Asian Journal of Information Technology 6 (2): 258-262, 2007

On the Problems and Solutions of Static Analysis for Software Testing

[1]N. Srinivasan and [2]P. Thambidurai
[1]Department of Computer Applications, Sathyabama University, Chennai, India
[2]Department of Computer Science and Engineering and Information Technology,
Pondicherry Engineering College, Pondicherry, India

Abstract: Software testing is any activity aimed at evaluating an attribute or capability of a program or system and determining that it meets its required results. Although crucial to software quality and widely deployed by programmers and testers, software testing still remains an art, due to limited understanding of the principles of software. The difficulty in software testing stems from the complexity of software: we can not completely test a program with moderate complexity. Testing is more than just debugging. The purpose of testing can be quality assurance, verification and validation, or reliability estimation. Testing can be used as a generic metric as well. Correctness testing and reliability testing are two major areas of testing. Software testing is a trade-off between budget, time and quality. In our approach, we focus the problems and solutions and the static analysis of those problems and solutions for software testing.

Key words: Software life cycle, software design, software performance, software maintanance, Dynamic Data Exchange (DDE), Object Linking Embedding (OLE), Commercial-Off-The-Shelf(COTS)

INTRODUCTION

Software testing is an essential phase in the modern software life cycle. It is the process of revealing software defects and evaluating software quality by executing the software (Ben and Marliss, 1997; Burnstein, 2003; Galin, 2004; Gao *et al*., 2003; Ginac, 1998; Horch, 2003; Myers, 1979). A well-designed test case may reveal previously undetected software defects. Software testing, defects repair and software reliability are closely related to one another. Thorough software testing can ensure the software quality by reexamining the requirements analysis, design and coding after the software has been created. A good process in software development uses top-down techniques. In the software design phase, people analyze and define the problem domain. Then they perform an analysis of the software requirements to build the data domain functions, quality requirements constraints and validation standards. In the software development phase, they turn the concept of software design into source code using suitable programming language(s). For software developers, software testing is the inverse process of software development in some sense. Prior to the software testing phase in the software life cycle, people usually construct the real software from abstract concepts. While at the software testing phase, people usually want to design a set of representative test cases to deconstruct the developed software by detecting the flaws injected during the various software development phases. Some basic testing principles are listed as follows:

Present the expected testing results when designing test cases. A design case should have two parts, i.e., the precise descriptions of both input data and their correct consequences. A good test case should have a higher chance to reveal the hidden defects (Fig. 1).

Separate the software testing team from the software development team, since the philosophies of the two teams are different. The former is intentionally destructive while the latter is constructive. Therefore, software testing should be performed by the trained testers who are not in the software development group.

Design invalid test cases. A program should be capable of running properly in different operation situations. For instance, it should work well in the presence of invalid inputs that are intentional or unplanned. The program should be able to reject the invalid inputs and give out the error information on possible reasons, together with corresponding counteractive measures.

Perform regression testing each time the software-under-test is revised, as new defects may be brought up by the software modification. In regression testing, the tester may find the newly incurred software defects using previous test cases.

Corresponding Author: N. Srinivasan, Department of Computer Applications, Sathyabama University, Chennai, India

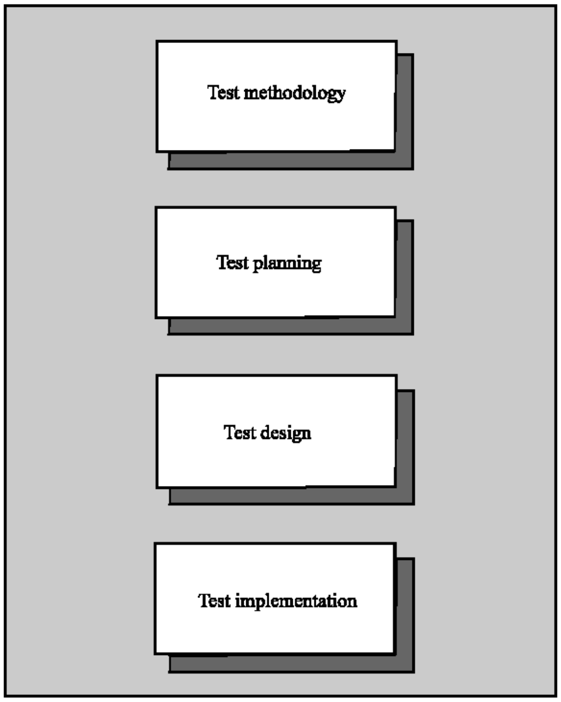

Fig.1: Software testing stages

The tester should concentrate on the error-prone program segments. It has been demonstrated that the more defects you reveal in a program segment, the more chances you can find other software defects in this segment. Generally speaking, the existence of additional defects in a special chunk of software code is proportional to the detected number of software defects in that segment.

RELATED WORK

Software performance: Software systems are becoming increasingly complex. In the arena of real-time measurement and control, the software may be distributed, embedded and highly responsive. The software is usually made up of a large amount of in-house developed components, Commercial-Off-The-Shelf (COTS) components and newly developed components. This trend makes the integrated software rather complicated and more prone to be out of service. As a result, the process of verification and validation for such software-intensive systems requires a larger number of test cases and more meticulous testing than conventional automation software systems.

Table 1: Typical software quality factors insoftware testing

Functionality	Engineering	Adaptability
(exterior quality)	(interior quality)	(future quality)
Correctness	Efficiency	Flexibility
Reliability	Testability	Reusability
Usability	Documentation	Maintainability
Integrity	Structure	

Embedded systems are involved in almost every facet of modern life and they are playing an ever-increasing role in the monitoring and control of potentially dangerous industrial applications (Douglass, 2000; 2003). It illustrates the basic structure of a real-time monitoring and control system. In a basic embedded measurement and control loop, a sensor measures the monitored variables, a microprocessor-based controller determines how the error between the actual and target measurements could be corrected and an actuator executes the command to drive the controlled variables close to the target values. Such operations are repetitious when the system runs. In this basic control loop, there are at least three types of faults that may occur during the system operations.

One commonly encountered fault is component malfunction, such as sensor or actuator faults. Also in the embedded measurement and control system, the limited system resources such as CPU, memory and bandwidth should be properly allocated for each task. Otherwise, sampling jitter and control delay may occur. Furthermore, control delay and packet loss during data transmission should also be taken into account in networked and embedded control system designs. As a result, the testing regarding software availability, reliability, survivability, flexibility, durability, security, reusability and maintainability is essential for the safety-critical, real-time automation system (Table 1). There are several factors that make testing of distributed and embedded real-time software difficult. The first is complexity. The large number of potential test paths overwhelms software testing even for a small network, let alone the testing for large-scale distributed systems. For such software testing, only a small number of paths can be examined. Therefore, the thoroughness of software testing cannot be ensured. Second, the real-time constraints intensify the software testing, as the software-under-test often demands a complex test environment to accurately evaluate the software performance in different implementation scenarios.

Furthermore, in object-oriented software, defects caused by encapsulation, inheritance and polymorphism must be carefully detected (Ambler, 2004).

Software maintenance: There are usually four phases experienced by the released software: enhancement, maturity, obsolescence and termination (Norris and Rigby, 1992). The distinction between any adjacent phases is not strict and could be rather blurred in the phase transition. Given that software systems often need to be changed to accommodate changing environments, it is important to establish a safe and well-controlled mechanism for modification and update. In practice, software maintenance often consumes the most time in the software life cycle.

It explains that software maintenance continues throughout the software life cycle. The cost of software maintenance can occupy 40-70% of the total software expenditure (Norris and Rigby, 1992). Software maintenance has two main tasks: identify the unexposed defects after the software has been installed on the customer site and adapt to various operating conditions and ever-changing user requests.

It can be regarded as the iteration of software development and testing whenever any new defects are found or certain part of the software needs to be updated to fulfill the new requirements. There are four types of software maintenance (Norris and Rigby, 1992).

Corrective maintenance: After installation at the user sites, the latent software defects appear and therefore revision is needed to ensure the proper running of the software. This is of critical importance for software quality assurance and can be viewed as a type of software testing.

Adaptive maintenance: It ascertains that the released software can adapt to new requirements, which were not in the previous design specification. Both changing user requirements and operating platform make the adaptive maintenance necessary.

Perfective maintenance: New technologies need to be incorporated into the existing software to improve its performance. In the software development phases, it is possible that the desired technology has not been available, or the technology employed is not the best for the application. In such cases, end users may often want the software to be upgraded using novel technology. For example, in the early industrial automation software, data exchange among different applications was realized by the traditional clipboard.

Later, the occurrence of Dynamic Data Exchange (DDE) technology made the data exchange more powerful for industrial automation software. More recently, the concept of Object Linking and Embedding (OLE) automation made data communication among different applications in a software system easier and more flexible. Hence, each time when a newer or more suitable technology is available, the software may need to be modified to incorporate any new developments to meet the often changing and tougher requirements.

Preventive maintenance: It involves making changes to the software that, in themselves, improve neither correctness nor performance but make future maintenance activities easier to be carried out.

PROBLEMS

The confused test team: A project manager fears that his team will not complete the test activities by an aggressive deadline. You make a house call and as you interview the test team, you find that the members have serious doubts about whether the chosen methods are appropriate and, in fact, whether they work at all. Your first feeling is that the team is upset over things other than work and is not focusing on how to perform its tasks.

The project is on an aggressive delivery schedule and, therefore, many of the available trained engineering resources are working 24 h a day on the design team. In actuality, a lack of technical leadership leaves the test team flabbergasted and unable to complete the test work on time.

The test-maintenance failure: After 16 months of creating test specifications from a requirements specification, the requirements Organization publishes a new version of the requirements specification. At this point, only very limited traceability exists from the written tests to the corresponding requirements specification. Consequently, locating the tests that you need to update according to the new requirements specifications requires another several months.

Manual testing: A test team is spending most of its time running test cases but is executing the tests slowly. It takes as much as a day just to test one new feature of a system and often the tests fail due to system timeouts.

Executing full regression tests has been so expensive that the team avoids doing so whenever possible. Needless to say, the test execution is manual.

The uncertainty principle: Uncertainty introduced by the testing method is virtually unavoidable. The following C code example is a classic, although oversimplified, illustration of the problem:

```
if (x != 0)
y = x;
else
assert (0);
x+=2;
```

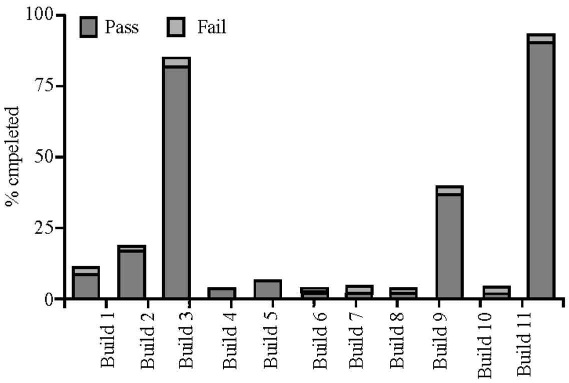

Fig.2: Percentage of tests complete by build, using traditional regression testing methods

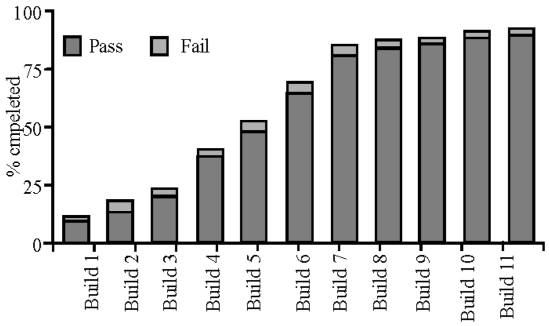

Fig.3: Cumulative percentage of tests completes using traditional regression testing methods

The above code behaves differently if compiled in debug mode than it behaves if you compile it in release mode. Because it is in C, the "assert" macro expands to nothing and the x+=2 statement takes the place of the assertion after the "else" statement. Other, often more difficult, examples include optimizers being too aggressive when test code in a system otherwise affects its size, speed, or behavior and when system or component simulators produce incorrect results. A problem that occurs during testing is not repeatable when running a non-test session and a feature that worked just fine during testing fails in real life.

Selecting the right tests: You work with a group that tests software for maintaining a communications network. The software you are testing comprises statistics-gathering nodes, an analysis module and a user interface for connecting to other components. The system is distributed and the user interface runs on a PC. The test group has started constructing test cases using an automated tool to test the system through the user interface.

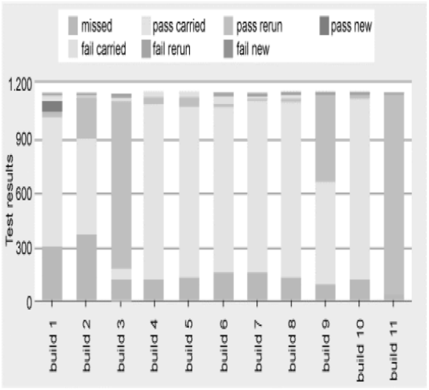

Fig. 4: CTA cumulative test results using traditional regression test methods

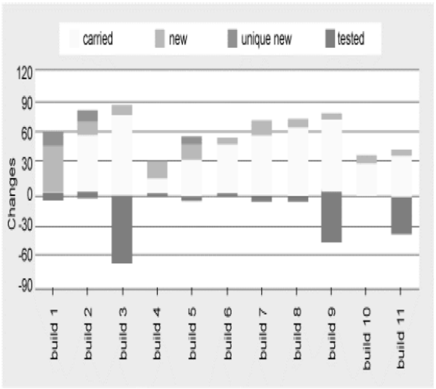

Fig. 5: Changes versus testing completed by build

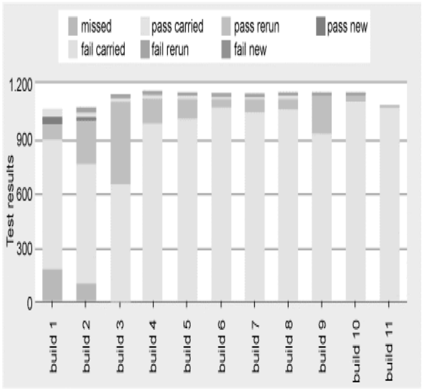

Fig. 6: CTA cumulative test results using targeted test methods

RESULTS AND DISCUSSION

Traditional Vs targeted testing: Figure 2-7 Testing activities can fail in many ways, however, you can prevent most problems with the following practices: form a well-

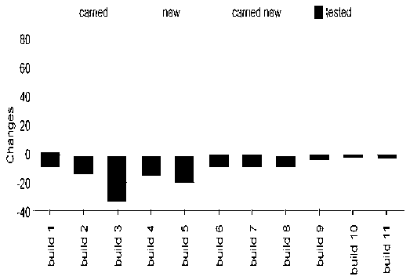

Fig.7: Changes versus testing completed by build using targeted test methods

qualified test team with the appropriate means for performing the tests at hand, make testing an integral part of software development, employ change-management processes, ensure requirement traceability to and from tests, automate test specification and execution and design for testability.

Given the complexity of current and anticipated software and communications systems, you should expect software testing to become even more complicated.

Consequently, even more potent tools and methodologies will emerge over time. Manual testing is becoming a less viable alternative and integration with the overall design processes and tools will prove necessary to keep pace in testing these complex current and future systems.

CONCLUSION

Software testing is an art. Most of the testing methods and practices are not very different from 20 years ago. It is nowhere near maturity, although there are many tools and techniques available to use. Good testing also requires a tester's creativity, experience and intuition, together with proper techniques.

Testing is more than just debugging. Testing is not only used to locate defects and correct them. It is also used in validation, verification process and reliability measurement.

Testing is expensive. Automation is a good way to cut down cost and time. Testing efficiency and effectiveness is the criteria for coverage-based testing techniques.

Complete testing is infeasible. Complexity is the root of the problem. At some point, software testing has to be stopped and product has to be shipped. The stopping time can be decided by the trade-off of time and budget. Or if the reliability estimate of the software product meets requirement.

Testing may not be the most effective method to improve software quality. Alternative methods, such as inspection and clean-room engineering, may be even better.

REFERENCES

Ambler, S.W., 2004. The Object Primer: Agile Model-Driven Development with UML 2.0. Cambridge, UK: Cambridge University Press.

Ben-Menachem, M. and G.S. Marliss, 1997. Software Quality: Producing Practical, Consistent Software, Slaying the Software Dragon Series. Boston, MA: International Thomson Computer Press.

Burnstein, I., 2003. Practical Software Testing: A Process-Oriented Approach, New York: Springer.

Douglass, B.P., 2000. Real-Time UML: Developing Efficient Objects for Embedded Systems, (2nd Edn.), Reading, MA: Addison-Wesley.

Douglass, C., 2003. Safety-critical software certification: Open source operating systems less suitable than proprietary? COTS. J., pp: 54-59.

Galin, D., 2004. Software Quality Assurance: From Theory to Implementation, Reading, MA: Pearson/Addison Wesley.

Gao, J.Z., H.S.J. Tsao and Y. Wu, 2003. Testing and Quality Assurance For Component-Based Software. Norwood, MA: Artech House.

Ginac, F.P., 1998. Customer Oriented Software Quality Assurance. Upper Saddle River, NJ: Prentice Hall.

Horch, J.W., 2003. Practical Guide to Software Quality Management, (2nd Edn.), Norwood, MA: Artech House.

Myers, G., 1979. The Art of Software Testing, New York: Wiley.

Norris, M. and P. Rigby, 1992. Software Engineering Explained. Chichester, England: Wiley.