ESEC/ACM FSE/WOOR’97 Zurich, September 22-26, 1997

On Built-in Tests in Object-Oriented Reengineering

Y. Wang, G. King, I. Court, M. Ross and G. Staples

Research Centre for Systems Engineering
Southampton Institute, Southampton SO14 0YN, UK
Tel: +44 1703 319773, Fax: +44 1703 334441
Email: wang_s@solent.ac.uk or *yingxu.wang@comlab.oxford.ac.uk*

Abstract: This paper provides a new approach for object-oriented reengineering. One of the difficulty in software testing and maintenance has been identified as caused by the convention that code and its tests are developed and described separately. This paper develops a method of built-in test (BIT) for OO reengineering. The advantage of this method is that the BITs in reengineered OO software (OOS) can be inherited and reused as that of code. Therefore building tests into the conventional software during OO reengineering can be a significant reengineering approach for the existing OO legacy systems.

Key words: Software engineering, OO reengineering, OOP, built-in test, test reuse, testable software

1. Introduction

Object-oriented programming (OOP) [1,2] is the natural extension of the structured programming [3] and abstract data types (ADTs) [4] with new concept of abstraction, encapsulation, inheritance and reusability. The most powerful feature of OOP is the availability of reuse by inheriting code and structure information at object and system levels. A recent development on OOP is the test-built-in software [5-9] in which the tests are self-contained rather than in separated documents, so that tests can be inherited and reused for the first time as that of code in OOS development and OO reengineering.

This paper develops a method of built-in test (BIT) for OO reengineering. Reengineered OOS, with BITs at object and system levels, is self testable, test reusable, and easy maintainable. The BIT method can be fitted into the conventional OOP techniques and OO compilers directly and has drawn much interests in the software industry.

2. Testing problems in software reengineering

Conventional testing and maintenance of software are generally application-specific and hardly reusable, especially for a purchased software module or package. Even within a software development organisation, coding, testing and maintenance are carried out by different teams and are described in separate documents. These conventions make testing and maintenance particularly difficult.

BIT is a new philosophy contributing towards the development of testable and maintainable software, and a new approach for OO reengineering. Testing and maintenance of conventional software focuses on the existing objects and systems; the testable OOS design method draws attention to build test mechanisms into objects and systems during design and coding, so that the succeeding testing and maintenance processes can be simplified. The most interesting feature of BIT technique is that tests, for the first time, can be inherited and reused in the same way as that of code in the conventional OO software. The BITs enables the OOS more testable and maintainable for both newly developed and/or reengineered software, because of its self containment of code, structure, as well as tests within a single source file. Thus the maintenance team and the end users of the OOS are no longer need to redesign and reanalyse the code, class structure and tests for a software system in maintenance and testing.

3. Built-in tests in OO reengineering

This section develops the OO reengineering method with BITs based on the authors' recent work [5,7]. By reengineering the conventional software with OO and the BIT methods, highly testable, test reusable, maintainable and self-testable software can be implemented at object and system levels.

3.1 Built-in-test at object level

The general structure of an object in OOP is shown in Fig.1. An object consists of two structural parts: the interface and the implementation. The interface of an object is the only mean for external access to the member functions contained in the object. The implementation of an object is the description of codes for all member functions.

```
Class class-name {
                    // interface
                      data declaration;
                      constructor declaration;
                      destructor declaration;
                      function declarations;

                    // implementation
                      constructor;
                      destructor;
                      functions;
                   } [object-name-list];
```

Fig.1 A typical prototype of an object

An object is reusable because of its features of encapsulation and inheritability. A prototype of test-built-in object can be developed as shown in Fig.2. It is an extension of the conventional object structure by embedding test declarations in the interface and test cases in the implementation. Then the tests can be inherited and reused in the same way as that of member functions within the object.

```
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
                             } TestableObject;
```

Fig.2 An object with BITs

The BITs have the same syntactical functions as that of the standard constructor and destructor in an object. Therefore we have no difficulty to fit the BITs into an object via C++ or any other OO compilers.

OO reengineering with BIT can be carried out at object level as shown in Fig.2. In the normal mode, an testable object has the same structure as the conventional object. The static and dynamic behaviours of the BIT object are the same as those of the conventional objects. The member functions can be called by ObjectName::FunctionName; and the BITs are stand-by and without any effect to the run-time efficiency of the object.

In the test/maintenance mode, the BITs in a testable object can be activated by calling the test cases as member functions, ie:

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

Each TestCaseN consists of a test driver and some test cases for the specific object. The test results can be automatically reported by the BIT driver.

3.2 Built-in-test at system level

The BIT method developed at object level can be naturally extended to the system level in OO reengineering. An OOS with built-in testing classes and subsystem are shown in Fig.3. Where modules 1.n, 3.k and 2.m are the built-in testing classes for the fully reusable, partially reusable and application-specific function subsystems respectively. The subsystem 4 is a newly built-in testing subsystem for the entire system.

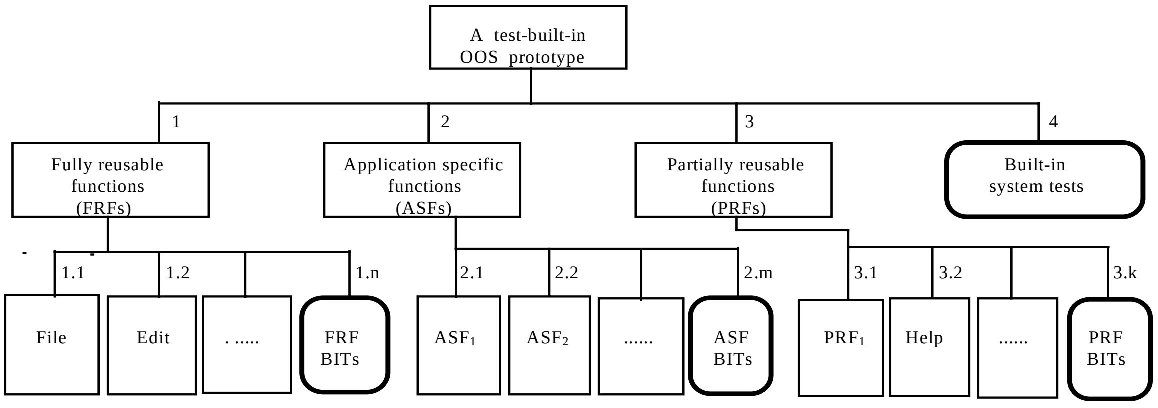
A test-built-in OOS prototype
1
2
3
4
Fully reusable functions (FRFs)
Application specific functions (ASFs)
Partially reusable functions (PRFs)
Built-in system tests
1.1
1.2
1.n
2.1
2.2
2.m
3.1
3.2
3.k
File
Edit
......
FRF BITs
ASF1
ASF2
......
ASF BITs
PRF1
Help
......
PRF BITs

Fig.3 Deployment of built-in tests in OOS

Within each classes, such as in the ASF_1, ASF_2, PRF_1 and etc, the testable mechanisms described in section 3.1 can be adopted in every object. By this way, ideal OO reengineered software with the BITs is implemented. The most interesting features of the BIT reengineered software are that it is test inheritable, test reusable and self testable at object, class and system levels.

In the normal mode, similar to the BIT object, the static and dynamic behaviours of the OO reengineered system with BIT are the same as those of the conventional ones. The FRF, PRF and ASF functions in the test-built-in OOS can be called by ObjectName::FunctionName; and the built-in testing classes are stand-by and without any effect on the run-time efficiency of the testable OOS.

In the test/maintenance mode, the BITs at object, class and system levels can be activated by calling the test cases at the corresponding levels as member functions: ObjectName::TestCaseN. Each TestCaseN consists of a test driver and some test cases for a specific object. The test results can be automatically reported by the BIT driver.

Applying the BIT method in reengineering an existing OO system can improve the software design quality, testability, test reusability and maintainability dramatically. Several institutions are planning to apply the BIT method in their system reengineering and implementation.

4. Conclusions

We are seeking new methodologies supplementary to OOP in software development and OO reengineering. This paper extends software reuse methods from code to the BITs. Incorporating the BITs during OO reengineering makes the existing OOS highly testable, test reusable and maintainable based on the self-contained testing mechanisms. Combining the BIT method with the OOP approach, high quality and productivity can be benefited in reengineering the existing systems.

The BIT methods provide a new approach for OO reengineering methodologies. The most interesting features of the BIT method are that the BITs can be systematically reused at object and system levels. Incorporating the BIT method with the OO framework techniques in OO reengineering, the existing OO legacy systems can be well reengineered for higher design quality on system testability, maintainability and reliability.

Acknowledgements

The authors would like to acknowledge the support of the European Software Institute, IVF Centre for Software Engineering, and BCS QSig.

References

[1] Snyder, A. [1987], Inheritance and the Development of Encapsulated Software Components, in *Research Directions in Object-Oriented Programming*, (Shriver and Wagner, eds.), MIT Press, pp.165-188.
[2] Stroustrup, B. [1986], The C++ Programming Language, *Addison-Wesley publishing Company.*
[3] Hoare, C.A.R., E-W. Dijkstra and O-J. Dahl [1972], Structured Programming, *Academic Press, New York.*
[4] Liskov, B. and Zilles, S [1974], Programming with Abstract Data Types, *ACM SIGPLAN Notices, Vol.9,* pp.50-59.
[5] Wang, Y., G. King, Court, I., Ross, M. and Staples, G. [1997], On Testable Object-Oriented Programming, *ACM Software Engineering Notes,* Vol. 22, No. 4, pp. 84-90.
[6] Wang, Y., Trujillo J. and Palomar M. [1997], On a Metric of Software Testability, *Journal of the Spanish Computer Society (Novatica),* Vol. 125, Jan/Feb Issue, pp. 10-13.
[7] Wang, Y., Staples, G., Ross, M., King, G. and Court, I. [1996] On a Method to Develop Testable Software, *Proc. of IEEE European Testing Workshop (IEEE ETW'96),* Montpellier, France, June, pp. 176-180.
[8] Wang, Y., Staples G., Ross M. King, G., and Court I. [1996], A New Approach to Develop Testable Software, *Proceedings of the International Conference on Software Quality (ICSQ'96),* Ottawa, Canada, October, pp. 322-334.
[9] Wang, Y. [1995] On Testable Software and Built-in Test Reuse in OOP, *technical Report of Oxford University Computing Laboratory*, OUCL-WANG-95002.