Analysis for Class-Component Testability

SUPAPORN KANSOMKEAT
Faculty of Engineering, Chulalongkorn University
Bangkok, 10330, THAILAND
supaporn.k@student.chula.ac.th

JEFF OFFUTT
Information and Software Engineering
George Mason University
Fairfax, VA 22030, USA
offutt@ise.gmu.edu

WANCHAI RIVEPIBOON
Faculty of Engineering, Chulalongkorn University
Bangkok, 10330, THAILAND
wanchai.r@chula.ac.th

Abstract: - Testability is a quality factor used to predict the amount of effort required for software testing and to indicate the difficulty of revealing faults. Also, it can be used to estimate the difficulty of software testing. This paper presents a quantitative testability analysis method for a software component that can be used when the source program is not available, but the bytecode is (as in Java .class files). This process analyzes the testability of each location to evaluate the component testability. The testability of a location is analyzed by computing the probability that the location will be executed and, if the location contains a fault, the execution will cause the fault to be revealed as a failure. This analysis process helps developers measure component testability and determine whether the component testability should be increased before the component is reused. In addition, low testability locations are identified. This paper uses an example to evaluate the ability of this process to indicate component testability.

Key-Words: - Software Testing, Software Testability, Testability Analysis, Bytecode, Mutation Analysis

1 Introduction

Testability is a quality factor that attempts to predict how much effort will be required for software testing and to estimate the difficulty of causing a fault to result in a failure (called *revealing* the fault). Several definitions for 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. Binder [2] defined testability in term of controllability and observability. *Controllability* is the probability that users are able to control the internal state through the component's inputs. *Observability* is the probability that users are able to observe outputs of components. If users cannot control the inputs, they cannot be sure what caused a given output. If users cannot observe the outputs of a component under test, they cannot be sure if the execution was correct. Voas and Miller [16, 17] explained that testability enhances software testing. Their definition of software testability focuses on the tendency for existing faults to be revealed during testing.

Numerous previous papers [6, 9, 12] have tried to create test cases that consist of inputs and expected output pairs. A testing process attempts to reveal software faults by executing the program on inputs and comparing the outputs of the execution with the expected outputs. Software testing is time consuming and very costly. Tools that provide information about software testability can help developers by helping to evaluate costs before test planning and execution, help target testing in specific locations, and help designers decide whether to increase testability of reusable components before use.

Numerous approaches to estimating testability have been proposed in the literature [10, 11, 14, 15, 16, 17, 18]. McCabe [11] proposed to evaluate software complexity by measuring the cyclomatic number based on the number of execution paths in control flow graphs, up to, but not including loops. This complexity measure is assumed to estimate the number of test cases needed in terms of the number of execution paths (although testability was not mentioned directly). Le Traon and Rabach [14, 15] proposed a testability measure for data flow designs that is based on controllability and observability of the components in the system. Controllability was defined as the ease of generating inputs to a

component from the inputs of the system and observability was defined as the ease of propagating the outputs of a component to the final outputs of the system. In particular, the authors proposed a *predictive* testability analysis that is appropriate during design-specification. They are interested in controllability and observability in ways that are similar to our approach, but because our approach is based on the implementation rather than the design, our measure has more precision than Le Traon and Rabach's. Voas et al. [16, 17, 18] used program testability to predict it's the ability to *hide* faults. They defined *fault sensitivity* to analyze testability as multiplying the probabilities that (1) the location containing a fault is **executed**, (2) the fault corrupts, or **infects**, the program's state, and (3) the corrupted state **propagates** to the output. This is named PIE analysis after *propagation*, *infection* and *execution*. A high fault sensitivity value indicates high testability. Lin and Lin [10] measured the testability of software by tracing the source code to estimate the three factors in the PIE analysis. Similar to the technique by Voas et al., our testability analysis technique is closely related to fault-based testing and mutation testing. In their work, infection is modeled by making syntactic changes to a program to model faults. This infection technique cannot directly be used if the source code is not available. They also made random changes to the program's data state to estimate propagation. This can be imprecise because a high percentage of the changes are unstable and will always result in failure.

Object-oriented software is increasingly becoming popular for system development, partly because it emphasizes portability and reusability. Java classes are compiled into portable binary class files that contain statements called *bytecode*. The class-components are included in Java libraries without source code, thus the source is not always available.

This paper presents a class-component testability analysis technique that does not require access to the source program. The testability analysis focuses on the fault revealing ability of class-components based on data flow analysis. A fault can be revealed when a program segment that contains the fault is executed and the fault affects the output. Thus, higher controllability refers to the ease of executing all locations that can contain faults, thereby locating faulty segments. Similarly, greater observability means the fault has a greater chance of propagating to the output. Therefore, we define the testability measurement as the product of two probabilities, execution probability and propagation probability.

The remainder of this paper is organized as follows. Section 2 introduces the background. Section 3 describes our class-component testability analysis model and section 4 describes the testability analysis mechanism. Section 5 presents a case study. Finally, conclusions are presented in section 6.

2 Background

This paper presents a technique for measuring class-component testability. This technique focuses on program statements that contain definitions and uses of variables as in data flow analysis. Each location is analyzed to estimate its execution and propagation probability. A fault injection approach similar to mutation analysis is used for propagation probability analysis. This section provides brief overviews for data flow and mutation analysis.

2.1 Data Flow Analysis

Data flow analysis [13] tries to ensure that correct values of program variables are stored into memory, and the same values are subsequently 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 *def-use pair* (du-pair) of a variable is an ordered pair of a def and a use, with the limitation that there must be an execution path from the def to the use without any intervening redefinition of the variable.

2.2 Mutation Analysis

Mutation analysis [4, 5] is often used to assess the adequacy of a test set. It is a fault-based testing strategy that starts with a program to be tested and makes numerous small syntactic changes to the original program, creating *mutants*. If a test set can cause behavioral differences between the original and mutant program, the mutant is said to be *killed* by the test. The product of mutation analysis is a measure called the *mutation score*, which indicates the percentage of mutants killed by a test set. Mutants are obtained by applying *mutation operators* that introduce the simple changes into the original program. Examples include changing arithmetic operators, logical operators, and variable references.

3 Class-Component Testability Analysis Model

Component testability analysis can be used to estimate the difficulty of testing components, aiding planning and execution of testing. Also, testability can be used to determine whether the component should be modified to increase its testability before reuse.

There are two general testability analysis approaches. The first focuses on predicting the effort needed for testing. For example, if software has high complexity, more effort may be needed to satisfy a test criterion. The second approach focuses on the fault revealing ability during testing. This work adapts the second approach for testability analysis.

This paper defines *testability analysis* as the probability that existing faults will be revealed during testing. In this research, testability analysis is concentrated on data flow analysis, as described in Section 2. A *data state* is a set of mappings between variables and their values. The def statements modify the data state. For instance, the data state {(*a, undefined*)} is changed to {(*a, 5*)} after executing the def statement a=5.

The defs and uses of data flow analysis can be used to find faults. Thus, **the locations of defs and uses are used to analyze testability in this research**. The remainder of this paper refers to the def and use statements as *def locations* and *use locations*. The goal of this work is to find a model that can be used to assess the testability of a class-component. Our model analyzes testability by measuring the fault revealing ability of a class-component. To evaluate the testability of a location, the location containing a fault must be executed. If the execution results in unexpected output, then the testability is higher. If the probability of a location *l* being executed, the *execution probability* is *E(l)*, and the probability that an incorrect data state at *l* affects the output, the *propagation probability*, is *P(l)*, then the testability of *l*, *T*(*l*), is defined as follows:

$$T(l) = E(l) * P(l)$$

3.1 Execution Analysis

Execution analysis executes a class-component and records the locations executed by each input. The execution probability of a location *l*, *E*(*l*), is estimated to be the percentage of inputs that execute that location. The algorithm for finding an execution probability of a particular location is as follows:

1. Assign a location number to each def and each use location of a class-component.
2. Initialize an array *counter* to zeroes, where the size of *counter* is the number of def and use locations in a class-component.
3. Execute a class-component by using an input. If a location *l* is executed, the element of array *counter* that corresponds to the location *l* is increased the value by one.
4. Repeat algorithm step 3 *m* times with different inputs.
5. Divide each element of *counter* by *m* to calculate an execution probability of each location. For example the execution probability of a location *l* is estimated to be *counter*[*l*]. The execution probability of location *l* is *counter[l]/m*.

3.2 Propagation Analysis

Propagation analysis estimates the probability that an incorrect data state caused by a faulty location will propagate to the output. The incorrect data state is generated by injecting a fault into the data state. This is called a *data state mutation*. This generation process is similar to the mutation process in mutation analysis [4, 5]. In this work, the data state mutation creates mutants at defs and uses. At defs, the data state is mutated by redefining the variable value. At uses, the data state is mutated by changing the variable value to an incorrect value. Each def and use location *l* is repeatedly mutated to create a set of mutants. Each mutant is executed and the execution output is checked against the original output. The propagation probability of a location *l*, *P*(*l*), is the percentage of mutants of *l* that will produce incorrect output. The algorithm for finding a propagation probability of a particular location is as follows:

1. Create *n* mutants by data state mutation for a def or use location *l*.
2. Initialize an integer variable *count* to 0.
3. Execute a mutant *M* with one input.
4. Compare the execution output with the output of the original program on the same input. If the outputs differ, propagation has occurred and *count* is incremented.
5. Repeat algorithm steps 3 to 4 *p* times, each time with different inputs.
6. Divide *count* by *p* to calculate the propagation probability of the mutant *M*.
7. Repeat algorithm steps 2 to 6 *n* times, each time with a different mutant.

8. Compute the average of mutant probabilities to be the propagation probability of location *l*.

For each def and use location *l*, the testability of the location, $T(l)$, is the execution probability multiplied by the propagation probability. The testability measure of a class-component (***T***) is the average of the testability of all def and use locations. If a component has *k* def and use locations, the testability measure is calculated as follows:

$$\boldsymbol{T} = (\sum T(i) \quad [i = 1, 2, \ldots, k]) / k$$

4 Testability Analysis Mechanism

The execution algorithm in Section 3 needs to keep track of which locations are executed. The usual way to track this kind of information is to *instrument* the source of the program, then compile the instrumented program. However, this research assumes the source is not available, so a way is needed to analyze class-components at a lower level.

Java programs are written and compiled into portable binary class files. Each class is represented by a single file that contains bytecode instructions. This file is dynamically loaded into an interpreter (Java Virtual Machine, JVM) and executed. To analyze the testability of a class-component, we modify bytecode instructions before they are loaded into JVM.

An open source tool from Apache/Jakarta called BCEL—Byte Code Engineering Library [1, 3] is used to access and modify bytecode instructions. The BCEL system can analyze, create and manipulate Java bytecode files. The BCEL API has three parts:

1. A package of classes that gives a "static" view of class files, that is, it is not intended to modify bytecode. This is used to analyze Java classes without requiring the source files to be available. The main data structure is called JavaClass, which gives the API its name.
2. A package to dynamically generate or modify JavaClass objects. It can be used to insert analysis code, to strip unnecessary information from class files, or to implement the code generator back-end of a Java compiler.
3. Various code examples and utilities like a class file viewer, a tool to convert class files into HTML, and a converter from class files to the Jasmin assembly language.

Our analysis process uses the first and second packages to extract information and modify bytecode instructions. The process is presented in the next subsection.

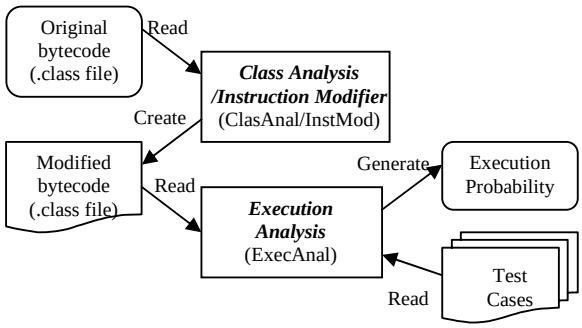

Fig.1 Execution Analysis Phase

```
JavaClass j_class = new ClassParser (args[0]).parse();
ClassGen cg               = new ClassGen (j_class);
ConstantPoolGen cpg = cg.getConstantPool();

cg.addField (new FieldGen (Constants.ACC_PUBLIC,
   new ArrayType (Type.INT,1),"count",cpg).getField());
j_class.setConstantPool (cpg.getFinalConstantPool());
```

Fig.2 Example statements to add the field count

4.1 Execution Analysis Phase

This phase implements execution analysis from section 3.1. The process of execution analysis is illustrated in Fig.1. There are two major components: (1) *Class Analysis / Instruction Modifier* (ClasAnal/InstMod) and (2) *Execution Analysis* (ExecAnal). *ClasAnal/InstMod* modifies the Java bytecode (original bytecode) and automatically creates the modified bytecode. By using BCEL, the Java bytecode is accessed to make the following manipulations:

- Add fields to be the counters of execution.
- Add a method to initialize the field values and to set environment values.
- Insert instructions into Java bytecode at every def and use location. These are recognized by the PUTFIELD instruction (defs) and the GETFIELD instruction (uses). The inserted instructions update the counters when each location is executed.

These modifications require complex manipulations of the Java bytecode. For example, when a new field is added into the Java bytecode, the constant pool must be modified, as shown in Fig.2. *ExecAnal* estimates the execution probability by invoking the modified bytecode with test cases. When the modified bytecode is invoked with each test case, the corresponding counters are updated.

These counters are used to compute the execution probability.

4.2 Propagation Analysis Phase

Fig.3 illustrates propagation analysis, following the process in section 3.2. Propagation analysis has two major components: (1) *Mutation Generator* (MutaGen) and (2) *Propagation Analysis* (PropAnal). The *MutaGen* accesses Java bytecode (original bytecode) and manipulates bytecode instructions to automatically generate mutants by data state mutation. Data state mutants are created by inserting new instructions into the Java bytecode using BCEL. Several mutants are generated for each def and use location. For example, to create a mutant at a use location, instructions are inserted to use an incorrect value, as shown in Fig.4. *PropAnal* invokes each mutant with test cases. Each execution output of a mutant is compared with the execution output of original class. The results of comparison are used to generate the propagation probability. This testability analysis mechanism is illustrated in the following section through a case study.

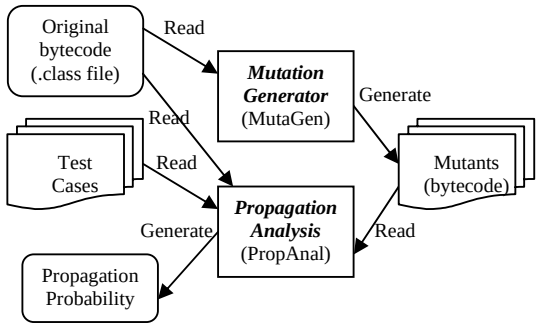

Fig.3 Propagation Analysis Phase

```
class_name = java_class.getClassName();
int ref = cpg.lookupFieldref
     (class_name,wrong_field_name,field_signature);
...
FindPattern f = new FindPattern (il);
String pat = "`GETFIELD"';
InstructionHandle ih = f.search (pat);
if (ih != null) {
    InstructionList patch  = new InstructionList();
    patch.append (new POP());
    patch.append (new ALOAD(0));
    patch.append (new GETFIELD(ref));
    il.append (ih, patch);
}
```

Fig.4 Example statements for data state mutation at the use location

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

Fig.5 Vending Machine Example

5 Case Study

This section presents results from a case study that applied class-component testability analysis to the vending machine example from Harrold et al. [7]. The Java source code for the vending machine is shown in Fig.5, however, it should be noted that this source code is not available to a component tester. As said in section 1, testability is based on

controllability and observability. To create a vending machine class with higher observability, output statements are added. The original class is named *vending1* and the new class is named *vending2*.

The goal of this case study was to determine whether the proposed testability analysis technique can, in fact, obtain different results on two classes that have the same functionality but obviously different observabilities. Harrold et al. [7] published 25 test cases for vending machine class, which we used for both *vending1* and *vending2*. In this class, a test case is a sequence of method calls to the class-component under test. A short example is the sequence "*initialVariables(); vend(2)*". *Vending1* has 19 def and use locations and *vending2* has 23. These locations are found by examining the bytecode. One source statement can represent several def and use locations, for example, the statement "curQtr = curQtr+1;" at line 16 in Fig.5 is two locations in the bytecode (one def and one use). Fig.6 shows bytecode instructions of method *addQtr* in Fig.5.

By using data state mutation, 38 mutants were created for *vending1* seeding faults into the 19 def and use locations, and 46 mutants were created for *vending2*. Table 1 shows the execution probability, *E(l)*, propagation probability, *P(l)*, and testability, *T(l)*, for each location (*l*) in *vending1* and *vending2*. The average testability measurements are 0.166 for *vending1* and 0.236 for *vending2*, thus *vending2* has a higher testability measure than *vending1*. Because the observability was increased as stated before, *vending2* has higher testability than the *vending1*, therefore, these results support the idea that our testability analysis can be used to measure the testability of class-components.

```
0:  aload_0
1:  aload_0
2:  getfield      CoinBox.curQtr I (3)
5:  iconst_1
6:  iadd
7:  putfield      CoinBox.curQtr I (3)
10: return
```

Fig.6 Bytecode Instructions of Method addQtr

6 Conclusions

This paper has proposed a method to measure testability of a class-component whose source is not available. The testability measure concentrates on the fault revealing ability of a class-component based on data flow analysis and considering def and use locations. To measure the class-component testability, we analyze execution and propagation probabilities from the bytecode in binary class files. The execution probability is the percentage of faulty locations executed. The propagation probability is the percentage of faulty locations for which an input caused incorrect output. The case study shows that our measurement can estimate class-component testability with reasonable accuracy. The testability analysis technique in this paper is fully quantitative and automated tool support is currently under development. Because this technique is based on the program's implementation, the results should be more precise than techniques that depend on class diagrams to estimate testability. Testability analysis can be used to predict the ease (or difficulty) of component testing and can be used to determine whether the component testability should be increased before the component is reused. Future plans are to apply the testability measurement to other aspects of software development.

7 Acknowledgments

This work was supported in part by Thailand's Commission of Higher Education (MOE), and by Center of Excellence in Software Engineering, Department 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. The second author is a part-time faculty researcher at the National Institute of Standards and Technology (NIST).

References:

[1] Apache Software Foundation, BCEL: Byte Code Engineering Library, Part of the Apache/Jakarta project, 2002-2003. http://jakarta.apache.org/bcel/ (accessed November 2005).

[2] R. V. Binder, Design for testability with object-oriented systems, *Communications of the ACM*, Vol.37, No.9, September 1994, pp.87-101.

[3] M. Dahm, Byte code engineering with the JavaClass API, *Technical Report B-17-98*, Freie Universität Berlin, Institut für Informatik, January 1999.

[4] R. A. DeMillo, R. J. Lipton, and F. G. Sayward, Hints on test data selection: Help for the

practicing programmer, *IEEE Computer*, Vol.11, No.4, April 1978, pp.34-41.

[5] R. A. DeMillo and A. J. Offutt, Constraint-based automatic test data generation, *IEEE Transactions on Software Engineering*, Vol.17, No.9, September 1991, pp.900-910.

[6] R. DeMillo and J. Offutt, Experimental Results from an Automatic Test Case Generator, *ACM Transactions on Software Engineering Methodology*, Vol.2, No.2, April 1993, pp.109-175.

[7] M.J. Harrold, A. Orso, D. Rosenblum, G. Rothermel, M.L. Soffa, & H. Do, Using component metadata to support the regression testing of component-based software, *Proceedings of IEEE International Conference on Software Maintenance*, Florence, Italy, 2001, pp.154-163.

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

[9] S. Kansomkeat, and W. Rivepiboon, Automated-generating test case using UML statechart diagrams, *Proceedings of the 2003 annual research conference of the South African institute of computer scientists and information*, Fourways, South Africa, September 2003, pp.296-300.

[10] J. C. Lin, and S. W. Lin, An Analytic Software Testability Model, *Proceedings of 11th of Asian Test Symposium (ATS'02)*, Guam, USA, November 2002, pp.278-283.

[11] T. J. McCabe, A Complexity Measure, *IEEE Transactions on Software Engineering*, Vol.2, No.4, December 1967, pp.308-320.

[12] J. Offutt and A. Abdurazik, Generating test cases from UML Specifications, *Proceedings of 2th International Conference on the Unified Modeling Language (UML99)*, Fort Collins, CO, October 1999, pp.416-429.

[13] S. Rapps and E. J. Weyuker, Selecting software test data using data flow information, *IEEE Transactions on Software Engineering*, Vol.11, No.4, April 1985, pp.367-375.

[14] Y. L. Le Traon, and C. Robach, Testability measurements for data flow designs, *Proceedings of the Fourth International Software Metrics Symposium*, New Mexico, USA, 1997, pp.91-98.

[15] Y. L. Traon, F. Ouabdesselam, and C. Robach, Analyzing testability on data-flow designs, *Proceedings of 11th International Symposium on Software Reliability Engineering (ISSRE'00)*, San Jose, CA, October 2000, pp.162–173.

[16] J. M. Voas and K. W. Miller, The revealing power of a test case, *Journal of Software Testing, Verification, and Reliability*, Vol.2, No.1, May 1992, pp.25-42.

[17] J. M. Voas, and Miller K. W, Software testability: The new verification, *IEEE Software*, Vol.12, No.3, May 1995, pp.17-28.

[18] J. M. Voas, PIE: A dynamic failure-based technique, *IEEE Transactions on Software Engineering*, Vol.18, No.8, August 1992, pp.717-727.

Table 1. ***Testability Measurement of vending1 and vending2***

vending1				*vending2*			
l	***E(l)***	***P(l)***	***T(l)***	***l***	***E(l)***	***P(l)***	***T(l)***
1	1.00	0.00	0.00	1	1.00	0.12	0.12
2	1.00	0.32	0.32	2	1.00	0.48	0.48
3	1.00	0.00	0.00	3	1.00	0.02	0.02
4	1.00	0.28	0.28	4	1.00	0.28	0.28
5	0.84	0.42	0.35	5	0.84	0.72	0.60
6	0.84	0.52	0.44	6	0.84	0.64	0.54
7	0.36	0.08	0.03	7	0.84	0.76	0.64
8	0.80	0.44	0.35	8	0.36	0.08	0.03
9	0.80	0.32	0.26	9	0.80	0.52	0.42
10	0.52	0.16	0.08	10	0.80	0.74	0.59
11	0.28	0.16	0.04	11	0.80	0.32	0.26
12	0.24	0.00	0.00	12	0.52	0.52	0.27
13	0.24	0.00	0.00	13	0.52	0.12	0.06
14	0.24	0.18	0.04	14	0.28	0.16	0.04
15	0.24	0.26	0.06	15	0.24	0.24	0.06
16	0.80	0.74	0.59	16	0.24	0.04	0.01
17	0.36	0.28	0.10	17	0.24	0.04	0.01
18	0.36	0.28	0.10	18	0.24	0.22	0.05
19	0.36	0.28	0.10	19	0.24	0.22	0.05
Average Testability			**0.166**	20	0.80	0.74	0.59
				21	0.36	0.28	0.10
				22	0.36	0.28	0.10
				23	0.36	0.28	0.10
				Average Testability			**0.236**