Robert V. Binder

Design for Testability in Object-Oriented Systems

Testability is the relative ease and expense of revealing software faults. This article maps the testability terrain for object-oriented development to assist the reader in finding relatively shorter and cheaper paths to high reliability. Software testing adds value by revealing faults. It is fundamentally an economic problem characterized by a continuum between two goals. A *reliability-driven* process uses testing to produce evidence that a pre-release reliability goal has been met. Time and money are expended on testing until the reliability goal is attained. This view of testing is typically associated with stringent, quantifiable reliability requirements. Other things being equal, a more testable system will reduce the time and cost needed to meet reliability goals.

A *resource-limited* process views testing as a way to remove as many rough edges from a system as time or money permits. Testing continues until available test resources have been expended. Measurement of test adequacy or system reliability are incidental to the decision to release the system. This is the typical view of testing. Other things being equal, a more testable system provides increased reliability for a fixed testing budget.

Regardless of which goal is emphasized, testability is important for both ad hoc developers and organizations with a high level of process maturity. Testability reduces cost in a reliability-driven process and increases reliability in a resource-limited process.

Design for testability (DFT) is a strategy to align the development process so that testing is maximally effective under either a reliability-driven or resource-limited regime. Object-oriented systems present some unique obstacles to testability as well as sharing many with conventional implementations. The cost and difficulty of testing object-oriented systems can be reduced by following some basic design principles and planning for test. Broadly conceived, software testability is a result of six factors:

- Characteristics of the representation
- Characteristics of the implementation
- Built-in test capabilities
- The test suite (test cases and associated information)
- The test support environment
- The software process in which testing is conducted

These six factors are the spine of the testability fishbone (Figure I). We'll consider each in turn. Before examining the major and minor bones of object-oriented testability, we review related approaches. Although research and practice in software testing has had relatively little to say about testability, it has been given considerable attention in the engineering and manufacturing of VLSI (very large-scale integration) devices.

The Concept of Testability

Testability has two key facets: *controllability* and *observability*. To test a component, you must be able to control its input (and internal state) and observe its output. If you cannot control the input, you cannot be sure what has caused a given output. If you cannot observe the output of a component under test, you cannot be sure how a given input has been processed. With controllable input and observable output, compliance can be decided. These concepts also have an analogous, formal meaning (beyond the scope of this article) in control system state-space models [20]. There are many obstacles to controllability and observability. In general, they result from the fact that a component under test must be embedded in another system. Removing obstacles to controllable input and observable output is the primary concern of design for testability. As a practical matter, testability cannot be considered apart from the software process. Process *capability* is equally important.

Design for testability in VLSI. The concept of testability has played an increasingly important role in the design and manufacture of integrated circuits (chips or ICs). Chip testing is performed to identify defects resulting from chip manufacturing or physical failures due to a wide range of causes (vibration, heat, cosmic rays). *Design verification* is the engineering development process that assures functional correctness. Chip testing assumes complete functional correctness. In contrast, software testing assumes perfect replication and the presence of functional faults.

Chips are tested by applying input bit strings and observing output bit strings. For example, with a simple AND gate, we expect that a logical 1 is produced only when both input lines are also logical 1. There are three main approaches to integrated circuit design for test.[1] 1) *Ad hoc* solutions are component-specific. These include the ability to disconnect parts, adding more test points, and reducing a network to smaller, more tractable components. 2) The *structured approach* refers to design rules that reduce or make controllable state variables in sequential circuits. Testing a sequential circuit (a state machine) is considerably more difficult than testing a combinational circuit (a decision table). For sequential circuits, these techniques aim to reduce the test problem to one of generating combinational inputs. 3) *Built-in Test* (BIT) adds standard test circuits. The density of logic gates on chips has steadily increased (about an order of magnitude in the last 10 years), while the feasible number of pins (external interfaces) remained about the same.

With present-day VLSI, on-chip testing capabilities are a necessity. Without these capabilities, it is impossible to provide adequate controllability and observability. ANSI/IEEE Standard 1 *149.1* defines a boundary-scan architecture for BIT functions [14]. Individual chips (embedded in a single package) may be accessed via four to six additional external test lines and a few standardized test circuits on every IC. This provides the ability to place an embedded chip or board into test mode, transmit (or generate) test bit strings to specific embedded components, and capture output bit strings. The basic BIT capabilities can be extended to provide complex, automatically initiated test suites called *Built-in-Self Test* (BIST). Since approval in 1990, ANSI/IEEE Standard I 149.1 has been rapidly adopted by IC manufacturers. Interoperability and reliability of components and systems have improved.

Software testability has received little consideration relative to the large number of books and articles published on other aspects of software testing. Definitions of testability from two U.S. software engineering standards appear in Figure 2.

Some notion of testability is implicit in most testing strategies. Structural and functional test case design methods are typically based on a program or system model and a related fault hypothesis. For example, data flow testing assumes the paths formed by definitions and uses of variables are a good place to look for errors. These models imply a kind of general testability criteria, since certain programs may be easier or harder to test according to these models. For example, it may be prohibitively difficult and expensive to

[1] Design for test in VLSI is given a comprehensive treatment by Milton Ambramovici, Melvin A. Breuer, and Arthur D. Friedman in *Digital Systems Testing and Testable* (New York, IEEE Press, 1990) and by Hideo Fujiwara in *Logic Testing and Design for Testability* (Cambridge, Mass.: MIT Press, 1985). An overview of DFT is presented by Daniel P. Siewiorek and Robert S. Swarz in *Reliable Computer Systems* (2d ed. Digital Press, Bedford, Mass., 1992.) The history, motivation, and key features of IEEE/ANSI Standard 1149.1, are summarized by Maunder and Tulloss in "Testability on TAP," *IEEE Spectrum 29*, 2 (Feb. 1992), 34–37. Hagge and Russell offer an excellent summary of DFT approaches and an integrated process model for development and manufacturing of testable multichip modules in "High-yield assembly of multichip modules through known-good ICs and effective test strategies." *Proceedings of the IEEE. 80*, 12 (Dec. 1992), 1965–1994.

devise path test cases for a program with spaghetti code or self-modifying control. Qualitative discussions of testability appear in [15] and [22]. In general, testable software is small, simple, and explicit.

A pragmatic analysis of unit-level testability is presented in [15]. When a module is embedded in an application system, it may be difficult to fully exercise it (lack of controllability) and the immediate outputs may be obscured (lack of observability). User interface bindings may reduce testability. Accessing and tracing module input and output may require special tools. Several strategies are suggested to increase controllability and observability. Information hiding and separation of concerns improves testability. Test scaffolding (test drivers and stubs) improves controllability and observability. Even with scaffolding, it is often useful to add explicit built-in test functions to report internal state on demand. "With simple modules, this separation of concerns is useful—with complex ones, it is invaluable." Assertions increase and retain observability when the module under test is subsequently integrated with the rest of an application system.[2]

A formal investigation of testability based on abstract data types and their interfaces is given in [14]. Two practices are identified that reduce testability. Global or **local** state variables reduce observability of procedure or function output, since response is not determined by input explicitly passed to the unit under test. A function or procedure that cannot produce every possible value in the domain of its associated type lacks controllability; part of the output domain cannot be produced under any circumstance. Modules can be made perfectly observable by adding explicit arguments to replace access to global or local state variables. Similarly, they can be made perfectly controllable by defining a data type that exactly corresponds to the domain of the output variables. The number of changes that must be made to obtain perfect observability and controllability is an index of testability.

An observability metric for every statement in a program is presented in [24]. The metric is based on the observable effect of program mutations. "A program's testability is a prediction of its ability to hide faults when the program is black-box tested with inputs selected randomly from a particular input distribution." Testability is defined as sensitivity to faults. Sensitivity is determined by systematically mutating a program and its data, running the mutated program, and comparing the output of the mutant to the output of the original program. Given some input set, statement sensitivity is a product of three probabilities: frequency of statement execution, probability that a fault will result in an incorrect data state, and the probability that a fault will result in an incorrect output. A highly sensitive statement is very likely to produce different output when mutated. A less sensitive statement is less likely to produce different output. Lower sensitivity implies that faults are more likely to be hidden, and therefore relatively more testing will be required to reveal them. Sensitivity is computed for each statement. Program sensitivity (testability) is the minimum of all statements sensitivities. Hence, highly sensitive programs are highly testable.

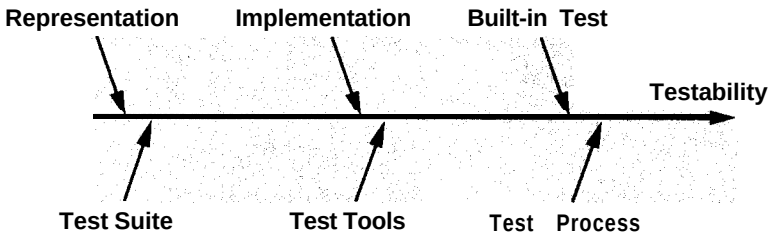


Figure 1. **The testability fishbone: Main facets**

4.3.4 Traceability of requirements to test cases. The contractor shall document the traceability of the requirements in the Software Requirements Specifications (SRS) and Interface Requirements Specifications (IRS) that are satisfied or partially satisfied by each test case identified in the Software Test Description (STD). The contractor shall document this traceability in the STD for each Computer Software Configuration Item (CSCI).

10.3.2 Testability of requirements. A requirement is considered to be testable if an objective and feasible test can be designed to determine whether the requirement is met by the software.

DOD 2167A

Testability. (1) The degree to which a system or component facilitates the establishment of test criteria and the performance of tests to determine whether those criteria have been met. (2) The degree to which a requirement is stated in terms that permit establishment of test criteria and performance of tests to determine whether those criteria have been met.

IEEE 610.12

Figure 2. **Testability definitions**

Testability Factors

The preceding definitions of software testability focus on technical aspects. As a practical matter, testability is as much a process issue as it is a technical problem. There are six primary testability factors (see Figure 1). Each may facilitate or hinder testing in many ways. A usable representation is necessary to develop test cases. Im-

[2] An asset-tion consists of a condition and an associated exception routine. This may be implemented by the assert() macro in C and C++. The assertion statement is simply a Boolean expression that is evaluated when the statement is reached during program execution. If the condition evaluates to true, nothing happens. If the condition is false, the program is terminated or an exception is activated.

plementation characteristics determine controllability and observability. Built-in test capabilities can improve controllability and observability and decouple test capabilities from application features. An adequate and usable test suite (test cases and plan for their use) is necessary. Test tools are necessary for effective testing. High leverage is available with an integrated tool set. Without an effective organizational approach to testing and its antecedents, technical testability is irrelevant.

Representation. The presence of a representation and its usefulness in test development is a critical testability factor. System representations range from natural language statements about desired capabilities to detailed formal specifications. Testing without a representation is simply experimental pi-otolyping-it cannot be decided that a test has passed or failed without an explicit description of the expected result. The best that can be said of "testing" without a representation is that it may force production of a partial representation as part of the test plan. There are many approaches to developing object-oriented representations, generically known as object-oriented analysis (OOA) and object-oriented design (OOD). For example, [17] outlines a systematic approach to OOA/D which explicitly addresses **development of** test cases from the representation. Figure 3 shows testability facets of representations.

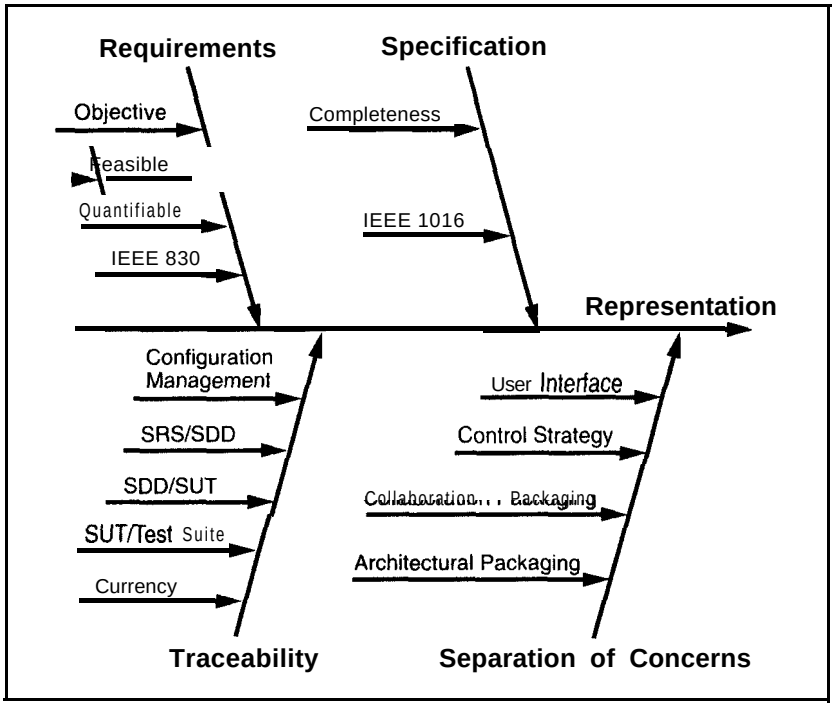
Requirements include both narrative capability statements and abstract system models. The standards shown in Figure 2 identify essential characteristics for testability. *Objective* requirements are phrased or modeled in such a way that compliance in the system under test (SUT) is not subject to idiosyncratic interpretation. *Feasible* capabilities may be developed and implemented with existing technology. While not all capabilities benefit from a quantitative definition, it certainly eases testing to quantify those that can. For example, "average response time will be 1 second or less," instead of "response time will be quick." IEEE/ANSI standard 830 defines the desirable aspects of software requirements specification: unambiguous, complete, verifiable, consistent, modifiable, traceable, feasible, and useful for maintenance [11]. Each of these attributes contributes to testability.

Specifications describe the detail design for an implementation. IEEE/ANSI standard 1016 provides a definition of a complete and usable software design description (SDD) [13]. An SDD should define the organization of software components, dependencies, interfaces, and detail of algorithms and data structures. *Completeness* means the specification covers all aspects of the system and that there are no "to be determined" items.

Traceability is a simple, common sense notion that can prevent a wide range of problems. It is simply a bookkeeping operation that records the antecedent of any work product. In testing, we are interested in being able to find which software component implements a given specification, and which specification implements a given requirement. Without this information, it is practically impossible to develop complete and accurate test plans for even a small system. *Currency* means that the specification mirrors the system as programmed. If specifications are not current, they result in incorrect test plans.

Separation of concerns is a first principle of software engineering that is directly related to controllability and observability. A component that can act independently of others is more readily controllable. Separation of concerns is application- and environment-specific, but we suggest a few key issues here. If user *interface* capabilities are entwined with basic functions it will be more difficult to test each function. The *control strategy* (mechanism for allowable sequence of component activation) of object-oriented programming tends to be less explicit (but no less important) than conventional systems. If control is embedded in the structure of the interfaces, it is more difficult to devise explicit tests than if control is an explicit function of a control object. *Collaboration Packaging* refers to the structure of classes participating in a responsibility. *Architectural Packaging* refers to the way classes are allocated to tasks in the target environment and how their runtime interfaces are composed. To the extent that packaging and capabilities are orthogonal, they will be more testable. In a distributed target environment, for example, a single responsibility imple-

Figure 3. Testability: Representation



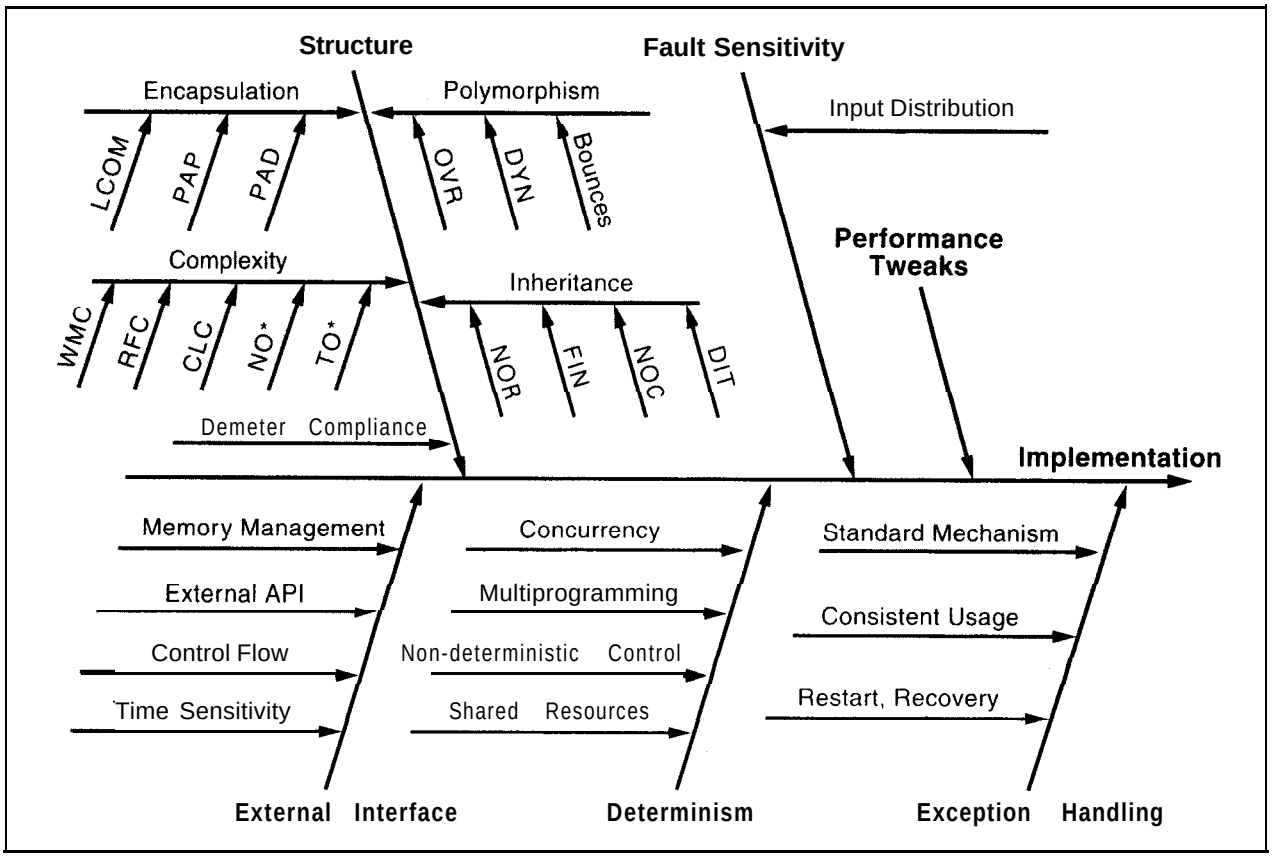


Figure 4. Testability: Implementation

mented by objects residing on several hosts will probably be more difficult to test than the same responsibility on a single host.

Implementation. Figure 4 shows the Implementation testability fishbone. An object-oriented program that complies with generally accepted principles of object-oriented programming poses the fewest obstacles to testing. *Structural* testability can be assessed by a few simple metrics. These are summarized in Tables I through 4. A metric may indicate complexity, scope of testing, or both. The effect of all complexity metrics is the same: a relatively high value of the metric indicates decreased testability; a relatively low value indicates increased testability. Scope metrics indicate the quantity of tests; the number of tests is proportional to the value of the metric. High scope and low complexity suggests many simple tests are indicated; low scope and high complexity suggests a few difficult tests.

For example, with high coupling among classes (CBO) it is typically more difficult to control the class under test (CUT), thus reducing testability. Good design usually improves testability but even well-designed applications may be inherently difficult to test. These metrics provide information useful for resolving design trade-offs and assessing relative difficulty and cost of testing; they are not some kind of "score" to be maximized or minimized.

The *Law of Demeter* constrains uses (message sends) to "preferred supplier classes," in effect superclasses, classes of instance variables, and classes of arguments passed to the class under test [18]. Compliance reduces the number of interfaces, the number of stubs and drivers that may be needed, and the number of integration test interfaces. *Fault sensitivity* [24] was discussed previously. Low sensitivity corresponds with low testability. *Performance tweaks* tend to reduce the correspondence of the implementation and a specification and can interfere with both controllability and observability. *External* Interfaces often create difficult testability problems. For example, with an "event-driven" GUI, it may be very difficult to test application server classes without using the GUI. However, the GUI may decrease both controllability and observability of a CUT. *Determinism* is the extent to which the CUT does not require asynchronous cooperation with other tasks. A high degree of concurrency poses obstacles to repeatable tests. It may be difficult to reproduce and isolate the cause of a failure: the SUT, the environment, or a particular input pattern. Testable *exception handling* requires consistent usage of language-supported features and a related design strategy [20]. Exceptions are typically less controllable than other application functions and may require simulation of failure modes. This may be difficult and time consuming.

Built-in test. Built-in test (BIT) capability provides explicit separation of test and application functionality. The systematic addition of set/reset, reporters, and assertions to a class is a

Table 1. Testability and encapsulation

Encapsulation metric	Complexity effect	Scope effect
LCOM (Lack of Cohesion in Methods). Number of groups of instance variables used by one method only [10]. High LCOM means more states must be tested to prove absence of side effects among methods.	√	
PAP (Percent Public And Protected). Percentage of data members in a C++ class that are public or protected. Indicates proportion of class data visible to other objects [21]. High PAP means more opportunities for side effects among classes	√	
PAD (Public Access to Data members). The number of external accesses to a class's public or protected data members. Measures violations of encapsulation [21]. High PAD means more states to test to prove absence of side effects among classes.	√	

Table 2. Testability and inheritance

Inheritance metric	Complexity effect	Scope effect
NOR (Number of Root Classes, RootCtn.) The number of distinct class hierarchies employed by a program [21]. NOR is one dimension of number of components to test.		√
FIN (Fan In). The number of classes from which a class is derived. FIN > 1 is only possible with multiple inheritance [21]. High FIN increases the possibility of' incorrect bindings. In most cases, all of the inherited methods will need to be retested in the CUT.	√	
NOC (Number of Children). The number of classes derived from a specific parent class. It indicates the number of classes which will be directly impacted by a change to the parent (class fan-out) [10]. In most cases, all of the parent methods will need to be retested in the child CUT.		√
DIT (Depth of Inheritance Tree). Indicates the level of a class within its hierarchy [10]. In most cases all inherited methods will need to be retested in the CUT.		√

Table 3. Testability and polymorphism

Polymorphism metric	Complexity effect	Scope effect
OVR (Percent of non-overloaded calls, Pctcall.) Percent of calls throughout the SUT that are not made to overload modules [21]. Overloading may result in unanticipated binding. High OVR indicates more opportunities for faults.	√	
DYN (Percent of dynamic calls.) Percent of messages throughout the SUT whose target is determined at run time. Dynamic binding may result in unanticipated binding. High DYN means that many test cases will be needed to exercise all the bindings of a method.‡		√
Bounce-C. Count of the number of yo-yo paths visible to a CUT. A yo-yo path is a path that traverses several supplier class hierarchies due to dynamic binding. A bounce may result in unanticipated binding. High bounce indicates more opportunities for faults.‡	√	
Bounce-S. Count of the number of yo-yo paths in SUT. A yo-yo path is a path that travel-ses several supplier class hierarchies due IO dynamic binding. A bounce may result in unanticipated binding. High bounce indicates more opportunities for faults.‡	√	

‡Under investigation by the author.

simple way to provide effective control and observation. Attempts to approximate BIT with application methods is a partial solution at best. If a standard test interface is included in all classes, additional development overhead is minimal and the potential payback is great. The BIT fishbone is shown in Figure 5.

Without set/reset, effective state-based testing is a practical impossibility. A set/reset method allows an object to be set to a predefined internal state, regardless of its current state. Several approaches are possible. The BIT can offer a menu of states with noncontrollable instance variable values, a menu of states with controllable values, or direct manipulation of instance variables.

A reporter returns the concrete (internal, private) state of an object. If the CUT does not offer complete reporting of abstract (external, public) state, the reporter must provide it.[1] A reporter must be trustworthy—we need to be sure its reports are accurate. Reporter code should be subjected to careful scrutiny. For example, the code could be proved correct and then tested with low-level debug probes.

Useful set/reset and reporter methods necessarily violate encapsulation. BIT services should not be used to accomplish application requirements. A *safety* provision is advisable to prevent inadvertent or willful misuse of BIT services. Source *code control* uses compiler-directives to include or exclude BIT features. Either a BIT or a non-BIT version of the SUT can be built. This is problematic because the tested implementation is not necessarily the released implementation. There are many horror stories about systems that inexplicably fail when an apparently innocuous code segment is added or removed. Source code control should be used with due diligence. *Mode control* provides a global parameter to toggle test mode and normal mode. In normal mode, BIT services are disabled. Any BIT invocation is treated as a no-op or an exception. In test mode, BIT services are enabled. *Password control* requires a password of some kind to activate BIT functions. The driver must supply a password value in a parameter defined as part of the BIT interface. A BIT request without the password would be treated as a no-op or an exception. There are many possible safety schemes. Regardless of the BIT approach, some form of encapsulation safety should be provided.

Assertions can be used for many purposes: pre-conditions, post-conditions, class invariants, and sampling of intermediate results. For example, they can monitor long-running algorithms to detect failure to converge. After class testing, a class must be embedded in an application system. Once embedded, assertions continue to provide built-in testing and reporting. They contribute to testability by proving built-in observability. A driver is a special-purpose class that activates the CUT-drivers are discussed later in this article.

Test suite. A test suite is a collection of test cases and plans to use them. The Test Suite fishbone is shown in Figure 6. There are many possible approaches to the development of test cases [8]. A comprehensive list of test suite components is offered in IEEE/ANSI standard 829 [2]. It defines the contents of a test plan, test design specification, test case specification, test procedures, test item transmittal report, test log, test incident report, and test summary. A test suite lacking in these features may pose practical obstacles to testability. A test case schema for object-oriented programs is presented in [7].

Expected results are necessary for test cases. An oracle is a mechanism for producing expected results. Typically, expected test results are prepared by manual calculation or simple simulation. Oracle automation and class testing are discussed in [16]. A system for which an oracle is infeasible is fundamentally untestable. For example, the quantity of input or output to consider may be intractably large, or there may be no way to obtain expected output, *a priori*. Such systems require special verification strategies [25].

A test suite represents an asset ac-

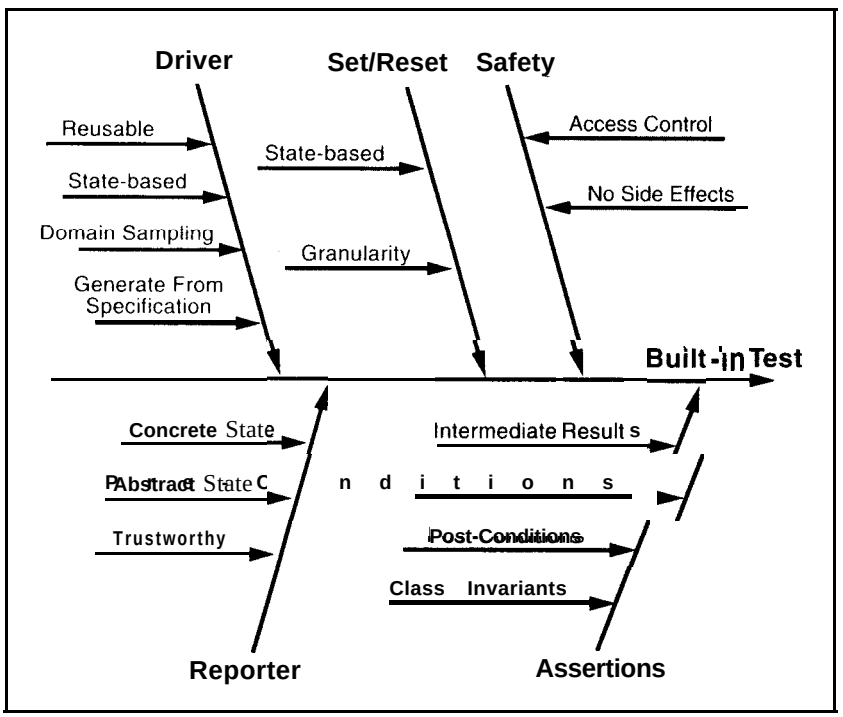


Figure 5. **Testability: Built-in test**

[1] Abstract state is implementation-independent and may be reported via an object's interface. Concrete state is implementation-dependent and is typically not reported. For example, suppose a collection is implemented by a linked list. The abstract state of the collection includes the contents and number of items in the collection; concrete state includes pointers to the first, last, and most recently used nodes [19].

Table 4. Testability and complexity

Complexity metric	Complexity effect	Scope effect
WMC (Weighted Methods per Class) is the sum of the cyclomatic complexity of methods implemented within a class [IO]. Higher complexity is often associated with a greater number of faults; a proportionally greater number of test cases will be required for decision coverage.	√	√
RFC (Response for a Class) is a count of methods implemented within a class. plus the number of methods used by an object not due to inheritance | IO]. High RFC indicates a class will either require high number of stubs or must he embedded with many other (lasses before it (an be tested. Each such interface will need to be tested.	√	√
CBO (Coupling Between Objects.) The number of non-inheritance couples to classes outside the CUT [10]. High CBO indicates a class will either require high number of stubs or must be embedded with many other classes before it can be tested. Each such interface will need to he tested..	√	√
CLC (Class Complexity .) The cyclomatic complexity of the control graph formed by a union of all method control graphs with a state tranisition graph for the class.	√	√
NOM (Nominal Number of Methods per class.) 20 is suggested as an upper limit. NNOM = NNOF + NNOP.‡		√
NOF (Nominal Number of Functions per class.) Functions report state, but do not change it.‡		√
NOP (Nominal Number of Procedures per class.) Procedures change state. ‡	√	√
TOM (Total Number of Methods per class.) Same as NNOM, hut includes all inherited methods. TNOM = TNOF + TNOP. More methods require more testing.‡		
TOF (Total Number of Functions per class.)‡		√
TOP (Total Number of' Procedures per class.) If there ate *n* procedures with no sequential activation constraint, at least n^2 tests are necessary to verifj state control.‡		√

‡Under investigation by the author

Table 5. DFT strategies

DFT approach	VLSI	Class test
None	Manual probes on available test points	Testing embedded class with white/black box tests, use of debugger and ad hoc inline trace
Ad hoc	IC-specific test points	Driver/stub per class
Structured	Addition of set/reset and report circuits to provide combinational test	Driver-stub per class, test interface provides set/reset and report methods
Standardized	Standard Test Access Port and related functions across all components (IEEE 1149.1)	Driver/class pair with standard driver API delivered with class library
Self-Test	Built-in Self-Test (BIST). On-chip ability to generate and execute tests using standard BIT	Generic driver, executable specification packaged with the implementation

quired at considerable cost and should be treated accordingly. The reuse of a test suite (regression testing) provides economic justification for careful test development. Over time, a reusable test suite increases the amount oftesting that can be done, and hence increases testability. Effective test suite reuse requires configuration management control and traceability to successive versions of the representation and implementation.

Test case development is a challenging activity that is susceptible lo errors similar to those occurring in software development. Faulty test cases can be problematic, iii that a correct implementation may be rejected o r incorrect implementation may be accepted. Established techniques for work product review and defect prevention will improve test suites and increase their contribution to testability

Test tools. Testing requires automation. Without automation less testing will be done or greater cost will be incurred to achieve a given reliability goal. I he absence of tools inhibits testability. The components of a test environment have been analyzed elsewhere [6]. Basic components include configuration management, test suite manager, static analyzer, instrumentor, run-time trace, comparator, reporting, and integrated debut. Test case development is supported by input capture systems, script editors, spec-based generator, code-based generators, input data generators, and support for other developer-defined tests. The test bed (test environment) needs functions to initialize a system and its environment, execute test scripts, and replay scripts under predefined conditions. There are many commercial offerings for such tools. These components, with the necessary changes, are equally useful for object-oriented implementations. Several test tools for object-oriented systems have recently become commercially available. Intel-operability is a key concern, as it is with CASE in general. The Test Tool fishbone is shown in Figure 7.

Process capability. Overall software process capability and maturity can significantly facilitate or hinder testability. The Process Capability fishbone is shown in Figure 8. The key process abilities of the defined level for software product engineering [23].

Constancy of purpose is necessary foi continuous quality improvement [12]. Staff and management need to be empowered to perform adequate testing. This requires sufficient funding, accountability, and an unequivocal. sincere commitment to establishing and improving the process.

The technical approach must be *effective.* I he over all development process must be defined and r epeatable [23]. A *customer-oriented* test process is driven by customer expectations for cost and reliability. Systems should not be tested unless a *readiness assessment* has been performed. This prevents waste of testing resources on trivial problems more easily prevented by design or programming. *Closed-loop feedback* is necessary to assess testing effectiveness. For example, if the desired level of post-release reliability is not obtained, a process improvement cycle should be initiated to determine and correct the cause of the problem. Testing is a challenging task requiring high *staff capability*. This results from adequate tool and method training, motiv a- tion, and appropriate education and experience.

Testing is most effective when it is viewed as an essential component of a system of production. It is neither a magic filter to remove problems with an upstream source nor a dumping ground for hastily developed code. An *integrated test strategy* views testing in context. Vertical integration means there is a well-defined path among test processes for classes, clusters of classes, and application systems. Horizontal integration means that testability is considered over all stages of development: representation, (requirements and design), implementation (code), test suite (testing), and subsequent iterations of reuse and maintenance. Verification and validation integration means that testing is used in a balanced, optimal manner with other quality assurance practices: prototyping, inspections, and reviews for all stages and work products.

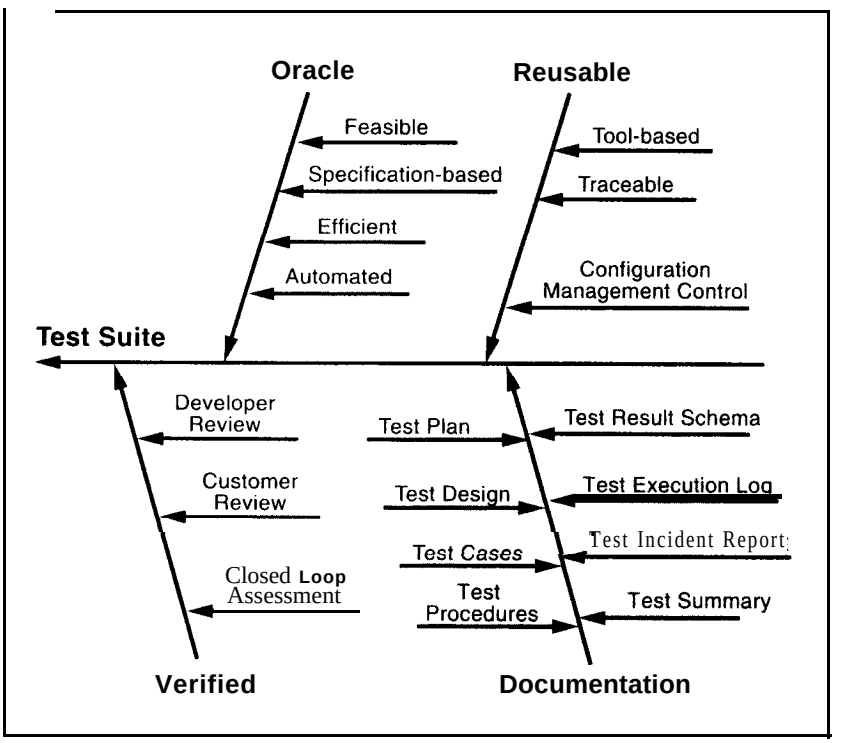


Figure 6. **Testability: Test suite**

Toward High Testability For Object-Oriented Systems

Software testing is an economic problem closely intertwined with nearly all major technical issues in software engineering. The implication of the foregoing analysis is hardly startling: testability results from good software engineering practice and an effective software process. Figure 9 shows the entire testability fishbone. We've seen how these basic facets contribute to testability. However, a manual approach to testing will quickly reach practical limits that are unsatisfactory given the extent of testing needed to assure high reusability and reliability. Just as high testability for VLSI requires an explicit test infrastructure, so will high testability for large object-oriented systems. Advanced test automation is needed.

An extension of the common notion of a driver and BIT functions suggests some interesting possibilities for advanced test automation. An instructive comparison of levels of abstraction in VLSI and class libraries is provided in [11]. Standardization is needed to achieve a high level of economic efficiency and interoperability. The "gauge" is central to this vision. A gauge is a reusable software component used to test a single class and is routinely produced as part of the implementation. An outline strategy for implementation of the gauge concept follows. Table 5 compares VLSI DFT strategies and class testing. Possible configurations for corresponding object-oriented DFT strategies are depicted in Figure 10 and summarized in Table 5.

Further discussion of elements in Table 5 follows:

- No *DFT*. To the extent that class testing is done, there is no systematic DFT. The implementation is developed and the tester copes. This typically means it is difficult to control and observe the CUT. This approach will result in the highest

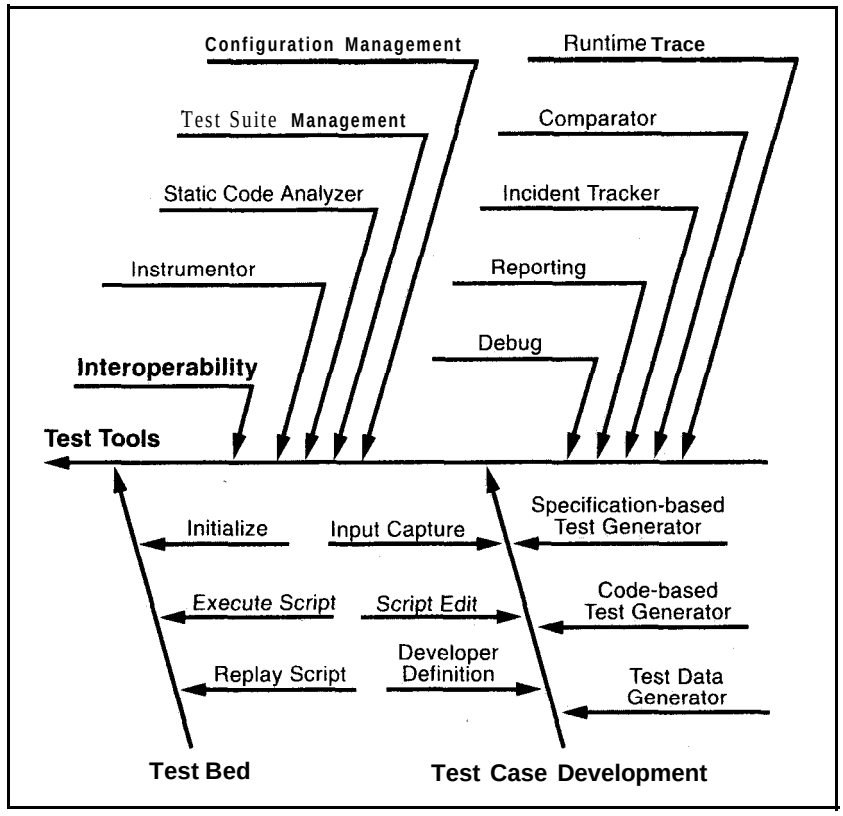


Figure 7. Testability: Test automation

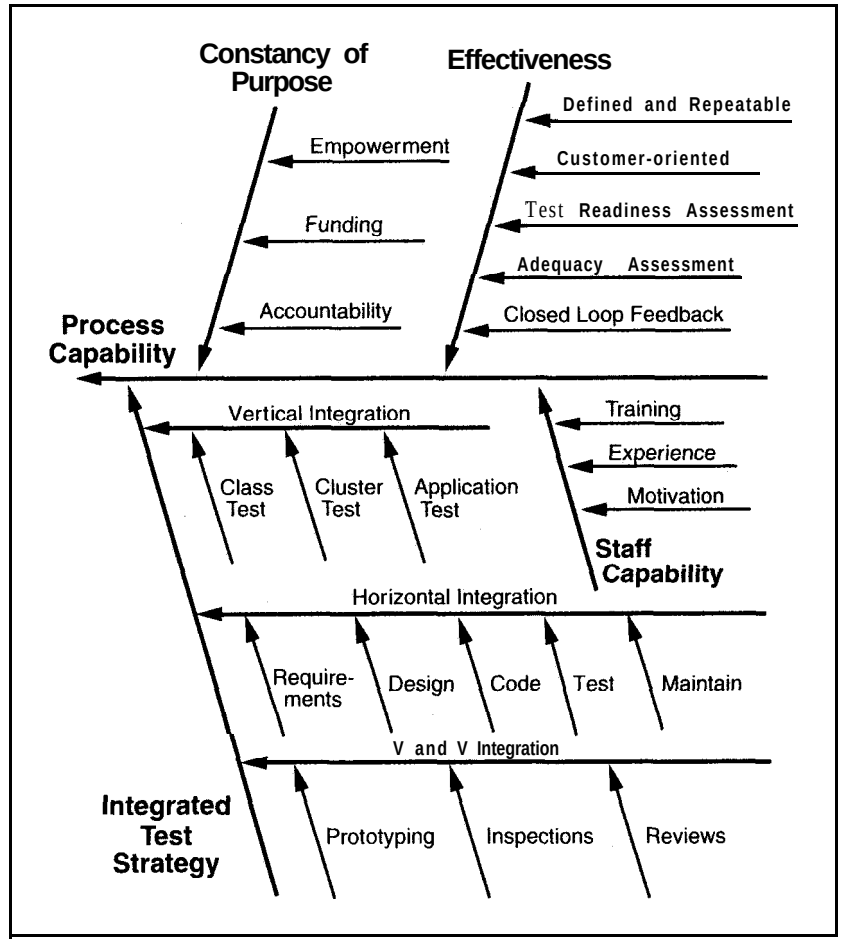


Figure 8. Testability: Process capability

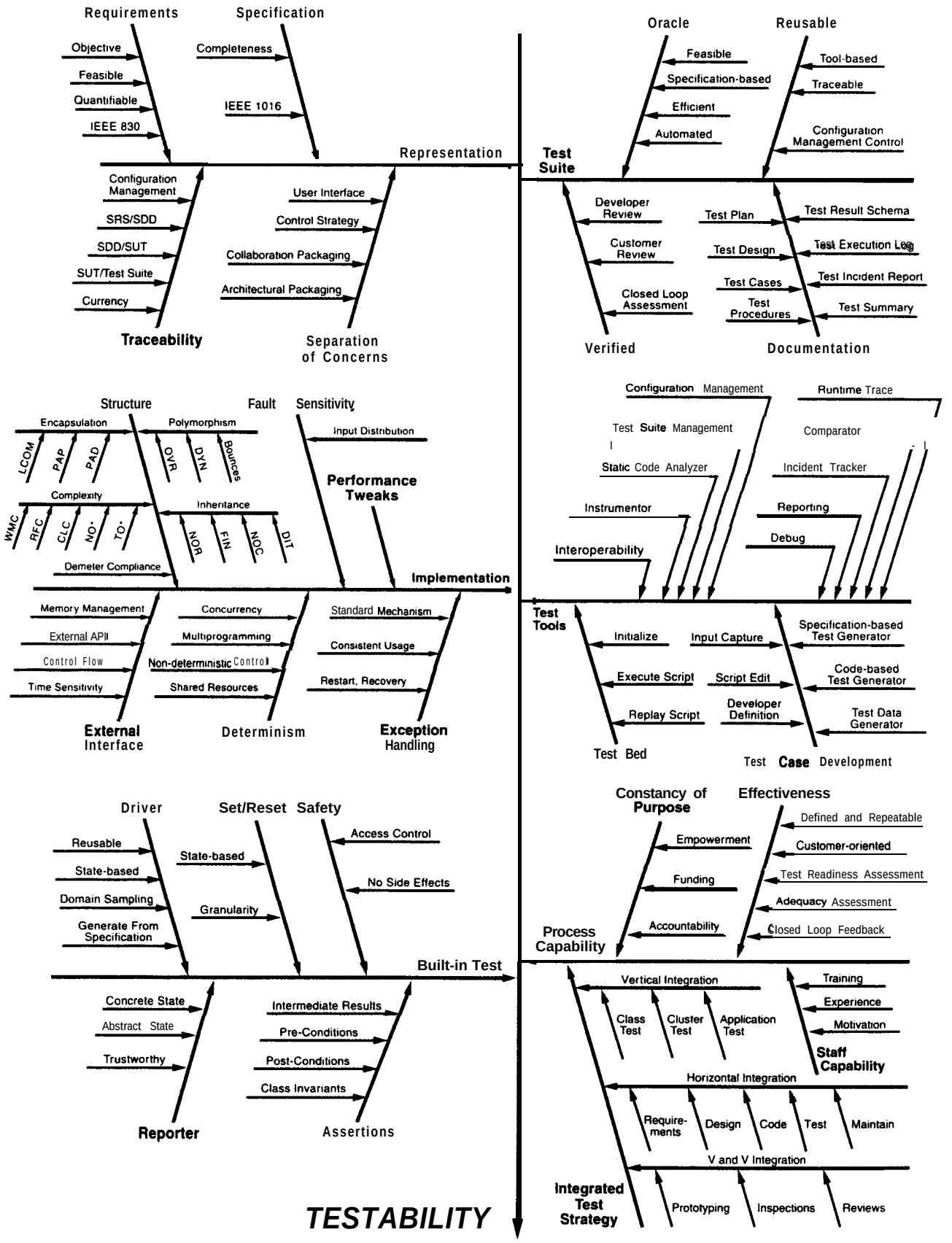


Figure 9. The testability fishbone: All facets

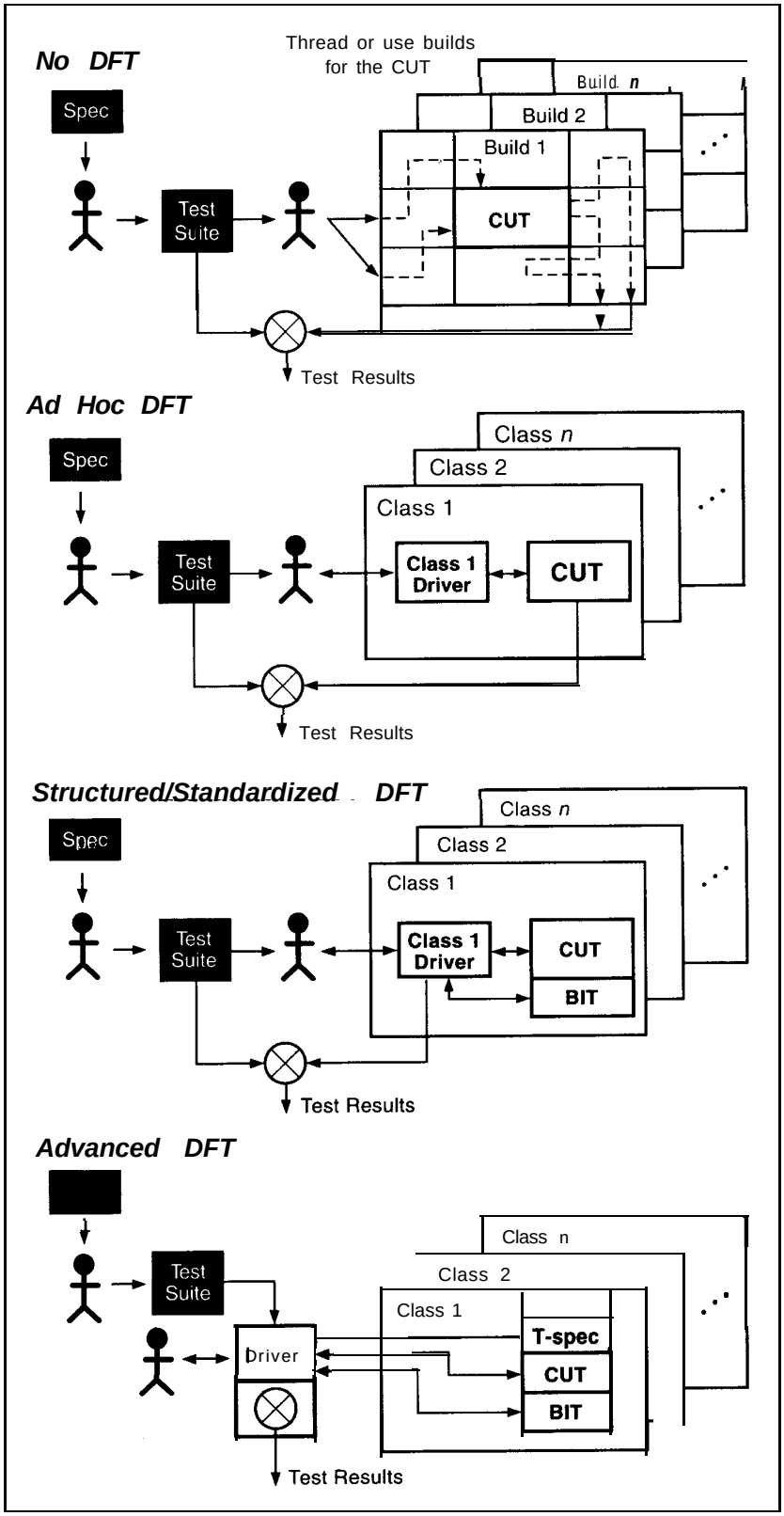


Figure 10. DFT configurations

possible test cost or least reliability, depending on which basic testing strategy is followed.

- *Ad hoc DFT.* Class testing is done by developing a driver for each class and stubs (as needed) for the class's servers. Assertions and other forms of inline instrumentation may be added. This improves controllability and observability. However, the absence of explicit set/reset can result in lengthy setup sequences and reliance on debug probes to verify state. This results in a relatively high test cost or less reliability, depending on the basic testing strategy.
- *Structured DFT.* The ad hoc strategy is augmented by including a standard BIT and BIT application programmer interface (API) in all classes. As discussed previously, minimal BIT requires state set/reset, state reporting, and BIT-safe implementation. This can offer complete controllability and observability. However, a single driver must be developed for each class. To develop the driver, the CUT interface is hard-coded in the driver. The requisite interface information is obtained by manual inspection of the CUT. Test cases must be manually prepared for each class.
- *Standardized DFT.* The agreement on and adoption of an industrywide VLSI standard for structured DFT improved IC interoperability and reliability, as well as reducing costs. Individual IC manufacturers with a high degree of vertical market integration practiced structured DFT long before the adoption of IEEE 1149. As the VLSI industry became less vertically integrated, however, interoperability needs provided a win-win economic opportunity, leading to the rapid and widespread adoption of 1149. An analysis of how a similar scenario might develop for commercial class libraries is beyond the scope of this article. However, within a single development organization, there are clear benefits to a standard test API for class drivers and class BIT: one API to learn instead of many, more time for better testing, and consistent test capability over the entire library.
- *Self-Test DFT.* The class testing analog to the structured strategy for calls for a driver to be paired with every class. This has several negatives. With n classes to test there are

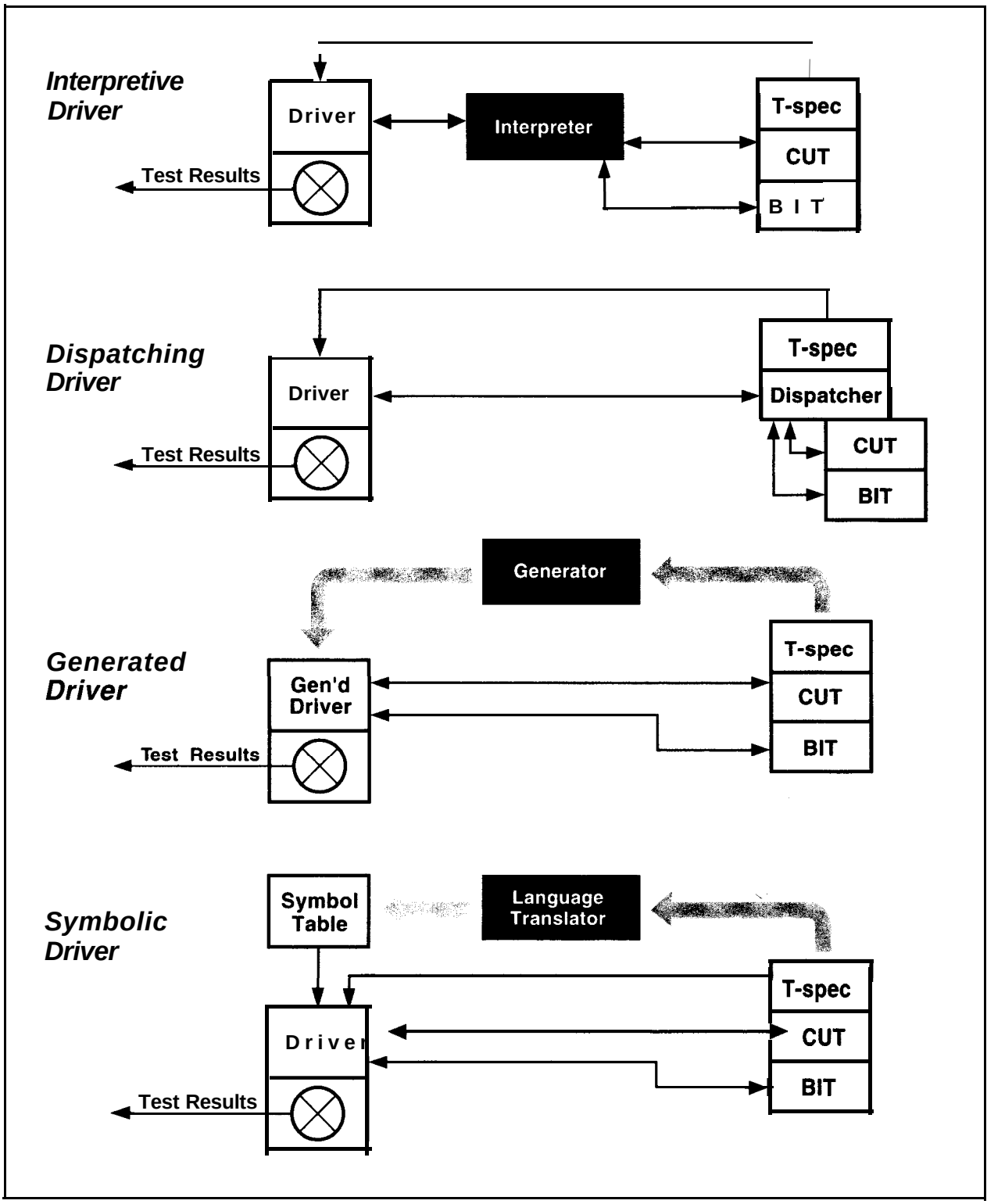


Figure 11. Advanced DFT architectures

now n additional interfaces to synchronize, the additional development and maintenance effort for every class driver, and limited automation of test case generation, execution, and evaluation.

The use of an implementation as the exclusive basis for automated testing is essentially tautological and therefore insufficient. There must be a specification and a means to generate expected results from the specifi-

```
t_spec
        = class_name
        + class_invariant          – Boolean expression on all instance variables.
        + {super_class
        + initial_state            – Is a state
        + {state}
        + {transition}

state
        = state_name
        + post_condition           – Boolean expression on all instance variables.

transition
        = accepting state_name     – Is a state
        + method_name
        + pre_condition            – Boolean expression on all instance variables
                                     and arguments.
        + resulting_state_name     – Is a state
        + activation_sequence      – Methods activated in this transition
```

Figure 12. **FREE test specification**

cation. Ideally, a single generic driver would automatically generate a test suite from an embedded test specification (*t-spec*), activate the CUT with this test suite, and evaluate the results. The need for a driver paired with every class is obviated. The self-test procedure would consist of the following steps:

1. The driver requests the t-spec from the CUT.
2. The driver generates a test pattern from the t-spec.
3. The driver generates test case values by splattering the input parameter domains (random sampling of from off, on, and in regions in the domain).
4. The driver uses set/reset to control the state of the CUT.
5. The driver sends the test case message(s) to the CUT.
6. The driver evaluates the response of the CUT.
7. The driver evaluates the state of the CUT with the reporter.

The feasibility of this scheme depends on solving two problems. First, the test pattern method names and arguments must be bound to messages accepted by the CUT. The manual step of inspecting the CUT to prepare the requisite interface must be automated. There are several general solutions to the interface binding problem, which are depicted in Figure 11. Further discussion of elements of Figure 11 follows:

- Interpreter. The test specification exported from the CUT contains the names necessary to create a bindable CUT message. The driver uses these names to generate a program passed to the interpreter. The interpreter accomplishes the binding in the normal manner.
- Test dispatcher. Instead of activating features in the CUT directly, the driver activates test methods in the CUT which in turn activate application methods in the CUT. The same test method interface is used throughout the application system. The dispatcher selects the actual message interface to use and then sends the message. Control is returned to the BIT dispatcher, and then back to the driver.
- Driver generator. Instead of attempting to drive the CUT directly from a generic driver, the generic driver reads the CUT source code and generates an intermediate driver to activate the CUT under control of the primary driver. This scheme has been investigated in the ASTOOT system [13].
- Language extension. The language translator is modified to accept syntax extensions for test specifications. After translation, portions of the symbol table and the test specification are saved as a separate file that can be directly processed by the driver. The scheme is essentially an extension of the technique used to run programs under a symbolic debugger. Thus, automatic test generation could be developed by simply using the same input as a symbolic debugger.

A suitable test specification language with respect to some testing methodology must be constructed. Figure 12 provides a sketch of a specification language to automate the state-based testing approach in the FREE methodology [9]. Inclusion of such a specification would not impose an onerous coding burden. For example, the Eiffel language provides an elegant executable specification for pre-conditions, post-conditions, and class invariants [19]. Only state control information needs to be added. Inclusion of an explicit test specification would also serve other important software engineering goals:

- Provide an explicit, specification-based test plan for every class.
- Prevent fragmentation of specification information and the implementation.
- Provide an explicit, readable specification embedded with the implementation. This would facilitate reuse and understandability.

The BIST (built-in self test) approach has limitations-it is not a replacement for all forms of testing. Unless a general solution for an automated oracle is found (this is not likely), manual preparation of test cases will continue to be necessary. The test specification language in Figure 12 can define the bounds of acceptable input and output, but does not permit determination of the exact correctness of a given input and output. For example, given only the pre- and post-conditions for a square root function, we could not decide whether any given value was indeed the square root of the argument (e.g., is 4.25 the square root of 18.0625?), but we could decide that was within the bounds (post-conditions) of a square root function. The t-spec should be consistent with all other representations. A full discussion of considerations and implementation of automatic consistence support is beyond the scope this article.

Conclusion

Testability reduces total test cost in a reliability-driven process and increases reliability in a resource-limited test process. Thus, regardless of the testing strategy employed, it is in the interest of a development organization to improve testability. Testability of an object-oriented system is a result of six primary factors: representation, implementation, built-in test, the test suite, the test support environment, and process capability. Each influences controllability and observability of the implementation, or basic operational effectiveness of the test process. Nearly all the techniques and technology for achieving high testability are well established, but require financial commitment, planning, and conscious effort. The advanced built-in test capabilities sketched here do not yet exist, but are feasible with existing technology. C

References

1. ***ANSI/IEEE Standard 830-1 984: Standard*** *for Software* ***Requirements*** *Specifications.* The Institute of Electrical and Electronic Engineers, New York, 1984.
2. ***ANSI/IEEE*** Standard *829-1983:* ***IEEE Standard*** *for* ***Software Test Documentation.*** The Institute of Electrical and Electronic Engineers, New York, 1987.
3. ***ANSI/IEEE Standard 1016-1 987: IEEE*** Recommended Practice *for* Software ***Design*** *Descriptions.* The Institute of Electrical and Electronic Engineers, New York, 1987.
4. ***ANSI/IEEE*** Standard ***1149.1-1 990: IEEE Standard Test Access Port and*** Boundary-Scan ***Architecture.*** The Institute of Electrical and Electronic Engineers, New York, 1990.
5. Beizer, B. ***Software*** System ***Testing*** and Quality Assurance. Van Nostrand Reinhold, New York, 1984.
6. Beizer, B. Software ***Testing Techniques,*** Second ed. Van Nostrand Reinhold, New York, 1990.
7. Berard, E. ***Essays on Object-Oriented Software Engineering.*** Prentice-Hall, Englewood Cliffs, N.J., 1993.
8. Binder, R.V. ***Testing Object-Oriented Programs: A Survey.*** RBSC-94-002. Robert Binder Systems Consulting, Inc., Chicago, 1994.
9. Binder, R.V. ***The FREE*** *Approach to* ***Testing Object-Oriented Systems.*** RBSC-94-003, Robert Binder Systems Consulting, Inc., Chicago, 1994.
10. Chidamber, S.R. and Kemerer, C.F. A metrics suite for object-oriented design. ***IEEE*** Trans. ***Softw.*** Eng. 20, 6 (Jun. 1994), 476-493.
11. Cox, B.J. Planning the software industrial revolution. ***IEEE Softw. Nov.*** 1990, 25-33.
12. Deming, W.E. ***Out of the*** *Crisis.* MIT Center for Advanced Engineering Study, Cambridge, Mass., 1986.
13. Doong, R.K. and Frankl, P. ***The*** *AS-TOOT approach to testing object-oriented* programs. Tech. Rep. PUCS-104-91. Polytechnic University, Brooklyn, N.Y, 1991.
14. Freedman, R.S. Testability of software components. ***IEEE*** Trans. ***Softw. Eng.*** 17, 6, (June 1991), pp 553-64.
15. Hoffman, D. Hardware testing and software ICs. ***Proceedings, Northwest Software*** *Quality* ***Conference*** (Portland Ore., Sept. 1989), pp 234-22.
16. Hoffman, D. and Strooper, P. A case study in class testing. ***In Proceedings*** *of CASCON 93.* IBM Toronto Laboratory, Oct. 1993, pp 472-482.
17. Jacobson, I, Christerson, M., Jonsson, P. and Overgaard, G. *Object-Oriented Software Engineering.* Addison-Wesley, Reading, Mass., 1992.
18. Lieberherr, K.J. and Holland, I.M. Assuring good style for object-oriented programs. *IEEE Softw.* (Sept. 1989), pp. 38-49.
19. Meyer, B. *Object-Oriented Software* ***Construction.*** Prentice-Hall, Englewood Cliffs, N.J., 1988.
20. Ogata, K. ***Modern Control*** Engineering. Second ed. Prentice-Hall, Englewood Cliffs, N.J., 1990.
21. 00 tool aids software testing. ***The Outlook.*** Fall 1993, McCabe & Associates, Columbia, Maryland.
22. Ould, M.A., and Unwin, ***C. Testing*** in ***Software*** Development. Cambridge University Press, Cambridge, UK, 1986.
23. Paulk, M.C., Curtis, B., Chrissis, M.B., and Weber, C.V. *Capability* ***Maturity Model for*** Software, ***Version 1.1.*** CMU/SEI-93-TR-24. Software Engineering Institute, Pittsburgh, Pa., 1993.
24. Voas, J., Morell, L., and Miller, K. Predicting where faults can hide from testing. ***IEEE Softw.*** (March 1991), pp 41-8.
25. Weyuker, E.J. On testing non-testable programs. ***Comput.*** *J.* 1982 25, 4, pp 465-70.

About the Author:

ROBERT V. BINDER is president of RBSC, Inc. He is researching state-based testing for reliability engineering in object-oriented systems and a systems engineering methodology for client-server development. **Author's Present Address:** RBSC, Inc., 3 First National Plaza, Suite 1400, Chicago, IL 60602; email: rbinder @chinet.com