

Component Testing with Intelligent Test Artifacts

Michael Silverstein
SilverMark, Inc.

msilverstein@silvermark.com
<http://www.silvermark.com>
(919) 858-8300 x29

Component testing is the act of subdividing an object-oriented software system into units of particular granularity, applying stimuli to the component's interface and validating the correct responses to those stimuli, in the form of either a state change or reaction in the component, or elsewhere in the system.

Often, component testing is performed by developers who's primary mental focus is that of the system under test and component tests to be performed. In most cases, the test case architecture, if any, evolves as an afterthought, driven mostly by the tests themselves. Some developers who have been through this process more than once write their own test frameworks.

In the last few years, design patterns have become a currency for communicating common problems and their solutions within a context. Testing is no exception. Publications such as [Binder 99, McGregor 99, Firesmith 96, Beck 94] have documented some common design solutions to component test automation problems in terms of design patterns.

This article presents a set of design patterns commonly encountered when creating automated test frameworks and application domain specific test cases, and introduces the notion of **test artifacts** as a test component architectural model for implementing those patterns.

This article begins by presenting several patterns for automated component test framework design, some familiar and some not, with implementation strategies. It then shifts focus from test framework design to test case design. And briefly applies the solutions discussed earlier as a basis for test case implementation.

Common component test design questions

The challenge to test framework developers is to recognize the common problems and associated implementation activities, to the extent that a scalable and easily maintainable automated test architecture can be applied as a foundation for implementing tests. Our experience has shown that for most types of component testing, the component test developer must eventually find answers to one or more of the following questions, and possibly the questions that their answers pose:

- How do I create a specifically configured instance of a component under test?
- How do I manage the application of a stimulus to the component under test?
- How do I organize stimuli into reusable groups of arbitrary granularity?
- How do I validate the state of the component and system under test after one or more stimuli?
 - How do I manage reference objects?
 - ♣ How do I compare references objects to target objects, so that just the states that are important are compared?
 - How do I intelligently traverse the state of my object under test
- How do I validate that expected exceptions are raised.
- How do I manage variations on test inputs in order to drive different paths with the same test stimuli?
- How do I map development activities to the tests that validate their correctness?

Pattern responses to design questions

This section presents component test design patterns, with implementation strategies. An effort was made to keep implementation strategies language neutral, however, wherever use of code is unavoidable examples are presented in the Java™ programming language.

Pattern: Test Component

Intent: Implement a reusable component model to represent aspects of automated testing of arbitrary granularity.

Context

Generic component architectures are not new. For example, Java™ has Java Beans™, and its variants. To date, the model for component tests has been limited to units of gross execution, such as the test suite and test case [Beck 94, Fowler 99], which is useful, but falls short in the role of a general organizational and behavioral principal for automated testing. What is required is a higher level of abstraction, under which specific variants may serve as architectural building blocks for particular aspects of automated testing.

Applicability

- This pattern is indicated at many levels of testing. Test components may express test organization or behavior and are in effect, miniature, automated testing problem specific frameworks.

Strategy/Implementation

- Create a hierarchy of test component classes, called **test artifacts**, each of which embodies specific test execution behavior, as well as the properties required to configure instances for specific applications of the artifact.
- Artifacts may be singular or composite. The specification for a composite artifact contains specifications for other artifacts.
 - A specification for an artifact indicates a specific configuration of an artifact instance, that is, an artifact instance whose properties are set in a certain way. The combination of an artifact type and the specific configuration of an instance of it results in a reusable component whose behavior is applicable to a certain test-related task.
 - Reusability is achieved with the creation of an artifact type that can locate a specifically configured artifact instances based on some well known identifier. An artifact reference is itself, a specifically configured artifact instance.
 - Composite artifacts execute by executing each contained artifact, in turn.
- Common artifact configuration properties include:
 - **Name** – Used to identify an artifact so that it can be located.
 - **Description** – More extensive descriptive information about the artifact, either as text or a reference to some external document.
 - **Expected exception** – A reference to an exception that is expected to be signaled as a result of execution of the artifact. Execution of the artifact is said to have failed if the exception is not raised.
 - **Precondition action** – Actions to set and/or verify the state of the system under test prior to further execution of the artifact.
 - **Invariant action** – A set of assertions that express general consistency constraints that apply to every class instance as a whole, regardless of stimulus. If specified, this is executed after execution of the artifact.
 - **Synchronization action** – An action that checks for some state transition that indicates the completion of some asynchronous transaction. In the case of an artifact that initiates an asynchronous transaction, the intent of the synchronization action would be to is to pause execution of a test until the transaction has completed.
 - **Classification of organizational intent** – such as, use case scenario [Jackobsen 93], requirements implementation, fix for known problem, etc.
- The specification for an artifact's configuration should be defined one and only one place, to ensure that multiple definitions of equivalent artifacts do not become a maintenance problem.

Consequences

- Abstracting typical testing patterns into a palette of reusable, individually configurable component types that embody common testing behavior enables test developers to focus less on test implementation and more on the problem of exercising the system under test.
- Generally, automated component test frameworks offer class or method level granularity to test developers. Artifacts offer arbitrary levels of test organization beyond the traditional class or method level. One or more

specifically configured test artifacts may represent a complete test. At minimum, a complete test may be as simple as instantiating a single class, or it may incorporate multiple artifacts as building blocks.

- Test artifacts support either a bottom-up test development process, where basic reusable artifacts are created first as components to be used in more broadly scoped artifacts, or a top-down development process, where broadly scoped artifacts reference utility artifacts that are not yet completely defined.
- Test artifacts benefit test tool writers because artifacts present a facade for potentially complex objects with a high degree of functionality, and places test behavior in the test framework instead of the test cases.
- As in most systems that employ a high degree of reuse, test development speed increases as more reusable components become available.
- Defining test artifacts in one and only one place eases maintenance because changes to test code due to changes in the system under test are localized.

Known Uses

There are many precedents for implementing reusable components outside of the automated component testing domain.

Pattern: Test Sequence

Intent: Present a sequence of test actions of arbitrary granularity as a discrete, reusable unit.

Context

A prerequisite for creating tests via composition of components is a container in which to hold them. Because of the sequential nature of tests, this container must be a composite of sequentially executed components. It is also extremely helpful to be able to subdivide tests into conceptual units for the purpose of enhancing readability and understandability.

Fine-grained executable units enable automated testing tools to better track test execution progress, and enable the tools to provide better test editing interfaces for users by allowing different task specific views of individual actions to be performed.

Applicability

This pattern is indicated wherever test creation through composition is to be employed, or where it is advantageous to subdivide a test into sequentially executed units.

Strategy/Implementation

Create a composite **test sequence** artifact that is composed of specifications for other artifacts. The execution behavior of this artifact is to execute each of its components in a well defined order under specified rules. For example, one may implement an **abort on failure** rule to provide a convenient shorthand for implementing a scheme where execution ceases on the first component failure encountered. Instead of repeatedly coding a check-and-return-if-fail after each component's execution, the artifact embodies the intelligence to manage the execution flow as specified by the rule.

Consequences

By simply extending the test artifact concept to include a composite artifact type, test creation through composition is achieved.

Because composite artifacts are responsible for sequencing through their components during execution, they may also embody general execution rules, such as ceasing execution on failure. Execution rules such as this are often used to dictate the level of granularity of the actions to be performed within the sequence.

Pattern: Configured Instance

Intent

Provide an instance of an object configured to a particular state.

Context

One of the foundations of component testing is the use of objects in a particular state as:

- inputs to individual stimuli (parameters)

- comparison targets (gold standard objects)
- test case starting state

During the process of writing component test code, creating instances of test input objects is a tangential activity that is distracting from the overall goal of stimulating the component under test, and often where a majority of the test development effort is expended.

Unless the interface for an object is accessible exclusively via a global, static namespace, it is likely that instances of the object, and the objects that it recursively contains, must be instantiated or extracted from persisted location.

Furthermore, there are usually multiple ways of instantiating an object, so there needs to be a way of encapsulating the appropriate instantiation strategy for the component so it can be easily referenced and applied to a test. For example, Java and C++ classes may offer multiple constructors for instantiation. Smalltalk offers a much looser interface. This pattern offers a solution to encapsulating instantiation or materialization of test objects as a reusable operation.

Strategy

A good, basic approach is to implement methods in some test class that return specifically configured instances of test objects. This test class is sometimes referred to as an Oracle [Binder 99].

A further abstraction would be to implement an artifact that, given a specification, returns a configured object instance. This may be implemented as an extension of the *test sequence* composite artifact, with the additional behavior of initially instantiating or retrieving the object under consideration. Once an object is present, it may be immediately returned or first acted upon by one or more actions to be performed in sequence.

Instantiation approach

This approach assumes instantiation of an object of some class. The specification requires:

- The class to instantiate
- A constructor method. The constructor may require recursive references to other configured objects, as in [Siepmann 94].
- Additional stimuli to apply to the newly instantiated object, possibly in the form of references to one or more existing scenarios.

The artifact needs to implement behavior to instantiate the object according to the specified constructor, recursively requesting any artifacts specified for constructor parameters to provide values, execute any additional stimuli on the object, and return it.

Materialization approach

This approach assumes the object exists in some persistent state and may be retrieved via a given strategy. The specification requires:

- The location (table/database/file) of the object

The artifact needs to implement behavior to retrieve the object, most likely through some framework interface, execute any additional stimuli on the object, and return it.

Known Uses

ObjectCreator in [Siepmann 94] implements a strategy for creating instances of test objects as specified by an object description language.

Related Patterns

This pattern is in many ways similar to the Builder pattern, which separates the construction of an object from its representation. Configured Instance certainly separates construction from implementation, however, the focus of this pattern is to encapsulate any operations that are required to provide a configured instance of an object, which may include implementation of the Builder pattern, or it may simply require materializing an externally persisted object.

Pattern: Test Stimulus

Intent: Encapsulate a stimulus/response validation operation for a component under test for singly or multiply threaded

systems.

Context

- The basic unit of execution of a test case is the application of a stimulus.
- The next level of organization for testing contains multiple discrete stimuli for a particular purpose – a use case scenario.
- Optionally, the response to a stimulus may be validated in terms of a change in the state of the component under test, or some area of the system under test, as well as an object returned directly by the stimulus.
- In multithreaded systems, the responses to a stimulus may be temporally non-deterministic.

Applicability

This pattern is indicated in any situation where a discrete stimulus or related sequence of stimuli need to be applied to a component, with response validation.

Strategy/Implementation

Create an artifact that embodies any or all of:

- A stimulus on some object defined as one of:
 - a reference to a target object and the action to perform on the target object (message send)
 - The selector for a method on a test class that will perform the stimulus action when requested
 - A reference to an artifact that will perform the stimulus action when requested
 - Some other executable representation of the stimulus
- precondition state validation
- postcondition state or return value validation
- invariant validation
- a synchronization mechanism for suspending test execution after executing the stimulus, until an expected state is sensed, or time limit is exceeded so that long running asynchronous transactions do not

Instances of implementors of this pattern are typically components of *test sequence* artifacts or test sequence artifacts themselves. As such, validation properties may be specified as references to validating artifacts. Precondition, postcondition and invariant validation, as well as state synchronization are all specified under the general definition common to all test artifacts, so no special work is required to implement those for this pattern.

Related Patterns

Command in [Gamma+ 95] describes a pattern for encapsulating requests. Test Stimulus differs because it also encapsulates expected response validation, and synchronization.

Pattern: Nested State

Intent: Provide a mechanism for specifying and traversing a chain of object references to extract a specific state.

Context

Many components present a primary interface through a single object that typically follows some implementation of the façade pattern, as described in [Gamma+ 95]. If testability were the primary motivating design force, all public sub-object interfaces and states for a component would be promoted to the interface of this façade object so they were easily accessible for validation. Unfortunately, this may lead to code bloat and the need to maintain methods that do not contribute to the overall purpose of the system beyond aiding testability.

Strategy/Implementation

Implement an artifact that provides a means to specify an access scheme for a particular state of a given object. This scheme may be a reference to a method that implements a daisy chained traversal of the objects, for example, `return`

```
myObject.x().y().z();
```

Another possibility would be to build on a composite artifact whose sequence of stimuli is used to access successive states in the chain. This is a little less straightforward, however it does present the opportunity to build off of a state returned by an artifact reference. For example:

In the definition for the artifact that returns state 'z', each component applies a stimulus to the result of the execution the previous component.

Consequences

A configured nested state artifact becomes a permanent, reusable access method for a particular state within a component. Tests that need to validate this state need only reference this artifact by its root artifact/name ordered pair, passing the object under consideration. If the path to the state ever changes, only a change in the artifact's configuration is needed.

Known uses

VisualWorks Smalltalk uses what is known as an access path in the ProtocolAdaptor class to describe a path to a particular object from a root object in terms of a collection of message selectors to be sequentially sent to the root object and subsequently the objects returned.

Pattern: Aggregate State

Intent: Represent the state of a component in terms of an aggregate of the states of its subcomponents. An aggregate state may or may not represent the complete, recursive state of the component and all each subcomponent.

Context

Most components, with the exception of primitives, can be described as a federation of one or more objects that are either contained by or collaborate with the component. Broadly speaking, each of these objects contributes its own state to that of the whole. The recursive state of a component is the value set of instance variable values obtained by recursively expanding each instance variable value [Binder 99]. A component has one *complete* recursive state, but may have many aggregate state representations, or *views*, depending on which states are of interest. For example, one might choose to ignore the current time state of a component for comparisons so as to avoid mismatches due to test execution at different times.

Applicability

This pattern represents a strategy for specifying the subcomponent states of interest, traversing the subcomponents to extract those states, and representing multiple states as a single state.

This pattern is applicable when a single, representation of all or some subset of the entire recursive state of a component is required, such as for validation purposes.

Strategy

Populate a composite **aggregate state** artifact with components that, when executed, extract individual states from the

object under consideration. *Nested state* artifacts lend themselves well to this purpose. The execution time behavior of this aggregate state artifact is to gather and return a collection of the resultant execution values each of its components as a flattened representation of the total aggregate state of the object under consideration.

Implement comparison operators that take into account the complete set of states

Consequences

A configured aggregate state artifact acts as a reusable representation of a particular view of an object's state. Used in conjunction with nested state artifacts, the state of an entire object tree may be captured. This type of artifact serves as a useful shorthand for comparing objects. For example, for a particular view of the state a root object and object tree one might normally code something like this in Java™:

```
return customer.x().equals(referenceCustomer.x()) &
       customer.x().y().equals(referenceCustomer.x().y()) &
       customer.x().y().z().equals(referenceCustomer.x().y().z())
       ... etc.
```

Given an aggregate state artifact instance configured for some set of states on an object of a particular type, comparison of an object and a presumably equivalent reference object is a matter of instantiating the artifact for the objects to be compared and then sending equals() to the artifacts.

```
(new MyArtifacts().getArtifactNamed(
    "General Customer State").subject(customer).equals(
    (new MyArtifacts().getArtifactNamed(
        "General Customer State").subject(referenceCustomer)));
```

Note: This example syntax for instantiating an artifact is not mandatory

Known uses

Any object that externalizes referenced or contained object states through delegation

Related Patterns

- The Memento pattern captures and externalizes an object's internal state [Gamma+ 95].
- Implementations of this pattern make use of implementations of Nested State for specifying and extracting states.
- The façade pattern [Gamma+ 95] presents a single interface for a cluster of objects.

Pattern: State Impression

Intent Express a particular view of the state of a component in a compact, but human readable form

Context

By and large, simple objects like strings and integers are much simpler to compare to one another than complex objects, and also much simpler to capture and store for later reference. Simple objects also have the added benefit of being easily recognizable by the human reader, which often proves advantageous when debugging. It becomes advantageous then, to be able to easily generate a simple representation of a view of a component's state so that rapid comparison to reference state can be accomplished. A view of a component's state means the union of some, but not necessarily all the instance state of the objects that comprise the component. The term *Impression*, is used to mean a simple representation of a complex object's state, in much the same way that artistic Impressionism conveys an impression of its subject without a detailed rendering.

Applicability

This pattern is indicated in situations where simple object state views are required, typically for comparison or debugging purposes.

Strategy/Implementation

Implement one method for each view of the states of interest. Within each method:

- Employ a strategy for extracting the required states. The aggregate state pattern describes a scheme for performing this.
- Append String representations of those states to a String or Stream, as well as any other syntactic sugar required to make the information more readable, and return.

The aggregate state artifact may be easily extended to include a `toString()` method that gathers its composite states in a String form. Because artifacts include a name property, it is conceivable that the names of the component states might be used in the impression's String representation.

Known uses

Implementations of impressions include `#printString` and `#asString` in Smalltalk, as well as `toString()` in Java.

Related Patterns

Implementations of this pattern may make use of Nested State for specifying and extracting states.

Pattern: State Validation

Intent: Assert that a component is in an expected state. Define this assertion as a reusable component.

Context

State validation involves comparing object state to some known, expected state. Validation of expected state may be encoded in several ways:

- as a comparison to one or more *Gold Standard* reference objects
- as simple comparisons to primitives or literals
- as a test for some known object's inclusion or exclusion within a collection type
- as a comparison between object impressions

Applicability

- Use this pattern when the state of an object at a particular moment requires validation.
- Use this pattern when scaling up to broadly scoped system-level validations from individual object validations.

Strategy/Implementation

A common strategy is to implement an *assert(boolean)* test API [Beck 99] that logs the success or failure of a contained expression, during test execution. This is a simple, but flexible solution that is applicable to many cases.

A common test artifact property discussed earlier is a postcondition validation action, which embodies a definition for some validation action to be performed on artifact execution completion. This has the advantage of tying the artifact's action to an immediate validation of its success. Having this kind of close, structural relationship between stimulus and response validation enhances understanding of the test, especially if the test framework provides user interface or results reporting that maintains the link between the two.

Possibilities for defining a validation action may be:

- an expression that evaluates to a boolean. This is similar to the assert API discussed above, which would most likely be called under the covers during execution.
- a reference to an artifact that performs the validation. In situations where the same validation is performed repeatedly (invariants, for example) defining and then referencing an artifact whose sole purpose is validation of a particular object type's state becomes advantageous, especially if many state comparisons are involved. A similar alternative, discussed above, is to implement an *aggregate state* artifact and use it as a comparison wrapper for an object under consideration and a reference object.

Consequences

A useful application of validation artifacts is to create a family of them to validate common, expected states. Consider a simple example of a *Loan* object with states, new, overdue, and complete, among others. Once artifacts have been created that validate the configurations that these states embody, they may be easily 'plugged in' as postcondition

properties, as appropriate. When the internal representation of a state changes, it is a simple matter to update the validation artifact's specification without regard to whether it is referenced in one or one hundred places.

Collaborators

Implementors of the nested state and aggregate state patterns may be used to extract state.

Pattern: Object Variation

Intent: Organize configured object variations as a single, reusable resource.

Context

When testing components at the interface level, the path from component input to output often varies depending on input values. Therefore, it is advantageous to be able to pass input variations to a test for the purpose of driving execution of the different paths, both inter-object and intra-object. Ideally, one would be able to describe a specific set of variations for a particular class, referencing that configuration as needed. Any reference to a set of variations in a test case would automatically result in execution of that test case across each value.

Strategy/Implementation

Design an *object variation* test artifact that presents an interface for enumerating, or sequentially accessing its values.

Values may be provided from references to configured artifacts, calculated values or any other values.

Any composite artifact that implements a sequential access mechanism, and whose components return useful values may serve as an object variation artifact. Components that are likely to return useful values are *configured instance* artifacts, and *stimulus* artifacts.

Another approach would be to implement an artifact that implements a sequential access mechanism (an Enumerator, for example) and generates values based on some particular strategy. Some simpler strategies are generating an Integer interval, or random values. Thanks to encapsulation, as long as an object variation artifact implements the required interface, the source of the objects is irrelevant. The concept may also be extended to support retrieval of objects from a database.

The essence of the solution is to extend the test execution framework to accept an object (object variation artifact) that implements a sequential access mechanism as an execution modifier, whereby execution of a given test element is repeated over the elements returned by the object variation artifact. The variation artifact implements a strategy for generating or retrieving objects, in some cases based on its own particular configuration. In the example of the Integer interval, that may configuration may be an upper and lower bound. For database retrieval, it may be the database, table, and some SQL.

The final implementation consideration concerns the case where multiple variation artifacts are specified. In this case, the elements provided by each of the variation artifacts must be mixed according to some consistent and desirable rule (one-by-one, each-with-every, etc.). The recommended approach is to encode the rules in a hierarchy of policy objects to be used by the execution framework to govern retrieval of variation artifact elements. The advantage to encoding this behavior in a separate class is that instances may be plugged in to the execution framework as appropriate to the given situation, as an execution behavior modifier.

Consequences

Useful variations are themselves reusable units that may be shared among test developers.

Even though any artifact that implements a sequencing interface may serve as an object variation provider, it is important to note that care should be taken that the objects that are returned are appropriate for use and of the correct type.

Related Patterns

The Oracle pattern [Binder 99] describes approaches for providing test data to tests.

Applying artifacts

This article introduced several test artifact types as responses to common automated component test design patterns. Test artifacts may be used as an architectural foundation for implementing automated component tests. Test artifacts are implemented by classes whose instances embody particular execution behavior. The behavior for an artifact instance is affected by the configuration of its properties.

In practice, artifacts are instantiated, configured and returned by test methods within classes organized by purpose. Traditionally classes known as test suites [Beck 99, Binder 99], provide specifications for test cases and supporting methods. This class-based organization is applicable to artifacts as well, where each artifact is represented by an instance method.

For unit tests, a suite class may appear in a hierarchy designed to mimic the hierarchy of the classes under test [McGregor+96]. For example, consider object, **A**. A parallel test suite class **A_T** may be defined for the purpose of testing **A**, with subclasses of **A_T** following along subclasses of **A**, as appropriate. Using this architecture one is encouraged to implement methods that return artifact instances configured specifically for the class under consideration.

In the above diagram, **ClassA** and **ClassB** are classes under test. **ClassATest** and **ClassBTest** are test suites arranged in a hierarchy that parallels the classes they test. **ClassATest** implements methods (not indicated) that each return a configured artifact instance. The test artifacts are:

"Configured instance of A #1"	A configured instance artifact - configured to, when executed, return an instance of ClassA in a particular state.
"Configured instance of A #2"	A configured instance artifact configured to, when executed, return an instance of ClassA in a particular state that is assumed to differ in some useful and interesting way from the above instance
"Variations of A"	A variation artifact - provides object variations and is configured to refer to and enumerate the two configured instances above.
"Sequence of stimuli on instances of A"	A test sequence artifact - represents a sequence of stimuli on some instance of ClassA, that is, a unit test case. It is configured to take as input the "Variations of A" artifact, which means that the sequence of stimuli are performed on each of the two instances of ClassA defined above.

The above is a very simple, and in some ways contrived test, but it does show potential relationships between classes and artifacts. A more likely usage would be to employ variations of **ClassA** as input to constructors and other methods on some other **ClassX**. For example,

In the above example, a “Sequence of stimuli on instances of X” (a test case for **ClassX**), is passed, variations of **ClassA**. Though not shown here, the test case for **ClassX** uses instances of **ClassA** as parameters for constructors and other message sends on **ClassX**.

As a side note, for cluster or subsystem tests, it is less reasonable to organize according to the structure of the objects under test, and more likely that a suite class will appear in a flatter test hierarchy that takes a more functional perspective of the system under test.

Regardless, as a matter of policy, a well designed suite should always be targeted toward testing some component under test, [Dwyer 99] whether it is a single class, cluster, or subsystem.

Effectiveness

The approach described here is most effective when applied to tests for more than just a handful of classes. Our experience has shown that the value of building tests upon a framework of reusable test components such as the artifacts described here, increases as the scope of the tests and the number of reusable artifact definitions grows.

It is also important to note that tests and test components are development work products in themselves and should be designed along with the rest of the system under test. A common question developers should ask themselves as they design code, and that should certainly be asked at reviews, is “how do I prove that this works?” It is also important to publicize and share reusable test components, especially *configured instance* and *object variation* artifacts, as they become available.

Conclusion

Widespread adoption of component architectures in the software industry is helping increase developer productivity and software quality. Applying the same architectural principles and design for reuse to component tests as the components under test is a sensible approach.

This article has described a component architecture for automated component testing frameworks, based on a family of objects called test artifacts that move common testing behavior, frequently implemented within automated component tests, from the tests themselves to the test artifact objects. The benefits of this approach are

- Less time is spent implementing component tests because the framework implements common testing behavior
- The test structure suggested by the artifacts themselves guides test developers down well defined implementation paths, making tests consistent and providing test developers with design momentum.
- Because artifacts can be used to completely describe tests, they may serve as a basis for test tool implementation, and targets for automated test generation.

References and further reading

[Beck 94] Kent Beck, *Simple Smalltalk Testing*. The Smalltalk Report 4(2):16-18, October 1994

[Binder 99] Robert V. Binder, *Testing Object-Oriented Systems: Models, Patterns, and Tools*. Addison-Wesley Object Technology Series. 1999

[Dwyer 99] Graham Dwyer and Graham Freeburn, *Business Object Scenarios: a fifth-generation approach to automated testing*. In *Software Test Automation : Effective Use of Test Execution Tools*. Edited by Mark Fewster and Dorothy Graham Addison-Wesley Pub Co. 1999

- [Firesmith 96] **Donald G. Firesmith**, *Pattern Language for Testing Object-Oriented Software*, Object Magazine, Vol. 5, No. 8, SIGS Publications Inc., New York, New York, January 1996, pages 32-38.
- [Gamma+ 95] **Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides**, *Design Patterns : Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional Computing. 1995
- [Fowler 99] **Martin Fowler**, *A UML Testing Framework*. *Software Development Magazine*. April, 1999
- [Fewster 99] **Mark Fewster, Dorothy Graham** *Software Test Automation : Effective Use of Test Execution Tools*. Addison-Wesley Pub Co. 1999
- [Hetzel 84] **Bill Hetzel**, *Software Testing*, QED Information Sciences, Inc. 1984
- [Jeffries 99] **Ronald E. Jeffries**, *Extreme Testing*, *Software Testing and Quality Engineering* 1(2):23-26 March/April 1999
- [Jorgenson 94] **Paul C. Jorgenson and Carl Erickson**,. *Object-Oriented Integration Testing*. CACM, 37(9):30-38 September 1994
- [Kaner 93] **Cen Kaner, Jack Falk, Hung Quoc**, *Testing Computer Software*, International Thompson Computer Press 1993
- [Marick 95] **Brian Marick**, *The Craft of Software Testing*, Prentice Hall. 1995
- [McGregor 94] **John D. McGregor**, Testing Object-Oriented Systems Tutorial Notes In *The Conference on Object-Oriented Systems, Languages and Applications*, October 1994
- [McGregor 96a] **John D. McGregor and Timothy Korson**, *Integrating Object-Oriented Testing and Development Processes*. CACM, 37(9):59-77 September 1994
- [McGregor 96b] **John D. McGregor and A. Kare**, *PACT: An Architecture for Object-Oriented Component Testing*, in the Proceedings of The Ninth International Software Quality Week in San Francisco, California, SR Institute, San Francisco, 22 May 1996.
- [McGregor 99] **John D. McGregor** *Test Patterns: Please Stand By*, *Journal of Object-Oriented Programming*, 12(3): 14-19, June 1999
- [Murphy 94] **Gail C. Murphy, Paul Townsend, and Pok Sze Wong**, *Experiences With Cluster and Class Testing*. CACM, 37(9):39-47 September 1994
- [Myers 79] **Glenford Myers**, *The Art of Software Testing*, John Wiley & Sons, Inc. 1979
- [Poston 94] **Robert M. Poston**, *Automated testing from Object Models*. CACM, 37(9):48-58 September 1994.
- [Siegel 96] **Shel Siegel**, *Object Oriented Software Testing, a Hierarchical Approach*. John Wiley & Sons, Inc. 1996
- [Siepmann 94] **Ernst Siepmann, A. Richard Newton**: TOBAC: A Test Case Browser for Testing Object-Oriented Software. ISSTA 1994: 154-168