

# On Built-in Test Reuse in Object-Oriented Framework Design

Yingxu Wang<sup>1</sup> Graham King<sup>2</sup> Mohamed Fayad<sup>3</sup> Dilip Patel<sup>1</sup> Ian Court<sup>2</sup> Geoff Staples<sup>2</sup> Margaret Ross<sup>2</sup>

<sup>1</sup> School of Computing, Information Systems and Mathematics, South Bank University, London  
103 Borough Road, London SE1 0AA, UK  
(wangy@sbu.ac.uk, dilip@sbu.ac.uk)

<sup>2</sup> Research Centre for Systems Engineering, Southampton Institute  
Southampton SO14 0YN, UK  
(graham.king@solent.ac.uk)

<sup>3</sup> Dept. of Computer Science, University of Nevada  
Reno, Nevada 89557, USA  
(fayad@cs.unr.edu)

**ABSTRACT:** Object-oriented frameworks have extended reusability of software from code modules to architectural and domain information. This paper further extends software reusability from code and architecture to built-in tests (BITs) in object-oriented framework development. Methods for embedding BITs at object and object-oriented framework levels are addressed. Behaviours of objects and object-oriented frameworks with BITs in the normal and test modes are analyzed. Systematic reuse methods of BITs in object-oriented framework development are provided. The most interesting development in the paper is that the BITs in object-oriented frameworks can be inherited and reused as that of code. Therefore testability and maintainability of the test-built-in object-oriented frameworks can be improved by the BIT approach. The BIT method can be used in analysis, design and coding of object-oriented frameworks.

**Key words:** Software engineering, object-oriented technology, framework, pattern, code reuse, framework reuse, test reuse, built-in test, testable software

## 1. Introduction

Design methodologies of object-oriented frameworks have been discussed in [1-4]. New features and approaches in object-oriented software testing have been explored in [5-10]. This paper investigates a practical built-in test (BIT) method and reuse approach of the BITs in object-oriented framework development.

Conventional testing of software is generally application-specific and hardly reusable, especially for a purchased software module or package. Even within a software development organisation, software and tests are developed by different teams and are described in separate documents. Those make test reuse particularly difficult [5,11,12].

The general concept of BIT was introduced in [5, 13-15] and the idea for inheriting and reusing conventional tests in object-oriented software was reported in [16,15]. This paper develops a practical method to incorporate both the BIT and the test reuse techniques. Applications and impacts of the reusable BITs in object-oriented framework development are explored systematically.

## 2. Built-in tests in object-oriented frameworks

The BITs are a new kind of software tests which are explicitly described in the source code of software as member functions (methods). The BITs are stand-by in normal mode and can be activated in test mode. Object-oriented frameworks extend reusability of software from code

modules to architectural and domain information. The reusable BIT method intends to further extend software reusability from code and frameworks to tests at object and object-oriented framework levels.

The BIT is a new philosophy contributing towards design of testable software. Testing of conventional object-oriented software focuses on generation of tests for existing objects and frameworks; the testable object-oriented framework design method draws attention to build testability into objects and frameworks, so that the succeeding testing processes can be simplified and reusable. The most interesting feature of the BIT techniques is that tests can be inherited and reused in the same way as that of code in conventional object-oriented frameworks. This technology can be used in analysis, design and coding of components in object-oriented framework development [17].

## 2.1 Built-in tests at object level

By embedding test declarations in the interface and test cases in the implementation of a conventional object structure [18], a prototype of BIT object can be designed as shown in Fig.1.

```
Class class-name {
    // interface
    Data declaration;
    Constructor declaration;
    Destructor declaration;
    Function declarations;
    Tests declaration;    // Built-in test declarations

    // implementation
    Constructor;
    Destructor;
    Functions;
    TestCases;           // Built-in test cases as new
                        // member functions (methods)
} TestableObject;
```

Fig.1 An object with built-in tests

The BITs can be a standard component in a test-built-in object structure. The BITs have the same syntactical functions as that of the standard constructor and destructor in an object. Therefore the BITs can be inherited and reused in the same way as that of functions within the object. The BITs can well fit into an object and object-oriented framework via C++, JAVA or any other object-oriented language compilers.

A BIT object has the same behaviours as that of the conventional one when the normal functions are called. But if the BITs are called as those of member functions in the test mode, eg:

```
TestableObject :: TestCase1;
TestableObject :: TestCase2;
.....
TestableObject :: TestCaseN;
```

the BIT object will be automatically tested and corresponding results are reported.

## 2.2 Built-in tests at object-oriented framework level

It has been proven that if each object can be tested, a system which contains the objects can be tested from bottom-up. Thus the same test-built-in method described in Section 2.1 can be extended naturally to the object-oriented framework level. An object-oriented framework with a BIT subsystem and BIT classes are shown in Fig.2. Where modules 1.n, 3.k and 2.m are the BIT class

clusters for the fully testable, partially testable and application specific subsystems respectively. Subsystem 4 is a global BIT subsystem for testing the entire framework by pre-designed event-driven threads and scenarios. The BIT classes and subsystems may introduce additional coupling for testing between classes. By limiting the coupling to be active in the test mode only, the BIT approach will not increase the complexity of an object-oriented framework.

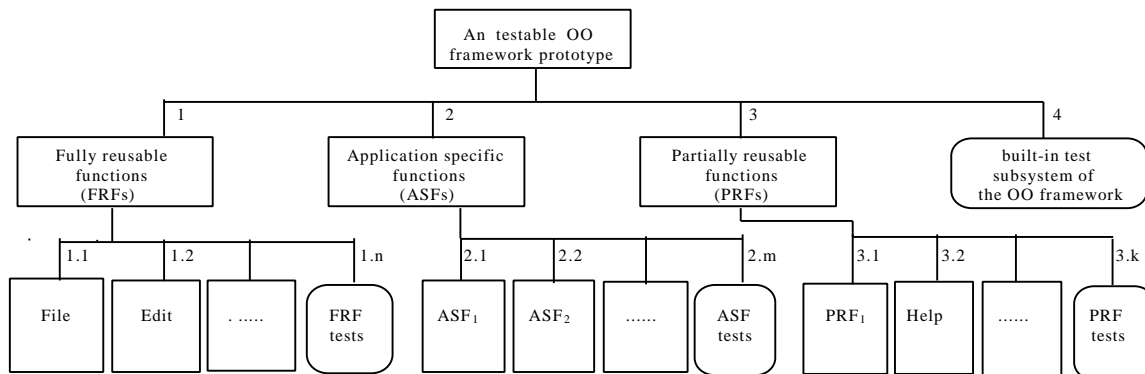


Fig.2 A prototype of test-built-in object-oriented framework

It is noteworthy, for the end-users of a BIT framework, that the BITs for the FRFs and part of the PRFs and the global BIT subsystem already exist. Therefore new BITs which an applied object-oriented framework needs to concentrate on are the ASFs and part of the PRFs and of the global subsystems. In this approach, an ideal object-oriented framework can be implemented, which is testable, test inheritable and reusable at object, class cluster and framework levels. The test-built-in framework also possesses the feature of easy maintenance since it is self-contained of code, structure as well as tests within a single source file. Thus, maintenance team and end-users of the BIT object-oriented frameworks no longer need to redesign and reanalyse the code, class structure and tests during testing and maintenance.

### 3. Reuse of built-in tests in object-oriented frameworks

Corresponding to the deployment of the BITs in an object-oriented framework, reuse of the BITs can be implemented at object, class cluster and object-oriented framework levels.

#### 3.1 Reuse of BITs at object level

Functions of a BIT object can be categorised into normal mode and test mode as shown in Fig.3. The former is applied for code reuse and the latter for test reuse.

In the normal mode, a BIT object has the same functions as that of conventional objects. The static and dynamic behaviours are the same as those of the conventional ones. The application-specific member functions can be called by `ObjectName::FunctionName`; and the BITs are stand-by and without any effect to run-time efficiency of the object.

In the test mode, the BITs in a test-built-in object can be activated by calling the test cases as member functions: `ObjectName::TestCasesI`. Each `TestCasesI` consists of a BIT driver and test cases for the specific object. Test results can be automatically reported by the BIT driver.

#### 3.2 Reuse of BITs at object-oriented framework level

Similar to the BIT object, an object-oriented framework with reusable BITs has the normal mode and test mode as shown in Fig. 4. The former is applied for code reuse and the latter for test reuse.

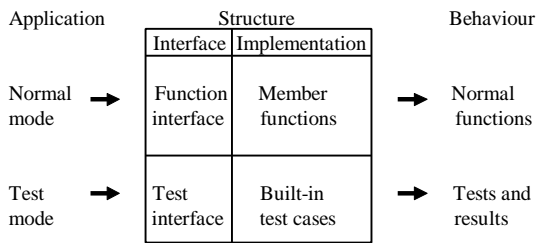


Fig.3 Reuse modes of a test-built-in object

```

Class BITsBinarySearch {
// Interface
// Member functions
BITsBinarySearch(); // The constructor
~BITsBinarySearch(); // The destructor
int BinarySearch (int Key; int DataSet[10]); // The conventional object
void BIT1(); // The built-in-tests 1...3
// Implementation
// =====
// Part 1: The conventional function code
// =====
int BinarySearch (int Key, int DataSet[10])
{
// The conventional object
// Assume: DataSet is ordered
// LastElement -FirstElement >=0
// and FirstElement >=0
// Input: Key to be found in the DataSet
// Output: TestElemIndex

Private:
int bott, top, i;
int found;

found = false;
Bott = 1;
Top = ArraySize (DataSet); // The last element in DataSet
while (bott <= top) && (not found)
{
i = floor ((bott + top)/2);
if DataSet[i] == Key
Found = true;
else if DataSet[i] < Key
Bott = i +1
else Top = i +1;
}
if found == true
return i; // The index of the element
else return 0; // An indicator of not existence
}

// =====
// Part 2: The BITs
// =====

// BIT case 1
// -----
void BIT1()
{
// BIT case 1: Array size of 1, key in array
private:
int DataSet[1] = {16};
int Key = 16;

```

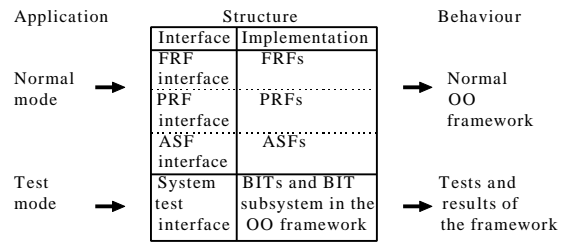


Fig.4 Reuse modes of a test-built-in object-oriented framework

```

int StdElemIndex = 1;
int TestElemIndex;
char TestResult1[5];
// Test implementation
TestElemIndex = BinarySearch (Key, DataSet);
// Test analysis
cout << "StdElemIndex1 = " << StdElemIndex << "\n";
cout << "TestElemIndex1 = " << TestElemIndex << "\n";
if TestElemIndex == StdElemIndex
TestResult1 = "OK";
else TestResult1 = "FALSE";
cout << "TestResult1: " << TestResult1 << "\n";
}
}

```

Fig. 5 A BIT object of binary search

```

Class DatabaseQuery: public BITsBinarySearch
{
// Part 1: The inherited conventional functions
// =====
int DatabaseQueryBinarySearch (int Key, int DataSet[10]) :
BITsBinarySearch::BinarySearch(int Key; int DataSet[10]);

// Part 2: The inherited BIT functions
// =====
void BIT1() : BITsBinarySearch::BIT1();

// Part 3: The newly developed BITs
// =====

// BIT case 2
// -----
void BIT2()
{
// BIT case 2: Even array size, key 1st element in array
Private:
int DataSet [6] = {16,18,21,23,29,33};
int Key = 16;
int StdElemIndex = 1;
int TestElemIndex;
char TestResult2 [5];
// Test implementation
TestElemIndex = BinarySearch (Key, DataSet);
// Test analysis
cout << "StdElemIndex2 = " << StdElemIndex << "\n";
cout << "TestElemIndex2 = " << TestElemIndex << "\n";
if TestElemIndex == StdElemIndex
TestResult4 = "OK";
else TestResult2 = "FALSE";
cout << "TestResult2: " << TestResult2 << "\n";
}
}

```

Fig. 6 Inheritability and reusability of BITs to support component-based maintenance

In the normal mode, a BIT object-oriented framework performs the same functions as the conventional frameworks. Its static and dynamic behaviours are the same as those of the conventional ones. The normal FRF, PRF and ASF functions in the test-built-in object-oriented framework can be called by `ObjectName::FunctionName`; and the BIT class clusters and subsystem are stand-by and without any effect to the run-time efficiency of the object-oriented framework.

A BIT object-oriented framework has testing mechanisms ready at system, class cluster and object levels from top-down as shown in Figs. 2 and 4. The end-users of an applied BIT object-oriented framework can call and reuse all BITs as member functions in the test mode. Framework users can also embed additional BITs in the application-specific subsystem and classes of a framework. As limitation of space, case studies of the BITs method will be reported separately.

#### **4. A case study on reuse of built-in tests**

A typical binary search object with the BITs as special member functions is show in Fig.5. The BIT object is implemented in two parts: the conventional functions and the BIT functions. Only one of the possible test cases have been built-in to show the method of BITs.

In the normal mode, the conventional functions described in Fig.5 can be executed by calling: `BITsBinarySearch::BinarySearch(int Key, int DataSet[10])`. In the test mode, the embedded BIT components in the class can be reused by calling the following `BITsBinarySearch::BIT1()`.

Approach to reuse the BIT components in maintenance is shown in Fig.6. When a new object, `DatabaseQuery`, is newly developed, the BIT components (part 2, Fig.6) developed in the `BITsBinarySearch` can be inherited and reused directly as that of the conventional member functions (part 1, Fig. 6). Also additional BITs can be incorporated into the new object as shown in part 3 of Fig. 6.

In the new BIT object `DatabaseQuery` listed in Fig. 4, the existing BITs developed in the `BITsBinarySearch` object can still be activated by calling: `DatabaseQuery::BIT1()`; (equivalent to `BITsBinarySearch::BIT1()`); and the new BITs supplemented in the `DatabaseQuery` object can be activated in the same way: `DatabaseQuery::BIT2()`.

#### **5. Conclusions**

This paper has developed a method to incorporate the reusable built-in tests (BITs) into object-oriented frameworks. The reuse approaches to the BITs in object-oriented framework development have been analyzed. The BIT method has extended the reusability of object-oriented frameworks from code to tests, so that highly testable and test reusable object-oriented frameworks can be developed on the same platform of the conventional object-oriented software.

The BIT method is a natural advance and supplementary to the conventional object-oriented technology for object-oriented framework development. A wide range of applications of the BIT method has been found in analysis, design and coding of components and object-oriented frameworks. It is significant that the BIT method can incorporate any test cases generated by black-box (functional) and/or white-box (structural) methods as components in object-oriented frameworks.

#### **Acknowledgements**

The authors would like to acknowledge the support of European Software Institute and the IVF Centre for Software Engineering. We would like to thank the referees for their valuable comments.

## References

- [1] Fayad, M. and Schmidt, D. [1997], Object-Oriented Application Frameworks, *Communications of the ACM*, Vol.40, No.10, October.
- [2] Buschmann, F., R. Meunier, H. Rohnert, P. Sommerlad and M. Stal [1996], Pattern-Oriented Software Architecture - A System of Patterns, Wiley, New York, 1996.
- [3] Jacobson, I. M. Christerson, P. Jonsson and G. Overgaard [1992], Object-Oriented Software Engineering, Addison-Wesley.
- [4] Schmidt, D.C. [1995], Using Design Patterns to Develop Reusable Object-Oriented Communication Software, *Communications of the ACM*, Vol.38, No.10, pp.65-74.
- [5] Binder, R.V. [1994], Design for Testability in Object-Oriented Systems, *Communications of the ACM*, Vol. 37, No. 9, Sept., pp. 87-101.
- [6] McGregor, J.D. and T.D. Korson [1994], Integrating Object-Oriented Testing and Development Processes, *Communications of the ACM*, Vol. 37, No. 9, Sept., pp. 59-77.
- [7] Jorgensen, P.C. and Erickson, C. [1994], Object-Oriented Integration Testing, *Communications of the ACM*, Vol. 37, No. 9, Sept., pp. 30-38.
- [8] Poston, R.M. [1994], Automated Testing from Object Models, *Communications of the ACM*, Vol. 37, No. 9, Sept., pp. 48-58.
- [9] Murphy, G.C., P. Townsend and P. S. Wong [1994], Experiences with Cluster and Class Testing, *Communications of the ACM*, Vol. 37, No. 9, Sept., pp. 39-47.
- [10] Arnold, T.R. and W.A. Fuson [1994], Testing “in a Perfect World”, *Communications of the ACM*, Vol. 37, No. 9, Sept., pp. 78-86.
- [11] Freedman, R.S. [1991], Testability of Software Components, *IEEE Transactions on Software Engineering*, Vol.17, No.6, June, pp.553-564.
- [12] Voas, J.M. and Miller, K.M. [1995], Software Testability: The New Verification, *IEEE Software*, Vol.12, No.3, May, pp.17-28.
- [13] Firesmith, D.G. [1994], Testing Object-Oriented Software, *Proceedings of Object Expo. Europe*, Sept., pp. 97 - 130.
- [14] Wang, Y. [1988], Testability Theory and Its Application in the Design of Digital Switching Systems, *Journal of Telecommunication Switching*, Vol.17, pp.30-36.
- [15] Wang, Y., Court, I., Ross, M., Staples, G. and King, G. [1997], On Testable Object-Oriented Programming, *ACM Software Engineering Notes*, July, Vol.22, No.4, pp.84-90.
- [16] Harrold, M.J., J.D. McGregor and K.J. Fitzpatrick [1991], Incremental Testing of Object-Oriented Class Structures, *Proceedings of the 14<sup>th</sup> International Conference on Software Engineering*, March.
- [17] Wang, Y., King, G., Court, I., Ross, M. and Staples, G. [1997], On Built-in Tests in Object-Oriented Reengineering, *Proceedings of 5th ACM Symposium on FSE/6th European Conference on Software Engineering/Workshop on Object-Oriented Reengineering (FSE/ESEC/WOOR'97)*, pp. 3.6.1-3.6.5.
- [18] Stroustrup, B. [1986], The C++ Programming Language, Addison-Wesley publishing Company.

Permission to make digital/hard copy of part or all of this work for personal or classroom use is granted without fee provided that the copies are not made or distributed for profit or commercial advantage, the copyright notice, the title of the publication, and its date appear, and notice is given that copying is by permission of the ACM, Inc. To copy otherwise, to republish, to post on servers, or to redistribute to lists, requires prior specific permission and/or a fee.

© 2000 ACM 0360-0300/00/0300es