Chapter 8
Issues and Strategies

J. C. Huang
Department of Computer Science
University of Houston

Regression testing

Regression testing should be performed to a program after it is modified to remove an error or to add a function.

The process is to rerun all or some of the previous tests to assure that no new errors have been introduced through the changes or to verify that the software still performs the same functions in the same manner as its older version.

Stages of program testing

Unit testing
Integration testing
System testing

Strategies for integration testing

Non-incremental (big bang) approach

Incremental approach:
- Bottom-up integration
- Top-down integration
- Sandwich integration
- Phased integration

Bottom-up integration testing

Units are tested in isolation from one another in an artificial environment known as a test driver (harness), which consists of the driver programs and data necessary to exercise the modules.

Bottom-up integration testing (continued)

Advantages:

Unit testing is eased by a system structure that is composed of small, loosely coupled modules

Bottom-up integration testing (continued)

Disadvantages:

1. The necessity to write and debug test harness for the modules and subsystems. Test harness preparation can amount to 50% or more of the coding and debugging effort.
2. The necessity to deal with the complexity that results from combining modules and subsystems into larger and larger units.

Top-down integration testing

This approach starts with the main routine and one or two immediately subordinate routines in the system structure. After this top-level skeleton has been thoroughly tested, it becomes the test harness for its immediately subordinate routines.

Top-down integration requires the use of program stubs to simulate the effect of lower-level routines that are called by those being tested.

Top-down integration testing (continued)

Advantages:

1. System integration is distributed throughout the implementation phase; modules are integrated as they are developed.
2. Top-level interfaces are tested first and most often.
3. The top-level routines provide a natural test harness for lower-level routines.
4. Errors are localized to the new modules and interfaces that are being added.

Top-down integration testing (continued)

Disadvantages:

1. It may become difficult to find top-level input data that will exercise a lower-level module in a particularly desired manner.
2. The evolving system may be very expensive to run as a test harness for new routines.
3. It may be costly to relink and re-execute a system each time a new routine is added.
4. It may not be possible to use program stubs to simulate modules below the current level.

Sandwich integration testing

This approach is predominately top-down, but bottom-up techniques are used on some modules and subsystems. This mix alleviates many of the problems encountered in pure top-down testing and retain the advantages of top-down integration at the subsystem and system level.

Phased integration testing

In the phased approach, units at the next level are integrated all at once, and the system is tested before integrating all modules at the following level.

Unit testing: to do or not to do?

Unit testing is a point of contention.

Some believe that unit tests allow the engineers to find the nature and location of faults quickly, and thus recommend that the units should be thoroughly tested before integration. Others recommend integration strategies in which units are not tested in isolation, but are tested only after integration into a system.

Experimental results

Some experimental results indicate that the top-down strategy generally produces the most reliable systems and is the most effective in terms of defect detection.

Testing object-oriented programs

Most of the test methods discussed in the preceding chapters were developed based on the traditional function- or procedure-oriented paradigm.

They have been found to be inadequate for testing object-oriented software.

Testing object-oriented programs (continued)

We need to deal with problems introduced by the new features of an object-oriented programming language. Those features, such as encapsulation, inheritance, polymorphism, and dynamic binding, cause challenging problems in software testing.

Impact of encapsulation

Encapsulation affects program testing in two ways:

1. The basic testable unit will no longer be a subprogram, and
2. Traditional strategies for integration testing are no longer applicable.

Integration testing for OO programs

For an object-oriented program, the smallest testable unit will be a class (or a comparable object-oriented program unit). Given that the methods associated with each method interface often take advantage of the underlying implementation, it is difficult to see how each method can be tested in isolation.

Weyuker's 5th axiom: antiextensionality

For two different algorithms that compute the same function (i.e., they are "semantically identical"), an adequate test set for one algorithm is not necessarily an adequate test set for the other.

This says that, if we replace an inherited method with a locally defined method that performs the same function, the test cases for the inherited method is not guaranteed to be adequate for the locally defined method.

Weyuker's 6th axiom: general multiple-change axiom

If two programs are of the same shape, a test for one will not necessarily be adequate for the other.

This axiom tells us that when the same items are inherited along different ancestor paths then different test sets will be needed.

Weyuker's 7th axiom: antidecomposition

Something tested in one context will have to be retested if the context changes.

For example, suppose a method is created within the context of a given class, and further suppose that a specialization is created (e.g., a subclass or a derived class) based on this class, and that the specialization inherits the tested method from the generalization. Then, even if the method is tested within the context of the generalization, there is no guarantee that the same method is appropriate within the context of the specialization.

Weyuker's 8th axiom: anticomposition

Adequately testing each unit in isolation is usually insufficient to be considered as the same as adequately testing the entire (integrated) program.

Suppose that we change the underlying implementation for a given object, but keep the interface intact. We might suspect that we could get away with simply retesting the modified object in isolation. This is not the case. We will have to retest all dependent units.

Problems with integration testing

Integrating "subprograms" into a class, one at a time, testing the whole as we go, may not be an option.

For example, there are usually direct or indirect interactions among the components that make up the class. One method may require that the object be in a specific state, which can only be set by another encapsulated method, or a combination of encapsulated methods.

Impact of information hiding

Advocates of object-oriented technology are fond of citing the black-box nature of objects because a user of an object is denied access to the underlying implementation. This creates problems during testing.

Problems with unit testing

Software unit testing is performed to find errors in a software component, such as a module. Unlike a traditional program where functional modules and functional procedures are considered as it's components, an object-oriented program consists of three levels of components:

1. functions defined in classes,
2. classes, and
3. modules consisting of a collection of classes.

Questions to be addressed in research

Since classes are the major components in an object-oriented program, testers have to find answers to the following questions:

1. Can existing unit testing techniques be applied to objects and classes?
2. What test-case selection criteria can be used in unit testing for object-oriented software?
3. How can unit test for an OO program be performed in a systematic way?

Flow graph-based testing

There are several techniques for testing classes individually, i.e., for unit testing. Most of them involve in sending a sequence of messages constructed based on the specification to an object of the class and observe the resulting effect on the object.

Random vs. systematic

The test-case generation techniques for classes fall into two categories: random or systematic. A random technique generates message sequences entirely at random; most existing techniques use this approach. A systematic technique uses a repeatable procedure to generate a fixed sequence satisfying certain criteria.

Flow graph for OO programs

Zweben *et al.* showed that it is possible to associate a flow graph with a class module. A node in such a flow graph represents a message; and an edge, say, from node A to node B, represents the possibility that A might be sent, followed by B. Using this notion of a "class-flow graph", a collection of systematic coverage criteria for class modules can be defined.

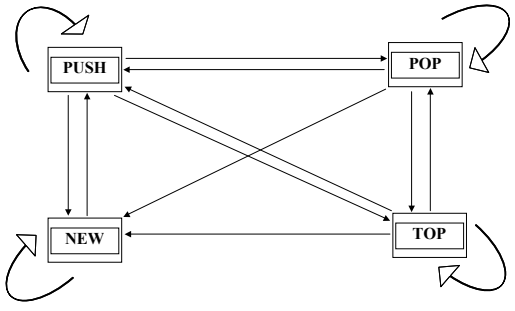
Algebraic and model-based specifications

Algebraic specification defines the behavior of a module in terms of a set of axioms that characterize the equivalence of combinations of operations, whereas *model-based* specifications involve the individual modeling of each operation in the class. The following table presents both algebraic and model-based specifications for a stack class.

Two versions of stack specification

Algebraic	*Model-Based*
module STACK_TEMPLATE(type T) **type STACK** **functions** NEW: → STACK PUSH: STACK × T → STACK POP: STACK → STACK TOP: STACK → T **domain conditions** POP(s): not (s = NEW) TOP(s): not (s = NEW) **axioms** (1) not (PUSH(s, x) = NEW) (2) POP (PUSH(s, x)) = s (3) TOP (PUSH(s, x)) = x **end** STACK_TEMPLATE	**module** STACK_TEMPLATE (type *T)* **type** STACK is modeled by STRING **interface** **operations** New (s: STACK) **ensures** s = e **operations** Push (s: STACK, x:T) **ensures** (s = #x ° #s) ∧ (x = #x) **operations** POP (s: STACK) **requires** not (s = e) **ensures** ∃x ∋ #x = x ° s **operations** Top (s: STACK, x:T) **requires** not (s = e) **ensures** (∃t ∋ s = x ° t) ∧ (s = #s) **end STACK_TEMPLATE**

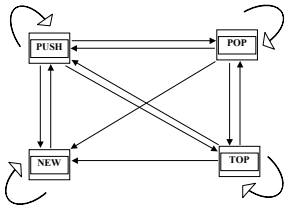
The operation of	defines	uses
NEW	s t a c k	<>
PUSH	s t a c k , T	s t a c k
POP	s t a c k	s t a c k
TOP	T	s t a c k



Flow graph for a simple stack module.

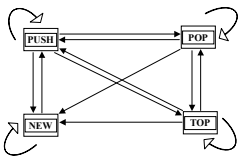
A test sequence for *node* coverage

NEW, PUSH, TOP, POP



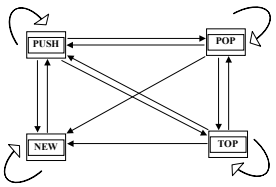
A test sequence for *edge* coverage

NEW, NEW, PUSH, NEW, PUSH, PUSH, POP, PUSH, TOP, TOP, PUSH, POP, POP, NEW, PUSH, PUSH, PUSH, POP, TOP, POP, TOP, NEW



A test sequence for *define* coverage

NEW, PUSH, PUSH, POP, TOP, PUSH



Other criteria

Other test-case selection criteria for OO software can be found in the literature, but they remain to be experimental in nature, and not in common use at this time.