Proceedings of the 23rd IASTED International Multi-Conference
SOFTWARE ENGINEERING
February 15-17, 2005, Innsbruck, Austria

INCREASING CLASS-COMPONENT TESTABILITY

Supaporn Kansomkeat
Department of Computer Engineering
Faculty of Engineering
Chulalongkorn Univesity
Bangkok, 10330, Thailand. (+66) 2218-6988
Supaporn.K@Student.chula.ac.th

Jeff Offutt
Information and Software Engng
George Mason University
Fairfax, VA 22030, USA
(+1) 703-993-1654 / 1651
ofut@ise.gmu.edu

Wanchai Rivepiboon
Department of Computer Engineering
Faculty of Engineering
Chulalongkorn Univesity
Bangkok, 10330, Thailand (+66) 2218-6988
wanchai.r@chula.ac.th

ABSTRACT
Testability has many effects on software. In general, increasing testability makes detecting faults easier. However, increasing testability of third party software components is difficult because the source is usually not available. This paper introduces a method to increase component testability. This method helps a user test when the component is reused during integration. First, we analyze a component to gather definition and use information about method and class variables. Then, this information is used to increase component testability to support component testing. Increased testability helps to detect errors, and helps testers observe state variables and generate inputs for testing. This paper uses an example to report the effort (in terms of test cases) and effectiveness (in terms of killed mutants).

KEY WORDS
Software Testing, Software Testability, Component Software

1. Introduction

Software testing is one of the most common ways to assure software quality and reliability, and is made easier by high software testability. Several different definitions of testability have been published. According to the 1990 IEEE standard glossary [8], testability is the "degree to which a component facilitates the establishment of test criteria and the performance of tests to determine whether those criteria have been met." Voas and Miller [14] explained that testability enhances testing and claimed that increasing component testability is a primary key to improving the testability of component-based software. Their definition of software testability focuses on the "probability that a piece of software will fail on its next execution during testing if the software includes a fault." Binder [2] defined testability in term of *controllability* and *observability*. Controllability is the probability that users are able to *control* component's inputs (and internal state). Observability is the probability that users are able to *observe* component's outputs. If users cannot control the input, they cannot be sure what caused a given output. If users cannot observe the output of a component under test, they cannot be sure if the execution was correct. Likewise, Freedman [4] considered testability based on the notions of observability and controllability. Observability captures the degree to which a component can be observed to generate the correct output for a given input. Controllability refers to the ease of producing all values of its specified output domain.

With object-oriented software, testing needs to place more emphasis on testing the connections among components [9]. Researchers [5, 7, 13] have proposed using information from developers to test object-oriented components. Gallagher and Offutt [5] use information about object states for integration testing, including a finite state machine and a list of which state variables each method defines and uses. Harrold and Rothermel [7] use the data flow relations from the program source to guide the selection of tests. Tsai, Stobart and Parrington [13] seek the definitions and uses of data members from code statements for testing classes. Although they are interested in definitions and uses of data, which are similar to this work, their paper used all variables and source code for testing.

An important property of a software component is *implementation transparency*, which means the implementation is not available. This means that testers have very little information about the internal state of the component. This lack of information makes it difficult to apply directly traditional white-box techniques [15] and difficult to fully exercise the software (lack of controllability) and also difficult to know the result of execution (lack of observability).

Testers can increase testability in several ways:
- gather information from components without source code
- increase observability to monitor outputs
- increase controllability to support inputs
- choose test criteria and generate test cases for component testing based on the criteria

This research project specifically assumes the component is objected-oriented. Furthermore, no access to the source is assumed. First, our method analyzes a compiled component to extract definition and use information. Then, the collected information is used to

increase component testability. This information provides ways to detect some errors, to observe state variables and criteria [9]. This method does not consider inheritance and polymorphism relationships.

2. Background

Several definitions of testability were given in Section 1. Our work defines component testability as the degree to which a component supports *detection*, *observability* and *controllability*. Detection focuses on the ease of detecting faults. Observability focuses on the ease of observing outputs. This means that the component supports ways to observe or monitor the results of testing. Moreover, an observable component allows not only the output of tests to be observed, but also intermediate values. Controllability focuses on the ease of controlling component's inputs. This means that a component supports ways to supply inputs that exercise the component as necessary.

This research presents techniques to create and supply tests for a black box component, and applies it by using data flow relations between methods to guide the selection of tests. Beizer [1] defined integration testing as focusing on testing interfaces between methods. Coupling between methods measures the dependency relations by the data and control flow interconnections between methods. Thus, brief overviews of data flow analysis and coupling-based testing are given in this section.

2.1 Data Flow Analysis

Data flow testing [12] requires tests to execute paths from statements that contain assignments to variables in a program, to statements where variables are used. A definition (*def*) is a statement where a variable's value is stored into memory. A *use* is a statement where a variable's value is accessed. A *definition-use pair* (or du-pair) of a variable is an ordered pair of a definition and a use, such that there is a path from the def to the use. Data flow testing criteria are used to select particular definition-use associations to test. Two of the most simple data flow testing criteria were first defined by Laski and Korel [10]. They proposed the *all-definitions* criterion, which requires that a test should cover a path from each definition to at least one use, and the *all-uses* criterion, which requires a test to cover a path from each def to **all reachable** uses.

2.2 Coupling-based Testing

Jin and Offutt [9] proposed *coupling-based testing* (CBT) as an application of data flow testing to the integration level. CBT requires the program to execute from definitions of variable in a caller to uses of the corresponding variables in the callee unit. The variables can be parameters, global and non-local variables, and to generate tests for when a component is reused in new environment. Test cases are generated to satisfy coupling external references. Unfortunately, directly applying either the all-defs or the all-uses criterion is very expensive, both in terms of number of du-pairs and the difficulty of resolving the paths. Therefore, CBT is only concerned with definitions of variables just **before** calls (*last-defs*) and uses of variables just **after** calls (*first-uses*). The criteria are based on the following definitions:

- A *Coupling-def* is a statement that contains a last-def that can reach a first use in another method on at least one execution path
- A *Coupling-use* is a statement that contains a first use that can be reached by a last-def in another method on at least one execution path
- A *coupling path* is a path from a coupling-def to a coupling-use

Four levels of coupling-based integration test coverage criteria are defined between two units:

- *Call-coupling* requires that the test cases cover all call-sites of the called method in the caller method
- *All-coupling-defs* requires that, for each coupling-def of a variable in the caller, the test cases cover at least one coupling path to at least one reachable coupling-use
- *All-coupling-uses* requires that, for each coupling-def of a variable in the caller, the test cases contains at least one coupling path to each reachable coupling-use
- *All-coupling-paths* requires that the test cases covers all coupling paths from each coupling-def of a variable to all reachable coupling-uses

These testing criteria cannot be directly applied to component-based software because the black-box nature of components prohibits full control flow and data flow analysis. This paper introduces a new analysis method that extracts just the essential information for data flow criteria from Java bytecode.

3. Component Analysis

After a developer implements an object-oriented component, it is assumed to be included into a library without the source code. Moreover, the developer does not provide information about component testing. Because of this lack of information, approaches that rely on detailed analysis of the program, such as the object testing approaches [5, 7, 13] discussed in section 1, cannot be performed. Therefore, we need a process to analyze components to find def and use information without using the source.

This analysis only considers class's state variables that describe of the class. For each method, the process gathers the definition and use information for every state variable. This information is collected from the intermediate form in Java bytecode, and is called *DU-M* (Definitions and Uses of Methods). Moreover, the first

uses and the last definitions of variables are indicated by the locations from the intermediate form in Java bytecode. This collected information is stored in the *DU-M* repository.

The analysis process is as follows. First, a Java class, .class file, is transformed into the intermediate form in Java bytecode by *Decompiler*. Then, the Java bytecode is analyzed to collect the definition and use information. The *Decompiler*, DJ Java Decompiler, is freeware supported by Author NavExpress [11]. The *DU-M* analyzer was developed for this research and parses the intermediate form in Java bytecode to create *DU-M*s. The *DU-M*s are used to increase component testability, as explained in the following subsections.

Figure 1 shows the collected information for a variable x, *uddu* (*d* indicates a definition and *u* indicates a use). The location of the first use is at 1, and last definitions are at 10 and 19.

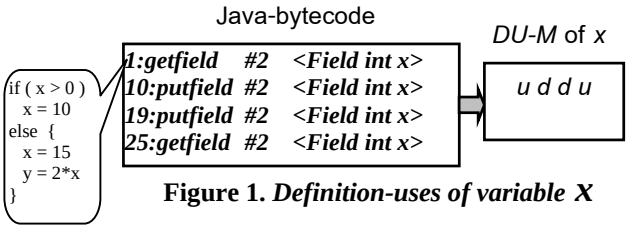

Figure 1. ***Definition-uses of variable X***

3.1 An Example

We illustrate our technique with the example of a vending machine. The machine accepts quarter coins and dispenses products when two quarters are received. CoinBox has four state variables. *total* keeps track of how many quarters have been received over a sequence of transactions. *curQtr* keeps track of how many quarters have been received within one transaction. *seleTy* keeps track of a user's current selection and *availSeleVals* is a constant that stores the selections that are currently available in the vending machine (these are modeled as integers for convenience in the example). Users can add quarters into the machine (*addQtr()*), ask the machine to return the quarters inserted and not consumed (*returnQtr ()*), and ask the machine to vend a specific item (*vend()*). The machine does not dispense the item if it is not available, if the quarters are insufficient, or if the selection is invalid.

The Java source code for this example is shown in Figure 2, however, it should be noted that this source code is not available to a component tester. Each callout in this figure is part of the java-bytecode for the associated method. Each line in the callouts contains the location (statement number), *getfield* or *putfield* (*getfield* indicates a use and *putfield* indicates a definition), the field number, and the field name used to generate the *DU-M*. After the callout, we denote the first use by *first-use* and the last definition by *last-def* of variables of each method. The *DU-M*s of each method of CoinBox example is shown in Table 1. The column DU-M shows the sequence of definition and use of a variable. The column First-use and Last-def show the location of first use and last definition of each variable in each method.

Note that testers can use the CoinBox component to control only inputs of quarters and selections. State variables total, curQtr, seleTy and availSeleVals are private, thus cannot be directly controlled by testers. For example when two quarters are added into the machine, it dispenses a product (shown at line 36 in Figure 2). However, if line 38 is removed, the machine still dispenses an item. This would represent a fault that causes the state variable curQtr is incorrect. To increase controllability of state variables, the *DU-M*s keep state variables to help select inputs to states that should be tested.

Method Name	Variable Name	DU-M	First-use	Last-def
CoinBox()	total	d		26
	curQtr	d		31
	seleTy	d		36
	availSeleVals	d		21
addQtr()	curQtr	u d	2	7
returnQtr	curQtr	d		2
vend()	seleTy	d u	29	7
	curQtr	u u u d u	11	109
	total	u d	94	99
available()	availSeleVals	u u	4	
	seleTy	u	18	

Table 1. ***The DU-Ms of CoinBox***

4. Component Testability

As said in Section 1, component testability generally refers to how easy it is to test. A high degree of testability indicates that any existing faults can be revealed relatively easily during testing, outputs of state variables can be observed during testing and inputs can easily be selected to satisfy some testing criteria. To increase testability, we provide processes to detect errors and increase observability and controllability.

Before components are integrated with exiting software, they should be tested. The *DU-M* repository from the previous section can be used to perform preliminary checks and detect some faults. As shown in Figure 3, *Component Anomaly Detector* uses *DU-M*s. For example, a *dd* data flow anomaly occurs when a definition is followed by another definition without an intervening use. Figure 4 shows a dd anomaly that would occur if the programmer had made a mistake of writing "total = curQtr-VAL" instead of "curQtr = curQtr-VAL" at line 38 in Figure 2.

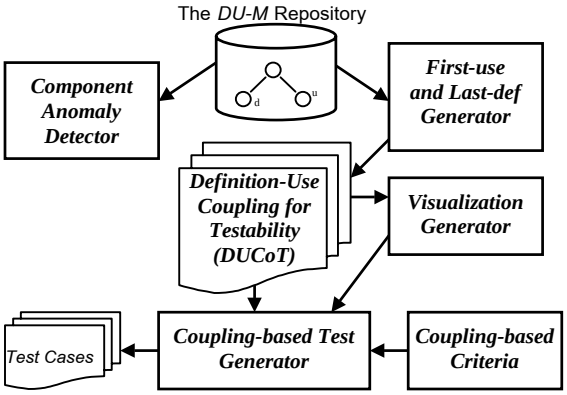

Figure 3. ***The Component Testability Process***

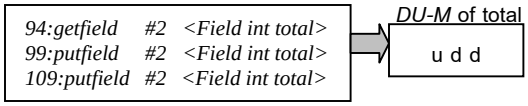

Figure 4. ***A dd anomaly example***

To increase observability and controllability, we collect definition-use pairs between the last definition in a method and the first use in other methods of variable by using *DU-M*s of component. These definition-use pairs of a variable are called *Definition-Use Coupling for Testing (DUCoT).* The *DUCoT*s for CoinBox's state variables are shown in Figure 5. This information is used to generate test cases. As shown in Figure 3, *First-use and Last-def Generator* creates a *DUCoT* for each variable. A *DUCoT* is defined next.

Definition The *DUCoT* of variable *V* is a tuple, *DUCoT* $(V) = (D_L, U_F)$

- D_L is a finite set of last definitions of variable *V*
- U_F is a finite set of first uses of variable *V*

```
DUCoT(availSelectionVals)  = ( {CoinBox(21)}, {available(4)} )

DUCoT(total)    = ( {CoinBox(26), vend(99)}, {vend(94)} )

DUCoT(curQtr) = ( {CoinBox(31), addQtr(7), returnQtr(2), vend(109)} ,
                  {addQtr(2), vend(11)} )

DUCoT(seleTy) = ( {CoinBox(36), vend(7)} ,
                  {vend(29), available(18)} )
```

Figure 5. ***The DUCoTs of CoinBox's state variables***

We define *observation points* to monitor the state variables of the component. Note that variables are given values in definition statements. Therefore, observation points should be located immediately before and after definitions of each variable. Also, the output of tests should be observation points to show all state variables of the component. As shown in Figure 3, *Visualization Generator* provides a method to monitor state variables by using the *DUCoT*s. This supports monitoring of the status of component variables during testing.

For example, when the test case *addQtr(7),vend(11)* is tested to cover *DUCoT(curQtr)*, the state variable *curQtr* must be monitored before and after calling *addQtr ()*. Furthermore, every state variable of CoinBox must be monitored after testing. Because *addQtr()* adds a quarter into the machine, the value of *curQtr* should be increased. The *Visualization Generator* uses Java reflection to show values of state variables.

```
1  class CoinBox {
2
3  private int total;
4  private int curQtr;
5  private int seleTy;
6  private int[] availSeleVals = {2,3,13};
7
8  public CoinBox() {          21:putfield #2 <Field int[] availSelectionVals> (last-def)
9  total = 0;                  26:putfield #3 <Field int total> (last-def)
10 curQtr = 0;                 31:putfield #4 <Field int curQtr> (last-def)
11 seleTy = 0;                 36:putfield #5 <Field int seleTy> (last-def)
12 }
13                             2:getfield #4 <Field int curQtr> (first-use)
14 void addQtr() {             7:putfield #4 <Field int curQtr> (last-def)
15         curQtr = curQtr + 1;  2:putfield #4 <Field int curQtr> (last-def)
16 }
17
18 void returnQtr() {          7:putfield #5 <Field int seleTy> (last-def)
19         curQtr = 0;         11:getfield #4 <Field int curQtr> (first-use)
20 }                           29:getfield #5 <Field int seleTy> (first-use)
21                             66:getfield #4 <Field int curQtr>
22 void vend( int selection ) { 94:getfield #3 <Field int total> (first-use)
23         int MAXSEL = 20;    99:putfield #3 <Field int total> (last-def)
24         int VAL = 2;       104:getfield #4 <Field int curQtr>
25         seleTy = selection; 109:putfield #4 <Field int curQtr> (last-def)
26         if ( curQtr == 0 ) 128:getfield #4 <Field int curQtr>
27           System.err.println ("No coins inserted");
28         else if ( seleTy > MAXSEL )
29               System.err.println ("Wrong selection ");
30         else if ( !available( ) )
31               System.err.println ("Selection unavailable");
32         else {
33             if ( curQtr < VAL )
34               System.err.println ("Not enough coins");
35             else {
36               System.err.println ("Take selection");
37               total = total+ VAL;
38               curQtr = curQtr - VAL;
39             }
40         }
41    System.out.println( "Current value = " + curQtr );
42 }                           4:getfield #2 <Field int[] availSelectionVals> (first-use)
43                            12:getfield #2 <Field int[] availSelectionVals>
44 boolean available( ) {     18:getfield #5 <Field int seleTy> (first-use)
45   for (int i = 0; i<availSeleVals.length; i++)
46       if (availSeleVals[i] == seleTy)
47             return true;
48   return false;
49 }
50 } // class CoinBox
```

Figure 2. ***Vending Machine example***

To control inputs to exercise state variables in a component, the *DUCoT*s are used to generate test inputs according to coupling criteria. This paper uses the coupling-based criteria proposed by Jin and Offutt [9], as defined in Section 2.2. Applying coupling-based testing to component testing requires some minor modification to the terminology.

All-coupling-defs requires that for each coupling-def, at least one test case executes a path from the def to at least one coupling-use. A version of *All-coupling-defs* for a component is given in Definition 1 and the *All-coupling-defs* for *curQtr* are shown in Table 2.

Definition 1 Let (D_L, U_F) be a *DUCoT* of variable *V*. The *All-coupling-defs* of variable *V* is defined as $AllCoD(V) = (D_L \times U_F) = \{ (d, u) \mid \forall d \in D_L$ and $\exists u \in U_F) \}$

#	*AllCoD (curQtr)*
1	CoinBox (31) , addQtr (2)
2	addQtr (7) , vend (11)
3	returnQtr (2) , addQtr (2)
4	vend (109) , vend (11)

Table 2. ***The All-coupling-defs of curQtr variable***

All-coupling-uses requires that for each coupling-def, at least one test case executes a path from the def to each reachable coupling-use. A version of *All-coupling-uses* for a component is given in Definition 2 and the *All-coupling-uses* for *curQtr* are shown in Table 3.

Definition 2 Let (D_L, U_F) be a *DUCoT* of variable *V*. The *All-coupling-uses* of variable *V* is defined as $AllCoU(V) = (D_L \times U_F) = \{ (d, u) \mid \forall d \in D_L$ and $\forall u \in U_F) \}$

#	*AllCoU(curQtr)*
1	CoinBox (31) , addQtr (2)
2	CoinBox (31) , vend (11)
3	addQtr (7) , addQtr (2)
4	addQtr (7) , vend (11)
5	returnQtr (2) , addQtr (2)
6	returnQtr (2) , vend (11)
7	vend (109) , addQtr (2)
8	vend (109) , vend (11)

Table 3. ***The All-coupling-uses of curQtr variable***

Test cases can be computed from *DUCoT*s according to the selected level of coupling-based criteria. As shown in Figure 3, the *Coupling-based Test Generator* uses *DUCoT*s to generate test cases depending on the selected criteria. How to use these ideas for testing is illustrated through a case study.

5. Case Study

This section illustrates component testing on the vending machine class in Figure 2. Following our criteria in section 4, we generate 9 test cases for *All-coupling-defs* and 15 test cases for *All-coupling-uses*. We derive sequences of method calls for each test. One sequence can be used to realize multiple test cases. For example, the sequence [*CoinBox(), returnQtr(), addQtr()*] is derived for the test case *returnQtr(2),addQtr(2)*. The sequence [*CoinBox(), addQtr(), addQtr(), vend(2), vend(3)*] is derived for test case *vend(109),vend(11)*. Moreover, this sequence covers the test case *addQtr(7),addQtr(2)*. In this way, 4 sequences are created for *All-coupling-defs* and 7 sequences are created for *All-coupling-uses* as shown in Table 4.

This example is based on the one used by Harrold et al. [6]. They published 25 test sequences that are grouped into 3 sets for the vending machine class in their paper. We use Harrold et al.'s 25 test sequences [6], named *AllSequence*, and 7 randomly selected test sequences from them to compare with our sequences. More precisely, we use 5 groups of 7 randomly selected test sequences, named *Sampling1*, *Sampling2*, …, *Sampling5*. Sampling1 selects 7 test sequences using prime number order. The Sampling2, Sampling3, Sampling4 and Sampling5 select 3, 2 and 2 test sequences at random from 3 sets (1-16, 17-20, 21-25) that are grouped from 25 test sequences [6].

	Test sequences	
AllCoU	1. CoinBox() addQtr() vend(1)	**AllCoD**
	2. CoinBox() addQtr() addQtr() vend(2) addQtr() addQtr() vend(3)	
	3. CoinBox() returnQtr() addQtr()	
	4. CoinBox() addQtr() addQtr() vend(2) vend(3)	
	5. CoinBox() returnQtr() vend(1)	
	6. CoinBox() vend(2)	
	7. CoinBox() addQtr() vend(21)	

Table 4. ***The Test Sequences***

The tests were evaluated by their ability to detect mutant-like faults. Delamaro et al. [3] proposed a set of interface mutation operators for integration testing. Our intent focuses on interface faults between methods, thus we use their operators to create faults for our case study. Six operators were used for our example; the others did not apply to our program. All possible mutants are generated according to these mutation operators, resulting in 40 mutants, as summarized in Table 5. These mutants were seeded into the vending machine class by hand.

Mutation operator	**Definition**	**Number of mutants**
DirVarRepPar	Replaces interface variable by each element of parameters of callee	2
DirVarRepGlo	Replaces interface variable by each element of global variables accessed by callee	18
IndVarRepLoc	Replaces non interface variable by each element of local variables in callee	10
DirVarAriNeg	Inserts arithmetic negation at interface variable uses	3
DirVarLogNeg	Inserts logical negation at interface variable uses	5
RetStaRep	Replaces return statement	2
Total		**40**

Table 5. ***The Mutation Operators and Number of Mutants***

Table 6 shows the results of executing each test set with mutants. The *All-coupling-uses* (AllCoU) tests resulted in a higher mutation score than any other sets, except Harrold et al.'s (AllSequence) tests. The *All-*

coupling-defs (AllCoD) tests resulted in a higher mutation score than three sets of sampling. Although the AllCoD tests had the same mutation score as two sets of sampling, it has a smaller number of test sequences. The AllSequence tests resulted in 100% mutation score, but 3.6 times more test sequences than AllCoU.

Criterion	Number of test cases	Number of test sequences	Killed mutants	Mutation Scores
AllCoD	9	4	37	**92.5%**
AllCoU	15	7	39	**97.5%**
Sampling1	-	7	9	22.5%
Sampling2	-	7	37	92.5%
Sampling3	-	7	19	47.5%
Sampling4	-	7	18	45%
Sampling5	-	7	37	92.5%
AllSequence	-	25	40	100%

Table 6. ***Results for a Vending Machine***

6. Conclusions

The purpose of this research is to increase component testability. We have defined a method to increase testability of an object-oriented component whose source is not available. This method sets up and develops facilities to support testing. First, we analyze a component to gather definition and use information. This information shows the internal state variables of the component. Then, it is used to increase testability. These facilities provide ways to detect errors, to observe state variables and to control inputs for component testing by supporting test cases generation. This generation relies on the selected level of coupling-based criteria. Therefore, test cases are able to control inputs to fully exercise a component and improve the controllability. Our test set effectiveness is illustrated by a case study. Moreover, the output and intermediate values for each test case can be observed.

This paper presents a way to increase testability in terms of detection, observability and controllability. In the future, we hope to apply our component testability ideas to another facet of controllability, specifically by allowing testers to control inputs with the goal of causing specific variables to have specific values during execution.

7. Acknowledgments

This work was supported in part by Thailand's Commission of Higher Education, MOE, and Center of Excellence in Software Engineering, Dept. of Computer Engineering, Faculty of Engineering, Chulalongkorn University. Thanks to the Department of Information and Software Engineering, School of Information Technology and Engineering, George Mason University, for hosting the first author during this research project.

References:

[1] B. Beizer, *Software Testing Techniques* (Van Nostrand Reinhold, Inc, New York NY, 2nd edition, 1990).

[2] R. V. Binder, Design for Testability with Object-Oriented Systems, *Communications of the ACM*, *37* (9), 1994, 87-101.

[3] M. E. Delamaro, J. C. Maldonado, and A. P. Mathur, Interface mutation: An approach for integration testing, *IEEE Transactions on Software Engineering*, *27*(3), 2001, 228–247.

[4] R. Freedman, Testability of software components, *IEEE Transactions on Software Engineering*, *17*(6), 1991, 553–563.

[5] L. Gallagher and J. Offutt, Integration Testing of Object-oriented Components Using Finite State Machines, Under review.

[6] M. J. Harrold, A. Orso, D. Rosenblum, G. Rothermel, M. L. Soffa, and H. Do, Using Component Metadata to Support the Regression Testing of Component-Based Software, *Proc. IEEE International Conference on Software Maintenance*, Florence, Italy, 2001, 154-163.

[7] M. J. Harrold and G. Rothermel, Peforming Data Flow Testing on Classes, *Proc. ACM SIGSOFT Foundation of Software Engineering*, New Orleans, LA, 1994, 154–163.

[8] IEEE Standard Glossary of Software Engineering Technology, ANSI/IEEE 610.12, IEEE Press (1990).

[9] Z. Jin and J. Offutt, Coupling-based criteria for integration testing, *The Journal of Software Testing, Verification, and Reliability*, *8*(3), 1998, 133–154.

[10] J. Laski and B. Korel, A Data Flow Oriented Program Testing Strategy, *IEEE Transaction on Software Engineering*, *9*(3), 1983, 347-354.

[11] Atanas Neshkov, "NavExpress DJ Java Decompiler", http://www.dj.navexpress.com.

[12] S. Rapps and E. J. Weyuker, Selecting Software Test Data Using Data Flow Information, *IEEE Transaction on Software Engineering*, *11*(4), 1985, 367-375.

[13] B.-Y. Tsai, S. Stobart and N. Parrington, Employing Data Flow Testing on Object-oriented Classes, *The IEE Proc. – Softw.*, *148*(2), 2001, 56-64.

[14] J. M. Voas and Miller K. W, Software Testability: The New Verification, *IEEE Software*, *12*(3), 1995, 17-28.

[15] Y. Wu, M. Chen, and J. Offutt, UML-Based Integration Testing for Component-Based Software, *Proc. 2nd International Conference on COTS-Based Software Systems*, Ottawa, Canada, 2003, 251–260.