Improving the Testability of Object Oriented Software Through Software Contracts

Yogesh Singh,
Prof.,School of IT,
GGSIP University
ys66@rediffmail.com

Anju Saha
Lecturer,School of IT
GGSIP University
anju_kochhar@yahoo.com

Abstract

Software testing is one of the most expensive phases of the software development life cycle. Testing object oriented software is more expensive due to various features like abstraction, inheritance etc. The cost of testing can be reduced by improving the software testability. Software testability of a class is generally measured in terms of the testing effort which is equal to the number of test cases required to test a class. Hence testability can be improved if the test cases can be reduced. Software contracts (method preconditions, method postcondtions, and class invariant) can be used in improving the testability of the software. This paper demonstartes how software contracts can be used to reduce the number of test cases and hence improve the testability of an object oriented software. To accomplish this, software contracts are instrumented in a class and test cases are designed for this class using the path testing technique and then it is compared with the class without instrumenting the software contracts. We found that the instrumentation of software contracts reduces the number of test cases and hence improves the testability.

1. Introduction

In today's world every sphere of life is dependent upon the software systems. Hence, it is of utmost importance that the software developed should have high quality and reliability. To achieve software quality and reliability, it is necessary to make the process of software testing more cost effective. One way to achieve this is to improve the software testability. The software testability is one of the important attributes of the software quality and it provides a guideline for testing. Bache and Mullerburg define the testability as the effort needed for the testing [4]. They claim that the most important characteristic contributing to the software testability is the number of test cases required to satisfy a given testing strategy.

A software contract is a formal specification of the semantics of a class and the methods of the class [2]. It has three elements: an invariant expression, a precondition and a postcondition for each method of the class. Although software contracts have been proved to improve the software testability in the previous studies [2,3] but none of the studies is known to the author which demonstrates the reduction of test cases, testing effort and hence the increase in the testability through the use of software contracts. This study is performed to show that software contracts improve the software testability and reduce the number of test cases required to test a class. We follow the definition of testability given by Bache and Mullerburg[4] and measure it in terms of the testing effort which is further measured as the number of test cases required to test a class. In this paper, we design the test cases for an object oriented class using the path testing technique [11] and then instrument this class with the software contracts and again we find the test cases of the class, which is instrumented by contracts. We find that the number of test cases required for an instrumented class is less than that for the one without software contracts. Hence the software contracts reduce the number of test cases required to test a class and improve the testability of the class by reducing the testing effort required to test a class.

The organization of this paper is as follows: Section 2 describes software testability as defined by different researchers and section 3 describes the software contracts. Section 4 describes the generation of the test cases and section 5 describes the conclusions.

2. Software Testability

Testability has been defined by a number of researchers with different points of view. Some of these definitions are based upon the concept of observability and controllability. Controllability refers to the ability to control the input and observability refers to the ability to observe the output of a class. Another most widely used definition is given by Voas. Some of the definitions of software testability are described below briefly.

Voas and Miller [8] have defined the software testability as the probability that software will fail on its next execution during testing if it contains faults. They have used sensitivity analysis to predict the testability. In that, the software and its mutant versions are repeatedly executed and estimation is made about the likelihood of the detection of the mutants. Depending upon this, the testing effort is concentrated on the parts of the software where there is low likelihood. It depends upon the testing technique used and also on the mutation testing method used for seeding the faults.

Binder has defined the software testability as the relative ease and expense of revealing software faults [5]. A highly testable system provides more reliability for a fixed testing cost and also reduces the total testing cost. Binder mentions that the software testability is a result of six factors: (1) characteristics of representation, which includes requirements and specification (2) the software development process, (3) the test suite, (4) built-in test capabilities, (5) the test support environment, and (6) characteristics of the implementation. The testability has two key factors: controllability and observability; the primary concern of the testability is to have more controllable input and observable output. Assertions increase and retain observability. He states that the testability can be accessed through software metrics.

DOI: 10.1145/1668862.1668869 http://doi.acm.org/10.1145/1668862.1668869

Bruntink and Deursen [6] predict the testability of a system based upon the number of test cases needed to test it and to the effort required to develop each test case. They used five java systems to find a correlation between class level metrics (number of attributes, number of methods etc.) and test metrics. They conducted the tests at the class testing level. Significant correlations were found with size related metrics but inheritance related metrics did not show any good correlation with test metrics. Such correlations are dependent on the testing technique followed and the testing criterion used.

Mouchawrab [7] has given a generic framework for object oriented software testability. A number of hypotheses are described which describe, under what conditions and how an attribute can relate to testability. The issue of the software testability is addressed at the design level, before the coding starts. The claim is that evaluating the testability at the analysis and design stage will reduce the testing cost. Design attributes that impact the testability for each testing activity are defined. For each attribute a set of measures is provided and model elements of UML models are identified which are required to evaluate the measures. The guidelines are provided on how the measurement of the testability helps in providing suggestions to change the design to improve the testability.

Baudry et al [9] suggests that only Class diagrams and Statecharts of UML are the main models that should be analysed for testability. In their study those parts of the software design are identified where complex interactions occur which cause problems in testing the software. They suggest that certain types of object interactions like: when one object changes the state of another object through dependencies makes testing difficult. These object interactions are called testability anti patterns which are identified by using a class dependency graph (CDG). These object interactions are the places where testability can be improved.

Jungmayr [10] provides a measure of the testability through the static analysis of the source code of software. He provides the testability metrics, which provide an idea of the effect of dependencies on the testability of the software. One of the testability metric is the average component dependency (ACD). ACD measures the average number of components that a given component depends on directly or indirectly. The number of such components needs to be tested in case of a change in the given component. However, ACD is an example of a metric that only deals with the effects of dependencies with relation to a particular component.

Most of the researchers measure the testability in terms of the testing effort. We follow this particular definition and show that software contracts reduce the testing effort and hence increase the software testability.

3. Software Contracts

A Software Contract is a formal specification of an object oriented class and its methods. It is made up of following three components.

1. Precondition of method: A precondition defines the conditions which should be true before the execution of the method of the class. Each method of a class has a precondition.
2. Postcondition of method: A postcondition defines the conditions which should be true after the execution of the method of the class. Each method of a class has a postcondition.
3. Class Invariant: Class invariant states the conditions which should be true for all the objects of a class. The invariant must be satisfied after the execution of a method if it was satisfied before the execution of the method. Invariant is a constraint that checks the consistency of the states of an object throughout its life cycle [1]. Invariant should be true after the creation of every object of a class.

A methodology based on software contracts called "Design by contract" was originally developed by Meyer [1] to improve the reliability of software and hence the quality of the software. Since reliability is a major component of software quality. Payne [2] has shown that the contracts improve the software testability as they increase the controllability and observability. Briand et al. [3] provides an approach in which the instrumented contracts (method precondition, post condition and class invariants) are used to improve the software testability. A case study is used to show that the contracts detect more number of failures. Also contracts reduce the number of methods and statements of the source code which one has to look to locate the faults on the detection of failures.

4. Generation of test cases

This section describes the path testing technique [11], a white box testing technique which is used to construct the test cases. The path testing is performed by exercising all the paths in the code at least once and is performed on the flow graph of every method in the class. For every method the test cases are created using the following steps:

1. The flow graph is constructed from the method of a class. A flow graph consists of nodes and edges, where nodes represent the executable blocks (sequence of statements between the two decisions) and edges represent the flow of control.
2. The flow graph of the method is represented as a UML activity diagram, where decisions in the flow graph represent the UML branches, blocks represent the action states and control flow represents the UML transitions.
3. Test cases are designed such that each transition in the activity diagram is exercised at least once. To accomplish this, the condition in each branch is examined and inputs are selected for the true branch as well as false branch. This process is repeated for each node to generate the test cases and the equivalent paths.

Now we take an example of the queue class and design its test cases through the path testing technique described above. The figures 1 and 2 show the source codes of the queue class without software contracts and a contract instrumented queue class. The queue class contains two methods add() and remove() which add and remove elements from the queue. The queue class in figure 2 is instrumented with software contracts. Every method has a precondition (shown as Pre) and a postcondition (shown as Post) and an invariant named queue_invariant.

DOI: 10.1145/1668862.1668869 http://doi.acm.org/10.1145/1668862.1668869

```
class queue
{
int q_Array[SIZE]; int front; int rear;
public:
queue( ){ front=0;  rear=0; }
void add(int element)
{ if (isfull())
{cout<<"queue is FULL “;}
q_Array[rear] = element;
rear++;
}
int remove()
{
if(isempty())
{cout<<"Queue Empty ";}
return(q_Array[front++]);
}};
int queue::isempty()
{
if(front == rear) return 1;
else return 0;
}
int queue::isfull()
{
if(rear == SIZE) return 1;
else return 0;
}};
```

Figure 1: C++ code for the queue class without software contracts

```
class queue
{
int q_Array[SIZE];
int front;
int rear;
public:
queue();
void add(int i);
int remove();
};
queue::queue(){
Pre(true);
front=0; rear=0;
Post(front ==0 && rear==0)
}
void queue::add(int i){
Pre (rear<=SIZE);
q_Array[rear] = i;
rear = rear + 1;
Post(q [rear-1] = i);
}
int queue::remove()
{ Pre (front!=rear);
element = q _Array[front];
front = front +1;
Post(! Is_Full ());
return element;
}
bool queue_invariant (){return front >= 0 && rear >= 0 && front <= size && rear<=size;}
```

Figure 2: C++ code for queue class using Software contracts

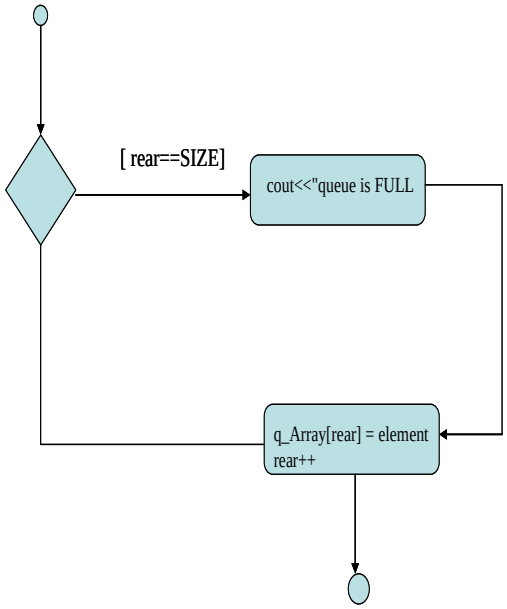

Figure 3: Flow graph for the implementation of add() method (UML activity diagram)

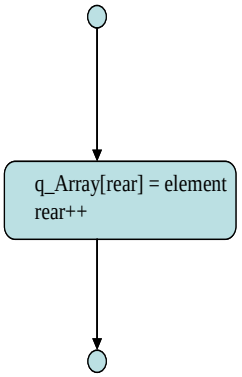

Figure 4: Flow graph for the implementation of add() method in the instrumented C++ class (UML activity diagram)

The figures 3 and 4 show the activity diagrams for the add() method of the queue class with and without contracts. The table 1 shows the test cases for the add() and remove() methods of the queue class in figures 1 and 2. The number of test cases for the queue class with software contracts is 2 and for queue class without contracts is 4. The number of test cases is less in case of a class with software contracts than for a class without contracts. Also, we can see in figures 1 and 2 that the lines of code are reduced in a class with software contracts. Hence we can say that software contracts help in reducing the number of test cases therefore testing effort is reduced and testability is improved.

DOI: 10.1145/1668862.1668869 http://doi.acm.org/10.1145/1668862.1668869

Methods	Test case for a C++ class without Software contracts	Path for a C++ class without Software contracts	Test case for a C++ class with Software contracts	Path for a C++ class with Software contracts
add()	rear == SIZE	cout<<"queue is FULL	Not required	Not required
	rear != SIZE	q_Array[rear]=element; rear++;	rear != SIZE	q_Array[rear] = i; rear = rear + 1;
remove()	front == rear	cout<<"Queue Empty "	Not required	Not required
	front != rear	return(q_Array[front++]);	front != rear	return(q_Array[front++]);

Table 1: Test cases for the methods of queue class for C++ in figure 1 and figure 2

5. Conclusions

The main goal of the study was to establish the relationship between software testability and software contracts. The software contracts reduce the testing effort and hence improve the testability of an object oriented class. The same is demonstrated through an example of a queue class written in C++. The results show that software contracts reduce the number of test cases by 50% to test a class. Hence, the software practitioners can make use of software contracts to reduce the testing effort and hence improve the testability of the software. This work can be extended with the following future work. First, this study has been conducted at the class level; it should be extended to the component level and system level testing. Second, this study may be extended to systems developed with different methodologies.

References

[1] B. Meyer (1997):Object-Oriented Software Construction. 2nd edition, Prentice Hall.

[2] J. E. Payne, R. T. Alexander, C. H. Hutchinson(1997): Design-for-Testability for Object Oriented Software, Object Magazine, SIGS Publications Inc., vol. 7, no.5, pp. 34-43

[3] L. C. Briand, Y. Labiche and H. Sun(2003): Investigating the Use of Analysis Contracts to Improve the Testability of Object-Oriented Code, Software - Practice and Experience, vol. 33 (7), pp. 637-672.

[4] Bache, R., and M. Mullerburg (1990): Measures of testability as a basis for quality assurance. Software Engineering Journal 5(2), 86-92.

[5] Binder, R.V. (1994): Design for testability in object-oriented systems. Communication of the ACM 37 (9) 87–101.

[6] Bruntink, M., and A.V. Deursen ((2006): An Empirical Study into Class Testability . Journal of systems and software. 79(9) ,1219–1232.

[7] Mouchawrab, S., Lionel C. Briand, Yvan Labiche (2005): A measurement framework for object-oriented software testability, Information and Software Technology. 47 , 979–997.

[8] Voas, J.M, K.W. Miller (1995): Software testability: the new verification, IEEE Software, 12 (3), 17–28.

[9] Baudry B., Traon Y.L (2005): Measuring design testability of a UML class diagram: Information and Software Technology, 47 (13), pp. 859-879.

[10] Stefan Jungmayr (2002): Identifying test-critical dependencies. In Proceedings of the International Conference on Software Maintenance, (ICSM '02), pages 404–413. IEEE Computer Society.

[11] Bernd Bruegge, Allen H. Dutoit, Object-Oriented Software Engineering: Using UML, Patterns and Java (2nd Edition)

DOI: 10.1145/1668862.1668869 http://doi.acm.org/10.1145/1668862.1668869