On Testable Object-Oriented Programming

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: A new philosophy contributing towards the design of testable object-oriented (*OO*) software is introduced in this paper. The testing of conventional *OO* software focuses on the generation of tests for existing objects and systems; the testable object-oriented programming (*TOOP*) method draws attention to building testabilities into objects and systems during coding or compiling, so that the succeeding processes in test generation and implementation can be simplified. A new method of *TOOP* is developed to improve the testability of *OO* software. Software testability at object level and system level is quantitatively modelled. A set of fundamental built-in testable mechanisms oriented to the basic control structures in objects is constructed in order to improve the testability of *OO* software in terms of test controllability and observability. The most interesting feature obtained by *TOOP* is that the built-in tests in any objects can be inherited and reused in the same way as that of codes or functions in conventional *OO* software.

Key words: Software engineering, software test, OOP, testable OOP, built-in test, basic control structures

1. Introduction

Approaches to programming have been changed dramatically since the invention of computers. The primary reasons for the change are: to accommodate the increasing complexity; to ensure correctness; and to improve the productivity of software. Object-oriented programming (*OOP*) has been broadly accepted since the 1980s when C++ emerged as a powerful *OOP* language [1,2]. *OOP* has taken the best ideas of structured programming and combined them with several new concepts such as abstraction, encapsulation, inheritance and reusability. An object is a specific instance of a class. A general form of the structure of an object in *OOP* is shown in Fig.1.

Conventionally, the design and testing of *OO* software (*OOS*) are relatively separate phases and independent activities. This causes many problems in test generation and implementation, such as over complexity, less than thorough testing, and extremely high cost [3-7].

How may the objects in *OOS* be made testable? How may tests be built into objects so that the tests can be inherited and reused as are functions in the objects? Oriented to the problems, a new testable *OOP* (*TOOP*) method is developed in this paper. The aim of *TOOP* is to build testable mechanisms into objects during coding or compiling, so that the succeeding processes of testing can be simplified, and the built-in tests for an object can be inherited and reused.

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

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

Fig.1 A typical prototype of an object

Conventional *OOS* is not fully testable or is difficult to test thoroughly [8, 9]; even a simple concatenated object with loops and multiple branches could be very hard to test within acceptable complexity. Therefore, it is still necessary to seek a new approach to *TOOS* development. In the following sections, a measurement model for the formal and quantitative description and assessment of the *OOS's* testability is created. A set of built-in testable mechanisms is developed at object level and *OOS* level. The method of *TOOP* and its application in improving the testability of *OOS* are provided.

2. Measurement model of *OOS* testability

This section derives the definition of testable *OOS*. Nature of software testability at flow control statement, object and system level is investigated hierarchically. This leads to the development of a measurement model for assessing the testability of *OOS* systematically and quantitatively.

2.1 Definition of testable *OOS*

As preparation for deriving the definition of testable *OOS*, we first introduce the fundamental concept of basic flow control structures [10, 11] in software.

Def. 1. The basic control structures *(BCSs)* of software are those fundamental statements which control the program flow, such as condition, iteration and sequence.

Based on the concept of *BCSs*, the testable *OOS* can be defined as follows.

Def. 2. The testable *OOS (TOOS)* is an *OO* software in which built -in testable mechanisms (as defined in Section 3), that can be activated independently in test mode, are adopted in all the *BCSs* through the software.

2.2 Controllability of *BCS*

The difficulties in testing conventional software at *BCS* level are often caused by *path-sensitivity* and *mutual-dependence* [12, 13] of the Boolean variables or expressions in a *BCS*. For instance, for an object given in Fig.2, p in BCS_2 and X in BCS_3 are path-sensitive since their specific values depend on the actually executed paths previous to them; and the control expression in BCS_3, $exp = X \wedge Y$, is mutually-dependent not only on X but also on Y.

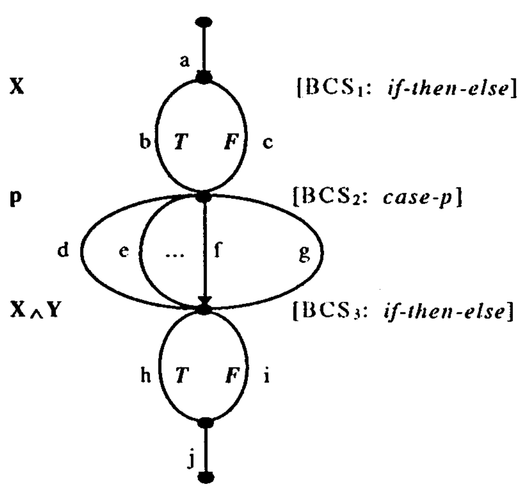

Fig.2 Flowgraph of an object

Def. 3. The controllability of a *BCS* in an object is the capability to independently assign the control variable or expression of the *BCS* at the interface of the object, so that any path(s) controlled by the *BCS* can be accessed independently.

Let C_{BCS_i} represent the controllability of BCS_i, the *BCS's* controllability can be quantitatively determined by

$$C_{BCS_i} = 1, \text{ if } BCS_i \text{ is independently assignable}$$
$$= 0, \text{ if } BCS_i \text{ is path-sensitive or mutually-dependent} \quad (1)$$

2.3 Object testability (*OTA*)

Design of tests for *OOS* concerns both stimulations and responses; Consequently, the testability of *OOS* needs to be studied via two aspects - test controllability (*TC*) and test observability (*TO*).

Def. 4. The test controllability *TC* of an object is defined as the capability to independently control all the *BCSs* within the object.

The physical meaning of *TC* is the ability to force an object to execute in any expected path in test mode, by assigning the control variables of each *BCS* within the tested object. *TC* can be determined by the following formula

$$TC = \frac{1}{n}\sum_{i=1}^{n} C_{BCS_i} \quad (2)$$

Where the C_{BCS_i} is the controllability of each *BCS* in the object.

The domain of *TC* is in $[0, 1]$. An object with $TC=1$ means it is fully controllable; $TC=0$ is non-controllable. Otherwise $0 < TC < 1$ implies it is partially controllable.

For example, in Fig.2, the BCS_3 determined by $X \wedge Y$ is non controllable because X may take a value of *false* or it is path-sensitive to p. The BCS_2 is also non controllable when the control variable p is sensitive to the paths b or c prior to it. In this case, the controllability of the object shown in Fig.2 can be calculated according to Formula 2 as

$$\begin{aligned} TC &= \frac{1}{n}\sum_{i=1}^{n} C_{BCS_i} \\ &= 1/3 * (C_{BCS_1} + C_{BCS_2} + C_{BCS_3}) \\ &= 1/3 * (1 + 0 + 0) \\ &= 0.333 \end{aligned}$$

Def. 5. The test observability *TO* of an object is the capacity to indicate the values of any variables within the path(s) sensitised by the current testing case.

TO has the same domain as *TC*. $TO=1$ or $TO=0$ represents a full or non observability respectively for an object. Since full observability is easier to be implemented by inserting a probe instrument such as *write* or *print* in the code, it is assumed that $TO \equiv 1$ from now on.

It is axiomatic that the higher both the *TC* and *TO*, the better the testability of an object. Thus based on the definitions of *TC* and *TO*, object testability (*OTA*) can be derived as below.

Def. 6 The object testability *OTA* is a product of its test controllability *TC* and observability *TO*, i.e.

$$OTA = f(TC, TO) = TC * TO \quad (3)$$

Formula 3 indicates that $OTA=1$ if both *TC* and *TO* are *1*;
$OTA=0$ if either *TC* or *TO* is *0*. More generally

$$0 \leq OTA \leq 1 \quad (4)$$

for $0 \leq TC \leq 1$ and $0 \leq TO \leq 1$.

Considering that $TO \equiv 1$, as well as the definition of *TC* in Formula 2, Formula 3 can be simplified in the form of

$$OTA = TC * 1 = \frac{1}{n}\sum_{i=1}^{n} C_{BCS_i} \quad (5)$$

Applying Formula 5 to evaluate the testability of the given object in Fig.2, its *OTA* can be determined as

$$\begin{aligned} OTA &= \frac{1}{n}\sum_{i=1}^{n} C_{BCS_i} \\ &= 1/3 * (C_{BCS_1} + C_{BCS_2} + C_{BCS_3}) \\ &= 1/3 * (1 + 0 + 0) \\ &= 0.333 \end{aligned}$$

The *OTA* of this object is much lower than *1* for the expected full testability, so it needs to be improved by the methods described in the following sections.

2.4 System testability (*STA*)

Def. 7 The system testability *STA* of *OOS* is defined as a mathematical mean of all the objects' testability obtained in the *OOS*.

The definition of *STA* can be expressed as

$$\begin{aligned} STA &= \frac{1}{m}\sum_{j=1}^{m} OTA_j \\ &= \frac{1}{m}\sum_{j=1}^{m} TC_j \\ &= \frac{1}{\sum_{j=1}^{m} n_j}\sum_{j=1}^{m}\sum_{i=1}^{n_j} C_{BCS_{ji}} \end{aligned} \quad (6)$$

Where m is the number of objects in the software, and n_j is the number of *BCSs* in the *j*th object.

Formula 6 indicates that if all the $C_{CBS_{ji}} = 1$ for $1 \leq j \leq m$ and $1 \leq i \leq n_j$, or $TC_j = 1$ or $OTA_j = 1$ for $1 \leq j \leq m$, then the system's testability $STA=1$ is obtained and the *OOS* is fully testable. Otherwise, the testability of the software needs to be improved by the testable *OOP* methods provided in the following sections.

By obtaining the testability measurement models of *OOS* at *BCS* level (C_{BCS}), object level (*OTA*) and system level (*STA*),
the following theorem of *TOOS* is reached.

Theorem 1. A given *OOS* is testable *iff*

a) $STA=1$; or
b) $OTA_j = 1$, $j = 1, 2, \ldots, m$; or
c) $C_{CBS_{ji}}=1$, $1 \leq j \leq m$, $1 \leq i \leq n_j$

is satisfied in the *OOS*. ❑

Theorem 1 as well as Formulae 5 and 6 indicate that the main approach to *TOOP* is to increase the test controllability at *BCS* level, so that the full testability can be obtained at object level and system level consequently.

3. Built-in testable structures in objects

This section describes a set of fundamental techniques that help to improve the testability of *OOS*. Emphasis will be put on how to build testable mechanisms into *BCSs*, so that the required full controllability can be fulfilled in any objects and thus in a software system.

3.1 Testable *BCS* schema in *OOS*

A general testable schema for the *BCSs* of an object can be constructed as follows

$$\begin{aligned} \exp &\Rightarrow \textit{mode} \wedge \exp \vee \textit{test} \\ &= (m_g \vee m_i) \wedge \exp \vee (t_g \vee t_i) \end{aligned} \quad (7)$$

Where, *exp* - initial Boolean expression in the *BCSs*
mode - mode selector, Boolean expression
test - test selector, Boolean expression
m_g - system global mode selector, Boolean
t_g - system global test selector, Boolean
m_i - individual mode selector for BCS_i, Boolean
t_i - individual test selector for BCS_i, Boolean

For describing the testable structures tidily in the following formulae, m_g and m_i in Formula 7 will be simply represented by m, t_g and t_i by t, and *exp* by e respectively.

3.2 *BCSs* with built-in testable mechanisms

The built-in testable mechanisms for *BCSs* can be formally described as follows, by which the *BCSs* in *OOS* can be testably constructed.

3.2.1 If-Then structure

The built-in testable mechanism for a *simple conditional* structure in an object can be defined as follows.

```
If e then P
    ⇒
    if [(m ∧ e) ∨ t]
                        then P;
```

(8)

3.2.2 If-Then-Else structure

Formula 9 describes the built-in testable mechanism of a *general conditional* structure in *OOS*.

```
If e then P else Q
    ⇒
    if [(m ∧ e) ∨ t]
       then P;
                        else Q;
```

(9)

3.2.3 While-Do structure

The built-in testable mechanism for *while-type iterative* structure of *OOS* can be represented in the following formula.

```
While e do P
    ⇒
    while [(m ∧ e) ∨ t]
                        do P;
```

(10)

3.2.4 Repeat-Until structure

The built-in testable mechanism of a *repeat-type iterative* structure in *OOS* can be expressed as follows.

```
Repeat P until e
    ⇒
    repeat P
                  until [(m ∧ e) ∨ t ];
```

(11)

3.2.5 For-Do structure

The testable mechanism of *for-do* iteration in *OOS* can be defined as follows. The main approach is to independently control the lower and upper bounds of the *for*-loop by introducing two predefined testable control variables t_a and t_b.

```
For i:=i_a to i_b do
   P;
   ⇒
   if [m ∧ (¬ t)]
       then // Normal mode
            For i:=i_a to i_b do
                P;
            else // Test mode
                {
                  i_a := t_a ;
                  i_b := t_b ;
                  // t_a , t_b are the built-in testing integers to
                     // control the iterating times of the for loop
                  For i:=i_a to i_b do
                      P;
                                                     };
```

(12)

3.2.6 Case structure

The testable mechanism of *case* structures can be described in the following formula. Notice that p is controlled via a predefined path selection variable k in the test mode.

```
Case p do
   { (0: P_0 / 1: P_1 / 2: P_2 / ... / n-1: P_{n-1});
     skip;
   }
   else {
       P_n ;
       skip;
       }
   ⇒
   if [m ∧ (¬ t)]
      then // Normal mode of case
          Case p do
              { (0: P_0 / 1: P_1 / 2: P_2 / ... / n-1: P_{n-1});
                skip;
              }
              else {
                  P_n ;
                  skip;
                  }
      else // Test mode of case
          {
```

```
        p := k;
            // k is a built-in integer variable for selecting the
paths
        Case p do
            { (0: P_0 / 1: P_1 / 2: P_2 / ... / n-1: P_{n-1});
              skip;
            }
            else { P_n ;
                   skip;
                 }
                                                                 };
```

(13)

Applying these built-in testable *BCSs* in *TOOP*, the full testability for objects and thus for the whole *OOS* can be obtained.

4. The approach to testable *OOP*

This section describes how to implement *TOOP* by building the testable *BCSs* developed in Section 3 into *OOS*, in order to meet the testable requirements described in Section 2. The inheritability and reusability of tests at object and system levels are illustrated.

Testable objects with built-in testability can be implemented by adopting the testable *BCSs* developed in Section 3, i.e., by replacing all the normal *BCSs* in an object with the corresponding testable *BCSs*. For example, the object described in Fig.2 can be testably redesigned by embedding the testable mechanisms into BCS_{1-3} by assigning

$$e_1 := (m_g \vee m_1) \wedge X \vee (t_g \wedge t_1) \tag{14}$$

$$p := e_2 \quad \text{if } (\neg(m_g \vee m_2)) \wedge (t_g \wedge t_2) \tag{15}$$

and

$$e_3 := (m_g \vee m_3) \wedge \langle X \wedge Y \rangle \vee (t_g \wedge t_3) \tag{16}$$

as shown in Fig.3.

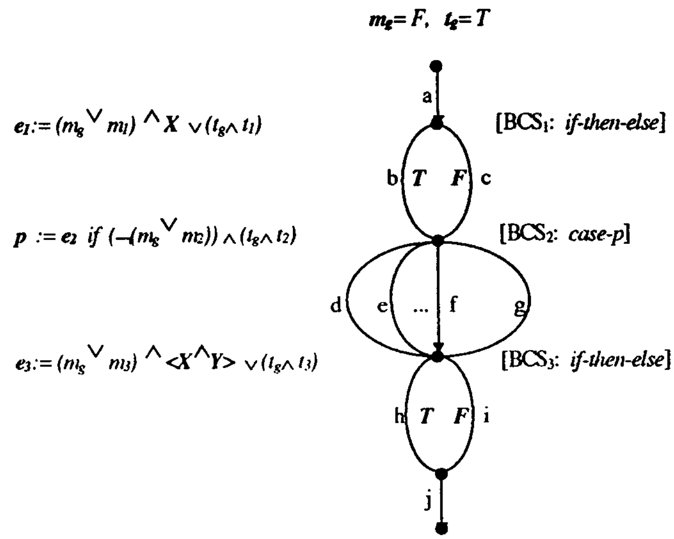

Fig.3 Flowgraph of the redesigned testable object

Where m_g is the Boolean global mode selector

m_g = true, if in *normal* mode
= *false*, if in *test* mode

(17)

t_g is the Boolean global test selector,

t_g = *true*, if in *test* mode
= *false*, if in *normal* mode

(18)

m_1, m_2 and m_3 are the individual mode selectors for BCS_{1-3}, t_1, t_2 and t_3 are the individual test selectors for BCS_{1-3}, which have the same definition as that of their corresponding global mode or test selectors given in Formulae 17 and 18. The Boolean variables e_1, e_2 and e_3 are the testable control expressions of BCS_{1-3} with the built-in testability. The rest of the variables, such as X, Y and p, are the original control variables or expressions in the object.

Since now the testable BCS_1, BCS_2 and BCS_3 can be independently controlled in test mode by letting m_g = F, t_g = T, as well as m_1, m_2 and/or m_3 be false, and t_1, t_2 and/or t_3 be true correspondingly, the testability of the object becomes

$$\begin{aligned} OTA &= \frac{1}{n}\sum_{i=1}^{n} C_{BCS_i} \\ &= 1/3 * (C_{BCS_1} + C_{BCS_2} + C_{BCS_3}) \\ &= 1/3 * (1 + 1 + 1) \\ &= 1 \end{aligned}$$

Therefore the expected full controllability and thus the full testability are realized in the object.

Referring to the general structure of a conventional object given in Fig.1, the testable mechanisms for an testable object need to be explicitly declared in the interface of the object, which include the control variables for test sensitising, the test cases and their corresponding responses as shown in Fig.4.

Class *class-name* {
  // interface
  data;
  constructor;
  destructor;

  test interface *{*
  test control variables
  Boolean m_g , // the global mode selector
    t_g , // the global test selector
    m_i , // the individual mode selector, $1 \le i \le n$,
      // n is the number of *BCSs* in the object
    t_i ; // the individual test selector, $1 \le i \le n$

  tests
  T_τ ; // the built-in tests, $1 \le \tau \le k$,
      // k is the number of the built-in tests
  test responses
  R_τ ; // the built-in expected test responses
      // corresponding to T_τ , $1 \le \tau \le k$
      }

  // function implementations;
  if (m_g $\wedge$ ($\neg$ t_g))
    then // *normal* mode
      functions;
    else // ***test*** **mode**
      {
      if $\neg m_i \wedge t_i$ // $i = 1, 2, ..., n$
        then ***testable BCS***$_i$;
        else *normal BCS*$_i$;
      ......
      }
} [*object-name-list*];

Fig. 4 The specification of a testable object

Adopting this approach, the implementation of a testable object and the sensitisation of the built-in tests are fully transparent to the testing team, so that testing can be carried out straightforwardly without knowing the internal structures of the tested objects, and without further generating tests for the objects. Thus testable objects with built-in inheritable and reusable testing mechanisms can be implemented.

Testable *OO* software *(TOOS)*, with built-in testability, can be implemented at system level if all the objects in it are testably designed in the above approach. The method of *TOOP* is suitable for developing testable codes manually or automatically. In the latter case, all the testable mechanisms can be automatically inserted into the conventional *OOS* by a special *TOOP* compiler with the testability built-in functions. For the time-critical software, a testable mode (for testing and debugging) and a normal mode (for execution) can be designed and compiled separately, so that both benefits of testability and run-time efficiency for the *TOOS* can be obtained.

5. Conclusions

There is a significant trend in the study of methodologies for testable software development. Coding and testing were usually separated in conventional *OOS* development [8,9,12-14]. The *TOOP* method is developed to support the implementation of design for testability in *OOS*, and of building testable mechanisms into objects during coding or compiling. A systematic approach to *TOOP* has been provided, and the testability of *OOS* at *BCS* level (C_{BCS}), object level (*OTA*) and system level (*STA*) are quantitatively modelled. A set of fundamental built-in testable mechanisms oriented to the common *BCSs* in objects has been created to improve the testability of *OOS* in the terms of test controllability (*TC*) and test observability (*TO*). Based on the *TOOP* method a new approach to develop *TOOS* is established which is applicable for both programmers and compiler designers. By adopting the philosophy of design for testability and by applying the *TOOP* method, the tests embedded in objects, as well as codes in it, can be inherited and reused for the first time, so that the testing and maintenance of *OOS* may be largely simplified.

Acknowledgements

The authors would like to thank C.A.R. Hoare for his support and comments. We wish thank our colleagues for their helpful suggestions.

References

[1] Stroustrup, B. [1986] The C++ Programming Language, *Addison-Wesley.*

[2] 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.

[3] Wang, Y. [1995] On the Design of Testable Software, *Research Report of Oxford University Computing Laboratory*, OUCL-WANG-95002.

[4] Voas, J.M. and Miller, K.M. [1995] Software Testability: The New Verification, *IEEE Software*, Vol.12, No.3, May, pp. 17-28.

[5] Freedman, R.S. [1991] Testability of Software Components, *IEEE Transactions on Software Engineering*, Vol.17, No.6, June, pp.553-564.

[6] 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.

[7] Wang, Y., King, G., Staples, G., Ross, M. and Court, I. [1996] Towards a Metric of Software Testability, *Proc. of 5th International Conference on Software Quality (5SQC)*, Dundee, UK, July, pp. 234-241.

[8] Tai, K. [1989] What to Do Beyond Branch Testing, *ACM Software Engineering Notes*, Vol. 14, No.2, pp.58- 61.

[9] Pressman, R. [1992] Software Engineering: A Practitioner's Approach (3rd ed.), *McGraw-Hill International Editions*, pp.595-630.

[10] McCabe, T. [1976] A Software Complexity Measure, *IEEE Trans. on Software Engineering*, Vol. 2, No. 6, pp. 308-320.

[11] McCabe, T. [1983] Structured Testing, IEEE Computer Society Press.

[12] Beizer, B. [1990] Software Testing Techniques (2nd ed), *Van Nostrand Reinhold*, pp.12-14.

[13] Beizer, B. [1984] Software System Testing and Quality Assurance, *Van Nostran Reinhold*, pp. 37-90.

[14] Meyer, B. [1992] Object-Oriented Software Construction, Prentice Hall International.