowledge, Participant X knew that the crew in a flight normally consists of the captain, at most one first officer, at most one second officer, the chief purser, several senior pursers, and several pursers. The actual composition of the crew varies according to the type of aircraft. After reading the above description and other parts of U_{meal}, Participant X was not sure whether in the ECC: (a) only the total number of crewmembers on the flight was specified, or (b) different numbers of crewmembers were specified for different types of crew. If scenario (a) is true, then the partial classification tree in Figure 3 may be constructed. Alternatively, if scenario (b) applies, then the partial classification tree in Figure 4 may be constructed instead. Note that the detection of this defect was mainly the result of the explicit consideration of constraints among classifications and classes during the construction of a classification tree for U_{meal}.

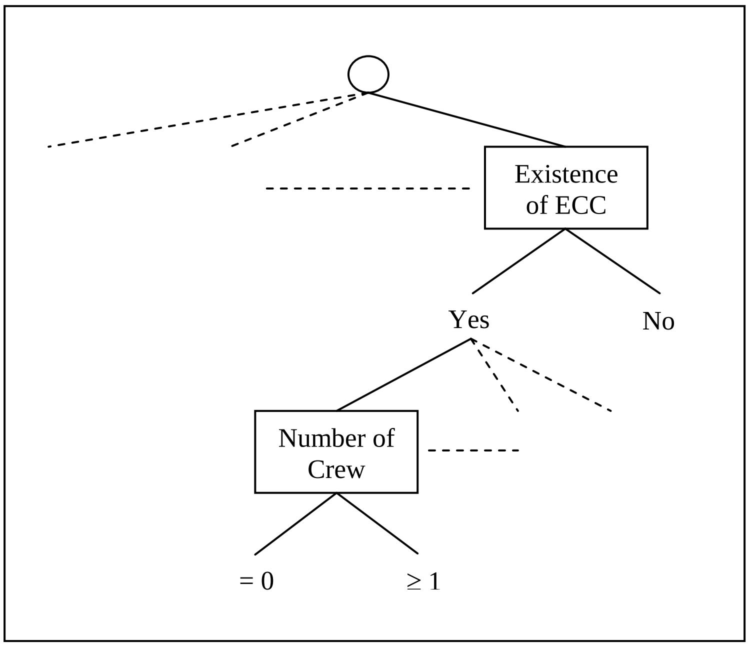


Figure 3. A Partial Classification Tree for U_{meal}

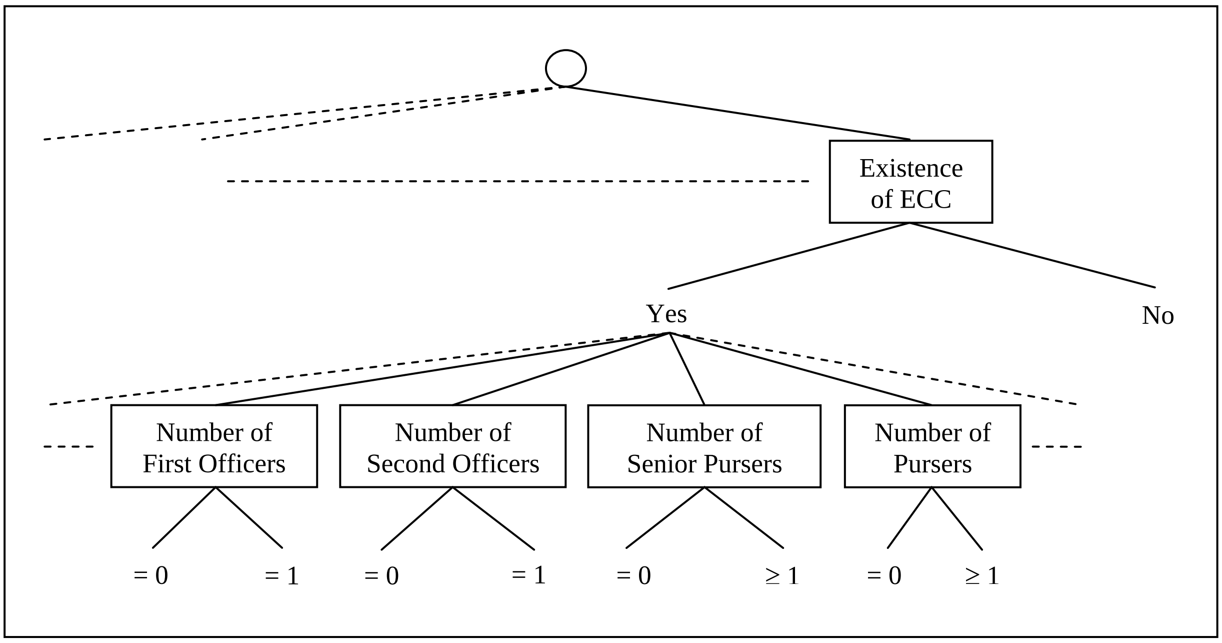


Figure 4. Another Partial Classification Tree for U_{meal}

Until the above defects were corrected, Participant X had been unable to construct a definite classification tree for U_{meal}. After correcting the defects, a classification tree and its corresponding test case table (see step (3) of Phase II in PBR_{CTM}) were constructed, from which a set of test cases was eventually generated (see step (4) of Phase II in PBR_{CTM}). We shall omit Phase III and steps (3) and (4) of Phase II because they were relatively straightforward and, in our case, no additional defects were detected in these steps/phases.

OTHER TESTING ASPECTS IN REQUIREMENTS INSPECTION

We mentioned earlier that testing should cover the functional and non-functional aspects of software. So far, we have used CTM as an example to illustrate how to perform requirements inspection (more specifically, PBR from the tester's perspective) related to the functional aspect of software. We, however, emphasize that our proposal of applying testing to requirements inspection is also applicable to the non-functional aspect of software, such as performance, security, reliability, and recovery.[16]

Take performance testing as an example. Many software systems have specific performance or efficiency requirements, normally stated in terms of system's response times and throughput rates under certain workload and configuration conditions. A major purpose of performance testing is to assess how the system runs under peak and continuous loads.

Obviously, an important prerequisite for performance testing is to include the relevant software performance or efficiency requirements in the specification in a precise manner. Otherwise, the software tester will have no basis to conclude whether the actual software performance meets user's expectation. Hence, software testers will naturally ask the following questions when inspecting the specification from the perspective of system performance:

- Are all expected processing times specified?
- Are all data transfer rates specified?
- Is the required level of system performance clearly specified for all different usage scenarios?
- Are all system throughput rates specified?
- …

Now, consider reliability testing, which is another type of non-functional testing. In today's competitive environment, every software system is expected to be reliable. In particular, ultra-high reliability is expected for "safety critical systems" such as those used in nuclear plants, weapon systems, aviation equipment, and medical devices. When software testers inspect the specification from the perspective of reliability testing, they will naturally ask the following questions:

- Is the expected bound for the mean time between failures (MTBF) or a similar metric specified?
- Are the consequences of software failure specified for each requirement?
- How will the system be expected to behave under exceptional or adverse input conditions?
- Does the specification contain enough information to determine how the reliability testing should be performed?
- Is a strategy for error detection specified?
- Is a strategy for error correction specified?
- …

In short, when software testers inspect the specification from a specific testing perspective, such as functional testing, performance testing, and reliability testing, this inspection approach will automatically provide some clues to the testers as to which types of information in the specification should be focused during the review process. In this way, potential defects in the specification will have a higher chance of being spotted.

SUMMARY

The quality of a requirements specification is of vital importance because it critically affects the quality of the resultant software system. Thus, the specification should be inspected at an early stage of the SDLC before it is used for software design so that the development work will not be based on an incorrect specification. The popular checklist approach to requirements inspection, however, poses difficulties to reviewers. This is because well written checklists are not generally available, and reviewers are left with an ill-defined responsibility of detecting all defects in the whole specification.

To support the task of early requirements inspection and to address the difficulties associated with the checklist inspection approach, in this paper, we recommend combining test case generation (an activity normally performed at a later stage of SDLC) with requirements inspection, and that it should be done as early as possible. To illustrate our recommendation, we have developed a technique, PBR_{CTM}, which incorporates PBR with CTM. The technique not only helps the reviewer detect requirements defects effectively, but also generates a set of test cases which can be used for testing the software later.

We have evaluated the technique PBR_{CTM} by means of a case study using a real-life catering specification. This specification contains numerous defects which were unknown to us before our study commences. Results show that PBR_{CTM} helps reviewers find genuine defects in the specification, and thus confirms the practicality and effectiveness of the technique. We have also demonstrated, using performance testing and reliability testing as examples, that PBR is also applicable to non-functional testing.

In summary, this paper conveys two important messages to the software community:
(a) requirements inspection is important and has to be done as earlier as possible in the SDLC, and
(b) requirements inspection is more effective when performed with the support of a systematic approach, and in this regard, testing techniques can be applied (such as PBR_{CTM}) to improve the defect-finding capabilities of inspections.

ACKNOWLEDGEMENT

The work described in this paper was partially supported by grants from the Research Grants Council of the Hong Kong Special Administrative Region, China (Project Nos. PolyU 5177/04E and CityU 1195/03E). A preliminary version of this article has earlier been presented at an academic conference.[15]

REFERENCES

[1] Fagan, M.E. “Advances in Software Inspections.” *IEEE Transactions on Software Engineering*. Vol. SE-12, no. 7, 1986, pp. 744–751.

[2] Doolan, E.P. “Experience with Fagan’s Inspection Method.” *Software: Practice and Experience*. Vol. 22, no. 2, 1992, pp. 173–182.

[3] Russell, G.W. “Experience with Inspection in Ultralarge-Scale Development.” *IEEE Software*. Vol. 8, no. 1, 1991, pp. 25–31.

[4] Basili, V.R., Green, S., and Laitenberger, O., Lanubile, F., Shull, F., Sørumgard, S., and Zelkowitz, M.V. “The Empirical Investigation of Perspective-Based Reading.” *Empirical Software Engineering: An International Journal*. Vol. 1, no. 2, 1996, pp. 133–164.

[5] Shull, F., Rus, I., and Basili, V. “How Perspective-Based Reading Can Improve Requirements Inspections.” *IEEE Computer*. Vol. 33, no. 7, 2000, pp. 73–79.

[6] Boehm, B.W. “The High Cost of Software.” In *Tutorial: Software Testing and Validation Techniques* (E. Miller and W.E. Howden, Eds.), IEEE Computer Society Press, NY, 1981, pp. 377–388.

[7] Woodward, M.R., Hedley, D., and Hennell, M.A. “Experience with Path Analysis and Testing of Programs.” *IEEE Transactions on Software Engineering*. Vol. SE-6, no. 2, 1980, pp. 278–286.

[8] Clarke, L.A., Podgurski, A., Pichardson, D.J., and Zeil, S.J. “A Formal Evaluation of Data Flow Path Selection Criteria.” *IEEE Transactions on Software Engineering*. Vol. 15, no. 11, 1989, pp. 1318–1332.

[9] Chen, T.Y., Poon, P.-L., and Tse, T.H. “A Choice Relation Framework for Supporting Category-Partition Test Case Generation.” *IEEE Transactions on Software Engineering*. Vol. 29, no. 7, 2003, pp. 577–593.

[10] Grochtmann, M. and Grimm, K. "Classification Trees for Partition Testing." *Software Testing, Verification and Reliability*. Vol. 3, no. 2, 1993, pp. 63–82.

[11] Chen, T.Y., Poon, P.L., and Tang, S.F. "A Systematic Method for Auditing User Acceptance Tests." *IS Audit and Control Journal*. Vol. 5, 1998, pp. 31-36.

[12] Yu, Y.T., Tang, S.-F., Poon, P.-L., and Chen, T.Y. "Improving the Cost-Effectiveness of a Test Suite for User Acceptance Tests." *Information Systems Control Journal*. Vol. 6, 2000, pp. 32–37.

[13] Chen, T.Y., Poon, P.L., and Tse, T.H. "An Integrated Classification-Tree Methodology for Test Case Generation." *International Journal of Software Engineering and Knowledge Engineering*. Vol. 10, no. 6, 2000, pp. 647–679.

[14] Pressman, R.S. *Software Engineering: A Practitioner's Approach* (*6th ed.*), McGraw-Hill, NY, 2005.

[15] Chen, T.Y., Poon, P.-L., Tang, S.-F., Tse, T.H., 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'02*), IEEE Computer Society Press, Los Alamitos, CA, 2002, pp. 221–229.

[16] Myers, G.J. *The Art of Sofware Testing* (*2nd ed.*), Wiley, NJ, 2004.

T. Y. Chen, PhD,
is a Chair Professor of Software Engineering in the Faculty of Information and Communication Technologies, Swinburne University of Technology, Australia. He has taught at The University of Hong Kong and The University of Melbourne for many years. His research interests include software quality, software testing, and software engineering. He is a member of the editorial board of Software Testing, Verification and Reliability. He was a co-recipient of the Michael Cangemi Best Book/Article Award from the Information Systems Audit and Control Association in 2001. His email address is tchen@ict.swin.edu.au.

Pak-Lok Poon, PhD, CISA, CSQA, MIEEE, MACM,
is an associate professor (information systems) of the School of Accounting and Finance at The Hong Kong Polytechnic University. His research interests include software testing, requirements inspection, computer audit and control, electronic commerce, and computers in education. He is on the editorial committee of the Information Systems Control Journal. He was a co-recipient of the Michael Cangemi Best Book/Article Award from the Information Systems Audit and Control Association in 2001. His email address is afplpoon@inet.polyu.edu.hk.

Sau-Fun Tang, MBus(IT), MIEEE, MACS,
is currently a PhD candidate in the Faculty of Information and Communication Technologies, Swinburne University of Technology, Australia. She has taught at The University of Hong Kong, The Hong Kong Polytechnic University, and Hong Kong Baptist University. Her main research interests are on software engineering, software testing, management information systems, electronic commerce, business process reengineering, and computers in education. She has more than seven years of commercial experience in software installation, maintenance, and support. She was a co-recipient of the Michael Cangemi Best Book/Article Award from the Information Systems Audit and Control Association in 2001. Her email address is satang@ict.swin.edu.au.

T.H. Tse, MBE, PhD, FBCS, FIMIS, FIMA, FHKIE,
is a professor at the Department of Computer Science of The University of Hong Kong. He was a visiting fellow at the University of Oxford. He is an editor of the Journal of Systems and Software and the steering committee chair of the International Conference on Quality Software. He is a fellow of the British Computer Society, a fellow of the Institute for the Management of Information Systems, a fellow of the Institute of Mathematics and its Applications, and a fellow of the Hong Kong Institute of Engineers. He was decorated with an MBE by The Queen. His email address is thtse@cs.hku.hk.

Yuen Tak Yu, PhD, MIEEE, MACM,
is an associate professor at the Department of Computer Science, City University of Hong Kong. His research interests include software engineering, software testing, software quality, e-commerce, and computers in education. He is on the editorial board of the International Journal of Web Engineering and Technology. He was a co-recipient of the Michael Cangemi Best Book/Article Award from the Information Systems Audit and Control Association in 2001. His email address is csytyu@cityu.edu.hk.