

# Adaptive Scenario-Based Object-Oriented Test Frameworks for Testing Embedded Systems

W. T. Tsai\*, Y. Na\*, R. Paul<sup>†</sup>, F. Lu\*, A. Saimi\*\*

\*Department of Computer Science and Engineering  
Arizona State University  
Tempe, AZ 85287

<sup>†</sup>Investment and Acquisition  
Department of Defense  
Washington, DC

\*\* Hitachi Software Engineering  
Yokohama, Japan

## Abstract

*This paper presents a process to develop adaptive object-oriented scenario-based test frameworks for testing embedded systems. Embedded systems often require rigorous testing due to the mission-critical nature of their applications, and they are often developed as a family of products. The process uses techniques such as design-for-change, design patterns, scenarios, ripple effect analysis, and regression testing. This paper then uses an example to illustrate this process by applying it to test a mobile phone system, and the framework constructed can facilitate generation of numerous test cases quickly with minimum effort, and it can also accommodate many changes suggested by another party without changing the overall structure of the framework.*

**Keywords:** Object-oriented test frameworks, scenarios, test scenarios, design patterns, test reusability, regression testing, and embedded systems,

## 1. Introduction

Products are constantly evolving as technology and market needs change. In many cases, a family of products will be developed rather than single products, and products evolve by adding, modifying or deleting features from earlier versions. Thus, all the products within a family are similar to each other in terms of functionality as well as design.

This paper addresses testing a family of embedded systems. As the system and its software keep on changing, testing must keep on changing to test the

modified system. New test cases must be developed to test new or modified features, and regression testing must be constantly applied [9] to ensure that the system remains consistent after modification.

Scenarios and use cases have been widely used in system and software development to specify system requirements. However, scenarios can also be for testing [1][11][13][14]. This paper also presents a scenario model in which a *scenario* describes a function from the end user' point of view, and scenarios are grouped in *scenario groups*; a *sub-scenario* is a decomposition of a scenario, and it represents a sub-function provided by the software.

This paper applies OO framework techniques to organize test scenarios and cases, and illustrates this idea to test a mobile phone system that consists of three parts: a Mobile Station Center (MSC) server, several Base Station (BS) servers, and clients. In this approach, requirements of a System Under Test (SUT) is first specified using the scenario model, then complex test scenarios are generated. An OO framework is then developed to represent these scenarios. Those that are likely changed are encapsulated by design patterns in the framework.

OO test frameworks have been proposed [3][6][12] and these frameworks often follow the OO structure such as inheritance graph. OO test frameworks have also been applied to test safety-critical embedded systems with 70% reduction in effort and cost [11]. Recently, several OO test frameworks are available such as *JUnit* [5], *Cactus* [2], *HttpUnit* [4], and *Mock Objects* [7], and these frameworks use design patterns extensive, and is often based on specific programming languages such as Java or specific applications such as Internet publishing.

This paper is organized as follows: section 2 introduces the scenario-based test framework, and an example framework is constructed using *JUni*. Section 3 describes the process of handling change scenarios and regression testing in the test framework. Section 4 discusses the production rate with the mobile phone example. Section 5 concludes this paper.

## 2. Scenario-Based Test Framework

Essentially, development of a scenario-based test framework follows the design-for-change initially proposed by Parnas [8]. While developing the SUT with the framework approach, one can develop the test framework by using the same scenarios. Specifically, this process is as follows:

- Specify system requirements by analyzing operational scenarios;
- Identify those areas of the systems that are likely to be changed;
- Develop the SUT with scenarios;
- Develop the OO test framework with scenarios.

By using the test framework, one can generate numerous test cases quickly with minimum effort. When the system is changed, a tester can then reuse many test scenarios embedded in the test framework to generate new test scenarios and cases, and to select test cases for regression testing.

### 2.1 Specify Scenarios

Scenarios are derived from system requirements, and they often represent the functional aspects from the end user's point of view. These scenarios can be atomic scenarios or complex scenarios. An atomic scenario verifies system correctness with individual functions, while a complex scenario executes multiple functions in a session to detect those defects that are difficult to be detected by atomic scenarios, and it can be used for stress testing if many atomic scenarios are involved. Figure 1 shows an atomic scenario.

```
{Scenario 1;
Description: start an MSC server successfully
with a valid port number;
Constraints: working environment is set
correctly;}
```

Figure 1: An Atomic Scenario

Scenarios can be grouped. Specifically, a scenario group is a collection of functional related scenarios or scenario groups. A sub-scenario is a decomposition of a scenario, representing a sub-function provided by the software.

A complex scenario can be formed by connecting atomic scenarios and complex scenarios by control constructs such as *sequencing, conditioned, concurrent, and iterative*. Pseudo code can be used to describe the complex scenario. Figure 2 shows a complex scenario.

```
{Scenario 1;
If ((the working environment is set up
correctly)AND (MSC server is started))
then
Scenario 2100;}}
```

Figure 2: A Complex Scenario Example  
\* Scenario 1 and Scenario 2 are atomic scenarios

Scenarios may relate to each other with the following relationships:

- *Independent*: two scenarios are independent with each other if one can happen regardless of the other.
- *Trigger/trigger-by*: the execution of a scenario triggers the other scenario to activate.
- *Mutually exclusive*: two scenarios are mutually exclusive if they cannot exist at the same time.
- *Related*: two scenarios are related to each other if they are used in the same scenario group or they are mutually exclusive.

These relationships can be used to eliminate infeasible or useless combinations of complex scenarios. For example, a complex scenario containing two scenarios that have *mutually exclusive* relationship with each other is infeasible. In fact, one needs to generate complex scenarios with sub-scenarios that are *related* or *independent*. In this way, one can eliminate many infeasible scenarios.

After scenarios are specified, one assigns risks to scenarios: those risky scenarios or those must be operational received high risks. These risk assignments can be useful in generating test cases (e.g., generating more test cases for risky scenarios) or in selecting test cases during regression testing.

## 2.2. Identify Change Scenarios

The design-for-change process identifies potential changes and encapsulates them in the design at the beginning, and Parnas suggested that hardware dependent parts, input and output, data structures, algorithms are likely to be changed. For those parts that are likely to be changed, one may find a suitable design pattern to encapsulate them, so that when the system is changed, the impact to the rest of system is minimized. In the mobile phone system, some features are not likely to be changed. For example, it must have at least one MSC server and one BS server, and both servers must provide *receive*, *send*, and *close* functions. These features can be viewed as invariants. On the other hand, various features are likely to be changed when system requirements are modified. For example, an MSC server adds a new function of sends an acknowledge message when it receives a request from a client.

## 2.3. Develop The SUT with Scenarios

Once system scenarios including change scenarios are identified, one can develop the SUT from those scenarios. Since a scenario group represents one functional aspect of the SUT, one can make a scenario group as an abstract class or an interface, and then one can organize all scenarios of a scenario group into subclasses of the abstract class. One may implement a scenario as a class or a method in a class. Similarly, a sub-scenario may be implemented as a method of a class. Note that developing the SUT with the framework approach is optional. The OO test framework can still be developed regardless whether the SUT is developed using the framework approach.

## 2.4 Develop The Test Framework with Scenarios

In addition to form the basis of the SUT design, scenarios can also be used to develop the test framework. Although SUT and test framework are developed with scenarios, they are different: the SUT focuses on implementation of the functionality, while the test framework focuses on testing the same functionality.

### 2.4.1 Implement a Test Framework Using JUnit

*JUnit* [5] is a framework for repeatable unit tests for Java-based applications. The test framework proposed in this paper is different from *JUnit* in several ways. It is scenario-based functional testing including unit testing and integration testing, and it is developed using design-for-change. However, it is

possible to implement the proposed test framework by using *JUnit*. Figure 3 shows the architecture of the scenario-based test framework. *FunctionalTesting* extends *JUnit*, and subclasses *UnitTesting*, *IntegrationTesting*. *UnitTesting* uses atomic scenarios to perform testing, while *IntegrationTesting* uses composite scenarios to perform integration testing, stress testing and focus testing. In this framework, one scenario corresponds to a class. For example, *scenario 1* has a corresponding test class *Scenario1Testing*.

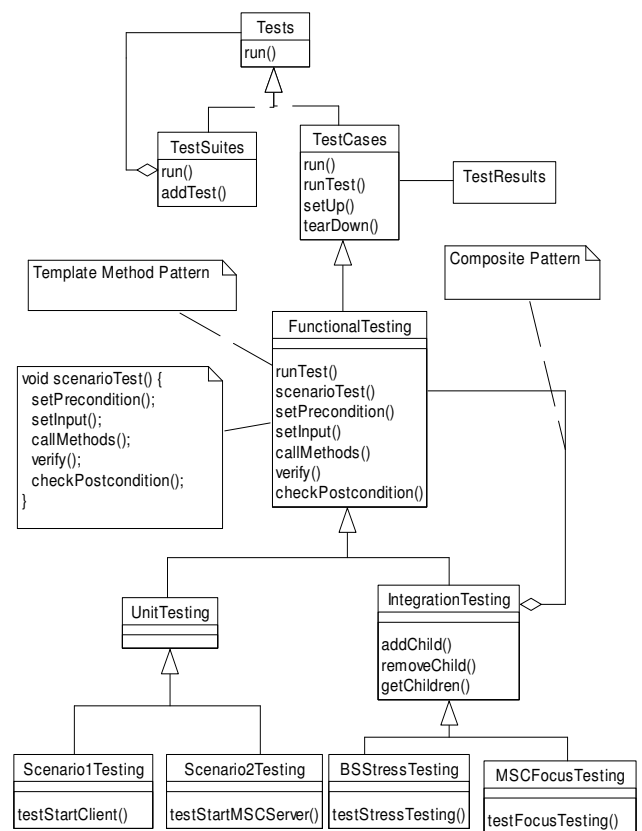


Figure 3: The Architecture of Test Framework

Scenario-based testing needs to perform the following steps: set testing preconditions such as testing environments, set inputs, run test (i.e., calls methods), verify the results, and check post-conditions. The test framework proposed in this paper implements this procedure by using *Template method* design pattern. In Figure 3, the *Template Method* pattern is used in the method *ScenarioTest()* that defines the common structure. *setPrecondition()*, *setInput()*, *verify()*, and *checkPostcondition()* are the primitive methods, which are the steps to perform scenario-based testing and can be rewritten in the

subclasses. The Composite design pattern is used to implement *IntegrationTesting*, which can generate complex scenarios to handle stress testing and focus testing.

### 2.4.2. Develop Test Cases from the Test Framework

The test framework provides a place where test scenarios can be generated, and once generated, test cases can be generated by examining the inputs and outputs of test scenarios based on various testing techniques such as partition testing, boundary value testing, random testing, and equivalent partition. In most cases, each test scenario is associated with multiple data. To generate test data, one can apply different technique to different data, and test cases are obtained from various combinations. For example, the test scenario “start an *MSCServer* with IP and port number” includes two associated data: IP address and port number. A correct IP address can be 144.169.25.7, and the correct port number is 222. When one generates test cases for this test scenario, one can use different technique like boundary value, random testing to select different test data.

## 3. Regression Testing and Ripple Effect Analysis

Once the system is modified, it is necessary to update the system so that it will remain consistent, and it is also necessary to re-validate the system. Essentially, it is necessary to perform the following tasks:

- Update the SUT;
- Perform ripple effect analysis (REA) to ensure that the components of the modified SUT is consistent with each other;
- Develop new test cases to test the modified feature; and
- Perform regression testing to ensure that those parts that are supposed to remain unchanged are indeed unchanged after modification.

### 3.1. Update the SUT

There are three types of changes: addition, deletion and modification. The addition/deletion/modification of a requirement item usually results in changing the corresponding operations on scenarios, which are then propagated to the other test artifacts including other scenarios and test cases.

### 3.2. Ripple Effect Analysis on the SUT

The REA process is an iterative process. Our scenario model enables scenario dependency and traceability

analysis, which can be used for scenario slicing and functional Ripple Effect Analysis (REA). The REA process is an iterative process of scenario-based regression tests selection: (1) A change request is first mapped to a set of potentially affected scenarios; (2) By traceability analysis, one can trace a scenario to its associates and identify and validate the set of other affected other software artifacts; (3) Scenario identification and validation iterate until no more ripples exist.

### 3.3 Develop New Test Cases for the Modified Feature

After the system has modified, including those that are affected by ripples are handled, it is necessary to test those modified features. For example, if the test cases for scenario “start an *MSC server successfully*” is changed by adding new feature into *send()* method in the *MSCServer*, one needs to develop new test cases for modified feature. In this example, new test cases are in table 1:

Parameters	Data selected
Acknowledge type	{"Successfully", "failure", " "}
Server name	{"149.169.25.7","149.169.25.6","149.169.25.8","149.159.22.3","000""III","xxxx", "49.169.22.6"}

Table 1: Test Cases for Modified Features

### 3.4 Regression Testing

Regression testing has the following steps [9]:

- Test case selection
- Test case re-validation
- Test case execution
- Failure identification
- Fault identification and mitigation

However, regression testing using OO test framework differs from conventional regression testing in test case selection. In most industrial test sites, test cases are listed in tables with cross references such as traceability to modules and requirement items. Test case selection in this kind of environment is to search all the test cases with appropriate cross references. However, test case selections in an OO test framework is different because test cases are attached to test scenarios embedded in the test framework rather than listed in a table. Furthermore, in conventional regression testing, one may use the default test case selection strategy, i.e., select all. In

fact, select all is frequently used in the computer industries [9]. However, it is no longer safe to select all the test cases in an OO test framework because many of them are specific to certain change scenarios, and if the current implementation does not use a specific change scenario, the related test cases cannot be used.

However, from a given implementation of the SUT, it is possible to identify all the scenarios that are applicable to it, and then its associated test scenarios can be traced and identified. Use all the test cases associated with those test scenarios as the default strategy.

Furthermore, some common regression testing strategies can be used here in combination. For example, one may use select all high-risk scenarios, highly-used scenarios, randomly selected scenarios, or use a time-wise strategy for regression testing. Specifically, different set of test cases can be selected at different time interval with sufficient overlap to ensure system reliability and quality. For example, module test cases can be re-run daily, while all integration test cases are re-run weekly and at major milestones.

#### 4. Evaluation

While using the test framework, one can generate numerous test cases quickly. In addition, if there is a change, the framework can be updated quickly to generate new test cases to test the modified features.

##### 4.1 Production Rate

With the test framework, numerous test cases can be generated via examining the inputs and outputs of the scenario, based on various testing techniques such as partition testing, boundary value testing, and random testing. In most cases, each scenario is associated with multiple data and each data has multiple attributes. To generate test data, one can apply different technique to different data and even to different attributes, and test cases are obtained from various combinations. The following test cases are generated once test scenarios are obtained from the framework as shown in table 2.

##### 4.2 Change Scenarios

To demonstrate the framework can adapt to changes, we asked a professional engineer to examine the original mobile phone system requirements, and suggest changes. She suggested ten changes, and these changes covered both the software and hardware aspects. The test framework handles all

Scenario	Number of Test Cases	Scenario	Number of Test Cases
Scenario 1	3,470	Scenario 13	8,920
Scenario 2	3,470	Scenario 14	10,020
Scenario 3	5,850	Scenario 15	3,920
Scenario 4	3,470	Scenario 16	2,030
Scenario 5	3,470	Scenario 17	8,422
Scenario 6	5,850	Scenario 18	9,000
Scenario 7	5,850	Scenario 19	22,394
Scenario 8	3,320	Scenario 20	5,020
Scenario 9	7,820	Scenario 21	3,490
Scenario 10	5,477	Scenario 22	9,650
Scenario 11	6,525	Scenario 23	7,046
Scenario 12	7,435		

Table 2: Test Case Generations

except one. The test framework could not handle the only change because we had no access to specific hardware mentioned. Changes suggested by the engineer that can be handled are as follows:

- Scenario 1: Add 911 calls, and it has higher priority than other phone calls.
- Scenario 2: Add an inter-state long distance call.
- Scenario 3: Add an international distance call.
- Scenario 4: Add a 1-800 call.
- Scenario 5: Add user account management.
- Scenario 6: Change user policy.
- Scenario 7: Add display message from the weather forecasting center.
- Scenario 8: Add a same type of BS.
- Scenario 9: Add a different type of BS.

The changes identified can be categorized as: input (Scenario 1-4), adding new feature (Scenario 5), business rule (Scenario 6), output (Scenario 7) and configuration (Scenario 8 and Scenario 9). The changes may affect the design, implementation and test cases. For example, Scenario 1 allows dialing three-digital telephone numbers. The corresponding method must be added in the system design and implementation. In addition, one also must generate the test scenario with input as three-digital number, and test cases. Table 3 shows how the framework accommodates 2 change scenarios. Once changed, new test cases can be generated quickly, in fact, 14,006 test cases were generated with minimum effort.

Change Scenarios	Change Procedure	Test Cases
Scenario 01: make 911 calls	This can be easily incorporated by adding a new method to class <i>MSCServer</i> .	New test cases can be generated by reusing method <i>testMSCStartup</i> and 456 test cases are generated quickly.
Scenario 02: add inter-state long distance call	This is done by adding a new method to class <i>MSCServer</i> .	By reusing test cases for test scenario "start an <i>MSCServer</i> ", one may generate 2.856 test cases quickly

Table 3: Change Scenarios

## 4.2 Coverage Analysis

In most existing testing techniques, coverage is evaluated with respect to the code or design of a SUT, such as statement coverage, condition coverage, and method/class coverage. [15] proposed an approach to express design by using MtSS, MgSS and MfSS that have been used to specify the message interaction in SUT. The MtSS specifies the sequence of messages that can be sent to a class; the MgSS specifies the message a particular method can send; the MfSS is an extension of MtSS and MgSS, and it specifies the sequence constraints on the interaction between framework objects and custom objects. In addition, the test framework enables coverage analysis with respect to requirements represented by scenarios. For example, one possible criteria can be *Number of scenarios tested/Total number of scenarios*. In addition, the test framework allows for coverage analysis combined with risk analysis and usage analysis, one can use *percentage of often-used scenarios covered* and *percentage of high-risk scenarios covered* to analyze the coverage. In this mobile phone example, we got 100% class coverage, often-used scenarios and high-risk scenarios covered. In addition to the SUT coverage, the test framework provides all atomic scenario coverage, and all complex scenarios with 3 and 4 combinations.

## 5. Conclusion

This paper presents a scenario-based test framework that enables test reusability and thus saves resource for a software organization. The proposed framework differs from previous approaches because it is based on scenarios, developed using design-for-change, handles complex test scenarios, and addresses issues such as ripple effect analysis, regression testing and test coverage in a test framework. By using this approach, numerous test cases can be generated quickly and many changes can be easily updated without changing the overall structure of the system.

## References:

- [1] X. Bai, W. T. Tsai, R. Paul, K. Feng, and L. Yu, "Scenario-based Modeling and Its Applications to Object-Oriented Analysis, Design, and Testing", Proc. of IEEE WORDS 2002.
- [2] Cactus project, at <http://jakarta.apache.org/cactus/index.html>.
- [3] D. G. Firesmith, "Pattern Language for Testing Object-Oriented Software", Object Magazine, Vol. 5, No. 9, Jan. 1996, pp. 42-45.
- [4] HttpUnit, <http://httpunit.sourceforge.net/>
- [5] JUnit.org, <http://www.junit.org/index.htm>.
- [6] J. D. McGregor, and A. Kare, "Parallel Architecture for Component Testing of Object-Oriented Software", Proc. of Annual Software Quality Week, Software Research Institute, May 1996.
- [7] T. Mackinnon, S. Freeman, and P. Craig. "Endo-Testing: Unit Testing with Mock Objects", Proc. Of extreme Programming and Flexible Processes in Software Engineering – XP2000, 2000.
- [8] S. McConnell, *Rapid Development: Taming Wild Software Schedules*, Microsoft Press, Redmond, WA, 1996
- [9] A.K. Onoma, W.T. Tsai, M. Poonawala, and H. Sukanuma, "Regression Testing in an Industrial Environment", Communications of the ACM, Vol. 41, No. 5, May 1998, pp. 81-86.
- [10] R. Paul, "End-to-End Integration Testing: Evaluating Software Quality in a Complex System", Proc. of Assurance System Conference, Tokyo, Japan, 2001, pp. 1-12.
- [11] M. Poonawala, S. Subramanian, R. Vishnuvajjala, W. T. Tsai, R. Mojdehbakhsh, and L. Elliott, "Testing Safety-Critical Systems - A Reuse-Oriented Approach", Proc. of 9th International Conference on SEKE, 1997, pp. 271-278.
- [12] W. T. Tsai, L. Yu, R. Paul, T. Liu, and A. Saimi, "Developing Adaptive Test Frameworks for Testing State-based Embedded Systems", to appear in Proc. of IDPT, 2002.
- [13] W.T. Tsai, X. Bai, R. Paul, and L. Yu, "Scenario-Based Functional Regression Testing", Proc. of IEEE COMPSAC, 2001, pp. 496-501.
- [14] W.T. Tsai, X. Bai, R. Paul, W. Shao, and V. Agarwal, "End-to-End Integration Testing Design", Proc. of IEEE COMPSAC, 2001, pp. 166-171.
- [15] W. T. Tsai, Y. Tu, W. Shao and E. Ebner, "Testing Extensible Design Patterns in Object-Oriented Frameworks through Hierarchical Scenario Templates", Proc. of IEEE COMPSAC, 1999, pp. 166-171.