2009 Fourth International Conference on Software Engineering Advances

Analysis of the Introduction of Testability Antipatterns During the Development Process

Muhammad Rabee Shaheen
Universités de Grenoble
Laboratoire d'Informatique de Grenoble (LIG)
Grenoble, France
Email: Muhammad-Rabee.Shaheen@imag.fr

Lydie du Bousquet
Universités de Grenoble
Laboratoire d'Informatique de Grenoble (LIG)
Grenoble, France
Email: Lydie.du-Bousquet@imag.fr

***Abstract*—Testability is a software characteristic that aims at producing systems easy to test. A testability antipattern is a factor that could affect negatively the testability of software. In this paper we compare the antipatterns at source code level and at different abstraction levels, in order to understand at which point they are introduced during the development.**

***Keywords*-testing; testability; antipatterns;**

I. Introduction

Software testing is one of the most expensive phases in the life cycle of software development. It is expensive in terms of time and money. It can represent 40% of the total development cost [1].

Practitioners and researchers are looking for solutions to decrease the cost of testing. One idea is generating tests automatically from the specification (or the model). Another idea is building systems easy to test, in order to reduce this cost. It is called "design for testing" or "design for testability" [9]. This idea relies on the observation that for a same problem, different solutions (with different designs) can be produced. Some of them are easier to test than others. "Design for testability" favours design solution(s) that would ease the test. To do that, one needs to identify the different types of weaknesses that make test process expensive. For instance, several testability metrics have been proposed for OO design and code such as the Chidamber and Kemerer set [12], [13]. Some of these metrics are used as quality predictor. A large part of researches focuses on the validation of these metrics with respect to their predictive abilities [3], [10], [11], [14], [15], [18], [24], [28], [29].

Another solution is detecting and avoiding "testability antipatterns". It is a design solution known to make test difficult (and/or known to increase the number of test to carry out) [4]. Two testability weaknesses have been described in [6]: ***self usage*** and ***interactions***. Both of them characterize dependency cycles in classes, which are known to make integration testing difficult [19].

To design for testability, it is important to understand when testability antipatterns are introduced during the development process (design and/or implementation phases) in order to detect and avoid them as early as possible.

This paper focuses on the analysis of the introduction of testability antipatterns during the development process. We studied 13 open-source Java applications to detect the self usage and interactions antipatterns occurrences at the source code and at different levels of abstraction. The objective was to observe how frequent, how complex the cycles in those applications and at which level they are introduced. We especially wanted to see if the cycles are mainly introduced during the modeling or the coding phase.

The remainder of this paper is organized as following. Section 2 introduces the concept of testability. Section 3 describes the antipatterns. Section 4 defines the abstraction levels that were used in this study. Section 5 presents our analytical study. Section 6 concludes and draws some perspectives.

II. Testability

Originally, testability was defined for hardware components. For several decades, testability has become a preoccupation for software systems. Several software-oriented definitions have been proposed for testability. Most of them characterize testability as the complexity or the effort required for testing. For some of them, software testability is expressed in terms of controllability and observability [16], [17], [25], [26]. Other testability definitions are related implicitly or explicitly with some specific testing method [2], [20]. In [2], testability is the effort needed for testing. For Binder, testability is the relative ease and expense of revealing software faults [9]. Other definitions allow a quantitative evaluation of the testing effort. For IEEE, it is also considered as "the degree to which a system or component facilitates the establishment of test criteria and the performance of tests to determine whether those criteria have been met" [21]. In [7], the testability is the probability that a test of the program on an input drawn from a specified probability distribution of the input is rejected, given a specified oracle and given that the program is faulty. In other

978-0-7695-3777-1/09 $26.00 © 2009 IEEE
DOI 10.1109/ICSEA.2009.27

words, it is the probability to observe an error at the next execution if there is a fault in the program.

In [9], Binder identifies six primary factors influencing testability: *representation*, *implementation*, *built-in test*, *test suites*, *testing environment/tool*, and *process* (see Fig. 1). He notices that "as practical matter, testability is as much a process issue as it is a technical problem".

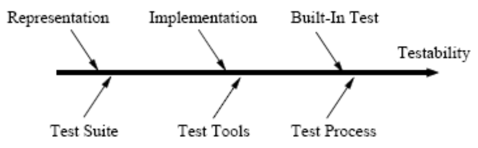

Figure 1. Testability main facets

III. Testability Antipatterns

To detect testability weaknesses, several metrics have been proposed [4], [6], [19]. Most of them focus on unit testing (either at a class or a method level). At integration level, difficulty of testing usually depends on dependencies among classes, which require an order in the integration. When a dependency cycle exists, it has to be *broken*. Detecting these cycles could be done either at ***presentation*** facet or at ***implementation*** facet Fig. 1.

A testability antipattern "describes undesirable configurations in the class diagram", contrary to design patterns represent good solutions to problems that arise when software is being developed in a particular context [4]. A testability antipattern could affect testability efforts intractable and make tests ineffective [8], [5]. Two testability antipatterns were proposed in [6], to capture the fact that there is a dependency cycle among the classes.

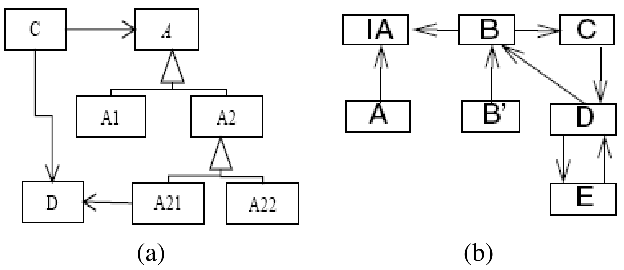

Figure 2. Class interaction and Cycles

An object-oriented system is a set of classes that communicate with each other. Some of these classes depend on others. In [19], two types of dependencies were defined. *Physical dependency* exists between two classes A and B if A cannot be *compiled* without B (e.g. inheritance). *Logical dependency* exists between A and B if a change happens to a B would require a change to A.

A dependency relationship could be either direct or indirect. It is indirect if the path between A and B includes other classes, otherwise it is direct. Figure 2(a) shows a dependency (interaction) between the classes C and D, which could be interpreted either directly or indirectly through the inheritance tree. Figure 2(b) represents the notion of class cycle, for example, one cycle exists between D and E, another one exists also between B and D through C. More classes in a cycle will increase the difficulty of the test, one reason for that is the need for more instances to be instantiated.

These types of dependency (interaction through more than one path and class cycle) should be avoided either by refactoring or by using class interfaces [4], [19]. Several studies [7], [27] focused on detecting and assessing the dependency. Jungmayr introduced a set of metrics to measure the dependability and its influence on the testability [19].

Since the cycles are introduced during the development at different levels, it would be interesting to have some tools that identify the cycles and allow the developer to remove them. The following section defines certain levels of abstraction that we used in this study to detect at which point(s) the cycles are introduced.

IV. Abstraction Levels

There are various methods of software development. Some of these methods propose *top-down* approach [23], that begins with describing the system at high level of abstraction (called model or high design level), then this description is refined progressively until getting the code (the final implementation).

In the following, we describe the different abstraction levels used in this study. These levels intuitively correspond to an approach that starts by highlighting the principle classes in the application, then it identifies their attributes, afterwards it defines their methods (signatures), then it defines the inner classes[1], and finally the complete coding.

The reason of taking the inner classes into account is a result of our previous study [22]. We found that inner classes participate significantly in forming the cycles. Indeed the inner classes do not appear early in the development, that is why the inner classes do not appear in the first and second levels of the abstraction levels that we define in this section.

Level 1 considers the attributes of the application classes, but neither considers the methods nor the attributes of an inner class type. In other words, it just considers attributes of simple types such as numbers, array, lists, string,...etc

[1]An inner class is a class that is defined in another class, it could be declared also inside the body of a method (either with name or without name).

and the attributes whose their types are classes which are not inner classes.

```
public class SampleExample{
 /* Attributes */
  int attribute_1;
  String attribute_2;
  byte[] attribute_array;
  Non_innerClassType attribute_complex;
}
```

Level 2 considers the attributes and the methods signatures of the application classes, but not those of inner class types, nor the methods which have one or more parameters of inner class types, nor the methods that return a value of an inner class type.

```
public class SampleExample{
 /* Attributes */
  int attribute_1;
  String attribute_2;
  byte[] attribute_array;
  Non_innerClassType attribute_complex;
 /* Methods */
  public int getAttribute_1(){
   return attribute_1;
  }
  public void setAttribute_1(int para0){}
  public void init(){}
}
```

Level 3 considers the attributes, methods signatures and inner classes. It also considers the attributes of inner class types, and the methods which have parameter(s) of inner class types, in addition to the methods that return a value of an inner class type.

```
public class SampleExample{
 /* Attributes */
  int attribute_1;
  String attribute_2;
  byte[] attribute_array;
  Non_innerClassType attribute_complex;
  InnerClassEx attribute_3;

 /* Inner Class Type */
  InnerClassEx(){
    // some attributes and methods
  }

 /* Methods */
  public int getAttribute_1(){
   return attribute_1;
  }
  public void setAttribute_1(int para0){}
  public void init(){}
  public void calc(int para0,
     InnerClassEx para1){}
}
```

Level 4 or Source code level represents the complete implementation of the class.

V. Quantitative Analysis

In this section we focus on the motivations of our study, the analysis of selected data and the results of this analysis.

A. Introduction

This work aims at studying several open source applications in order to detect and analyze the testability antipattern occurrences, and especially the class cycles, with the intention to understand at which point the cycles are introduced during the development. We were motivated by several questions:

- Is it frequent to find cycles in a model?
- What is the percent of/How many cycles are introduced during the coding?
- How many classes could be found in a cycle at the model and at the source code level?
- What is the size of smallest/ biggest cycle at the both levels?
- Are there any elements that favour the occurrence of cycles? Which ones?

To answer these questions we analyzed 13 open-source Java applications, from different domains. Table I gives small description for these applications, and their sizes expressed by the number of packages and classes. They represent 173 packages and more than 3260 classes and interfaces. They were chosen arbitrarily on the web mainly from sourceforge.

It would be interesting to collect data during the real development of the application. However, this is really difficult because the models that are used to build an application are not distributed with the application. Therefore, we chose to simulate the top-down approach using the reverse engineering.

To get the first three levels of abstraction, we developed a program based on Java reflection. It can extract the three levels from the byte code. We have compiled each model in order to get a model that is free of compilation errors. Then we have used Classycle's Analyser[2], an Eclipse plugin, to analyze the class dependencies in Java applications at the source code level and at the abstraction levels.

Obtaining the models using this method has two properties. First, all applications are consistent with their models. Second all models are expressed at the same level of abstraction. As a consequence, this method makes the comparison consistent.

B. Data Analysis

Our analysis is done at four levels, the three abstraction levels in addition to the source code level. For each level we identify the cycles which are present in each application, the maximum and minimum number of classes in each cycle.

[2]http://classycle.sourceforge.net/

Application	Description	# packages	# classes
1-NanoXML$^{(1)}$	small XML parser for Java	1	19
2-Chemical Evaluation Framework 1.001$^{(*)}$	software to assist in hazard assessment	3	127
3-Jsxe$^{(*)}$	Java simple XML editor	25	453
4-XMLMath$^{(*)}$	XML-based expression evaluator	2	80
5-HTMLCleaner$^{(*)}$	Transforms HTML a into a well-formed XML	1	27
6-MegaMek-v0.32.2$^{(*)}$	A networked Java clone of BattleTech	32	811
7-weirdx-1.0.32$^{(*)}$	A pure Java X Window System server	3	108
8-Java Gui Builder0.6.5a$^{(*)}$	Decouple GUI code from the rest of application	19	163
9-Bluepad v0.1$^{(*)}$	Turns cellphone to a remote PC controller	5	11
10-Jaxe$^{(*)}$	Java XML editor	7	178
11-JDom1-1$^{(2)}$	Access to XML data	7	68
12-JFreeChart-1.0.9$^{(*)}$	Create Charts	41	475
13-KoLmafia-v12.3$^{(*)}$	A interfacing tool with online adventure game	27	740
All	-	173	3260

(1) `http://nanoxml.cyberelf.be`; (2) `http://www.jdom.org/`; (*) `http://www.sourceforge.net/` ;

Table I
DATA SOURCE

The different levels of abstraction allow us to know how many cycles are introduced at each level. Although it is clear that the source code level will have the maximum number of cycles, and the *level 1* will have the minimum number of cycles, we do not know at which level the number of cycles begins to increases significantly, and at which level the cycle's complexity increases.

Table II shows for each application the maximum cycle size, minimum cycle size, number of cycles, total number of classes that form all the cycles, number of inner classes that appear in the cycles, and the percent of inner classes with respect to the total number of cycle classes.

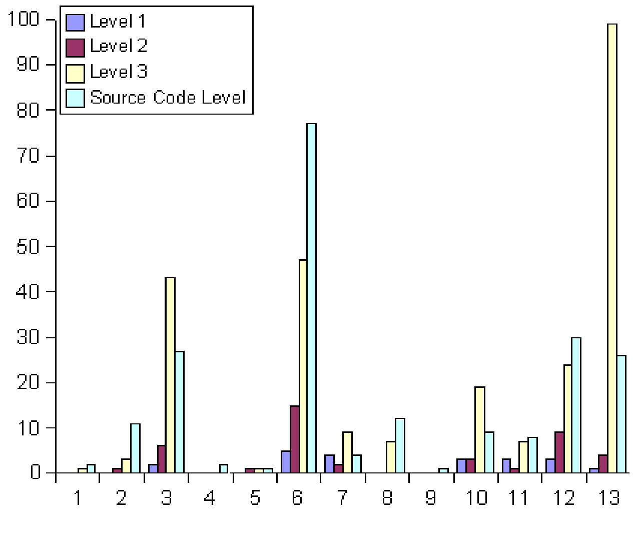

Figure 3. Number of cycles per application at each level

Table III shows, for each abstraction level, the maximum cycle size, minimum cycle size and number of cycles. From both Table II and III, we can observe that the number of cycles increases progressively from level one to level four (source code level).

Additionally, from Figure 3 one can notice that the number of cycles for the 3rd, 7th, 10th and 13th application at *level 3* is greater than the number of cycles at the source level. To interpret this observation we study the structure of the cycles, and we found that some cycles at the source level could be formed from a combination between two or more cycles found at *level 3*. This also could be seen by comparing the maximum number of classes that form the cycles at the source level and the level 3 (see Table II and III).

This analysis leads us to two conclusions. First, it is important to detect the cycles at the model level and to refactor them in order to eliminate them as many as possible and as soon as possible. Because the more there are cycles in the model, the more cycles will be in the coding phase, or the more complex they will be. And thus, the difficulty of testing is increased.

Second, it is not sufficient to analyze cycles at the model level, since new cycles would be introduced during the coding phase. This means that analyzing and refactoring of the code may be required to ease the test. Moreover, this also means that integration tests may not be built only from the model since it may not describe existing cycles at the source code.

C. Limitations of the work

In this study, we focused only on the relationships between classes regardless of the dependency between the packages. In this analysis we have simulated the development by defining three levels of abstraction and collecting data using reverse engineering. But these levels do not represent real development case, which is not possible due

Application/Cycle size	#Max	#Min	#Cycles	#Classes in all cycles	#Inner Classes	%Inner classes
NanoXML	2	2	2	2	1	50
CEF	38	2	11	105	91	86.67
JSXE	129	2	27	361	209	57.89
XmlMath	4	2	2	6	1	16.67
HtmlCleaner	21	21	1	21	0	0
MegaMek	101	2	77	571	321	56.22
Weirdx	58	2	4	65	10	15.38
JavaGUIBuilder	6	2	12	34	22	64.71
Bluepad	10	10	1	10	0	0
Jaxe	132	2	9	150	84	56
JDom	24	2	8	44	13	29.55
JFreeChart	36	2	30	121	22	18.18
KoLmafia	598	2	26	703	396	56.33
Total	-	-	184	2193	1170	53.35

Table II
MAX/MIN CYCLE SIZE AND NUMBER OF CYCLES AT THE SOURCE CODE LEVEL (LEVEL 4)

	Level 1			**Level 2**			**Level 3**		
Application/Cycle size	#Max	#Min	#Cycles	#Max	#Min	#Cycles	#Max	#Min	#Cycles
1-NanoXML	0	0	0	0	0	0	2	2	1
2-CEF	0	0	0	3	3	1	3	3	3
3-Jsxe	2	2	2	8	2	6	46	2	43
4-XMLMath	0	0	0	0	0	0	0	0	0
5-HTMLCleaner	0	0	0	8	8	1	10	10	1
6-MegaMek	13	2	5	39	2	15	45	2	47
7-weirdx	13	2	4	27	2	2	27	2	9
8-Java Gui Builder	0	0	0	0	0	0	3	2	7
9-Bluepad	0	0	0	0	0	0	0	0	0
10-Jaxe	4	3	3	9	3	3	24	2	19
11-JDom	3	3	3	8	8	1	10	2	7
12-JFreeChart	2	2	3	31	2	9	31	2	24
13-Kolmafia	2	2	1	29	2	4	29	2	99
Total			21			42			260

Table III
MAX/MIN CYCLE SIZE AND NUMBER OF CYCLES AT THE ABSTRACTION LEVELS

to the difficulty of obtaining the *real* models that are used for building the applications under study.

VI. CONCLUSION AND PERSPECTIVES

Testability is software factor that could be used to detect the different weaknesses that could increase the difficulty of the test. Antipatterns are weaknesses that reduce the testability of software. The antipatterns could be detected early in the life cycle of software development. The goals of our analysis were firstly, to find if there is any relationship between the occurrence of antipatterns and the characteristics of the code, secondly showing that eliminating cycles and antipatterns at model level does not eliminate the ones at code level. Therefore more tests will be needed at the coding phase.

This study based on 13 open-source Java applications. To summarize the results of this work we state that, the occurrence of cycles starts to appear during the modeling phase and it is important to detect them as early as possible. Moreover, detecting the cycles at the model level is important, but it is not sufficient to avoid all problems caused by the cycles. One reason behind that either the number of cycles introduced during any level of abstraction increases at the next refined level or the cycles may become more complex. Therefore, analyzing the cycles at each level of development is essential, and it is interesting to have some tools associated with the development environments that could detect the cycles during the different phases of development. Additionally, as a perspective of this work, it would be appreciated to continue the analysis with real data collected from real models, in order to confirm the presented results. Furthermore, since the cycles are not the only source of testability antipatterns, it would be interesting to study other sources of testability antipatterns. That may help developers to avoid the different sources of antipatterns or at least to avoid the most critical ones.

REFERENCES

[1] Beizer B. *Software Testing Techniques*. Van Nostrand Reinhold Company Inc., 1990.

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

[3] Victor R. Basili, Lionel C. Briand, and Walcélio L. Melo. A validation of object-oriented design metrics as quality indicators. *IEEE Trans. Software Eng.*, 22(10):751–761, 1996.

[4] B. Baudry, Y. Le Traon, G. Sunyé, and J.-M. Jézéquel. Measuring and improving design patterns testability. In *9th IEEE International Software Metrics Symposium (METRICS 2003)*, pages 50–, Sydney, Australia, September 2003.

[5] B. Baudry and Y. Le Traon. Measuring design testability of a UML class diagram. *Information & Software Technology*, 47(13):859–879, 2005.

[6] B. Baudry, Y. Le Traon, and Gerson Sunyé. Testability analysis of a uml class diagram. In *8th IEEE International Software Metrics Symposium (METRICS 2002)*, pages 54–, Ottawa, Canada, June 2002.

[7] A. Bertolino and L. Strigini. On the use of testability measures for dependability assessment. *IEEE Trans. Software Eng.*, 22(2):97–108, 1996.

[8] J. M. Bieman and C. Izurieta. Testing consequences of grime buildup in object oriented design patterns. *First International Conference on Software Testing ICST*, 2008.

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

[10] L. C. Briand, Jürgen Wüst, S. V. Ikonomovski, and H. Lounis. Investigating quality factors in object-oriented designs: An industrial case study. In *ICSE*, pages 345–354, 1999.

[11] M. Bruntink and A. van Deursen. An empirical study into class testability. *Journal of Systems and Software*, 79(9):1219–1232, September 2006.

[12] S. R. Chidamber and C. F. Kemerer. A metrics suite for object oriented design. *IEEE Trans. Soft. Eng.*, 20(6):476–493, 1994.

[13] F. Brito e Abreu and R. Carapuça. Candidate metrics for object-oriented software within a taxonomy framework. *Journal of Systems and Software*, 26(1):87–96, 1994.

[14] F. Brito e Abreu and W. L. Melo. Evaluating the impact of object-oriented design on software quality. In *IEEE METRICS*, pages 90–99, 1996.

[15] K. El Emam, S. Benlarbi, N. Goel, and Shesh N. Rai. The confounding effect of class size on the validity of object-oriented metrics. *IEEE Trans. Software Eng.*, 27(7):630–650, 2001.

[16] R. S. Freedman. Testability of software components. *IEEE Trans. Software Eng.*, 17(6):553–564, 1991.

[17] A. Goel, S. C. Gupta, and S. K. Wasan. Controllability mechanism for object-oriented software testing. In *10th (APSEC 2003)*, pages 98–107, Chiang Mai, Thailand, December 2003. IEEE Computer Society.

[18] T. Gyimóthy, R. Ferenc, and I. Siket. Empirical validation of object-oriented metrics on open source software for fault prediction. *IEEE Trans. Soft. Eng.*, 31(10):897–910, 2005.

[19] Stefan Jungmayr. Identifying test-critical dependencies. In *18th International Conference on Software Maintenance (ICSM*, pages 404–413. IEEE Computer Society, 2002.

[20] S. Mouchawrab, L. C. Briand, and Y. Labiche. A measurement framework for object-oriented software testability. *Information & Software Technology*, 47(15):979–997, 2005.

[21] Institute of Electrical and Electronics Engineers. IEEE Standard Computer Dictionary: A Compilation of IEEE Standard Computer Glossaries. Technical report, IEEE, New York, USA, 1990.

[22] Muhammad-Rabee Shaheen and Lydie du Bousquet. Quantitative analysis of testability antipatterns on open source java applications. In *12th ECOOP Workshop on Quantitative Approaches on Object Oriented Software Engineering*, 2008.

[23] I. Sommerville. *Software Engineering*. Addison Wesley, 2004.

[24] M.-H. Tang, M.-H. Kao, and M.-H. Chen. An empirical study on object-oriented metrics. In *6th IEEE International Software Metrics Symposium (METRICS'99)*, pages 242–249, Boca Raton, FL, USA, Nov. 1999. IEEE Computer Society.

[25] J.M. Voas and K. Miller. Semantic Metrics for Software Testability. *J. Systems Software*, 20:207–216, 1993.

[26] Y. Wang, G. King, I. Court, M. Ross, and G. Staples. On testable object-oriented programming. *SIGSOFT Softw. Eng. Notes*, 22(4):84–90, 1997.

[27] H. Yul Yang, E. D. Tempero, and R. Berrigan. Detecting indirect coupling. In *Australian Software Engineering Conference*, pages 212–221. IEEE Computer Society, 2005.

[28] P. Yu, T. Systä, and H. A. Müller. Predicting fault-proneness using OO metrics: An industrial case study. In *6th CSMR 2002*, pages 99–107, Budapest, Hungary, March 2002. IEEE Computer Society.

[29] Y. Zhou and H. Leung. Empirical analysis of object-oriented design metrics for predicting high and low severity faults. *IEEE Trans. Software Eng.*, 32(10):771–789, 2006.