Preface

The object-oriented (OO) paradigm adopts a more natural view of the application domain as consisting of objects, their behaviors, and their interactions. It provides a better understanding of requirements in terms of identifying and specifying the objects, their behaviors, the services provided by objects, object interactions, and the constraints. Furthermore, the seamless transition into OO design and programming facilitates design verification, code review, and maintenance. It is widely accepted that the OO paradigm will significantly increase software reusability, extendibility, interoperability, and reliability.

Software testing is an important software quality assurance activity to ensure that the benefits of OO programming will be realized. The objective of software testing is to uncover as many errors as possible with a minimum cost. A successful test should show that a program contains bugs rather than showing that the program works. Since software testing consumes 40–80 percent of the development costs, how to reduce its cost and improve its quality has always been a big challenge to the software engineering community.

During the last two decades, many software testing strategies, methods, and reliability models have been proposed. These include formal code review, equivalence partitioning, boundary value analysis, cause-effect analysis, basis path testing, control structure testing, mutation testing, and symbolic execution. Most of them were designed based on the traditional function-oriented paradigm. They have been found to be inadequate for testing OO systems.

OO software testing has to deal with new problems introduced by the powerful new features of OO languages. OO features (such as encapsulation, inheritance, polymorphism, and dynamic binding) provide visible benefits in software design and programming. However, these new features also raise challenging problems in the software testing and maintenance phases [wild92a] [lejt92a] [bind94a]. *Encapsulation* means modeling and storing with an object the attributes and the operations an object is capable of performing. The interaction between two or more objects becomes implicit in the code. This makes it difficult to understand object interactions and prepare test cases to test such interactions [leto86a]. *Inheritance* means properties defined for a class are inherited by its subclasses, unless it is otherwise stated. However, a method that is tested to be "correct" in the context of the base class does not guarantee that it will work "correctly" in the context of the derived class [perr90a]. *Polymorphism* means the ability to assume more than one form, both in terms of data and operations. That is, an attribute of an object may refer to more than one type of data, and an operation may have more than one implementation. *Dynamic binding* means code that implements an operation is unknown until run time. These features make testing more difficult because the exact data type and implementation cannot be determined statically, and the control flow of the OO program is less transparent [smit90a].

Currently, most software development organizations are still in the process of observing and/or transitioning to the OO paradigm. It is anticipated that OO software testing will receive much attention. More and more organizations will seek OO testing methods and techniques in the near future as more C++ and Java programs are developed.

The audience of this book includes OO program testers, OO program developers, OO software project managers, and OO researchers who have something to do with OO testing. OO software testers will learn OO software testing problems, various test methods and other aspects of OO development. Software developers usually conduct unit testing of the code they developed. The Unit Testing and Integration Testing chapter will be useful for OO developers. In addition, the Specification and Verification chapter contains an article on design for testability in which a set of design metrics is proposed. Reading this article, OO developers can learn about what should be considered during the design phase to produce a program with good testability.

The book can also be used for a graduate course in software engineering, preferably a special topic or seminar course on OO software testing. In fact, most of the articles in this book were used in a graduate level special topic course taught by one of the editors.

The book begins with a chapter on OO software testing problems to expose the reader to the differences between testing a conventional program and testing an OO program and focuses on the difficulties or challenges a tester would face when testing an OO program. This chapter also helps the reader to understand some subtle issues of OO programming.

Software verification is a software quality assurance activity closely related to software testing. Software verification checks for inconsistencies in a specification or code, for example, what should be checked when adding a subclass to a super class and what are valid invocation sequences of the methods of a class. Solutions to these problems are answered in the Specification and Verification chapter.

The Unit Testing and Integration Testing chapter addresses OO testing techniques. In conventional programming, the basic program unit for unit testing is usually a function, or a module. Is this still valid and effective for OO testing? If not, what is the unit? How should OO unit testing be conducted? This chapter offers some ideas and methods. Integration testing is a crucial phase in software testing in which the components of a software system are integrated and tested according to a certain integration strategy. In conventional programming, integration testing can be carried out either bottom-up, top-down, or a combination of the methods. Depending on the design approach that is used, these integration testing strategies may still be used. That is, the design approach must generate a module structure chart that shows a lattice structure of module invocation relationships. Based on this structure, the conventional integration testing strategies can be applied as usual. In this chapter, however, an OO program integration testing strategy is described.

Regression testing refers to retesting a program after modifications are made to ensure that the program still performs correctly according to its requirements. One simple approach to regression testing is to retest everything whenever a program is modified. However, this approach is not economical since the modification my affect only a small portion of a large program. In the Regression Testing chapter, we present to the reader approaches that can drastically reduce the regression test costs. One approach is to identify classes that are affected by the modification and retest only these classes. Another approach allows the regression tester to select from an existing set of test cases those that will cause the modified program to produce different output. This will further reduce the regression test costs, and will do so to a significant degree, because the tester only needs to focus on the test cases that will produce different results.

One significant difference between conventional program testing and OO software testing is object state testing [kung94c]. In conventional programming, state dependent behaviors are usually found in embedded systems in which the environment of the software system generates stimuli to the software that responds to the stimuli by executing appropriate actions. The software system may generate certain output to the environment as responses to the stimuli. In OO programming, many

objects may have state dependent behaviors and these objects may interact with each other intimately. For example, if the engine is in the "on" state, and the transmission is in the "forward" state, then releasing the brake will cause the car to move forward. If the transmission is in the "park" state, then the car will not move when the brake pedal is released. The Object State Testing chapter presents methods for testing object state dependent behaviors. One of the methods uses a reverse engineering approach to recover the state dependent behavior of an object from code. The article also describes a method for testing the interactions between the objects through their state dependent behavior. Another method proposes to use a state equivalence checker to determine whether two sequences of method invocations will result in the same state as expected. The first method is white-box object state testing. The tester derives test cases from the internal logic of the software and ensures that all statements are tested to a certain extent. For example, 80 percent or 100 percent of the statements are tested at least once. The second method is black-box object state testing, in which test cases are derived from requirements to ensure that the software correctly implements the requirements. Clearly, these two methods are complementary: one ensures that all of the requirements are fulfilled and the other ensures that the desired percentage of code is tested at least once.

After the general philosophy, principles, and OO testing methods, the Test Methodology chapter describes the steps for conducting OO testing. One methodology proposes to integrate the development process with the OO testing process. That is, during the OO development phases, verification and validation of the development products are performed and appropriate testing plans and test procedures are generated. Another methodology deals with testing a class inheritance hierarchy. This methodology recognizes the dependencies of the derived classes of the base classes and proposes a detailed procedure to test the classes in the hierarchy by reusing the test information of the based classes.

OO testing is tedious and time consuming. Tool support is important and necessary. Therefore, the last chapter presents some OO test tools and systems. We hope that more OO testing tools and systems will be developed in the near future and will be available in the commercial market.

This book is organized in such a way that it introduces the reader first to OO testing problems, then general discussions on OO software verification, followed by OO unit testing, integration testing, regression testing, object state testing methods, and finally OO testing methodologies and tool support. We believe that this organizational scheme covers both the overall aspects and the detailed aspects of OO software testing.

Each chapter begins with an editorial introduction and comments on the papers selected. It covers the motivation for selecting the papers, what the reader should look for when reading the papers, and insight and integration of the papers to provide the reader an integrated view of the section. We hope the introductions will help the reader understand the papers.

References

[leto86a] S. Letovski and E. Soloway, "Delocalized Plans and Program Comprehension," *IEEE Software*, Vol. 3, No. 3, May 1986, pp. 41–49.

[perr90a] D.E. Perry and G.E. Kaiser, "Adequate Testing and Object-Oriented Programming," *J. Object-Oriented Programming*, Vol. 2, Jan.-Feb. 1990, pp. 13–19.

[smit90a] M.D. Smith and D.J. Robson, "Object-Oriented Programming—The Problems of Validation," *Proc. IEEE Conf. Software Maintenance*, IEEE Computer Society Press, Los Alamitos, Calif., 1990, pp. 272–281.

[wild92a] N. Wilde and R. Huitt, "Maintenance Support for Object-Oriented Programs," *IEEE Trans. Software Eng.*, Vol. 18, No. 12, Dec. 1992, pp. 1038–1044.

[lejt92a] M. Lejter, S. Meyers, and S.P. Reiss, "Support for Maintaining Object-Oriented Programs," *IEEE Trans. Software Eng.*, Vol. 18, No. 12, Dec. 1992, pp. 1045–1052.

[kung94c] D. Kung, et al, "On Object State Testing," *Proc. COMPSAC '94*, IEEE Computer Society Press, Los Alamitos, Calif., 1994, pp. 222–227.

[bind94a] R. Binder (Guest Editor), "Object-Oriented Software Testing," *Comm. ACM,* Vol. 37, No. 9, Sept. 1994, pp. 28–29.