A Verification of the Correspondence between Design and Implementation Quality Attributes Using a Hierarchal Quality Model

Raed Shatnawi and Ahmad Alzu'bi

***Abstract*—software verification is very important activity to prolong software quality. Many software systems deviate from their design when implemented. Typically, software engineers expect a high correspondence between design and implementation artifacts to ensure the quality of the final product. In this paper, we validate the use of a quality model to verify the correspondence between the artifacts of a software design and implementation. The model uses software metrics to measure the differences between the design graphical models (UML diagrams) and the source code for three external quality attributes: reusability, extendibility and understandability. The significance of the differences is verified using inferential and descriptive statistical tests. The proposed model is validated on a real open-source system that was developed in C++. The proposed model can be used to investigate the differences in a software quality either at the system or component levels. Many differences in quality attributes have been identified in the case study. The correspondence model has shown many characteristics; it is flexible, extendible and accepts different forms of design (UML diagrams) and code notations.**

***Index Terms*—design correspondence, software quality, QMOOD, quality models.**

I- INTRODUCTION

Object-oriented (OO) construction makes people think that software can be constructed like a building. However, software systems are intangible and the match between design and implementation is not guaranteed as in conventional engineering sense. Therefore software complexity is high and even when software is well-planned by following a strict software development process. Software quality assurance is very important activity to maintain software quality. A software system should maintain quality attributes such as extendibility, reusability and understandability.

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)

These quality attributes can be a essential characteristics of software systems. For example, extendibility is very important for modular design where classes in a system have the suitable level of abstraction , are loosely coupled, and show a dynamic behavior (late binding) via polymorphism. Such characteristics can be measured using software metrics which give indicators of the level of the software quality. Software quality assurance, however, is time-consuming activity especially if worked at later stages. The correspondence between software artifacts is considered an indication of the quality assurance. However, software systems are intangible and few business requirements are stable [1]. Changes on a software system and its constituents hinder the consistency between software artifacts. Changes has ripple effects, i.e., changes in a part of the system break other parts of the system. Changes are not necessarily caused by bad practices, rather it might be caused by complexity. Moreover, systems evolve continuously, leading to more complexity [2][3]. Many factors can cause a lack of correspondence such as implementation mismatch (e.g. by mistake or by purpose), change of requirements, and the necessity to remove the bugs [4]. The inconsistency between implementation and design leaves the software unattached to the planned design. In addition, these changes causes the software to deviate from the external quality attributes such as understandability, reusability, and extendibility. Therefore, the software process should involve a quality assurance activity to ensure the match between design and implementation.

Software quality can be measured directly and indirectly for the artifacts of design and code. The object-oriented paradigm includes internal properties such as inheritance, polymorphism, and encapsulation. The internal properties reflect what we can measure directly in the software systems. Assessing quality by measuring internal properties offers an objective and an independent view of software quality [5]. OO paradigm deals with components such as classes, methods, and attributes. These components are measurable and require suitable metrics to use for quality assurance purposes. These internal properties can be used to indirectly assess software external attributes (i.e. quality factors) such as reusability, extendibility and understandability. The external quality attributes measure the product quality based on the viewpoint in consideration. In fact, we are interested in the properties and the attributes of the design and the implementation phases. Both phases are important to produce high quality software. They form the core of the quality assurance in the early development phases.

The aim of this research is to propose and validate a hierarchal quality model to verify the correspondence between OO design models and its implementation. The model uses metrics to measure the internal properties and external attributes to achieve this goal. The proposed model is validated empirically on an open-source system—Turaya.Crypt—that was developed in C++. Statistical tests were conducted at several levels (system and subsystems levels) to provide an evidence of the correspondence. The proposed model is a Quality Model for Object-Oriented Design (QMOOD) that was proposed and validated on commercial software systems [6]. QMOOD has shown that the properties and the attributes of a software system can be used to investigate the differences between the design and implementation artifacts from two perspectives, internal and external. In addition, software designers expect that the implementation will conform to their designs. The model helps them to draw a link between design and implementation phases to verify their expectations. Furthermore, the model can be applied to various object-oriented programming languages such as C++, Java and C#.

The rest of this paper is organized as follows: in Section 2, we discussed 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 case study and how data is collected are described. In Section 6, the correspondence model is applied on an OO software system. The conclusions and future trends are discussed in Section 7.

II- RELATED WORK

Many techniques were developed to assess the conformance between design and code. These techniques were based on different measures and models. Dennis et al. [4] have developed a quantitative technique for the assessment of correspondence between UML design and its implementation. Their technique proposed a maximal matching algorithm that uses three elements: classifier names, metric profiles, and structural properties of classifiers (i.e. Package information). Deniss et al [4] used the software reflexion model to visualize the differences between design and implementation using implementation relationships as inputs. In another study, Antoniol et al. [7] have compared different traceability recovery methods based on different properties. This technique complements their previous works described in [8][9], which focused on the traceability procedure itself. Both design and code were modeled using a Abstract Object Language (AOL) and then they compared both products to find inconsistencies by providing a similarity measure. In another related model, a software reflexion model [10] 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. The engineer then look for three kinds of relationships: convergence, Absence, and divergence. All these studies about the correspondence between software design and implementation focus on similarities and differences between two artifacts. However, in our work we focus on the deviations in quality factors such as reusability, extendibility and understandability. We use a hierarchal quality model to verify the correspondence between design and implementation artifacts. Our model considers the relationship between the internal and external characteristics of a software system. Our model uses object-oriented metrics to measure the internal properties and external attributes to characterize the correspondence between the design and implementation.

III- THE CORRESPONDENCE MODEL

Many software quality models were proposed to verify the evolution of software quality. McCall has proposed a quality model to assess software products [11]. The model has a hierarchical nature of defining software product qualities based on metrics of measurable components. In 2001, ISO 9126 [12] has developed standards for measuring software quality that are similar to the McCall's model [11]) in complexity, but differ in the definition of model processes. ISO 9126 classifies software quality into set of structured characteristics that are decomposed into sub-characteristics. Dromey built a bottom-up quality model [5] that links and explores the relationship between internal software characteristics and external software quality attributes using the appropriate metrics. Dromey's model fixes some problems of earlier models such as the dependency between quality attributes, and the effect of each attribute on the whole or some of quality attributes of software. In general, Dromey's model identifies the quality in three stages: identify high-level quality attributes, identify the product's components with its quality-carrying properties, and link the quality attributes to the product properties. QMOOD model [6] extends the Dromey's model. It is based on the ISO 9126 six attributes which are reviewed by QMOOD authors to get a new set of attributes. Design quality in QMOOD is assessed using six quality attributes: effectiveness, understandability, extendibility, reusability, and flexibility. These attributes are identified based on the author's experience and empirical knowledge from working on object-oriented systems. QMOOD can be used to measure the quality of software when evolves. Therefore we use the QMOOD in our research to evaluate the evolution from deign to implementation. The QMOOD model have been used to assess software quality in many studies. For example, O'Keeffe and O´Cinneide [13] have evaluate alternative designs that can produce a better quality using QMOOD. In another study, Hsueh et al. [14] have used QMOOD to validate the effectiveness of design patterns as heuristics for good quality. QMOOD provides an assessment of the quality of the software artifacts after evolution, which is consistent with the measurements of the external quality factors mentioned above; it gives one measurement for each quality factor.

QMOOD was proposed by Bansiya and Davis [6] to assess software evolution. QMOOD provides a direct and indirect measurement of the software quality. Fig. 1 depicts the use of

the QMOOD in our effort to find inconsistencies in the quality of design and code. The QMOOD model is used to determine the internal and external properties of software components and their relationships. An OO system has many components such as classes, objects, and the relationships between them. A software design and implementation artifacts share many internal properties such as: inheritance, encapsulation, and polymorphism. They also share external quality attributes such as: reusability, extendibility and understandability. These OO properties and attributes are used in this research to verify software quality. In this section, we select the OO internal properties, OO quality attributes, and OO metrics to be used in the verification model. The selected properties are: inheritance, encapsulation, polymorphism, abstraction, coupling, cohesion, messaging, composition, design size, and complexity. QMOOD model is composed of a combination of six attributes: reusability, flexibility, understandability, extendibility, functionality and effectiveness [6]. We select three attributes, reusability, extendibility, and understandability to be used in the verification. Table 1 shows the definitions of these attributes. The verification technique uses the same OO metrics defined in QMOOD [6]. Table 2 lists the definitions of ten metrics that are defined in QMOOD.

As shown in Fig. 1, the methodology deals with the artifacts of two phases of the software development life cycle, design models and source code. Software metrics are collected from both artifacts. The process of data collection and preparing information as inputs to the assessment model should be well-planned to ensure accuracy. In this research, as discussed in previous sections, the data will be collected for two phases in software development (design and implementation). The data is collected for design phase from UML design diagrams and specification, and from source files written by C++ language for implementation phase. In the design phase, metrics are collected from class diagrams that depict the object-oriented software in a hierarchical view with each class contains methods and attributes. In the design phase, class diagrams depict the object-oriented software in a hierarchical view. These class diagrams were prepared by the designers of Turaya.Crypt[1] software [15]. The collected data and metrics from class diagrams uses information such as class definition, methods, and attributes. Also the relationships between class diagrams give information about class collaborations, dependencies, inheritance, aggregation, and other properties of object oriented paradigm. In the implementation phase, the representative form is the source code which is written in C++.

In the data collection process, multiple tools were used to achieve higher accuracy. For instance, more than one tool was used to collect metrics from the source code: Resource Standard Metrics (RSM[2]) and Understand 2.0 tool[3].

[1]http://www.emscb.com/contents/pages/turaya.downloads.htm
[2] http://msquaredtechnologies.com
[3] http://www.scitools.com/prodcts/understand

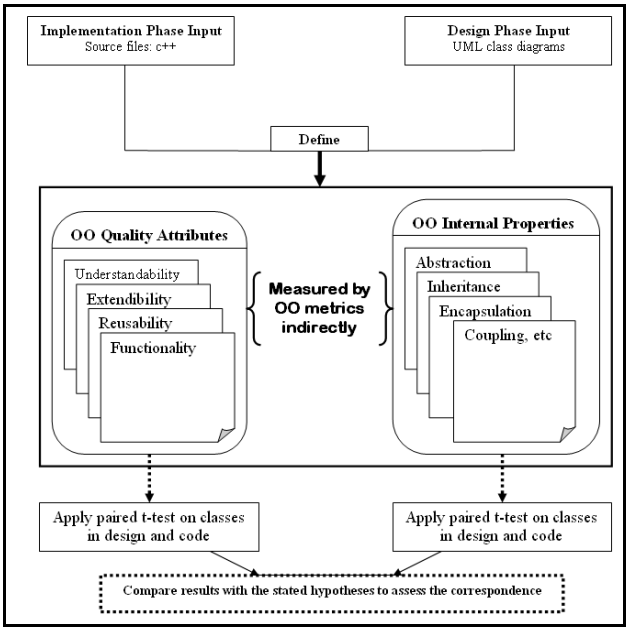

Fig. 1. The methodology of the correspondence verification.

TABLE 1
The definition of external attributes

Attribute	Definition
Reusability	Defines how we can reuse a pre-defined component in a new problem in object-oriented design and implementation with low effort.
Extendibility	Allows the incorporation of new requirements in design and implementation with existing properties.
Understandability	Measures the level of ability and easiness of learning the design and implementation, and it measures the degree of complexity.

Metrics of design diagrams are collected manually. In our methodology, to verify the correspondence between design and implementation through internal and external properties in each phase, we collect these metrics in both phases. QMOOD represents the link between the OO internal properties and OO quality attributes using the formulas that are shown in Table 3. The internal properties are measured by a set of metrics, i.e., every property has a precise mathematical formula. The metrics are used to generate a measurable link between internal and external attributes. These formulas show how object-oriented software's internal properties (in design and implementation) influence and affect the three external quality attributes [16].

IV- RESEARCH HYPOTHESES

To achieve the goals of this work, many hypotheses are stated for the internal properties and external quality attributes. The hypotheses for the internal properties were validated and discussed previously in [17]. We repeat the discussion of the internal properties in this work to connect it properly with the external quality factors. The hypotheses are divided into two groups to verify the correspondence at two levels (system and subsystem). The following list of hypotheses are validated for both levels.

The Null hypotheses of the internal properties are:

- There is no significant difference in a quality property (inheritance, polymorphism, encapsulation, abstraction, cohesion, composition, messaging, complexity, or coupling) between design and implementation at the system level.

TABLE 2
OO metrics to measure OO design and implementation properties

METRIC	Definition
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. In implementation, DIT (Depth of Inheritance Tree) metric is equivalent to ANA metric in design.
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.
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.
Cohesion Among Methods of Class (CAM)	Computes the relatedness among methods of a class based upon the parameter list of the methods. The metric is computed using the summation of the intersections of parameters of a method with the maximum independent set of all parameter types in the class. A metric closed to 1 is preferred (Range 0 to1).
Measure of Aggregation (MOA)	Counts the number of data declaration whose types are user defined classes, and it is realized by using attribute declaration.
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).
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*.
Class Interface Size (CIS)	The number of public methods in a class.
Number of Methods (NOM)	The number of all methods defined in a class. It is equivalent to WMC (Chidamber and Kemerer, 1994).

TABLE 3
Indexes for external attributes.

Index	Property/ Attribute Equation
Reusability	-0.25*Coupling + 0.25*Cohesion + 0.5 Messaging + 0.5*Design Size
Extendibility	0.5*Abstraction – 0.5*Coupling + 0.5*Inheritance + 0.5*Polymorphism
Understandability	-0.33*Abstraction+0.33*Encapsulation– 0.33*Coupling + 0.33*Cohesion – 0.33*Polymorphism - 0.33*Complexity – 0.33* Design Size

- There is no significant difference in a quality property (inheritance, polymorphism, encapsulation, abstraction, cohesion, composition, messaging, complexity, or coupling) between design and implementation at the subsystem level.

The Null hypotheses of the external attributes are:

- There is no significant difference in an external attribute (reusability, extendibility, and understandability) between design and implementation at the system level.
- There is no significant difference in an external attribute (reusability, extendibility, and understandability) between design and implementation at the subsystem level.

To accept or reject these hypotheses, we use the paired t-test [18][19]. The paired t-test assesses weather the means of two groups are statistically different from each other. The statistical tests are conducted at the 95% confidence level. A hypothesis is rejected if the statistical difference is significant (p-value ≤ 0.05). Otherwise, the hypothesis cannot be rejected. The assumption of the paired t-test is that the observations for each pair should be made under the same conditions. This is achieved in our assessment since the data were collected within the same environment. Also, we have a large number of classes (N=75), i.e. large degrees of freedom.

V- THE CASE STUDY AND DATA COLLECTION

Our model is validated on an object-oriented system—Turaya.Crypt (Secure Linux Hard-Disk Encryption). The system is based on the microkernel-based EMSCB [15] security kernel. This allows an architecture, Fig. 2, where the key critical information of a hard-disk encryption system is stored and handled in a special EMSCB service outside of Linux. This EMSCB service is the main part of the Turaya.Crypt project. This system consists of five subsystems which are listed in Table 4. Both the UML diagrams for the design phase and the source code files are available on EMSCB online repository[1]. Such system is built for reuse in Linux systems, therefore the system should have attributes such as reusability, extendibility, and understandability. Reusability of the system is critical and the design is built for reuse. The extendibility is required to extend the system into variants of other operating systems. Finally understandability is important to integrate this system with other operating systems. Fig. 2 shows the architecture of the system as provided in the software specification document.

TABLE 4
Turaya.Crypt subsystems.

Subsystem Name	Number of Classes
LinuxStub	5
HddEncServer	21
Server GUI	19
Launcher	5
LibUtils	25
Total number of classes	75

[1] http://svn.emscb.org/svn/emscb/trunk/apps/hddenc

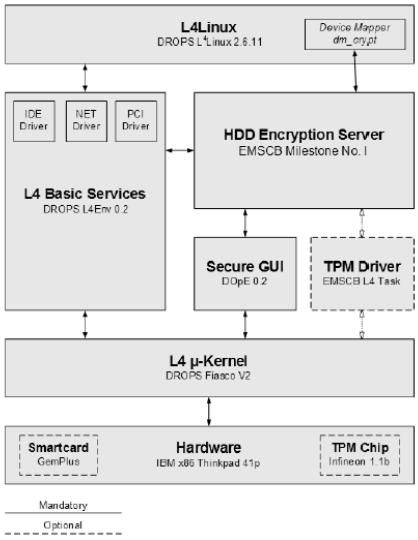

Fig. 2. The architecture of the secured Linux hard-disk encryption system

VI- RESULTS AND VERIFICATION

In this section, our verification technique is applied. The computed measurements of internal properties and external attributes are described using some descriptive statistics. Then, they are verified using the paired t-test at two levels: all classes in the system, and all classes for each subsystem.

A. Descriptive Verification

The descriptive statistics for the Turaya.Crypt system are shown in Table 5. These statistics can be used to investigate the main internal properties that the software system has in design and implementation. These descriptive statistics also help in conducting how the observed properties change from design to implementation. The differences between the group means could be used to depict how the property looks in design and implementation. For example, the difference in the inheritance property is 0 (both mean equal 0.06), i.e., there is no significant change between design and its implementation for inheritance property. We can conclude that the average of inheritance tree depth is the same in design and code. Another case, the difference in the complexity property (NOM metric) is 1.933; there is a deviation between design and its implementation for this property. NOM metric represents the complexity property, so there is a change for the number of methods in design and code classes. Fig. 3 depicts the differences between the means for each property. We can observe that for the metrics' means there is a deviation between the design classes and its counterparts in implementation for the properties: messaging, coupling, and complexity. The remaining properties show no deviations. But this descriptive investigation cannot be reliable to decide if those deviations represent significant differences or not. Fig. 4 depicts the differences for external quality attributes in design and code. We can observe that there is a deviation between means of metrics in design and code for the properties: extendibility and understandability, whereas there are no noticeable differences in reusability. This observation indicates that software reusability is sustained whereas extendibility and understandability degenerate from design.

TABLE 5
Descriptive statistics for all classes of Turaya.Crypt system.

Properties	Mean		Std. dev		Max		Min	
	Des.	Imp.	Des.	Imp.	Des.	Imp.	Des.	Imp.
Abstraction	0.15	0.24	0.36	0.49	1	2	0	0
Encapsulation	0.58	0.49	0.44	0.41	1	1	0	0
Inheritance	0.06	0.06	0.19	0.19	1	1	0	0
Messaging	5.85	6.73	5.67	7.14	35	36	0	0
Polymorphism	0.20	0.15	0.55	0.75	3	5	0	0
Composition	1.48	1.43	1.49	1.44	8	8	0	0
Coupling	1.60	2.75	1.46	3.71	8	23	0	0
Cohesion	0.42	0.47	0.37	0.36	1	1	0	0
Complexity	6.40	8.33	6.11	7.73	35	38	0	0

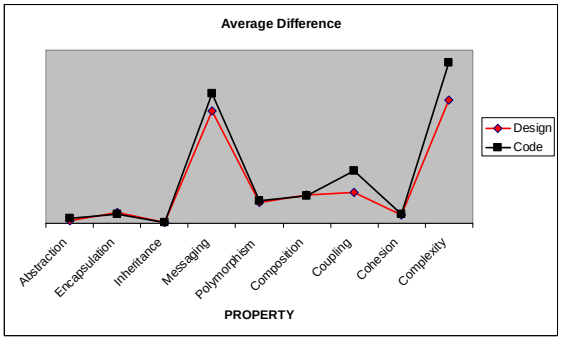

Fig. 3. The Means of internal properties.

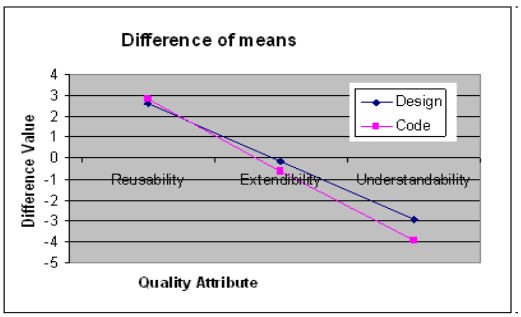

Fig. 4. The means of quality attributes.

B. Internal Properties Verification

In this section, we discuss the t-test results for the internal properties. Table 6 summarizes the paired t-test results for all internal properties. Calculations and results of all properties in

Table 6 are computed by SPSS[1]. The results of inheritance property (Pair 1) do not appear in the Table 6 because the standard error of the mean equals zero. So, the statistics cannot be computed. It indicates that there is a fully conformance between design and code for inheritance property. Therefore, inheritance structure is sustained as presumed in design.

For each null hypothesis, the letter (a) stands for system level and the letter (s) stands for subsystem level. From Table 6, the decisions for the null hypotheses stated for the internal properties are:

- ***H01***a, ***H02***a, and ***H07a*** hypotheses are accepted, i.e. there are no significant differences in inheritance, polymorphism and composition in design and implementation. We can notice that these properties measure the structure of the system and indicates a sustainability of the system shape, .i.e., high level design is sustained.

- ***H03***a, ***H04***a, ***H05a, H06a, H08a,*** and ***H09a*** hypotheses are rejected, i.e. there are significant differences in these properties in design and implementation. The properties that have deviations are involved in method implementation. Therefore, low-level design is not sustained in code.

The previous results are for all classes in design and its implementation. In the following, the calculations are repeated for the class pairs at the subsystems level. Turaya.Crypt consists of five subsystems. For each subsystem, the paired t-test is applied on its classes. Table 7 contains the results of paired samples t-test on the Server subsystem for the internal OO properties. Inheritance and polymorphism properties (i.e. pairs 1 and 2 respectively) do not appear in Table 7. In this test, statistics cannot be computed. Therefore, there is a full conformance between the design and the implementation for inheritance and polymorphism properties

Based on these results, the stated hypotheses of the internal properties of the server subsystem are:

- ***H01s, H02s, H07s,*** and ***H09s*** are accepted (i.e. there are no significant differences). These results are consistent with the whole system except the complexity is sustained in code.
- ***H03s, H04s, H05s, H06s,*** and ***H08s*** are rejected (i.e. there are significant differences). Again the low level properties are not consistent with design. Therefore, the low-level design is not sustained as presumed.

The t-tests have been applied for all other subsystems and then summarized in Table 8 for conciseness. Table 8 concentrates on the differences, for each OO property, between the design and the implementation of each subsystem. If the hypothesis is accepted, it is indicated by (✓) symbol. Otherwise, the hypothesis is rejected (i.e. sig<0.05). As shown in Table 8, LinuxStub and Launcher subsystems have the highest degree of correspondence for their internal properties in design and code. By reviewing the nature of these subsystems, each one has only five classes and structs with simple functionality; these subsystems are manageable, easy to understand, and easy to transform into code. In contrast, the Server subsystem is the core of the Turaya.Crypt system it may not be easy to achieve the conformance when the developer transforms the design diagrams into code. The Server subsystem has more coupling than other subsystems. Therefore, the Server subsystem is less reusable, extendible and understandable than other subsystems.

TABLE 6
Paired t-test's results of all internal properties.

Property	Paired Differences					Sig. (2-tailed)
	Mean Diff.	Std. Dev.	Std. Error	95% CI Lower	95% CI Upper	
Inheritance	-	-	-	-	-	-
Polymorphism	-0.067	0.379	0.044	-0.154	0.021	0.133
Encapsulation	0.095	0.333	0.038	0.019	0.172	**0.016**
Abstraction	-0.093	0.335	0.038	-0.170	-0.016	**0.019**
Coupling	-1.133	3.146	0.363	-1.859	-0.409	**0.003**
Cohesion	-0.052	0.181	.0209	-0.093	-0.011	**0.015**
Composition	0.053	0.279	.0322	-0.011	0.117	0.103
Messaging	-0.880	3.157	0.364	-1.606	-0.154	**0.018**
Complexity	-1.933	3.260	0.376	-2.685	-1.183	**0.000**

TABLE 7
The results of paired t-test applied on properties of the Server subsystem.

Property	Paired Differences					Sig. (2-tailed)
	Mean Diff.	Std. Dev.	Std. Error	95% CI Lower	95% CI Upper	
Abstraction	-0.190	0.402	0.087	-0.373	-0.007	**0.042**
Encapsulation	0.324	0.337	0.073	0.170	0.477	**0.000**
Coupling	-2.809	5.095	1.111	-5.128	-0.490	**0.020**
Cohesion	-0.106	0.217	0.047	-0.205	-0.007	**0.036**
Composition	0.190	0.512	0.111	-0.042	0.423	0.104
Messaging	-1.619	2.132	0.465	-2.589	-0.648	**0.002**
Complexity	-0.666	2.955	0.644	-2.011	0.678	0.314

TABLE 8
Results of Paired t-test applied for OO internal properties of subsystems.

Property	LinuxStub	Server	GUI	Launcher	LibUtils
Inheritance	✓	✓	✓	✓	✓
Polymorphism	✓	✓	✓	✓	✓
Encapsulation	✓			✓	✓
Abstraction	✓		✓	✓	✓
Coupling	✓		✓	✓	
Cohesion	✓			✓	✓
Composition	✓	✓	✓	✓	✓
Messaging	✓		✓	✓	
Complexity	✓	✓		✓	

[1] www.spss.com

C. External Attributes Verification

In this section, the same tests are applied for the external quality attributes. Tests will be applied on two levels as discussed in the previous section: system's classes and subsystem levels. Table 9 summarizes the paired t-test for the external quality attributes. From Table 9, the decisions for the hypotheses stated for the external quality attributes are:

- ***H12c,*** and ***H13c*** are rejected, i.e. there are significant differences of extendibility, and understandability respectively.
- ***H11c***: sig > 0.05; Accepted, i.e. there are no significant differences of reusability.

Two external attributes were affected by the changes made to the software code. From the mean difference, we can observe improvements on extendibility and understandability. The reusability is sustained in the code, whereas understandability and extendibility are not sustained. The statistical tests are also calculated for the external quality attributes of each subsystem. The tests of the hypotheses related to those attributes are also verified. Table 10 summarizes the results of applying the paired t-test on the external quality attributes in each subsystem. Again, the LinuxStub and Launcher have a full correspondence between the design and the implementation for its external quality attributes. It is observed that although the hypothesis of encapsulation is rejected in the class level, it is accepted in three of five subsystems. This leads to conclude that the subsystem in design and its package in implementation could have a high correspondence, but if at a system level they show low correspondence. The variety of property or attribute metrics in design and implementation decide if there is a significant difference between both phases. So, it is a good strategy to verify the correspondence of OO properties and attributes at the two levels. This verification discovers where the software developer has a difficulty in transforming the design artifacts into source code. The process of adapting changes, specifying none conformable units, and making changes will be easier for the future evolutions.

D. Results Discussion

Once all hypotheses are tested for correspondence, the differences in the software system between design and implementation can be determined. The results can be used to specify which parts (i.e. subsystem and classes) of the software system have differences and their effects on the software structure and quality needs. There are some properties and attributes that have an explicit evidence of the correspondence, but others have a lack of correspondence. Inheritance, polymorphism, abstraction, composition, understandability, and extendibility have the highest correspondence. The remaining properties and attributes have a relative correspondence. The most suitable and reliable way to determine why the software have a lack of correspondence in a certain property or attribute is to examine their relationships to design and implementation. The differences and their possible effects on external quality attributes are as follows:

- ***Changes in the complexity***: Software designers may have no complete conception of all methods required to achieve the functionality of the class or the object, and to achieve software needs. 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 software (or part) needs, but this change is not propagated back to the design. For Turaya.Crypt, there are 145 additional methods in the implementation (i.e. 66 public, 68 private, and 11 protected) and therefore implementation classes tend to be more complex than those in the design. The effect of adding or removing methods depends on their type; adding public methods increases the accessibility for other classes to those methods. If public methods are used by another software component (e.g. class), this causes an additional dependencies between software components, which increases the probability of coupling and decreases system understandability. The effect of new public methods is propagated to the functionality of classes and the software at all.

- ***Changes in the encapsulation:*** Introducing new (i.e. extra) private and protected attributes (i.e. larger DAM) 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 object-oriented software; the lack of correspondence for this OO property in Turaya.Crypt has significant effect on understandability. The lack of encapsulation violates the security of classes and unexpected side effects may happen, which increases the possibility of software defects.

- ***Changes in the coupling:*** Having extra public methods means extra dependencies, which increases the probability that a class has more coupled classes. Introducing additional attributes in implementation may cause extra relationships between classes or methods, especially if they are public attributes. The introduced methods and attributes in the case study actually cause these changes in coupling (i.e. larger DCC). An increase in coupling causes degenerations of the software reusability, extendibility and understandability.

- ***Changes in the cohesion:*** High cohesion is desired in a object-oriented software. In Turaya.Crypt, introducing extra related methods causes more relatedness between class methods in implementation. High coupling in the source code violates the desired convention to have low coupling (i.e. CAM is close to 1) and high cohesion. This is caused by introducing additional attributes in the implementation. A decrease in a class cohesiveness causes degenerations in software reusability and extendibility.

- ***Changes in the messaging:*** Introducing additional public methods (66 methods) causes more message passing (i.e. larger CIS) between classes through those methods' interfaces. The impact of large messaging in implementation affects the coupling between classes and their complexity. Deviations in the class interface have negative effects on software reusability.

- ***Changes in 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 becomes harder to understand and maintain.

- ***Changes in inheritance***: changing the inheritance structure has a major effect on software design. It changes the shape of software and affects software reusability.

- ***Changes in polymorphism***: The number of children represents the number of specializations and uses of a class. Changing parent classes (superclasses) also changes the polymorphism in the inheritance hierarchy. These changes could affect the degree of extendibility in a source code. Therefore, deviations in polymorphism have great effect on software structure and quality.

- ***Changes in composition***: changes on the software compositions affect an important characteristic which can change the whole shape of the software. The aggregation model is very important to be consistent between design and code to maintain the level of comprehension and understandability of the software structure.

VII- CONCLUSIONS AND FUTURE WORK

This research proposed and validated an empirical verification of consistency between design and code. A quality model, QMOOD, that connects the internal properties and external attribute was used to detect the inconsistencies in code. QMOOD uses OO metrics to directly measure software properties and indirectly measure software external attributes such as reusability, extendibility and understandability. The verification process helps in saving time, effort, and rework for future evolutions of the system. The model provides a quantitative indicator to the correspondence between low-level and high-level design and implementation. The technique uses internal properties of software (e.g. encapsulation, inheritance, and coupling) and external quality attributes (e.g. reusability, and understandability) that are measured using object-oriented metrics. The model was validated on OO open-source system developed to work under Linux system. The system shall support security functions at the kernel level, therefore the system is part of a large scale system. The studied system should have characteristics that allow the developers to reuse, extend and understand it. The reusability and extendibility are very important to use it for variations of Linux. In addition, the understandability is vital to achieve these attributes. The results of application of the proposed model gave indications of sustainability of the reusability at the system and subsystem levels, whereas the extendibility and understandability were not sustained in the code. These results help the developers to understand the nature of deviations from design and to understand the level of deviation: low-level or high-level designs. From our analysis, the core components of software tend to have high 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. Improvements can be done to enhance the efficiency of this technique. Some of these intended works are:

- In QMOOD quality model (as part of our calculations), there is only one object-oriented metric represents a certain internal property. It maybe more effective if we select more than one metric to represent the properties in design and implementation, and this requires more extensive work.
- Our verification model verifies only three high-level quality attributes. Other quality attributes can be verified in the same manner. This requires more empirical analysis for any additional attributes and their relationships to the selected internal properties and its metrics.
- The verification model has been applied on one case study because there is a limitation to access full UML software designs without a discloser agreement with the software authors. Applying the verification model on many systems provides more effective results and conclusions about the correspondence.

REFERENCES

[1]. I. Sommerville, *Software Engineering*, 8th ed. Addison Wesley, 2006.

[2]. M. Lehman and L. Belady, "Program Evolution: Processes of Software Change," London Academic Press, London, 1985.

[3]. S. Demeyer, S. Ducasse, and O. Nierstrasz. *Object-Oriented Reengineering Patterns*, Morgan Kaufmann, 2002.

[4]. 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.

[5]. G.R. Dromey. "A Model for Software Product Quality," *In IEEE Transactions of Software Engineering*, vol. 21, no. 2, 1995, pp. 146-162.

[6]. J. Bansiya and C. Davis, "A Hierarchical Model for Quality Assessment of Object-Oriented Designs," *In IEEE Transactions of Software Engineering*, vol. 28, no. 1, 2002, pp. 4-17.

[7]. 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, 2001, pp. 213–234.

[8]. 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. 1999, pp. 151–160.

[9]. G. Antoniol, B. Caprile, A. Potrich and P. Tonella, "Design-code traceability for object-oriented systems," *Annals of Software Engineering*, vol. 9, 2000, pp. 35–58.

[10]. 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, 1995, pp.18–28.

[11]. J. McCall, P. Richards and G. Walters, *Factors in Software Quality*. Nat'l Tech. Information Service. Springfield, Va. vols. 1, 2, and 3, AD/A-049-014/015/055, (1977).

[12]. ISO, International Organization for Standardization (2001). *ISO/IEC 9126-1:2001, Software engineering – Product quality, Part 1: Quality model*. JTC Information technology.

[13]. M. O'Keeffe, M. O´Cinneide, "Search-Based Refactoring for Software Maintenance," *The Journal of Systems and Software*, vol. 81 no. 4, 2008 pp. 502–516.

[14]. N. Hsueh, P. Chu, W. Chu, "A Quantitative Approach for Evaluating the Quality of Design Patterns," *The Journal of Systems and Software*, vol. 81, no. 8, 2008, pp. 1430–1439.

[15]. EMSCB -European Multilaterally Secure Computing Base (2006). *Turaya: Secure Linux Hard-Disk Encryption. Milestone No1.* Documentation is available at http://www.emscb.com/content/pages/turaya.downloads.htm. (Accessed 27/6/2009).

[16]. J. Bansiya (1997). *A Hierarchical Model for Quality Assessment of Object-Oriented Designs*. PhD Dissertation, Univ. of Alabama in Huntsville.

[17]. R. Shatnawi and A. Alzu'bi, "A Quantitative Verification of the Correspondence between Design and Implementation Artifacts Using Software Metrics," *Lecture Notes in Engineering and Computer Science:Proceedings of The International MultiConference of Engineers and Computer Scientists 2011, IMECS 2011, 16-18 March, 2011, Hong Kong,*, pp. 721-725.

[18]. C. Goulden (1956). *Methods of Statistical Analysis*, 2nd ed. New York: Wiley, pp. 50-55.

[19]. E. Pearson and M. Kendall, *Studies in the History of Statistics and Probability*, Darien, Conn: Hafner Publishing Company, 1970.