An Empirical Verification of Software Artifacts Using Software Metrics

Raed Shatnawi and Ahmad Alzu'bi

***Abstract*—In model-driven development, design understandability is very important to maintain software systems. Software developers use the design models in their endeavor to understand and maintain the final product. Typically, software developers expect consistency between design and implementation artifacts of a software system. However, software systems may deviate from design. Software verification is an important phase to prevent this deviation and to ensure that developers are building the product right. There are few research studies that were conducted to assess the correspondence of a software implementation to its design. In this paper, we validate the use of a hierarchal quality model to verify the correspondence between design and implementation artifacts. We have conducted this study on an open-source system that is built using C++. The model has shown that we can discover the differences between the design and implementation. We found that hierarchal models could be used to find differences at the low-level design properties and to find similarities at the architecture-level design properties.**

***Index Terms*— object-oriented metrics, software quality, QMOOD.**

I. INTRODUCTION

Software verification is very important to ensure that software consistency is maintained among consecutive software artifacts. Software verification has effect on both the process and product quality. However software verification is a time-consuming activity especially if worked only at later stages. In a matured software development process, a planned design (represented in models) is one way to ensure that the implementation matches the design. The correspondence between design and implementation artifacts is an indication of the stability of the system. However, few business requirements are stable [1] and change unexpectedly. The changes vary depending on many factors such as when those changes occur, the number of affected components, and the nature of changes. Changes that force the developers to redo major changes of the software are very costly, especially in the implementation phase which requires more time, rework, and review [2]. In addition, a lack of correspondence between early software activities has a ripple effect on later process activities such as testing and maintenance. The lack of correspondence can occur for many reasons such as implementation mismatch (e.g. by mistake or by purpose), change of requirements, and the necessity to remove bugs [3]. In the Object-oriented (OO) paradigm, software systems are composed of classes, methods, and attributes. These components appear in both artifacts of the design and implementation. The properties of these components are measurable in both the design and implementation The OO paradigm can be characterized using many properties such as inheritance, polymorphism, and encapsulation. Assessing quality by measuring internal properties offers an objective and an independent view of the quality [4]. Therefore, we can use these measurements to verify the quality of both design and implementation artifacts, i.e., verify the conformance between both artifacts. The aim of this research is to validate a hierarchal quality model to verify the conformance between OO design models, class diagrams in this research, and the software implementation. The quality model uses metrics to measure the internal properties to characterize both the design and implementation. The model is validated empirically on an open-source system—Turaya.Crypt—that is developed in C++. The proposed model can be used to investigate the differences between the design and implementation artifacts. In addition, the model helps software designers to draw a link between design and implementation phases to verify their expectations. A paired t-test is conducted at several levels (classes and subsystems) to provide an evidence of the correspondence. This helps developers or testers conducting a quantitative analysis to explore the conformance or the lack of correspondence between design and implementation.

The rest of the paper is organized as follows: in Section 2, we discuss the related work. In Section 3, the verification methodology and the experiment model are illustrated. In Section 4, research hypotheses are stated. In Section 5, the verification model is applied on an OO software system. The conclusions and future trends are discussed in Section 6.

II. RELATED WORK

An implementation is said to conform to its design if everything that was designed is implemented [3]. Few approaches have been proposed to verify or assess the correspondence between a software design and its

Manuscript received December 06, 2010; revised February 09, 2011.

Dr. R. Shatnawi is an assistant professor in the Software Engineering Department, Jordan University of Science and Technology, Irbid, Jordan 22110, (phone: 011962777562690, e-mail: raedamin@just.edu.jo.)

Ahmad Alzu'bi finished his master thesis from the Computer Science Department, Jordan University of Science and Technology, Irbid, Jordan 22110, (e-mail: agalzoubi06@cit.just.edu.jo)

implementation. Techniques were developed to assess the conformance based upon different measures and models. In [3], authors developed quantitative techniques for the assessment of correspondence between UML diagrams and implementation. For the assessment of correspondence, they used a metamodel which is inspired upon the UML metamodel. For both design and implementation they instantiate such a metamodel. The instantiation of the design metamodel is a subset of the UML model. The same holds for the implementation meta model. It is a subset of the actual implementation where they left out all implementation details. The correspondence between a design class and an implementation class has expressed in terms of a similarity value . They considered matching based on classifier names, matching based on metric profiles and matching using package information (i.e. structural properties of classifiers). Deniss et al [3] used the software reflexion model to visualize the differences between design and implementation using implementation relations as inputs. In another study, Antoniol et al. [5] have compared different traceability recovery methods, based on different class properties. This technique complements their previous works described in [6,7], which was focused on the traceability procedure itself. The design is modeled using OMT notations and the procedure accepts C++ source code. Both design and code are represented using a custom OO design description language, the Abstract Object Language (AOL). The process recovers an "as-is" design from the code in AOL, compares the recovered design with the actual AOL design, and helps the user to deal with inconsistencies by providing a similarity measure for the matched classes and pointing out the unmatched ones. Antoniol's et al. works [5]-[7] recovered design-code traceability links by computing the similarity between each pair of properties in design and code. They have used a maximum likelihood classifier [5], to get an optimal threshold value for the similarity measure. In another related model, a software reflexion model [8] technique was developed. The engineer defines a high-level model of interest, extracts a source model (such as a call graph or event interactions) from the source code, and defines a mapping between the two models. A software reflexion model is then computed to determine where the engineer's high-level model does or does not agree with the source model. If the relationship occurs in both the design and the implementation, then it is called a convergence. If the relationship occurs in the design but is absent in the implementation, then it is called an absence. If the relationship occurs in the implementation but not in the design, then it is called a divergence. All these studies about the correspondence between software design and implementation are different from our work in the conducted strategy for design and its implementation traceability. Our model uses a validated quality model (QMOOD) that assess the evolution of a software product form design to implementation. QMOOD can be used to measure both the design and implementation. QMOOD measures the components of a software system for eight OO properties. Therefore, it verifies software quality for the OO paradigm.

III. THE VERIFICATION FRAMEWORK

This research introduces a new model for the verification of the correspondence between design and implementation using software metrics. The measurements of both artifacts are derived from a hierarchal quality model [9] that was proposed and validated on commercial software systems to assess software evolution [10]. The intent of our verification technique is completely different from the intent of the QMOOD quality model. While the QMOOD model aims to assess high-level design quality attributes in OO designs, our research aims to verify the correspondence between software design and implementation using OO metrics. QMOOD uses eight OO metrics to measure design and implementation properties. The properties are: inheritance, encapsulation, polymorphism, abstraction, coupling, messaging, composition, and complexity. Appendix I shows the definitions of these properties and their measurements. The QMOOD is validated on an OO system—Turaya.Crypt (Secure Linux Hard-Disk Encryption) which consists of 75 classes. The system is based on the microkernel-based EMSCB [11] security kernel in Linux systems. This system consists of five subsystems which are listed in Table 1. Both The UML diagrams for the design phase and the source code files are available on EMSCB online repository[1]. Multiple tools were used to collect the metrics data to achieve higher accuracy. Two tools were used to collect metrics from the source code: Resource Standard Metrics (RSM)[2] and Understand 2.0[3]. Metrics from UML diagrams were collected by hand.

TABLE 1
Turaya.Crypt subsystems

Subsystem Name	#Classes
HddEncServer	21
LibUtils	25
Server GUI	19
Launcher	5
LinuxStub	5
Total	75

IV.RESEARCH HYPOTHESES

To verify the significance of the correspondence between software design and implementation, a set of hypotheses are stated for the internal properties. We conduct the verification at two levels (classes and subsystemes). Therefore, we want to validate the following hypotheses for both levels. The Null hypotheses (*H01 to H08*) of the internal properties:

- **H01c to H08c**: There is no significant difference in a quality property between design and implementation at the class level.

[1] http://svn.emscb.org/svn/emscb/trunk/apps/hddenc/.
[2] http://msquaredtechnologies.com.
[3] http://www.scitools.com/prodcts/understand.

ISBN: 978-988-18210-3-4
ISSN: 2078-0958 (Print); ISSN: 2078-0966 (Online)

- **H01s to H08s**: There is no significant difference in a quality property between design and implementation at the subsystem level.

For each null hypothesis, the letter (c) stands for class level and the letter (s) stands for subsystem level. To accept or reject these hypotheses, we use the paired t-test at the 95% confidence level [13, 14]. The significant value in this test decides whether the hypothesis is rejected or not. If the significance value of the samples (design and code) is larger than or equal to 0.05 then there is no significant difference, and the null hypothesis cannot be rejected, i.e., there is no significant difference in a quality property between design and implementation. Otherwise, the null hypothesis is rejected.

V. VERIFICATION ANALYSIS

In this section, our verification technique is applied on Turaya.Crypt system (EMSCB 2006). The collected metrics are described statistically. These metrics are verified using the paired t-test at two levels: all classes (for design and code) in the system, and all classes of each subsystem.

A. DESCRIPTIVE STATISTICS

For Turaya.Crypt system, the mean, standard deviation, maximum, and minimum values for the system classes are calculated. Table 2 lists these values for all metrics. These descriptive values help us to observe the differences between the properties of design and implementation. For example, MFA shows no mean differences between design and implementation. Although we can use the mean to compare the effect of implementation, we cannot draw conclusions without conducting a statistical test which we discuss thoroughly in next sections. From Table 2, we notice that the DAM's mean is smaller for implementation which is an indication of encapsulation violations. NOP's mean is also smaller for implementation which is a sign of flexibility reduction in the code. ANA and DCC mean values indicate an increase in the inheritance and coupling during implementation. CIS and NOM mean values increase for implementation which indicate that classes' complexity increase as well.

TABLE 2
Descriptive statistics for all classes of Turaya.Crypt system

Metric	Mean		Stdev.	
	Des.	Imp.	Des.	Imp.
MFA	0.06	0.06	0.19	0.19
DAM	0.58	0.49	0.44	0.41
NOP	0.20	0.15	0.55	0.75
ANA	0.15	0.24	0.36	0.49
DCC	1.60	2.75	1.46	3.71
CIS	5.85	6.73	5.67	7.14
MOA	1.48	1.43	1.49	1.44
NOM	6.40	8.33	6.11	7.73

B. PROPERTIES OF CLASSES

In this section, we discuss the paired t-test results for the internal properties at the class level which are summarized in Table 3. From Table 3, the decisions for the null hypotheses ***H01c***, ***H02c***, and ***H06c*** are accepted, i.e. there is no significant difference in these properties in design and implementation. While ***H03c***, ***H04c***, ***H05c, H07c,*** and ***H08c*** hypotheses are rejected, i.e. there is a significant difference in these properties in design and implementation. These differences are design violations that hinder software maintainability. We notice that low-level changes are more common in implementation and therefore there were significant differences on the properties such as encapsulation, abstraction, coupling, messaging and complexity. Whereas no significant design violations on higher-level design elements such as inheritance, polymorphism and composition.

C. PROPERTIES OF SUBSYSTEMS

The results in the previous section stand for all classes (i.e., 75 classes). In this section, the calculations are repeated for the class pairs at the subsystems level. We have conducted the paired t-test for each subsystem that has enough number of classes. Table 4 shows the results of the paired t-test on the server subsystem. Inheritance and polymorphism properties (i.e. pairs 1 and 2 respectively) do not appear in Table 4. In this test, if the standard error of the difference is zero for a particular property, the statistics cannot be computed. If the hypothesis is accepted, it is indicated by (✓) symbol. Otherwise, the hypothesis is rejected (i.e. sig<0.05). As shown in Table 5, the HddEnc Server subsystem have the lowest degree of correspondence for their internal properties in design and code. Four properties have significant differences. Fig. 1 shows that the Server subsystem is the core of the Turaya.Crypt system, which explains the reason why this subsystem has the most inconsistencies. The Server subsystem complexity is larger than others. This verification explores where the software programmer or developer have a difficulty in transforming the design artifacts into source code.

D. DISCUSSION

We can use the QMOOD to specify which parts (i.e. subsystems and classes) of the software system have differences and their effects on the software structure and quality needs. The tester or developer has many quantitative indicators to conduct the correspondence verification. In this study, we found some properties that have an explicit evidence of the correspondence, but others have a lack of correspondence. Inheritance, polymorphism and composition have the highest correspondence level. The differences and their possible effects on software properties are caused by micro changes during implementation. These changes can be summarized as follows:

Changes on complexity: software designers may not have a complete conception of how to achieve the functionality of the class or the object. New additional methods (i.e. larger NOM) are introduced in the implementation but missed in the design. One reason for adding methods is the change of

ISBN: 978-988-18210-3-4
ISSN: 2078-0958 (Print); ISSN: 2078-0966 (Online)

software needs during implementation, but this change is not reflected to design models [5]. For Turaya.Crypt, there are 145 additional methods in the implementation (i.e. 66 public, 68 private, and 11 protected). The classes in the implementation tend to be more complex than those in the design.

TABLE 3
Paired t-test's results of all internal properties

Property Pairs in design vs. code		Mean Diff.	Sig.
Pair 1[4]	Inheritance	-	-
Pair 2	Polymorphism	-0.066	0.13
Pair 3	Encapsulation	0.095	0.016
Pair 4	Abstraction	-0.09	0.019
Pair 5	Coupling	-1.13	0.003
Pair 6	Composition	0.053	0.103
Pair 7	Messaging	-0.88	0.018
Pair 8	Complexity	-1.93	0.000

TABLE 4
The results of paired t-test applied on properties of server subsystem

OO Property Design vs. Code		Mean Diff.	Sig.
Pair 3	Encapsulation	0.324	0.000
Pair 4	Abstraction	-0.190	0.042
Pair 5	Coupling	-2.809	0.020
Pair 6	Composition	0.190	0.104
Pair 7	Messaging	-1.619	0.002
Pair 8	Complexity	-0.666	0.314

TABLE 5
Results of paired t-test for subsystems

Property	All	HddEnc Server	GUI	LibUtils
Inheritance	✓	✓	✓	✓
Polymorphism	✓	✓	✓	✓
Encapsulation				✓
Abstraction			✓	✓
Coupling			✓	
Composition	✓	✓	✓	✓
Messaging			✓	
Complexity		✓		

Changes on Coupling*:* adding public methods increases the accessibility for other classes to those methods. If public methods are used by another software component, this causes an additional dependencies between software components and decreases system understandability.

Changes on encapsulation*:* introducing new private and protected attributes (i.e. larger DAM values) in implementation (78 public, 87 private, and 3 protected attributes) have less impact on the correspondence than public methods. This is because the private attributes are accessible only by other members in the class, and the protected attributes are accessed locally. Encapsulation and information hiding are considered to be a convenient programming tactic in OO software. The lack of encapsulation violates the security of classes and unexpected side effects may happen, which increases the possibility of software defects.

Changes on abstraction: introducing new subclasses during software implementation adds more specializations and does not leave the room for flexibility of change. If an inheritance hierarchy is long then the complexity of software increases and software becomes harder to understand and maintain.

Changes on messaging*:* introducing additional public methods (66 methods) causes more message passing (i.e. larger CIS) between classes. The impact of large messaging in implementation affects the coupling between classes and their complexity. It also has an adverse impact on software understandability.

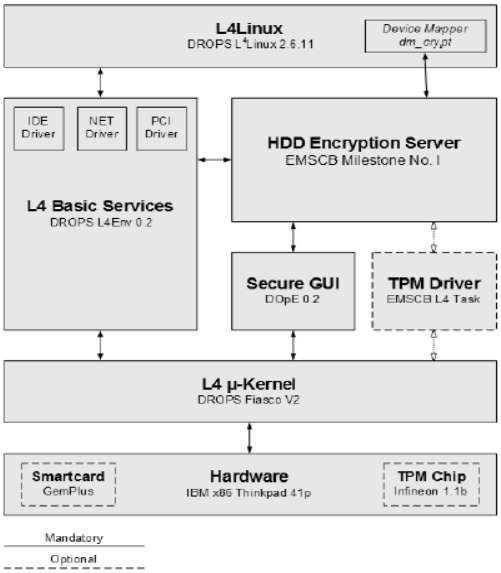

Fig. 1. The architecture of the secured Linux hard-disk encryption system

VI.CONCLUSIONS AND FUTURE WORK

A quantitative verification model for the correspondence between OO design and its implementation has been developed. The verification is based on using an OO metrics to measure the classes in design and implementation phases. It helps in the verification and validation processes, which saves time, effort, and rework. The technique uses internal properties of software (e.g. encapsulation, inheritance, and coupling) that are measured quantitatively using metrics. The software developer verifies the differences between metrics to estimate their magnitude on the software development. Software designs provide vital information about the developer's view of software's responsibilities, structure, and the collaboration among components. This eases the process

[4] The results of inheritance property (Pair 1) do not appear in the Table 3 because the standard error of mean equals zero. The statistics cannot be computed.

ISBN: 978-988-18210-3-4
ISSN: 2078-0958 (Print); ISSN: 2078-0966 (Online)

of specifying where differences occur, which classes deviate from design, and which subsystems or packages or milestones have more deviation. From our analysis, the core components of the system tend to have higher probability of differences in source code from its design. Their roles in the software increase their tendency to the change; they maybe edited frequently in implementation but not in design. The developed verification model is extendible; many metrics and tests can be applied based on the verification intent. However, improvements can be done to enhance this model. First, the verification model does not verify the cohesiveness of software components. Since the design models of the case study does not include details such as parameters of methods and the interconnections among methods and attributes. This requires more empirical analysis on designs that offer the ability to collect cohesion metrics form the design models. Second, the verification model has been applied on one case study because there is a limitation to access full software designs without a discloser agreement with the software authors. We plan to conduct similar studies on more case studies.

REFERENCES

[1] I. Sommerville, *Software Engineering*. 8th ed. Addison Wesley, 2006.
[2] R. Pressman, *Software Engineering: A Practitioner's Approach*. 6th ed. Professional computing series: McGraw-Hill. pp. 461–497, 2005
[3] J. Dennis, F. Christian and R. Michel, "Quantitative Techniques for the Assessment of Correspondence between UML Designs and Implementations," *Proceedings of the 9th QAOOSE workshop*, co-located with ECOOP 2005.
[4] G. Dormey, "A Model for Software Product Quality," In IEEE Trans. Software Eng.,vol. 21, no. 2. pp. 146–162, 1995.
[5] G. Antoniol, B. Caprile, A. Potrich and P. Tonella, "Design-code traceability recovery: selecting the basic linkage properties," *Science of Computer Programming*, vol. 40, no. 2-3, pp. 213–234, 2001.
[6] G. Antoniol, A. Potrich, P. Tonella and R. Fiutem, "Evolving Object Oriented Design to Improve Code Traceability," *In Proc. of the International Workshop on Program Comprehension*. Pittsburgh, PA, pp. 151–160, 1999.
[7] G. Antoniol, B. Caprile, A. Potrich and P. Tonella, "Design-code traceability for object-oriented systems", *Annals of Software Engineering*, pp. 35–58, 2000.
[8] C. Murphy, N. David and S. Kevin, "Software reflexion models: bridging the gap between source and high-level models," *SIGSOFT Software. Eng. Notes*, vol. 20, no.4. pp.18–28, 1995.
[9] J. Bansiya and C. Davis, "A Hierarchical Model for Quality Assessment of Object-Oriented Designs," *IEEE Transactions of Software Engineering*, vol. 28, no. 1. pp. 4-17, 2002.
[10] Bansiya J., *A Hierarchical Model for Quality Assessment of Object-Oriented Designs*. PhD Dissertation, Univ. of Alabama in Huntsville, 1997.
[11] EMSCB -European Multilaterally Secure Computing Base, "Turaya: Secure Linux Hard-Disk Encryption. Milestone No1. Documentation", available at www.emscb.com/content/pages/turaya.downloads.htm. (last accessed 27/6/2009), 2006.
[12] R. Chidamber and F. Kemerer, "A Metrics Suite for Object Oriented Design," *IEEE Transactions on Software Engineering*, vol. 20, no. 6, pp. 476–493, 1994.
[13] Goulden C., *Methods of Statistical Analysis*, 2nd ed. New York: Wiley, pp. 50–55, 1956.
[14] Pearson E. and Kendall M. (eds.), *Studies in the History of Statistics and Probability*. Darien, Conn: Hafner Publishing Company., 1970.

APPENDIX I

OO metrics to measure OO design and implementation properties.

Property	Metric Definition
Inheritance: Forms the (is-a) relationship, and occurred when a new class inherits methods and attributes from an existing class.	Measure of Functional Abstraction (MFA): The ratio of the number of methods inherited by a class to the total number of methods of the class (Range 0 to 1).
Encapsulation: Allows hiding the attributes (data) and methods (behavior) into the class. Hiding the definitions in OO software is desirable to protect them from outside access.	Data Access Metric (DAM): The ratio of the number of private and protected attributes to the total number of attributes declared in the class. Range is 0 to 1, and high values of DAM are desired.
Polymorphism: Allows using the same name in different contexts in the same scope.	Number Of Polymorphic Methods (NOP): This metric is a count of the methods that can exhibit polymorphic behavior, and such methods in C++ are marked as *virtual*.
Abstraction: Measures the generalization-specialization aspect in design, in which the number of descents of a class is determined.	Average Number of Ancestors (ANA): The average number of classes from which a class inherits information. It is determined by class inheritance structure in design by computing the number of classes along all paths from the root class to other classes in the inheritance structure.
Coupling: Indicates the degree of relationship in which one class interacts with other classes through its interfaces to achieve its functionality properly.	Direct Class Coupling (DCC): Count the different number of classes that a class is directly related to. It is determined through attributes declaration and message passing in methods.
Messaging: Stands for the services that a class provides to other classes.	Class Interface Size (CIS): The number of public methods in a class.
Composition: Measures "part-of", "has", "consist-of" relationships, which are aggregation relationships in OO design.	Measure of Aggregation (MOA): Counts the number of data declaration whose types are user defined classes, and it is realized by using attribute declaration.
Complexity: Depicts the difficulty of understanding and tracing software design and its implementation.	Number of Methods (NOM): The number of all methods defined in a class. It is equivalent to WMC [12].

ISBN: 978-988-18210-3-4
ISSN: 2078-0958 (Print); ISSN: 2078-0966 (Online)