Kuwait J. Sci. **39** (2A) pp. 131-154, 2012

The impact of inheritance on the internal quality attributes of Java classes

JEHAD AL DALLAL

Department of Information Science, Kuwait University, P.O. Box 5969, Safat 13060, Kuwait
E-mail: j.aldallal@ku.edu.kw

ABSTRACT

Inheritance is a key concept in object-oriented programming. Designing and implementing the inheritance relations in a software system properly is expected to improve the quality of the system. This paper uses the class flattening approach to empirically investigate the impact of inheritance on three internal quality attributes: size, cohesion, and coupling. This paper explains how to flatten Java classes and reports the application of the process on classes of five Java open-source systems. The size, cohesion, and coupling of the original and flattened classes were measured using 28 existing metrics, and the results are compared to empirically explore the relationship between inheritance and class quality. The results show that considering inheritance relations in software development significantly improves the class quality. In addition, the results of the study demonstrate that class flattening improves some cohesion metrics that indicate class quality.

Keywords: Object-oriented design, inheritance, quality metric, internal quality attribute, external quality attribute, class flattening.

INTRODUCTION

An object-oriented programming approach is built on several key concepts, such as data encapsulation and inheritance. Implementing these concepts properly results in high-quality systems (Fayed & Laitinen, 1998). A class in an object-oriented system is the basic unit of design, and it encapsulates a set of attributes and methods. Instead of redefining the attributes and methods that are included in other classes, a class can inherit these attributes and methods and only implement its unique attributes and methods, which results in reducing code redundancy and improving code testability and maintainability (Sheldon *et al.*, 2002). Representing a class with its inherited attributes and methods is called class flattening (Beyer *et al.*, 2001).

Several class quality attributes were proposed in the literature, and they are categorized into internal and external attributes. Internal quality attributes, such as size, cohesion, and coupling, can be measured using software artifacts such as

the software code. On the other hand, external quality attributes, such as reusability, maintainability, and testability, cannot be measured using only software artifacts (Morasca, 2009).

Researchers have proposed many metrics that use different approaches to measure the internal quality class attributes and indicate the external quality attributes. In addition, they use several approaches to validate their measures and explore correlations among the internal quality attributes and between internal attributes and external quality attributes. Inheritance is a concept that may have an impact on the quality of the system. However, the relationship between inheritance and size, coupling, and cohesion attributes has not been thoroughly investigated. For example, Beyer *et al.* (2001) and Chhikara *et al.* (2011) considered a few internal-quality metrics. They used a small number of classes in the empirical study, and did not discuss the practical influences of class flattening. Several works (Bieman & Kang, 1995; Briand *et al.*, 1998; Al Dallal & Briand, 2010; Al Dallal & Briand, 2012) discussed the impact of considering the inherited attributes and methods on quality measurement theoretically, but not empirically.

In this paper, we explain how to perform class flattening for Java classes. Our explanation considers the related key concepts of object-oriented programming, such as data encapsulation, overriding, and overloading. We provide four different views of a subclass in Java systems: (1) the original class, (2) the original class with the inherited attributes, (3) the original class with the inherited methods, and (4) the original class with both the inherited attributes and the inherited methods. We applied the class flattening process to all of the subclasses in the five Java open-source systems, which resulted in the construction of three versions of each subclass, in addition to the original version. Each version considers one of the four previously mentioned views. We used the class flattening idea to explore the relationship between inheritance and size, cohesion, and coupling attributes. We applied 28 metrics to measure the quality attributes for each version of the classes considered in the empirical study. The collection of the flattened subclasses and the classes without inheritance relations represents the system that does not consider the inheritance relations in its implementation. This version is expected to have lower quality than the system with inheritance relations. Therefore, to investigate the impact of inheritance on the quality of the system considered, we measured and compared the qualities, in terms of size, cohesion, and coupling of the different versions. To explore whether the quality indication of a specific metric among the considered metrics can be improved when considering the inherited attributes and/or the inherited methods, we used the metric to obtain the quality values for each of the four versions and compared the results. The version that has the best quality indication must be adopted.

The results of the empirical study show that implementing the inheritance relations in object-oriented systems enhances the quality of the systems. In addition, the results showed that, for some of the cohesion metrics, their quality indications were significantly improved when the inherited attributes were considered in the cohesion measurement.

In summary, the major contributions of this paper are as follows:

1 - It explains how to flatten Java classes.

2 - It empirically explores the importance of addressing whether to include/exclude inherited attributes and methods in size, cohesion, and coupling measurements.

3 - It empirically investigates the impact of inheritance on internal class quality attributes.

4 - It empirically explores the impact of class flattening on improving the quality indication of some cohesion metrics.

This paper is organized as follows. Section 2 reviews related work. Section 3 explains how to flatten Java classes. Section 4 explains the design of the empirical study. Section 5 reports and discusses the empirical study results and lists validity threats to the empirical study. Finally, Section 6 concludes the paper and discusses future work.

RELATED WORK

Several metrics have been proposed to measure the size of a class. The most commonly used metrics are lines of code (LOC), number of local methods (NOM), and number of local attributes (NOA).

Class coupling refers to the extent to which the class is coupled with other classes (Briand *et al.*, 1999a). Several coupling metrics have been proposed to assess class coupling (Briand *et al.*, 1997, Chidamber & Kemerer, 1991; Chidamber & Kemerer, 1994; Li & Henry, 1993; Lee *et al.*, 1995). In this paper, we consider five coupling metrics, including RFC, MPC, DAC1, DAC2, and OCMEC, as defined in Table 1. The theoretical validation of these coupling metrics has been studied by Briand *et al.*, (1999a).

Table 1. Definitions of the considered class coupling metrics.

Class Coupling Metric	Definition/Formula
Response for a class (RFC) (Chidamber & Kemerer 1994)	The number of methods directly invoked by the methods of class A.
Message Passing Coupling (MPC) (Li & Henry 1993)	The total number of method invocations in a class.
Data Abstraction Coupling (DAC1) (Li & Henry 1993)	The number of attributes that have types of other classes.
DAC2 (Li & Henry 1993)	The number of distinct classes used as types of the attributes of the class.
OCMEC (Briand *et al.*, 1997)	The number of distinct classes that are used as types of the parameters of the methods in the class.

Class cohesion refers to the relatedness of class members (i.e. methods and attributes) (Briand *et al.*, 1998). Many class cohesion metrics have been proposed in the literature (Briand *et al.*, 1998; Briand *et al.*, 1999b; Counsell *et al.*, 2006; Al Dallal & Briand, 2010; Al Dallal & Briand, 2012; Al Dallal, 2007; Al Dallal, 2012b; Chidamber & Kemerer, 1991; Chidamber & Kemerer, 1994; Hitz & Montazeri, 1995; Bieman & Kang, 1995; Chen *et al.*, 2002; Badri & Badri, 2004; Fernandez & Pena, 2006; Li & Henry, 1993; Zhou *et al.*, 2002; Zhou *et al.*, 2004; Chae *et al.*, 2004; Etzkorn *et al.*, 2004). In this paper, we consider 20 cohesion metrics, as defined in Table 2: LCOM1, LCOM2, LCOM3, LCOM4, LCOM5, Coh, TCC, LCC, DC_D, DC_I, CC, SCOM, LSCC, CBMC, ICBMC, OL_n, PCCC, CAMC, NHD, and MMAC. The last three metrics are applicable during the high-level design phase, whereas the rest are applicable during the low-level design phase. All of the selected metrics have been theoretically and empirically studied (Briand *et al.*, 1999b; Briand *et al.*, 2001; Al Dallal, 2010; Al Dallal, 2011a; Al Dallal, 2011b; Al Dallal, 2012a; Al Dallal, 2012c).

Table 2 (Part 1). Definitions of the considered class cohesion metrics (modified from Al Dallal 2011b)

Class Cohesion Metric	Definition/Formula
Lack of Cohesion of Methods (LCOM1) (Chidamber & Kemerer, 1991)	LCOM1 = Number of pairs of methods that do not share attributes.
LCOM2 (Chidamber & Kemerer, 1994)	P = Number of pairs of methods that do not share attributes. Q = Number of pairs of methods that share attributes. $LCOM2 = \begin{cases} P - Q & if \ P - Q \geq 0 \\ 0 & otherwise \end{cases}$
LCOM3 (Li & Henry, 1993)	LCOM3 = Number of connected components in the graph that represents each method as a node and the sharing of at least one attribute as an edge.
LCOM4 (Hitz &Montazeri, 1995)	Similar to LCOM3 and additional edges are used to represent method invocations.
LCOM5 (Henderson-Sellers, 1996)	LCOM5 = $(a\text{-}kl)/(l\text{-}kl)$, where l is the number of attributes, k is the number of methods, and a is the summation of the number of distinct attributes accessed by each method in a class.
Coh (Briand *et al*. 1998)	Coh = a/kl, where a, k, and l have the same definitions above.
Tight Class Cohesion (TCC) (Bieman & Kang, 1995)	TCC = Relative number of directly connected pairs of methods, where two methods are directly connected if they are directly connected to an attribute. A method m is directly connected to an attribute when the attribute appears within the method's body or within the body of a method invoked by method m directly or transitively.
Loose Class Cohesion (LCC) (Bieman & Kang, 1995)	LCC = Relative number of directly or transitively connected pairs of methods, where two methods are transitively connected if they are directly or indirectly connected to an attribute. A method m, directly connected to an attribute j, is indirectly connected to an attribute i when there is a method directly or transitively connected to both attributes i and j.
Degree of Cohesion-Direct (DC_D) (Badri & Badri, 2004)	DC_D = Relative number of directly connected pairs of methods, where two methods are directly connected if they satisfy the condition mentioned above for TCC or if the two methods directly or transitively invoke the same method.

Table 2 (Part 2). Definitions of the considered class cohesion metrics (modified from Al Dallal 2011b)

Class Cohesion Metric	Definition/Formula
Degree of Cohesion-Indirect (DC_I) (Badri, 2004)	DC_I = Relative number of directly or transitively connected pairs of methods, where two methods are transitively connected if they satisfy the same condition mentioned above for LCC or if the two methods directly or transitively invoke the same method.
Class Cohesion (CC) (Bonja & Kidanmariam, 2006)	CC = Ratio of the sum of the similarities between all pairs of methods to the total number of pairs of methods. The similarity between methods i and j is defined as: $Similarity(i,j) = \frac{\lvert I_i \cap I_j \rvert}{I_i \cup UI_j\rvert}$, where I_i and I_j are the sets of attributes referenced by methods i and j, respectively.
Class Cohesion Metric (SCOM) (Fernandez & Pena, 2006)	SCOM = Ratio of the sum of the similarities between all pairs of methods to the total number of pairs of methods. The similarity between methods i and j is defined as: $Similarity(i,j) = \frac{\lvert I_i \cap I_j \rvert}{\min(\lvert I_i \rvert, \lvert I_j \rvert)} . \frac{\lvert I_i \cup I_j \rvert}{l}$, where l is the number of attributes
Low-level design Similarity-based Class Cohesion (LSCC) (Al Dallal & Briand, 2012)	$LSCC(C) = \begin{cases} 0 & if\ k = 0 \ \ or \ \ l = 0, \\ 1 & if \ \ k = 1, \\ \frac{\sum_{i=1}^{l} x_x(x_i - 1)}{lk(k-1)} & otherwise. \end{cases}$ Where l is the number of attributes, k is the number of methods, and x_i is the number of methods that reference attribute i.
Cohesion Based on Member Connectivity (CBMC) (Chae *et al.*, 2000)	CBMC(G) = F_c(G)xF_s(G), where F_c(G) = |M(G)|/|N(G)|, M(G) = the number of glue methods in graph G, N(G) = the number of non-special methods represented in graph G, $F_s(G) = [\sum_{i=1}^{n} CBMC(G^i)]/n$, n = the number of child nodes of G, and glue methods is the minimum set of methods for which their removal causes the class-representative graph to become disjointed.
Improved Cohesion Based on Member Connectivity (ICBMC) (Xu & Zhou 2001, 2003)	ICBMC(G) = F_c(G)xF_s(G), where F_c(G) = |M(G)|/|N(G)|, M(G) = the number of edges in the cut set of G, N(G) = the number of non-special methods represented in graph G multiplied by the number of attributes, $F_s(G) = [\sum_{i=1}^{2} ICBMC(G^i)]/2$, and cut set is the minimum set of edges such that their removal causes the graph to become disjointed.

Table 2 (Part 3). Definitions of the considered class cohesion metrics (modified from Al Dallal 2011b)

Class Cohesion Metric	Definition/Formula
OL_n (Yang, 2002)	OL_n = The average strength of the attributes, wherein the strength of an attribute is the average strength of the methods that reference that attribute. The strength of a method is initially set to 1 and is computed, in each iteration, as the average strength of the attributes that it references, where n is the number of iterations that are used to compute OL.
Path Connectivity Class Cohesion (PCCC) (Al Dallal, 2012b)	$$PCCC(C) = \begin{cases} 0 & \text{if } l = 0 \text{ and } k > 1, \\ 1 & \text{if } l > 0 \text{ and } k = 0, \\ \frac{NSP(G_c)}{NSP(FG_c)} & \text{otherwise.} \end{cases},$$ where NSP is the number of simple paths in graph G_c, FG_c is the corresponding fully connected graph, and a simple path is a path in which each node occurs once at most.
Cohesion Among Methods in a Class (CAMC) (Counsell *et al.*, 2006)	CAMC = a/kl, where l is the number of distinct parameter types, k is the number of methods, and a is the summation of the number of distinct parameter types of each method in the class. Note that this formula is applied on the model that does not include the self parameter type that is used by all of the methods.
Normalized Hamming Distance (NHD) (Counsell *et al.*, 2006)	$NHD = 1 - \frac{2}{lk(k-1)} \sum_{j=1}^{l} x_j(k - x_j)$, where k and l are as defined above for CAMC, and x_j is the number of methods that have a parameter of type j.
Method-Method through Attributes Cohesion (MMAC) (Al Dallal & Briand, 2010)	$$MMAC(C) = \begin{cases} 0 & \text{if } k = 0 \text{ or } l = 0, \\ 1 & \text{if } k = 1, \\ \frac{\sum_{i=1}^{l} x_x(x_i - 1)}{lk(k-1)} & \text{otherwise.} \end{cases},$$ where x_i is the number of methods that have a parameter or a return of type j (the type must match one of the attribute types).

JAVA CLASS FLATTENING

Inheritance is a key concept in object-oriented programming. Implementing this concept implies distributing the features among the classes that have inheritance

relations. Generally, a subclass inherits the members of its direct and transitive superclasses. The members of a class are its attributes and methods. A subclass *c1* transitively inherits a superclass *c2* when the subclass *c1* directly inherits a superclass *c3* that directly or transitively inherits the superclass *c2*. Syntactically, Java reserves the keyword *extends* to indicate the inheritance relation and allows for single inheritance only. By default, a class without a declared superclass inherits the *Object* class. Flattening a class refers to the process of representing the class as it really is, which means considering all of its inherited attributes and methods (Beyer *et al.*, 2001). In this paper, we only consider local inheritance; we do not account for the inheritance of classes that are declared outside of the system under consideration for example library classes. In addition, we consider only explicit inheritance caused by class extension using *extends* keyword, and therefore, we do not consider flattening inner classes.

Java provides four accessibility levels for class members: public, package, protected, and private. A subclass can directly access its superclass members for any accessibility level except for private. Private members in a class can only be directly accessed within the class in which they are declared. In this paper, we refer to the private attributes and methods as *invisible* attributes and methods. The rest of the attributes and methods are *visible*.

In Java, attribute overriding occurs when an attribute of a subclass has the same name as an attribute in the direct or transitive superclass, regardless of the types of the two attributes. Method overriding occurs when a method in the subclass and a method in the superclass have identical signatures (i.e. method name and types, number, and ordering of the parameters). Attribute flattening refers to the process of pulling down the attributes of the superclass to the subclass during the class flattening process. Similarly, method flattening refers to the process of pulling down the methods of the superclass to the subclass during the class flattening process. In case a chain of superclasses exists, class flattening is applied for each pair of subclass and superclass, starting from the subclass that inherits the superclass with no declared superclass. The diagram given in Figure 1 summarizes the different possibilities of the attribute or method cases and the corresponding flattening actions.

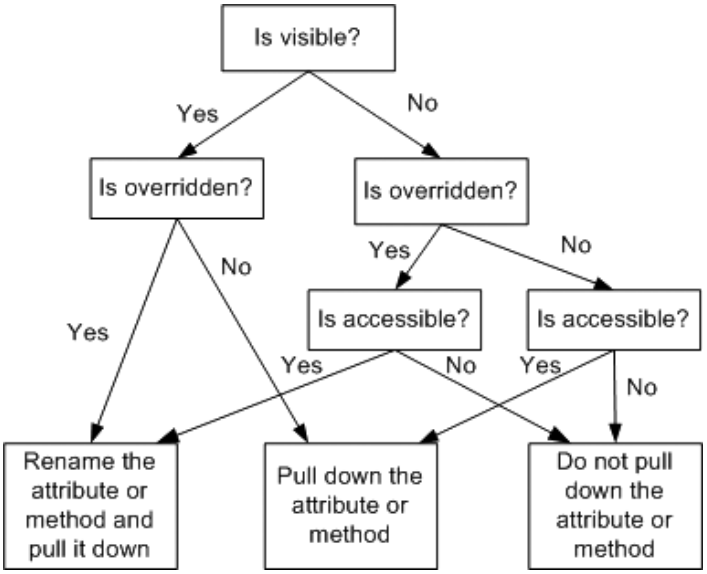

Fig.1. The different possibilities of the attribute or method cases and the corresponding flattening actions

DESIGN OF THE EMPIRICAL STUDY

The goal of the empirical study is to explore the impact of Java class flattening on the size, cohesion, and coupling quality attributes. As a result, the study indirectly investigates the effect of implementing inheritance relations in Java systems on the quality of the classes. In addition, for each of the considered size, cohesion, and coupling metrics, the empirical study investigates whether class flattening improves the ability of the metric to indicate the intended class quality aspect. Finally, the study empirically determines whether the changes in the quality values, when performing class flattening, are significant and, consequently, empirically determines whether it is important for software engineers to consider the problem of class flattening when using size, cohesion, and coupling metrics in supporting software quality decisions.

To perform this study, we selected five Java open-source software systems from different domains: Art of Illusion (2011), FreeMind (2011), Gantt Project (2011), JabRef (2011), and Openbravo (2011). Art of Illusion is a 3-D modeling, rendering, and animation studio system. FreeMind is a hierarchical editing system. Gantt Project is a project-scheduling application that features resource management, calendaring, and importing and exporting (MS Project, HTML, PDF, spreadsheets). JabRef is a graphical application for managing bibliographical databases. Finally, Openbravo is a point-of-sale application for

touch screens. The versions and sizes of the considered systems are reported in Table 3. The number of classes reported in the fourth column of Table 3 excludes the interfaces and abstract classes because most of the considered quality metrics are not applicable for interfaces and abstract classes, and that number includes all of the other classes. We selected these five open-source systems from http://sourceforge.net. The restrictions placed on the choice of these systems were that, they were (1) implemented using Java, (2) relatively large in terms of the number of classes, and (3) from different domains.

Table 3. The Java systems used in the empirical study

System	Version	LOC	Number of considered classes	Number and percentage of flattened classes
Art of Illusion	2.5	88 K	463	177 (38.2%)
FreeMind	0.8.0	64 K	378	170 (45.0%)
GanttProject	2.0.5	39 K	435	123 (28.3%)
JabRef	2.3 beta 2	48 K	540	111 (20.6%)
Openbravo	0.0.24	36 K	428	93 (21.7%)
Total	-	275 K	2,245	674 (30.0%)

We developed a tool to automate the flattening process. The tool takes the classes of a system as an input, analyzes the inheritance relations among the classes, and generates three flattened versions for each subclass in the system. The first flattened version of the subclass includes all of the directly and indirectly inherited attributes (that comply with the constraints discussed in Section 3) in addition to the code of the subclass itself. The second flattened version of the subclass includes all of the directly and indirectly inherited methods (that comply with the constraints discussed in Section 3) in addition to the code of the subclass itself. Finally, the third version of the subclass considers both the inherited attributes and methods. As a result, we considered four versions of each subclass, including the original version before flattening and the three flattened versions. The goal is to investigate the impact of each of these flattening scenarios on the size, cohesion, and coupling quality attributes. The number and percentage of subclasses (flattened classes) in each of the considered systems are reported in the last column of Table 3. This table shows that the percentages of the subclasses in the considered systems range from 20.6% to 45%. This observation indicates that the class flattening is expected to substantially affect the overall quality of the considered systems. This expectation will be confirmed with empirical evidence in the next section of this paper.

We developed our own Java tool to automate the size, cohesion, and coupling measurement process for Java classes. The tool was applied to each version of the classes considered in the five systems, which implies applying the tool on the 2,245 original classes in addition to the three flattened versions of each of the subclasses. For each of the classes considered, the tool analyzed the Java source code; extracted the required data; calculated the size, cohesion, and coupling values using the 28 metrics that were considered; and reported the results in an Excel spreadsheet. The selected metrics are listed in Tables 4, 5, and 6 and are summarized in Section 2. The selected quality metrics are widely applied, and they have been theoretically and empirically studied (Briand *et al.*, 1999a; Briand *et al.*, 1999b; Briand *et al.*, 2001; Al Dallal, 2010; Al Dallal, 2011b; Al Dallal, 2011a; Al Dallal, 2012b; Al Dallal & Morasca, 2012). It is important to note that these metrics do not represent all of the existing metrics in the literature. They are example metrics of different internal quality attributes used to investigate the general relationship between the internal quality attributes and inheritance and to show the importance of considering whether to include or exclude the inherited attributes and methods in quality measurement whenever a metric is proposed.

To investigate whether the changes in the size, cohesion, and coupling values were statistically significant, we applied the paired z-test (Hines *et al.*, 2003), a standard statistical technique applied to compare two independent sets of samples. We applied this technique because the quality values before and after performing the class flattening process are independent and the number of selected classes is sufficiently large. In our context, to apply the paired z-test for a metric, (1) the difference between the quality values before and after the change must be calculated for each class, (2) the means and standard deviation values of these differences must be obtained, and (3) the Z value must be calculated by multiplying the mean of the differences by the square root of the number of classes and dividing the result by the standard deviation of the differences. We applied the typical significance threshold (á = 0.05) to decide whether the differences between the quality values were significant. The corresponding critical Z value is 1.96. The differences were considered significant when the absolute value of Z was greater than that of the critical Z. We applied this analysis to compare the values of Scenarios 2 (attribute flattening), 3 (method flattening), and 4 (both attribute and method flattening) to the values of Scenario 1 (original code without flattening).

EMPIRICAL STUDY RESULTS AND THREATS TO VALIDITY

The empirical study considers the impact of inheritance on each of the considered internal quality attributes. The results of the empirical study are as follows.

Impact of inheritance on size metrics

Typically, additional code is added to the subclasses when they are flattened, which results in enlarging the sizes of the subclasses. Therefore, when assessing the size quality of a specific class, the software engineer must decide which type of quality to evaluate, based on the intended use of the size values. For example, if the software engineer is looking for small classes to be reused in other systems, then measuring the size of the classes without accounting for inheritance will give an incorrect size indication. This incorrect indication is, in this case, due to the need to consider the class of interest, and all of its direct and indirect superclasses for re-usability.

In our empirical study, the original number of methods in the flattened classes was 5,300, and it was substantially increased by 8,777 methods (166%) to 14,077 methods. In addition, the number of attributes in the subclasses was greatly increased by 183% from 2,424 attributes to 6,869 attributes. The flattening process caused the mean number of methods in the subclasses to increase from 7.86 to 20.89. Moreover, it caused the mean number of attributes in the subclasses to increase from 3.59 to 10.18. Figures 2 and 3 depict the distribution of the inherited methods and inherited attributes, respectively, in the flattened classes. For example, Figure 2 shows that 237 classes (35% of the flattened classes) include zero to five inherited methods. Figures 2 and 3 show that a higher percentage of the flattened classes had fewer inherited methods and attributes. The high percentage of change in the number of methods and attributes when the subclasses were flattened indicates the importance of studying the impact of inheritance on the quality values that are obtained by the size, cohesion, and coupling metrics.

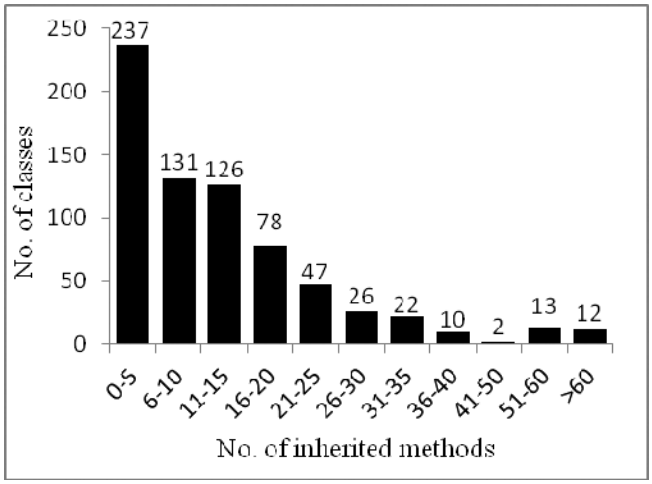

Fig. 2. The distribution of the inherited methods

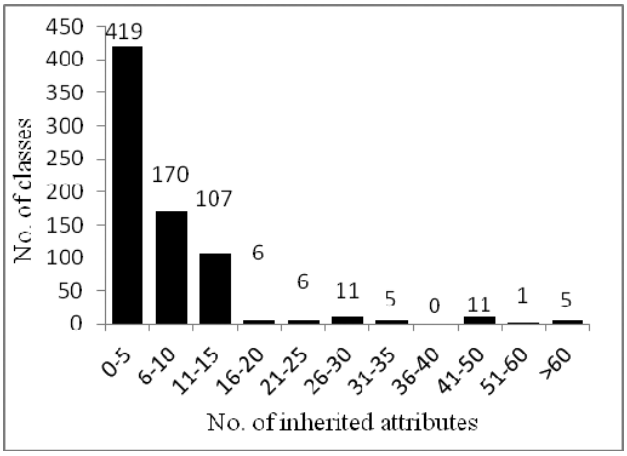

Fig. 3. The distribution of the inherited attributes

In each of the four scenarios considered, we obtained the descriptive statistics for each quality measure, including the minimum, 25% quartile, mean, median, 75% quartile, maximum value, and standard deviation. For each metric, the descriptive statistics cover all of the considered classes (i.e. the flattened and nonflattened classes). Because of space limitations, Tables 4, 5, and 6 show the mean for the quality values of the metrics considered when applied to the four scenarios. In addition, Tables 4, 5, and 6 report the percentage of the difference between the mean value of the metric in each scenario and the corresponding mean value in Scenario 1. Finally, Tables 4, 5, and 6 show the Z absolute values resulting from applying the paired z-test analysis. It is important to note that the Z value is undefined when none of the values change (e.g., attribute flattening does not change the NOM values). The p-values that resulted from the paired z-test analysis were below the threshold for all of the metrics in all scenarios for which Z values were defined, except for a few results indicated by Z values that are highlighted in boldface in Table 5. Having a p-value below its threshold and the Z absolute value above its threshold indicates that the changes in the quality values are significant when the classes were flattened.

Table 4. Paired z-test results and partial descriptive statistics for the considered size metrics

	Scenario 1: Original classes	Scenario 2: Attribute-flattened classes			Scenario 3: Method-flattened classes			Scenario 4: Both attribute- and method-flattened classes		
Metric	Mean	Mean	Percentage of Mean Diff.	|Z|	Mean	Percentage of Mean Diff.	|Z|	Mean	Percentage of Mean Diff.	|Z|
NOM	7.17	7.17	0.0%	-	11.08	54.5%	19.1	11.08	54.5%	19.1
NOA	4.20	6.18	47.1%	16.1	4.20	0.0%	-	6.18	47.1%	16.1
LOC	92.10	92.50	0.4%	3.0	120.56	30.9%	15.5	122.41	32.9%	15.8

For example, when accounting for all of the considered classes, Table 4 shows that the consideration of the inherited methods (Scenarios 3 and 4) caused the values of the NOM metric to change significantly ($|Z| > 1.96$ and the Z value is not highlighted in boldface, i.e., p-value < 0.05) and the mean value of the NOM metric to increase by 54.4%. Similarly, Table 4 shows that the consideration of the inherited attributes (Scenarios 2 and 4) caused the values of the NOA metric to change significantly and their mean to increase by 47.1%. Finally, Table 4 shows that the flattening process, applied in Scenario 4, caused the LOC values to increase significantly by 32.9%. As depicted in Table 4, the increase in the LOC values is mainly caused by the consideration of the inherited methods because the number of lines of code required to declare the attributes is typically much less than the number required for implementing the methods. Generally, the results reported in Table 4 indicate that flattening the subclasses causes the overall size quality of the system to worsen. This observation is apparent from the positive signs of the percentages of mean differences, which indicate that the mean size of the classes becomes larger (of worse quality). Therefore, the results provide empirical evidence that the inheritance feature of object-oriented programming improves the size quality attribute of the developed systems. Consequently, if the inheritance relations are not implemented well in the considered system, then the quality of the system becomes worse than the quality of the system in which the inheritance relations are implemented well.

Impact of inheritance on cohesion metrics

The cohesion of a flattened class is affected by three factors: (1) the cohesion among the members of the class before they are flattened, (2) the cohesion among the members of the superclass of the class of interest, and (3) the cohesion between the members of the class of interest and the members of its superclass. Typically, the members of the superclass are loosely cohesive to the members of the subclass; otherwise, they would have originally been placed within the subclass. Therefore, if the members of the classes are properly distributed among the classes that have inheritance relations, then the cohesion between the members of the subclasses and the members of the superclasses is expected to be relatively low, which causes the overall cohesion of the flattened class to weaken. Carefully analyzing the use of the obtained cohesion value has a great impact on the decision for whether to flatten the class of interest before measuring cohesion or not. For example, if the cohesion value is intended to indicate whether the class requires re-factoring, then measuring the cohesion of the flattened class will incorrectly indicate that the class is in need of refactoring, whereas it is indeed already re-factored into a subclass and a superclass.

In our empirical study, some of the considered metrics are undefined for some

classes. For example, TCC, LCC, DC_D, DC_I, CC, SCOM, LCOM5, and NHD are undefined for classes that consist of fewer than two methods. In addition, LCOM5, SCOM, Coh, and NHD are undefined for classes without any attributes. The cohesion metric applicability problem has been thoroughly studied by Al Dallal (2011a). To perform the same analysis on the same set of classes and, therefore, to compare the respective results in an unbiased manner, classes with undefined cohesion values must be excluded. Our empirical study did not attempt to compare the results across all of the cohesion metrics. Instead, the goal of the study was to compare the results across the four scenarios for each metric alone. Therefore, our analysis did not exclude the classes for which at least one metric is undefined. Instead, for each metric, we excluded the classes for which the cohesion value was undefined for at least one of the scenarios. For example, when analyzing the results of TCC, the classes that consisted of fewer than two methods were excluded. In this case, TCC was defined for the remainder of the classes in each of the four scenarios considered. It is important to note that some of these classes can be without any attributes; therefore, their LCOM5, SCOM, Coh, and NHD values are undefined. However, this fact does not affect the analysis of the TCC results because this paper does not compare these results with the results of LCOM5, SCOM, Coh, NHD, or any other metric. The second column of Table 5 reports the number of classes for which each of the cohesion metrics is defined. The rest of the table reports the z-test results and the partial descriptive statistics for the considered cohesion metrics.

The results reported in Table 5 lead to the following observations:

1 - Flattening the attributes causes the cohesion values to increase (the lack of cohesion values to decrease) when using metrics that are based on considering a pair of methods to be cohesive when the two methods share at least one common attribute. This observation is applicable for LCOM1, LCOM2, LCOM3, LCOM4, TCC, LCC, DC_D, and DC_I. The reason behind this increase in the cohesion values is that some methods in the subclass share a common attribute that is declared only in a direct or indirect superclass. In this case, when ignoring such an attribute, the methods are incorrectly considered as not being cohesive. Therefore, to improve the cohesion indication, these eight metrics must consider the attribute-flattened version of the class.

Table 5. Paired z-test results and partial descriptive statistics for the considered cohesion metrics

Metric	No. of Considered classes	Scenario 1: Original classes	Scenario 2: Attribute-flattened classes			Scenario 3: Method-flattened classes			Scenario 4: Both attribute- and method-flattened classes		
		Mean	Mean	Perce ntage of Mean Diff.	|Z|	Mean	Perc entage of Mean Diff.	|Z|	Mean	Perc entage of Mean Diff.	|Z|
LCOM1	1996	47.72	47.03	-1.4%	4.62	148.56	211.3%	10.48	138.72	190.7%	10.40
LCOM2	1996	35.53	34.47	-3.0%	3.83	135.53	281.4%	10.42	115.83	226.0%	10.19
LCOM3	2245	1.62	1.41	-13.0%	9.45	2.57	58.5%	10.38	1.86	15.0%	7.23
LCOM4	2245	1.62	1.40	-13.1%	9.47	2.56	58.7%	10.37	1.85	14.8%	7.15
LCOM5	1639	0.69	0.75	9.4%	14.27	0.73	6.2%	13.04	0.73	6.5%	12.27
LSCC	2245	0.30	0.28	-6.5%	10.34	0.25	-16.4%	13.29	0.25	-15.4%	12.45
CC	1639	0.30	0.29	-1.0%	2.94	0.25	-16.1%	14.31	0.26	-12.7%	12.52
SCOM	1639	0.35	0.31	-9.9%	12.25	0.29	-16.9%	14.70	0.29	-16.6%	14.34
Coh	1713	0.45	0.40	-10.9%	15.37	0.39	-12.8%	17.09	0.39	-13.2%	16.52
TCC	1746	0.34	0.35	2.6%	4.87	0.27	-21.9%	14.88	0.32	-7.4%	5.68
LCC	1746	0.41	0.42	2.9%	5.67	0.32	-21.0%	15.57	0.41	0.8%	**0.73**
DC_D	1996	0.42	0.43	1.6%	5.04	0.36	-15.1%	15.84	0.40	-4.8%	5.42
DC_I	1996	0.50	0.51	1.8%	6.06	0.42	-15.4%	16.33	0.51	1.3%	**1.64**
CBMC	1505	0.21	0.17	-17.0%	7.80	0.17	-18.9%	8.41	0.17	-19.6%	8.58
ICBMC	1505	0.19	0.16	-16.9%	7.34	0.15	-18.1%	7.72	0.15	-19.0%	7.93
OL_2	1505	0.21	0.17	-17.1%	7.82	0.17	-18.9%	8.39	0.17	-19.7%	8.56
PCCC	2245	0.49	0.47	-3.0%	6.63	0.38	-21.7%	17.07	0.38	-22.1%	16.97
CAMC	2245	0.55	0.55	0.0%	-	0.49	-11.3%	21.00	0.49	-11.3%	21.00
NHD	1940	0.51	0.51	0.0%	-	0.58	13.9%	19.80	0.58	13.9%	19.80
MMAC	2245	0.17	0.16	-2.2%	4.26	0.14	-14.7%	8.09	0.15	-11.4%	6.45

2 - Except for the improvement indicated in the previous point, as expected, the signs of the values of the percentage of mean difference for all of the metrics in all of the scenarios indicate that the cohesion of the classes becomes worse when the classes are flattened. Therefore, the results provide empirical evidence that the inheritance feature of object-oriented programming improves the cohesion quality attribute of the developed system. Consequently, if the inheritance relations are not implemented well in the system under consideration, then its quality, in terms of cohesion, becomes worse than the quality of the system in which the inheritance relations are implemented well.

3 - The attributes of the class of interest are not considered in CAMC and NHD measurements. Therefore, the cohesion values obtained using these two metrics were unaffected by attribute flattening (Scenario 2).

4 - The values of the LCC and DC_I metrics were insignificantly changed when the flattening process considers both attributes and methods (Scenario 4). These two metrics were affected by two factors. The first factor is that the

processes of flattening the attributes and methods had opposite effects on the cohesion values (i.e., flattening the attributes caused the LCC and DC_I values to increase, whereas flattening the methods caused the cohesion values to decrease). The second factor is that both LCC and DC_I metrics consider a pair of methods to be cohesive when they directly or *transitively* share a common attribute. Transitive sharing of attributes is not accounted for when using other similar metrics such as TCC and DC_D.

Impact of inheritance on coupling metrics

When a subclass is flattened, it becomes coupled not only with the classes to which the subclass is coupled, but also with the classes to which the superclasses of the class are coupled. This fact leads to the expectation that flattening subclasses causes an increase in their coupling with other classes.

Table 6 shows the impact of class flattening on the five coupling metrics considered. The table reports the corresponding z-test results and partial descriptive statistics. The results reported in Table 6 show that flattening the subclasses caused a significant increase in the values obtained from using all of the considered coupling metrics. This increase is caused by attribute flattening (Scenario 2) for DAC and DAC2 metrics and by method flattening (Scenario 3) for the other metrics. The former metrics only account for the coupling defined in the attribute declarations; whereas, the latter metrics only account for the coupling defined by the methods of the class. Increasing coupling values indicate that the quality of the code is worsening. Therefore, the results indicate that when the inheritance relations are not implemented well in the considered system (e.g., such as when the classes are flattened), the quality of the system, in terms of coupling, becomes worse than the quality of the system in which the inheritance relations are implemented well. In other words, the reported results provide empirical evidence that the inheritance feature of object-oriented programming improves the coupling quality attribute of the developed systems.

Table 6. Paired z-test results and partial descriptive statistics for the considered coupling metrics

	Scenario 1: Original classes	Scenario 2: Attribute-flattened classes			Scenario 3: Method-flattened classes			Scenario 4: Both attribute- and method-flattened classes		
Metric	Mean	Mean	Percentage of Mean Diff.	|Z|	Mean	Percentage of Mean Diff.	|Z|	Mean	Percentage of Mean Diff.	|Z|
DAC	2.85	4.20	47.3%	16.44	2.85	0.0%	-	4.20	47.3%	16.44
DAC2	1.92	2.78	44.5%	18.99	1.92	0.0%	-	2.78	44.5%	18.99
OCMEC	2.87	2.87	0.0%	-	3.69	28.4%	17.72	3.69	28.4%	17.72
MPC	39.80	39.80	0.0%	-	50.31	26.4%	11.76	50.31	26.4%	11.76
RFC	17.98	17.98	0.0%	-	22.89	27.4%	15.41	22.89	27.4%	15.41

Generally, the results support the hypothesis that implementing the inheritance relations in object-oriented programming significantly improves the overall quality of the system in terms of size, cohesion, and coupling. The applications that require localized measurements for the quality of the classes under consideration must not consider class flattening; otherwise, the obtained values would provide incorrect quality indications. In contrast, the applications that require measuring the quality of the class as it really is (i.e., considering the inherited attributes and methods) must consider class flattening before quality measurements are taken; otherwise, the values obtained will provide incorrect quality indications. The only exception to these general observations is that some of the cohesion metrics were found to indicate cohesion when the attributes were flattened better than when the inheritance is not accounted for. The obtained results provide indirect empirical evidence for the inverse relationship between the cohesion and coupling attributes. That is, classes with high cohesion have low coupling. The exploration of this relationship in accordance with the data obtained in this research is straightforward. However, such an exploration is out of the scope of this paper; therefore, it is not included here.

Threats to validity

Several internal, external, and construct threats to validity may restrict the generality and limit the interpretation of our results. Internal threats to validity include the fact, that our analysis relies on the claim that the selected systems are of good quality, and that the developers of the systems correctly designed the inheritance relations among the classes in the systems. However, our claim is supported by the fact that the selected systems are of good quality in comparison to systems that are widely known for their good quality (Al Dallal, 2012d). Therefore, it is strongly expected that the inheritance relations in the selected systems were designed well. Another internal validity threat is related to the metrics that were selected to explore the impact of inheritance on internal quality attributes. Many more metrics are proposed in the literature to measure the size, cohesion, and coupling of a class. However, it was not our goal to include all of these metrics. Instead, we selected several metrics that use different measuring approaches to determine whether or not there was evidence that implementing inheritance relations improves the internal quality of Java systems. Our results demonstrate that the selected metrics were able to achieve the intended purpose.

External threats to validity include the fact that all of the systems that were considered are open-source systems that may not be representative of all industrial domains. However, this practice is common in the research community. Although differences in design quality and reliability between open source and industrial systems have been investigated (Samoladas *et al.*, 2003; Samoladas *et al.*, 2008; Spinellis *et al.*, 2009), there is not yet a clear, general

result upon which we can rely. All of the considered systems were implemented in Java. Other programming languages, such as C++, allow for multiple inheritance relations. However, in this case, the version of the system that includes the flattened classes is expected to have a larger size, less cohesion, and higher coupling mean values than that constructed in the case of single inheritance. Therefore, in the case of a language that allows for multiple inheritance relations, it is expected that the impact of inheritance on the selected metrics will be stronger than in the case of languages with single inheritance relations. The related empirical study by Beyer et al. (2001) was based on C++ classes, and their general conclusions regarding the impact of inheritance on size, cohesion, and coupling metrics were identical to those drawn from our study, which is based on Java classes. Another external threat to validity is the fact that, although they are not artificial examples, the selected systems may not be representative, in terms of the number and sizes of classes. To generalize the results, different systems written in different programming languages, selected from different domains, and including real-life and large-scale software should be taken into account in similar large-scale evaluations.

Finally, construct threats to validity include the fact that, we relied on the existing literature to assess the theoretical validity (i.e., the construct validity) of the selected size, cohesion, and coupling metrics (i.e., to assess whether the metrics we used for size, cohesion, and coupling could truly be considered size, cohesion, and coupling measures). An in-depth examination of the theoretical validity of size, cohesion, and coupling metrics is beyond the scope of our paper.

CONCLUSIONS AND FUTURE WORK

This paper demonstrates how to flatten Java classes and explains which of the superclass attributes and methods must be considered in class flattening. It shows how to perform attribute and method flattening to obtain different views of the class that inherits other classes. The class flattening was applied to perform the empirical study that explores the impact of inheritance on internal quality attributes. Four versions (i.e. the original and the three flattened versions) of five selected systems and 28 size, coupling, and cohesion metrics were considered in the empirical study. Comparing the quality values obtained using the 28 metrics across the four versions allowed us to statistically analyze the impact of class flattening on the quality values. In general, the results indicate that class flattening caused the size, cohesion, and coupling quality attributes to weaken. This observation led to the conclusion that properly implementing inheritance relations in Java systems improves the quality of the systems. The quality results obtained were used also to investigate whether it is better to consider inherited attributes and/or methods in quality measurement.

The results show that considering the inherited attributes in cohesion measurement improved the ability of some of the metrics to indicate cohesion.

In the future, we plan to empirically study the impact of class flattening when using the internal quality attributes to indicate the external quality attributes. This study will prove or disprove our theoretically based expectations. The performed empirical study can be replicated using larger industrial systems and other internal quality attributes, such as complexity.

ACKNOWLEDGEMENT

The author would like to acknowledge the support of this work by Kuwait University Research Grant WI06/09. In addition, the author would like to thank Anas Abdin for assisting in collecting the quality results.

REFERENCES

Al Dallal, J. 2007. A design-based cohesion metric for object-oriented classes. International Journal of Computer Science and Engineering 1(3): 195-200.

Al Dallal, J. 2010. Mathematical validation of object-oriented class cohesion metrics. International Journal of Computers 4(2):45-52.

Al Dallal, J. 2011a. Improving the applicability of object-oriented class cohesion metric. Information and Software Technology 53(9): 914-928.

Al Dallal, J. 2011b. Measuring the discriminative power of object-oriented class cohesion metrics. IEEE Transactions on Software Engineering 37(6): 788-804.

Al Dallal, J. 2012a. The impact of accounting for special methods in the measurement of object-oriented class cohesion on refactoring and fault prediction activities. Journal of Systems and Software 85(5): 1042-1057.

Al Dallal, J. 2012b. Fault prediction and the discriminative powers of connectivity-based object-oriented class cohesion metrics. Information and Software Technology 54(4): 396-416.

Al Dallal, J. 2012c. Incorporating transitive relations in low-level design-based class cohesion measurement. Software: Practice and Experience, in press.

Al Dallal, J. 2012d. Constructing models for predicting extract subclass refactoring opportunities using object-oriented quality metrics, Information and Software Technology, 54(10):1125-1141.

Al Dallal, J. & Briand, L. 2010. An object-oriented high-level design-based class cohesion metric. Information and Software Technology 52(12): 1346-1361.

Al Dallal, J. & Briand, L. 2012. A Precise method-method interaction-based cohesion metric for object-oriented classes. ACM Transactions on Software Engineering and Methodology (TOSEM) 21(2): 8:1-8:34.

Al Dallal, J. & Morasca, S. 2012. Predicting object-oriented class reusability using internal quality attributes, Empirical Software Engineering, in press.

Art of Illusion: http://sourceforge.net/projects/aoi/. Accessed July 2011.

Badri, L & Badri, M. 2004. A Proposal of a new class cohesion criterion: an empirical study. Journal of Object Technology 3(4): 145-159.

Beyer, D., Lewerentz, C. & Simon, F. 2001. Impact of inheritance on metrics for size, coupling, and cohesion in object oriented. The 10th International Workshop on Software Measurement: New Approaches in Software Measurement, Berlin, pp. 1-17.

Bieman, J. & Kang, B. 1995. Cohesion and reuse in an object-oriented system. *Proceedings of the 1995 Symposium on Software reusability, Seattle, Washington, United States, pp. 259-262.*

Bonja, C. & Kidanmariam, E. 2006. Metrics for class cohesion and similarity between methods. *Proceedings of the 44th Annual ACM Southeast Regional Conference, Melbourne, Florida, pp. 91-95.*

Briand, L., Daly, J. & Wuest, J. 1998. A unified framework for cohesion measurement in object-oriented systems. Empirical Software Engineering - An International Journal 3(1): 65-117.

Briand, L., Daly, J. & Wuest, J. 1999a. A unified framework for coupling measurement in object-oriented systems. IEEE Transactions on Software Engineering 25(1): 91-121.

Briand, L., Devanbu, P. & Melo, W. 1997. An investigation into coupling measures for C++. *Proceedings of the 19th International Conference on Software Engineering, Boston, Massachusetts, United States, pp. 412-421.*

Briand, L., Morasca, S. & Basili, V.R. 1999b. Defining and validating measures for object-based high-level design. IEEE Transactions on Software Engineering 25(5): 722-743.

Briand, L., Wüst, J. & Lounis, H. 2001. Replicated Case Studies for Investigating Quality Factors in Object-Oriented Designs. Empirical Software Engineering 6(1): 11-58.

Chae, H.S., Kwon, Y.R. & Bae, D. 2000. A cohesion measure for object-oriented classes. Software-Practice & Experience 30(12): 1405-1431.

Chae, H.S., Kwon, Y.R. & Bae, D. 2004. Improving cohesion metrics for classes by considering dependent instance variables. IEEE Transactions on Software Engineering 30(11): 826-832.

Chen, Z., Zhou, Y. & Xu, B. 2002. A novel approach to measuring class cohesion based on dependence analysis. *Proceedings of the International Conference on Software Maintenance, pp. 377-384.*

Chhikara, A., Chhillar, R. & Khatri, S. 2011. Evaluating the impact of different types of inheritance on the object oriented software metrics. International Journal of Enterprise Computing and Business Systems 1(2): 1-7.

Chidamber, S. & Kemerer, C. 1991. Towards a Metrics Suite for Object-Oriented Design. Object-Oriented Programming Systems, Languages and Applications (OOPSLA), Special Issue of SIGPLAN Notices 26(10): 197-211.

Chidamber, S. & Kemerer, C. 1994. A Metrics suite for object Oriented Design. IEEE Transactions on Software Engineering 20(6): 476-493.

Counsell, S., Swift, S. & Crampton, J. 2006. The interpretation and utility of three cohesion metrics for object-oriented design. ACM Transactions on Software Engineering and Methodology (TOSEM) 15(2): 123-149.

Etzkorn, L., Gholston, S., Fortune, J., Stein, C., Utley, D., Farrington, P. & Cox, G. 2004. A comparison of cohesion metrics for object-oriented systems. Information and Software Technology 46(10): 677-687.

Fayed, M. & Laitinen, M. 1998. Transition to Object-Oriented Software Development. John Wiley & Sons, 1st edition.

Fernández, L. & Peña, R. 2006. A sensitive metric of class cohesion. International Journal of Information Theories and Applications 13(1): 82-91.

FreeMind: http://freemind.sourceforge.net/. Accessed July 2011.

GanttProject: http://sourceforge.net/projects/ganttproject/. Accessed July 2011.

Henderson-Sellers, B. 1996. Object-Oriented Metrics Measures of Complexity. Prentice-Hall.

Hines, W., Montgomery, D., Goldsman, D., & Borror, G. 2003. Probability and statistics in engineering, 4th edition. John Wiley & Sons Inc.

Hitz, M. & Montazeri, B. 1995. Measuring coupling and cohesion in object oriented systems. *Proceedings of the International Symposium on Applied Corporate Computing, pp. 25-27.*

JabRef: http://sourceforge.net/projects/jabref/. Accessed July 2011.

Lee, Y., Liang, B., Wu, S. & Wang, F. 1995. Measuring the coupling and cohesion of an object-oriented program based on information flow. *In Proceedings of International Conference on Software Quality, Maribor, Slovenia, pp. 81-90.*

Li, W. & Henry, S.M. 1993. Object-oriented metrics that predict maintainability. Journal of Systems and Software 23(2): 111-122.

Morasca, S. 2009. A probability-based approach for measuring external attributes of software artifacts. The 3rd International Symposium on Empirical Software Engineering and Measurement, Florida, USA.

Openbravo: http://sourceforge.net/projects/openbravopos. Accessed July 2011.

Samoladas, I., Bibi, S., Stamelos, I. & Bleris, G.L. 2003. Exploring the quality of free/open-source software: a case study on an ERP/CRM system. 9th Panhellenic Conference in Informatics, Thessaloniki, Greece.

Samoladas, I., Gousios, G., Spinellis, D. & Stamelos, I. 2008. The SQO-OSS quality model: measurement based open-source software evaluation. Open Source Development, Communities and Quality 275: 237-248.

Sheldon, F., Jerath, K. & Chung, H. 2002. Metrics for maintainability of class inheritance hierarchies. Journal of Software Maintenance and Evolution: Research and Practice 14(3): 147-160.

Spinellis, D., Gousios, G., Karakoidas, V., Louridas, P., Adams, P.J., Samoladas, I. Stamelos I. 2009. Evaluating the quality of open source software. Electronic Notes in Theoretical Computer Science 233: 5-28.

Xu, B. & Zhou, Y. 2001. Comments on 'A cohesion measure for object-oriented classes' by Chae, H.S., Kwon, Y.R. & Bae, D.H. (Softw. Pract. Exper. 2000, 30: 1405-1431), Software-Practice & Experience 31(14): 1381-1388.

Xu, B. & Zhou, Y. 2003. More comments on 'A cohesion measure for object-oriented classes' by Chae, H.S, Kwon, Y.R. & Bae, D.H. (Softw. Pract. Exper. 2000, 30: 1405-1431), Software-Practice & Experience 33(6): 583-588.

Yang, X. 2002. Research on Class Cohesion Measures. M.S. Thesis Department of Computer Science and Engineering, Southeast University.

Zhou, Y., Lu, J., Lu, H. & Xu, B. 2004. A comparative study of graph theory-based class cohesion measures. ACM SIGSOFT Software Engineering Notes 29(2): 13-13.

Zhou, Y., Xu, B., Zhao, J. & Yang, H. 2002. ICBMC: an improved cohesion measure for classes. *Proceedings of International Conference on Software Maintenance*, pp. 44-53.

Submitted : *1/4/2012*
Revised : *12/6/2012*
Accepted : *2/7/2012*

تأثير توريث وحدات برامج لغة الجافا على عوامل الجودة الداخلية

جهاد الدلال

قسم علوم المعلومات – جامعة الكويت – ص.ب 5969 – الصفاة – الكويت 13060
j.aldallal@ku.edu.kw

خلاصة

التوريث هو أحد المبادئ الرئيسية في البرمجة شيئية التوجه. أن تصميم وتنفيذ علاقات التوارث في النظم البرمجية بشكل صحيح يتوقع أن تحسن جودة النظم البرمجية. هذا البحث يستخدم طريقة تسطيح الوحدة لاستكشاف بشكل تجريبي تأثير التوارث على ثلاثة عوامل للجودة الداخلية وهي الحجم و التماسك والاقتران. نبين هنا كيفية تسطيح وحدات الجافا ونطبق هذه الكيفية على الوحدات في خمسة نظم جافا مفتوحة. استخدمنا 28 طريقة قياس مختلفة للعوامل الثلاثة المختارة، واستخدمنا النتائج لاستكشاف العلاقة بين التوارث وجودة الوحدة. تشير النتائج إلى أن اعتبار التوارث في بناء البرامج يحسن بشكل ذو شأن جودة الوحدات، كما أن النتائج تشير إلى أن تسطيح الوحدات يحسن بعض مقاييس التماسك للدلالة على جودة الوحدات.