Increasing the Testability of Object-Oriented Frameworks with Built-in Tests

Taewoong Jeon[1] Sungyoung Lee[2] Hyonwoo Seung[3]

[1]Dept. Computer Science, Korea University
jeon@korea.ac.kr, +82-41-860-1346
208 Seochang-Dong, Chochiwon, Choongnam, Korea, 339-700

[2]Dept. Computer Engineering, Kyunghee University, sylee@oslab.kyunghee.ac.kr
[3]Dept. Computer Science, Seoul Women's University, hwseung@swu.ac.kr

Keywords: object-oriented framework, testability, built-in test(BIT), hook classes

Abstract: Object-oriented frameworks require thorough testing as they are reused repeatedly in developing numerous applications. Moreover, frameworks must be retested each time they are adapted and extended for reuse. Frameworks, however, have properties that make it difficult to control and observe the testing of the parts that were modified or extended. This paper describes a scheme of encapsulating test support code as built-in test (BIT) components and embedding them into the hook classes of an object-oriented framework so that defects caused by the modification and extension of the framework can be detected effectively and efficiently through testing. The test components built into a framework in this way increase the testability of the framework by making it easy to control and observe the process of framework testing without incurring changes or intervention to the framework code.

1. Introduction

One of the current trends in object-oriented technology is to develop software as a framework [1,2,3]. A framework supports efficient software development by providing a pre-implemented architecture common to a family of applications together with hot spots designed to be adapted to each particular application. An object-oriented framework consists of various cooperating abstract and concrete classes [4]. Some of the classes in the framework are designed as hook classes to serve as the hot spots that can be adapted and extended within the limits the architecture permits. That is, the framework is reused in application developments by adapting and extending the hot spots provided as hook classes according to the class inheritance and object composition mechanism.

Since frameworks are to be reused repeatedly in many application developments, they need thorough testing. Moreover, each time a framework is extended for reuse, it requires additional testing to check for possible progressive faults and regression faults. Systematic testing on a framework is therefore crucial for the reliability of framework-based applications. Frameworks, however, have properties that make it difficult to control and observe the process of framework testing needed whenever they are modified and extended for reuse in developing applications.

The context in which a framework is (re)used can be very complicated[5]. Furthermore, the types of architectural mismatches that can occur when multiple frameworks are reused together are vastly diverse and hard to predict[6]. Usually, the execution of the extended parts of the framework is controlled by the framework itself, which makes it difficult to set up initial test conditions of the framework and to drive test execution. Since it is not easy to predict the

starting point of the execution and to observe the result of the test, it is also difficult to detect occurrences of malfunctions. Consequently, it is not easy to force the classes adapted and composed during framework reuse to satisfy the constraints the framework assumes, or to discover constraint violations in advance. If changes are made to framework code arbitrarily to intermingle it with test support code to enhance the controllability and observability of framework testing, the reliability of testing can be compromised due to possible interferences of test code with the framework code.

In order to overcome such problems, this paper proposes a scheme for embedding test support code as BIT(Built-in Test) components into the hook classes of the framework without incurring changes or intervention to the framework code. The test components built into the framework in this way make the testing of the adapted framework more controllable and observable, and thereby enable us to effectively detect, through tests, the faults generated during framework adaptation or extension.

Many test methods on object oriented software have been introduced[7,8,9,10,11,12]. Most of them can be applied to framework testing. Our approach makes their application to framework testing more effective by directly addressing the testing problems specific to the hot spots of a framework and increasing the framework testability.

This paper is organized as follows. Section 2 describes framework testability and the types of test support components we considered for BIT-embedding. Section 3 describes framework hot spots. Section 4 describes the proposed design scheme for embedding built-in test components in framework hot spots. Section 5 presents an experimental example of applying the design scheme to the testing of a sample framework. Section 6 reviews related work. Section 7 concludes this paper.

2. Framework Testability

Software testability means ease of revealing software faults through tests[15,16,17]. For effective framework testing, high testability must be maintained when developing, adapting or extending a framework. Software testability can be affected directly or indirectly by many factors[15]. In this paper, we focus on the following four factors that have direct influence on framework testability:

1) controllability: the ability to set up and control test conditions
2) sensitivity: the ability to capture and expose traces of malfunctions in response to tests
3) observability: the ability to observe test results externally
4) oracle availability: the ability to determine or obtain expected test results

Besides testability, there are other important quality factors such as reliability, robustness, flexibility, modularity, and performance that a framework must maintain at a high level. Though testability is generally complementary to other quality factors, it could conflict with them in some cases. For example, increasing a framework's test sensitivity could cause malfunction by faults to occur more frequently, and consequently to reduce reliability and robustness. To take another example, if a framework's controllability and observability get higher, then reliability, flexibility, modularity or performance could be degraded since concealed information could be revealed and components of the framework might be possibly interfering with each other.

In order to increase framework testability without adverse effects on other quality factors, we designed a scheme for encapsulating test support codes as a set of tester components separated from the framework under testing and embedding them into the framework with little change to the framework. Since test support codes are embedded as BIT components with little

interference to the framework under test, enforcing testability does not sacrifice other quality factors such as reliability and modularity. Furthermore, the tester components are designed to be attached or detached as needed so that framework testability can be higher during tests and lower in operation in order to avoid performance degradation due to the overhead incurred by the execution of tester components.

The types of test support components presented in this paper to enforce framework testability are test controllers, test sensitizers, test monitors, test oracles, and test loggers. The test controller increases controllability by setting up and initializing test conditions for framework hot spots. The test sensitizer increases the framework's sensitivity to faults by capturing and leaving traces of malfunctions during test execution. The test monitor increases observability by monitoring actual results of test execution and judges pass/fail of the test by comparing the actual with the expected results. The test oracle helps the monitor judge pass/fail of the test by giving expected pre-conditions, post-conditions and invariants of the test cases. The test result, passed or failed, is sent to the test logger by the monitor. The test logger records and stores the test result according to the current test context.

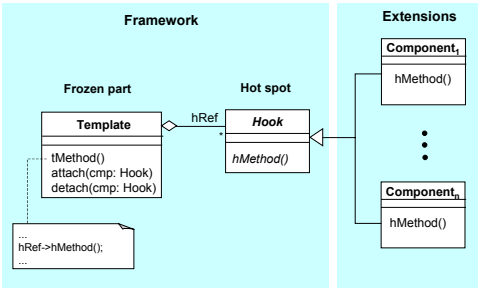
The pre-, post-conditions, and invariants for the framework hot spots are encapsulated in a suite of assertions that can be inspected by the tester components. The tester components embedded or attached to the framework support the framework test by watching through the assertions to see how the adapted or extended components of the framework observe the required contracts. Much work has been done on the type and grammar of the assertions and where to place them[18,19,20].

3. Framework Hot Spot

A framework is composed of frozen parts designed to be shared among applications without modification, and hot spots designed to be adapted to specific needs of the application[21,22]. A framework also prescribes rules of composing and interacting among the system components which must be observed when adapting hot spots. Those rules can be defined precisely using the design by contract principle[18,23,24].

The frozen and hot spots of an object-oriented framework are encapsulated in methods of classes. The method for a frozen spot is called a template method, and the method for a hot spot is called a hook method[21,22]. The class that contains template methods is called a template class, and the class that contains hook methods is called a hook class. The template class and the hook class can be of different classes or the same class. When they are different, the hook class is adapted and extended by composing an object from a subclass of the hook class into the template class through an instance variable which references the hook class object. When they are same, the hook method is adapted and extended by subclassing the unified template/hook class.

Pree classified the composition among the template and hook classes of a framework into 7 patterns [21]. An application built from an object-oriented framework provides its functions through interactions allowable among the template and hook class objects compounded according to the composition patterns. The variable functions implemented through adapting and extending the hook



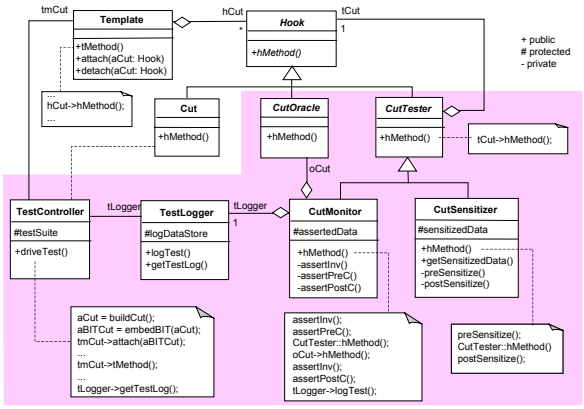
[Figure 1] Framework Hot Spot

classes, and the interactions among the frozen spots and extended hot spots of the framework must be (re)tested to check for possible progressive faults and regression faults. Figure 1 shows a framework hot spot in which a template class is associated with its hook class that can be extended into subclasses as represented by $Component_i$.

4. The Proposed Design Scheme of BIT-Embedded Framework Hot Spot

This section describes the proposed design scheme for embedding tester components into the hook classes of the framework in order to facilitate testing whether the framework's functional contracts are observed when the hook classes are extended to adapt to an application. The design scheme is illustrated using the UML diagramming notations and the C++ language.

The class diagram in Figure 2 shows the composition pattern of a BIT-embedded framework hot spot in which the tester components are embedded together with an extended component of the hook class as a framework CUT (class or component under test). The classes in the shaded area of the diagram represent the tester components embedded or attached to the framework hot spot. The TestController and TestLogger are the tester components attached from the outside of the CUT, while the other components are BITs embedded into the CUT. The CutTester is an abstract base class of the BIT components which provides the same interface as the CUT for the template class object through the hook class. The CutSensitizer and CutMonitor are designed as subclasses of the CutTester class, which is a subclass of the hook class. The BITs and the CUT form a chain structure connected through the tCut, which is an instance variable of the CutTester class. The Template object is composed with the CUT through the Hook class interface referenced by the hCut variable. Via a chain of the BIT components in between, the template object is connected to the CUT located at the end of the chain.



[Figure 2] Class Structure of BIT-Embedded Framework Hot Spot

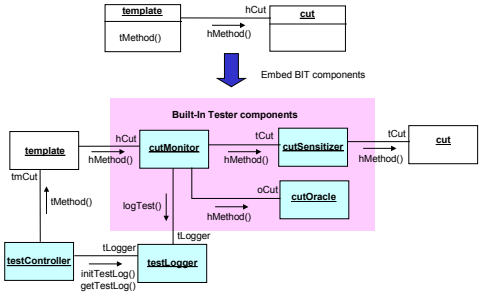
When its hMethod is called, each of the tester components embedded into the CUT as elements of the chain structure performs its own testing function before and after forwarding the hMethod call to the successor component connected through the tCut variable. During the process, however, the functional behavior between the Template object and the CUT is not interfered with by the tester components attached to the CUT. The design pattern of the tester components is similar to the Decorator design pattern[4]. Testing functions on the CUT are distributed over, and encapsulated by the tester components. The outermost CutMonitor plays the role of a proxy which watches the template object have access to the CUT[4].

The CutSensitizer in Figure 2 captures the clue data critical to the testing of the CUT before and after it calls the hMethod through tCut. For example, the CutSensitizer keeps the state before the CUT is called, so that it is possible to inspect post-conditions and invariants after calling. The CutMonitor checks if the contract is violated by inspecting the pre- and post-

conditions and invariants before and after the CUT calls the hook method. The test results are collected and recorded by the TestLogger.

The CutOracle referenced by the CutMonitor calculates the expected result after the CUT executes the hook method. The CutMonitor judges pass or fail of the test by comparing the expected result obtained from the CutOracle with the actual result obtained from the CutSensitizer.

The TestController initializes the other tester components and drives the test execution. Before starting the test execution, it generates an object structure instance of the CUT and BIT components, and embeds the generated BITs in the CUT instance. It then initializes the BIT-embedded CUT and the TestLogger to the condition required by the current test case.

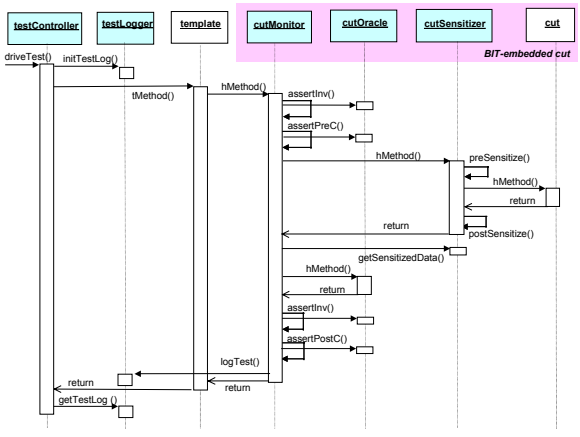


[Figure 3] Object Structure of BIT-Embedded Framework Hot Spot

Figure 3 shows an object structure which has a test monitor, sensitizer and oracle between the template object and the CUT object instantiated according to the BIT-embedded composition pattern in Figure 2. Figure 4 shows a sequence of interactions among the template, CUT and tester objects of Figure 3 when the tMethod of the template object is called by the TestController.

Referencing the hCut variable, the tMethod of the template object calls the hMethod of the hook class object. The invocation of the hMethod is forwarded to hMethod of the CUT by way of the CutMonitor and the CutSensitizer. The execution result returns to the tMethod of the template object by way of the CutSensitizer and the CutMonitor. With the help of the CutOracle, the CutMonitor checks the pre-, post-conditions and invariants of the CUT before and after the hMethod of the CutSensitizer is called, and passes the inspection result to the TestLogger. The CutSensitizer captures and stores the states of the CUT before and after the hMethod of the CUT is called. During this process, however, the interaction between the template object and the CUT is not interfered with by the CutMonitor, CutSensitizer and CutOracle.

The object structure of the hot spot in Figures 3 and 4 shows a 1:1 composition of a hook class object and a template class object. The BIT embedding method proposed in this paper supports the framework testing for 1:n composition where a template object is compounded with many hook class objects, as well as for 1:1 composition. For example, in the object structure allowed by Figure 2, where many hook objects are 1:n compounded with a template object, the BIT components are embedded into each of the hook objects. Although this paper exemplifies a case with only one hook method, testing can be easily extended to the



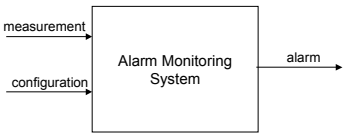
[Figure 4] Sequence Diagram of BIT-Embedded Framework Hot Spot

interactions between a template object and a hook object that has several hook methods. In such a case, the BIT-embedding process is defined for each hook method in the same way as the BIT components are embedded in the hMethod in Figure 2.

5. A Case Study

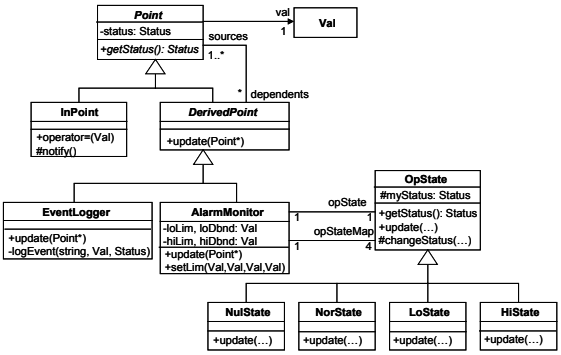
5.1 Alarm Monitoring System

This section describes a case study that we performed to justify our approach. The framework exemplified here is an alarm monitoring framework that we implemented in C++ for use in applications such as process control monitoring systems. The framework can be extended to an alarm monitoring system [Figure 5] which raises an alarm upon change to an abnormal state according to the current measurement and configuration.



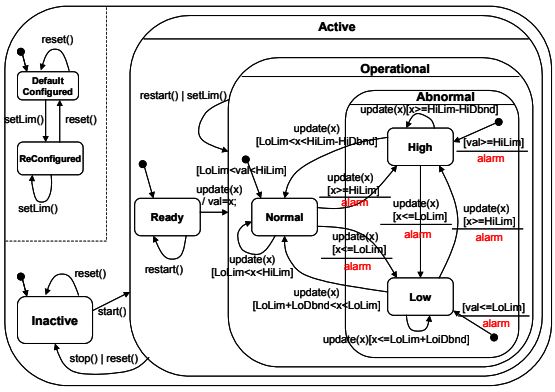
[Figure 5] Alarm Monitoring Svstem

The class diagram in Figure 6 shows a part of the composite structure of the alarm monitoring framework. The InPoint is the class which directly inputs measured values from the outside. The DerivedPoint class calculates new measured values or state values, using the measured values acquired from other Point class objects. AlarmMonitor, a subclass of the DerivedPoint class, senses state changes, depending on the changes of measured values, and gives an alarm upon change to an abnormal state. AlarmMonitor is a hook class compounded into the Point class, and can be extended to various subclasses depending on the applications.

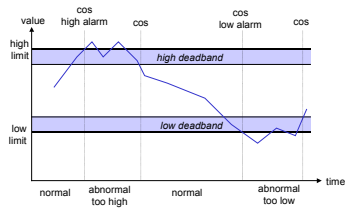


[Figure 6] Partial Class Structure of the Alarm Monitoring Framework

In the example framework, the AlarmMonitor will be the CUT upon which the tester components are attached, and have the state behavior which is specified in the Statechart[25] shown in Figure 7. If the measured values received as

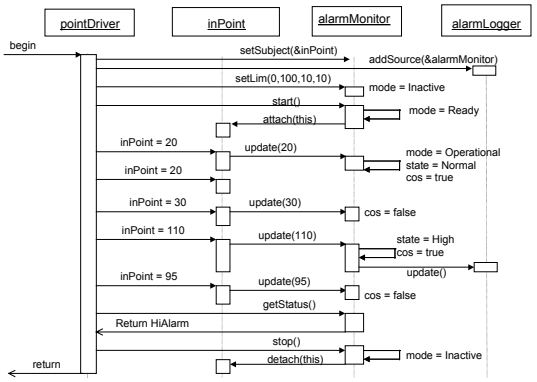


[Figure 7] State Behavior of the AlarmMonitor



[Figure 8] State Changes as Measured Value Changes

parameters when the update operation is called exceed either high or low limits, the AlarmMonitor raises high or low alarm. It also has deadbands to avoid repeating alarms that might occur when the measured values fluctuate between upper or lower boundaries. To illustrate the state behavior, figure 8 depicts how state changes as measured value changes. Figure 9 shows one possible sequence of interactions between the objects of an alarm monitoring system when a sequence of measured values comes from a single input source.

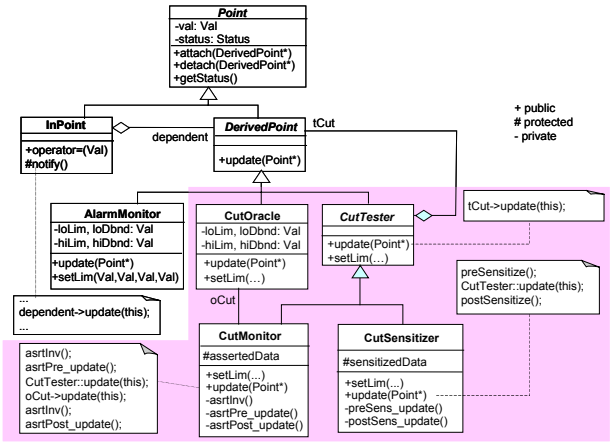


[Figure 9] Sequence Diagram of Alarm Monitoring System

5.2 BIT Design and Embedding

Figure 10 shows a class structure in which tester components, as BITs, are embedded into the CUT, AlarmMonitor. The classes in shaded area of the diagram indicate the embedded BITs, which are CutMonitor, CutSensitizer and CutOracle. The CutMonitor and the CutSensitizer are connected to each other through tCut, an instance variable of their superclass, CutTester. The CutSensitizer inspects and captures the internal states of the AlarmMonitor before and after it calls the update function through tCut. The CutOracle is designed to simulate the state behavior in Figure 7. The Cut Monitor is designed to check if the CUT violates the pre-, post-conditions and invariants, comparing the actual behavior from the CutSensitizer with the expected behavior from the CutOracle. Other tester components, such as a test controller which initializes test conditions and drives the framework testing, or a test logger which collects and records the test results, are also designed to be attached to the framework under testing. As test cases, we used sequences of update function calls whose parameters are the measured values. Each of such test cases represents the paths of a state transition spanning tree whose paths are the state transition sequences allowable within the operational mode in the statechart in Figure 7.

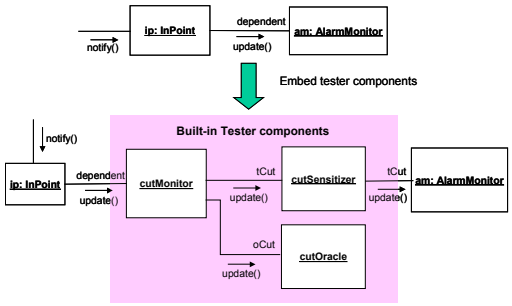
At the initial stage of the testing, the test controller instantiates the tester components and embeds them into the CUT. When the test controller attaches AlarmMonitor to the InPoint object, the BITs embedded in the AlarmMonitor are also attached together with the AlarmMonitor. The tester components were implemented and integrated with the framework under testing without incurring changes or intervention to the framework codes. The tester components can be dynamically detached and attached as needed at run-time.



[Figure 10] Class Structure of BIT-Embedded Alarm

Figure 11 shows how the tester component objects, designed as in Figure 10, are embedded into

the AlarmMonitor object which is compounded as a hook class object into the InPoint object. Upon testing, not only establishing test conditions but also monitoring test results are done transparently without intervening in the alarm monitoring function of the system.

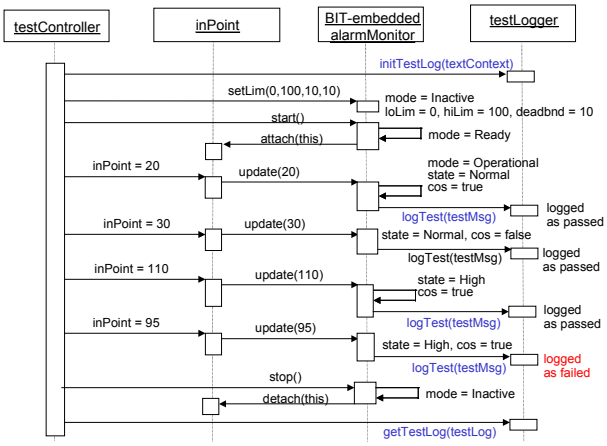


[Figure 11] Embedding BIT Components in an AlarmMonitor

5.3 A Test Scenario

In order to check if the tester components work properly in testing the BIT-embedded alarm monitoring system, we planted some errors in the system that may not be easily detected but cause abnormal state changes, and performed testing with test patterns as explained in the previous section. Two cases of testing have been done: with and without embedding BITs. In the case of tests with BITs, abnormal state changes caused by errors were recorded by the test logger without omission, although not all the known errors were detected. Meanwhile, in the case of tests without BITs, only a subset of the errors that could be detected by the BITs, such as giving a false alarm by misjudging a normal state to be abnormal, were detected by the alarm logger.

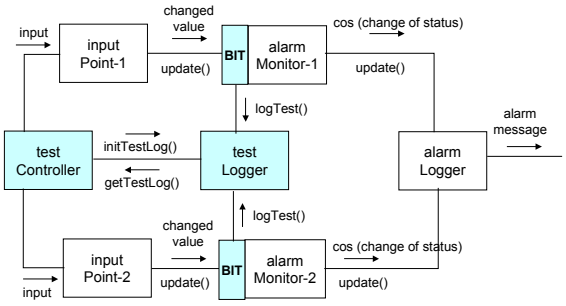
We give an example in which a malfunction caused by an error in the alarm monitor is detected by the tester components. Suppose the alarm monitor perceived the current state to be abnormal because a measured value exceeded the high limit, and another measured value below the high limit and within the deadband was input. In this case, the alarm monitor should still regard the current state as abnormal. But, due to an error in the alarm monitor, suppose the alarm monitor regarded the situation as returning to the normal state without considering the deadband. Figure 12 shows the interaction among test controller, InPoint object, BIT-embedded alarm monitor and test logger, under the supposed situation. (In the actual implementation, the caller's reference is passed as a parameter when the Inpoint object calls the update function. In Figure 12, however, measured values were used here as parameters, as a matter of convenience.) When a normal measured value of 95 which is within the deadband is passed from the inPoint to the alarm monitor, the BITs perceive the alarm monitor misjudge the situation as returning to the normal state. Then, the testLogger logs it as a failed test. Since the alarm monitor gives alarm only when the state changes to abnormal, such a malfunctioning cannot be easily detected without support of the tester components.



[Figure 12] Test Scenario of BIT-Embedded Alarm Monitor

Though we showed only the simplest test case in which a single input and a single alarm

monitor are involved, we did successfully apply our testing approach to more complicated cases in which multiple input sources and multiple alarm monitors were involved in a variety of configurations. Figure 13 shows one of such cases in which each alarm monitor has its own set of built-in test components except for the test controller and the test logger that are shared by the system.



[Figure 13] Possible Object Structure of BIT-Embedded Alarm Monitoring System

6. Related Work

Many testing methods for object-oriented software have been proposed[7,8,9,10,11,12]. For example, ASTOOT[9] offers an algebra-based class test method and support tools, while ClassBench[10] provides a state-based class test method and its support environment. An incremental test for the class hierarchy[11] and object-oriented integration testing methods[12] have been also introduced. The XUnit[26] provides testing frameworks for many programming languages to support writing repeatable unit tests of classes. For example, the JUnit[27] framework, which is a version of XUnit framework for Java, can be extended to implement a suite of test cases for a Java class, and any other test support code necessary to automate running the test cases. Edwards[28] proposed a strategy for automated generation of test drivers, test cases, and test oracles as BITs for components (or classes) given their specifications. Both the XUnit's and Edwards' strategies separate the testing infrastructure code from the units under testing.

Difficulties in testing adapted frameworks are well-known, and several solutions have been proposed in the literature[5,8,13,14]. Fayad, et al.[14] presented a method in which the test-case generating codes, packaged in Built-in tests(BITs) classes, are embedded into the framework, and reused in framework testing, being inherited and adapted during framework adaptation and extension. Binder[8,15] comprehensively presented test design patterns and methods to construct a test support environment. All of the testing approaches and methods mentioned above are useful for framework testing. Our method complements them by directly addressing the testing problems specific to the hot spots of the framework.

7. Conclusion

This paper has described a scheme for encapsulating test support code as built-in test (BIT) components and embedding them into the framework's hot spots so that defects caused by the modification and extension of the framework can be easily detected through testing. Using our scheme, test components can be designed and embedded into a framework to increase the controllability and observability of framework testing without incurring changes to the framework code and without affecting its functional behavior. Finally, the test components can be attached and detached dynamically to/from the framework as needed at run-time.

References

[1] Fayad, M.E., et al., *Building Application Frameworks*, John Wiley & Sons, Inc., 1999

[2] Fayad, M.E., et al., *Implementing Application Frameworks*, John Wiley & Sons, Inc., 1999

[3] Fayad, M.E. and Johnson, R.E., *Domain-Specific Application Frameworks*, John Wiley & Sons, Inc., 2000.

[4] Gamma, E., et al., *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley, 1995.

[5] Codenie, W, et al., "From Custom Applications to Domain-Specific Frameworks", *Comm. ACM*, 40(10), Oct. 1997, pp. 71-77.

[6] Garlan, D., et al., "Architectural Mismatch or Why its hard to build systems out of existing parts", *Proc. 17th Int'l Conf. Software Engineering*, Apr. 1995, pp. 179-185.

[7] Kung, D.C., Hsia, p. and Gao, J. (eds.), *Testing Object-Oriented Software*, IEEE CS Press, 1998.

[8] Binder, R.V., *Testing Object-Oriented Systems: Models, Patterns, and Tools*, Addison-Wesley, 2000

[9] Doong, R. and Frankl, P., "The ASTOOT Approach to Testing Object-Oriented Programs", *ACM Transactions on Software Engineering and Methodology,* 3(2), Apr. 1994, pp. 101-130.

[10] Hoffman, D. and Strooper, P., "ClassBench: a Framework for Automated Class Testing", *Software Maintenance: Practice and Experience*, 27(5), May 1997, pp. 573-597.

[11] Harrold, M.J., Mcgregor, J.D. and Fitzpatrick, K.J., "Incremental Testing of Object-Oriented Class Structures", *Proceedings of 14th Int'l Conference on Software Engineering,* May 1992, pp. 68-80.

[12] Jorgensen, P.C. and Erickson, C., "Object-Oriented Integration Testing", *Comm. ACM*, 37(9), Sept. 1994, pp. 30-38.

[13] Sparks, S, et al., "Managing Object-Oriented Framework Reuse", *IEEE Computer*, September 1996, 29(9), pp. 52-61.

[14] Fayad, M.E., Wang, Y. and King, G., "Built-In Test Reuse", In the Building Application Frameworks, Fayad, M.E., et al, John Wiley & Sons, Inc., 1999, pp. 488-491.

[15]Binder, R.V., "Design for Testability in Object-Oriented Systems", *Comm. ACM*, 37(9), Sep. 1994, pp. 87-101.

[16] Voas, J.M., Morell, L and Miller, K., "Predicting Where Faults Can Hide from Testing", *IEEE Software*, Mar. 1991, pp. 41-48.

[17] Voas, J.M. and Miller, K.W., "Software Testability: The New Verification", *IEEE Software*, 12(3), May 1995, pp. 17-28.

[18] Meyer, B., "Applying Design by Contract", *IEEE Computer*, Oct. 1992, pp. 40-51.

[19] Rosenblum, D.S., "A Practical Approach to Programming with Assertions", *IEEE Transactions on Software Engineering*, 21(1), Jan. 1995, pp. 19-31.

[20] Voas, J. and Kassab, L., "Using Assertions to Make Untestable Software More Testable", *Software Quality Professional Journal*, 1(4), Sep. 1999

[21] Pree, W., *Design Patterns for Object-Oriented Software Development*, Addison-Wesley, 1995.

[22] Schmid, H.A. "Systematic Framework Design by Generalization", *Comm. ACM*, 40(10), Oct. 1997, pp. 48-51.

[23] Helm, R, et al., "Contracts: Specifying Behavioral Compositions in Object-Oriented Systems", *Proc. OOPSLA'90*, Ottawa, Canada, 1990.

[24] Steyaert, P, et al., "Reuse Contracts: Managing the Evolution of Reusable Assets", *Proc. OOPSLA'96*, San Jose, CA, USA, Oct. 6-10, 1996.

[25] D. Harel, et. al., "STATEMATE: a Working Environment for the Development of Complex Reactive Systems", *IEEE Transactions on Software Engineering*, 16(4), April 1990, pp. 403-414.

[26] The XUnit Home Page, http://www.xprogramming.com/software.htm

[27] Gamma, E. and Beck, K. "JUnit A Cook's Tour", *Java Report*, May 1995

[28] Edwards, S. H., "A Framework for Practical Automated Black-Box Testing of Component-Based Software", *Software Testing, Verification and Reliability*, Vol. 11, 2001, pp. 97-111.