British Journal of Mathematics & Computer Science
16(5): 1-12, 2016, Article no.BJMCS.25851
ISSN: 2231-0851

SCIENCEDOMAIN *international*
www.sciencedomain.org

The Metrics of Multiple Inheritance and the Reusability of Code – Java and C++

Fawzi Albalooshi[1*]

[1]*Department of Computer Science, College of IT, University of Bahrain, Kingdom of Bahrain.*

Author's contribution

The sole author designed, analyzed, interpreted and prepared the manuscript.

Article Information

DOI: 10.9734/BJMCS/2016/25851
Editor(s):
(1) Qiang Duan, Information Sciences & Technology Department, The Pennsylvania State University, USA.
Reviewers:
(1) V. Krishna Priya, Bharathiar University, India.
(2) M. Bhanu Sridhar, GVP College of Engineering for Women, Vizag, India.
Complete Peer review History: http://sciencedomain.org/review-history/14573

Original Research Article

Received: 23rd March 2016
Accepted: 4th May 2016
Published: 11th May 2016

Abstract

One of the fundamental notions in object-oriented systems is multiple inheritance which enables developers to combine concepts and increase the reusability of resulting software. Two of the widely used object-oriented languages are Java and C++ that each has its own mechanism to implement multiple inheritance. The paper investigates the difference between the two languages' implementation of this important notion. CK software metrics have been widely used to measure object-oriented software designs and implementations and are well-known in the software engineering community. In this paper they are used to assess the two implementations of an object-oriented system having multiple inheritance relationships and in particularly the reusability factor. Reusability is evaluated using a combination of the CK metrics that have been designed specifically for the purpose. The results clearly show that the Java implementation compared to C++ has increased coupling and software complexity and lacks cohesion resulting to reduced software reusability.

Keywords: Multiple inheritance; software metrics; reusability; java; C++; CK metrics.

1 Introduction

Inheritance is a fundamental mechanism that distinguishes object-oriented (OO) method of software development from more traditional ones. According to Booch [1] "inheritance is a relationship among

**Corresponding author: E-mail: falblooshi@uob.edu.bh;*

classes wherein one class shares the structure and/or behavior defined in one (single inheritance) or more (multiple inheritance) other classes". The benefits of inheritance include information sharing between a subclass and its super class(es) and software reuse that ultimately results to reduced development time and effort.

Inheritance is of two types single and multiple. Single inheritance is the ability of a class to inherit the features of a single super class with more than a single inheritance level i.e. the super class could also be a subclass inheriting from a third class and so on. Multiple inheritance is the ability of a class to inherit from more than a single class. For example, a graphical image could inherit the properties of a geometrical shape and a picture as shown in Fig. 1. Stroustrup [2,3] states that multiple inheritance allows a user to combine independent concepts represented as classes into a composite concept represented as a derived class. For example, a user might specify a new kind of window by selecting a style of window interaction from a set of available interaction classes and a style of appearance from a set of display defining classes.

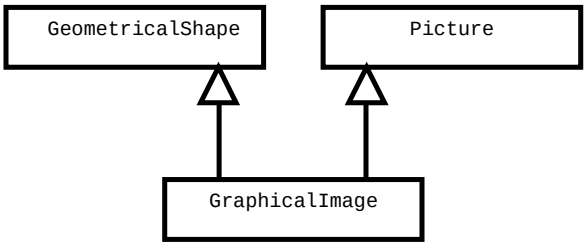

Fig. 1. Multiple inheritance

There is wide debate on the usefulness of multiple inheritance and whether the complexities associated with it justify its implementation. Though some researchers such as Stroustrup [2,3] are convinced that it can easily be implemented. He states that multiple inheritance avoids replication of information that would be experienced with single inheritance when attempting to represent combined concepts from more than one class. Booch [1] finds inheritance to be like a parachute in that it is good to have it on hand when you need it. According to Booch there are two problems associated with multiple inheritance and they are; how to deal with name collisions from super classes? And how to handle repeated inheritance? He presents solutions to these two problems. Other researchers [4] suggest that there is a real need for multiple inheritance for efficient object implementation. They justify their claim referring to the lack of multiple subtyping in the ADA 95 revision which was considered as a deficiency that was rectified in the newer version [5].

It is clear that multiple inheritance is a fundamental concept in many systems and the ability to incorporate it in system design and implementation will better structure the description of objects modeling their natural status and enabling further code reuse to that benefited from single inheritance.

2 Multiple Inheritance and C++

C++ is a widely used programming language and is considered as the most comprehensive due to its support to a variety of programming styles such as, procedural, modular, data abstraction, object-oriented, and generic programming [2,3]. It supports single and multiple inheritance a child class can inherit the properties (attributes and methods) of a single parent class and multiple parents. In cases where the parent classes define the same property the child class uses scope resolution to resolve the issue. Multiple inheritance is repeated single inheritance and in some cases the parent classes share a common ancestor property resulting the child class to have multiple copies of the same ancestor property. The compiler will be ambiguous on which version to use. This situation is referred to as the 'diamond problem'. C++ overcomes the problem with the use of virtual inheritance.

3 Multiple Inheritance and Java

Java has secured its position as the most widely used OO programming language due to many reasons including its network-centric independent platform and powerful collection of libraries of classes known as Java APIs (Application Programming Interface) [6]. Nevertheless, it has a limitation when it comes to implementing multiple inheritance which motivated researchers to think of ways to overcome.

In Java, a class inherits from its superclass and direct super-interfaces all methods that are public and protected. Classes can only support single inheritance from another class in which the child class can inherit the implementations of a super class. Java does not support multiple inheritance, however the language supports multiple inheritance of interfaces [7]. A strong reason that prevents Java from extending more than one class is to avoid issues related to multiple inheritance of attributes from more than one level which is referred at as the 'diamond problem' [8]. In which a sub-class inherits from two or more super classes that share the same ancestor resulting to more than one instance of the same ancestor state (attribute) present in the child class at the lower level of the inheritance hierarchy thus raising the issue of which instance of the ancestor state is valid and should be accessed? On the other hand, interfaces do not have state, thus do not pose such a threat, and the more recent Java 8 compiler resolves the issue of which default method a particular class uses. To overcome this shortcoming in Java, researchers investigated compromised solutions. Two of the reported work in the literature have a similar approach with minor differences are discussed in the following two paragraphs.

Thirunarayan et al. [9] investigated approximating multiple inheritance in Java by enabling a subclass C to inherit from a single superclass *A* and to implement an interface *IB* that is implemented by a class B in an effort to simulate multiple inheritance in Java. The example in Fig. 2 outlines the authors' solution to approximating multiple inheritance in Java. The class B is then incorporated as an inner class (with composition relationship) in the class C. The authors initially present three main difficulties with their solution. The first is that code reuse would be limited, but it is possible. The second is polymorphism and the third is overriding. Polymorphism could not be fully supported due to the fact that class C may not support all methods in B. Amendments to class B will require changes to the interface IB and to the class C. The third is in that overriding is a fundamental concept of inheritance but cannot easily be implemented with inner classes such as B and may require the modification of the parent class. The authors conclude that multiple inheritance can be simulated by the use of forwarding to achieve code reuse, interfaces to achieve polymorphism, and back-referencing to approximate overriding.

Tempro and Biddle [10] highlight the two main benefits of inheritance as code reuse and protocol conformance. Code defined in the parent class is reused by the child class and the child class responds to the message similarly to the parent class and can substitute it, thus achieving protocol conformance. The authors suggest that delegation can be used to simulate multiple inheritance in Java, but there are two main setbacks. The first is that in some cases the amount of code needed to achieve reuse is almost as much as the code being reused. The second is the difficulty in accessing objects imposed by the solution which renders classes to be highly coupled and less cohesive. Their solution is similar to that presented by Thirunarayan et al [9] as shown in Fig. 2 in which the class B is incorporated as an inner class within C and declaring an object b to implement it. In their paper they demonstrate that protocol conformance can be achieved by single inheritance and the use of Java's capability which allows the multiple implementation of Java interface classes. The technique they use is called 'interface-delegation' which require a child class to inherit from a single parent class and implements and delegates to as many interface classes resulting to the child class reusing all the parent classes. In addition to the two main drawbacks highlighted above the solution suffers from the following: first, protected fields and methods of the delegation object are only accessible to extending classes; second, the programmer does not have control over class libraries such as Java Core API thus creating interfaces for such classes is not possible; and third, delegation can be problematic in the presence of self-calls. The authors recommend that every class intended for reuse by inheritance (such as Java Core API library of classes) should also have a matching interface to enable such an approach in simulating multiple inheritance to be applicable.

The above two approaches in simulating multiple inheritance in Java proposed by the researchers is adopted and recommended by many Java developers as it is evident in online Java forums and posts. An approach recommended by Venners [11] uses composition (also referred at as inner class/object) instead of inheritance especially if code reuse is the goal. On the other hand, Lagorio et al. [12] completely replace inheritance with composition as presented in their framework titled FeatherJigsaw.

```
class A { // The primary class to be inherited
        public string a() { return a1();}
        protected string a1() {return "A";}}
interface IB { // Second class to be inherited declared as an interface
        public string b(IB self);
        public string b1();}
class B implements IB { // Implementation class for the interface IB
        public string b(IB self) {return self.b1(); }
        protected string b1() {return "B";}}
class C extends A implements IB { // Subclass inheriting from A and
                                  // implementing IB's interface
B b; // Innerclass as composition relationship
        public string b(IB self) {return b.b(this); }
        protected string b1() {return "C";}
        protected string a1() {return "C";}}
```

Fig. 2. Approximating multiple inheritance in java

4 Software Metrics

Metrics for Object-Oriented software has been a major research topic for more than two decades. A survey carried by Genero et al. [13] presented nine different initiatives to establish metrics for OO software such as CK [14], Li and Henry [15], MOOD [16], Lorenz and Kidd [17], Briand et al. [18], Marchesi [19], Harrison et al. [20], Bansiya et al. [21], and Genero et al. [22]. The CK [14] set of metrics has gained wide acceptance due to the fact that it was empirically tested by many researchers such as that reported in [23,24,25,26]. The originators of the CK [14] metrics realized the need for software measures or metrics to manage the software development process. They proposed a suite of six metrics for OO design and demonstrated their feasibility for process improvement. These are Weighted Methods Per Class (WMC), Depth of Inheritance Tree (DIT), Number of Children (NOC), Coupling between Object Classes (CBO), Response For a Class (RFC), and Lack of Cohesion in Methods (LCOM). In their work presented in [27] they demonstrate the use of CK metrics for managers responsible of software development efforts. Their advantage in predicting parts of the system that may be problematic as early as in the design or during implementation stages is presented. The empirical results across three financial services applications showed that metrics data can be collected on systems that were written in a variety of programming languages and on systems that were not yet coded. Another set of popular metrics was the MOOD [16] which was later extended to MOOD2 [28]. The set consists of six metrics for OO software. For the measurement of encapsulation Method Hiding Factor (MHF) and Attribute Hiding Factor (AHF) are proposed. To measure inheritance Method Inheritance Factor (MIF) and Attribute Inheritance Factor (AIF) metrics are proposed. The Coupling Factor (CF) measures coupling and the Polymorphism Factor (PF) measures polymorphism. The authors demonstrate how they can be used to measure systems. They assert that their set of metrics operate at the system level and are complementary to the CK metrics that operate at the class level.

5 Metrical Measurement and Comparison of Multiple Inheritance

5.1 The experiment sample programme

To determine the metrical difference of multiple inheritance in Java and C++ we devised a simple system as shown in Fig. 3. There are eight classes all together starting with Person, Student, and Parent classes at the

first level with each having one attribute and its associated get and set functions. At the second level three more classes are defined they are, FullTimeEmployee, FullTimeStudent, and FullTimeParent. FullTimeEmployee having an attribute and its associated get and set functions. FulTimeStudent and FullTimeParent are inheriting from two first level classes (multiple inheritance) each. Unlike the FullTimeEmployee class which declares the employee related attribute and inherits from Person the FullTimeStudent and FullTimeParent in addition to inheriting from Person each inherit from another class Student and Parent respectively. This is because the Student and Parent classes are further reused by the StudentEmployee and ParentStudentEmplyee classes, and to avoid the "diamond problem" the Student and Parent classes are independently declared (not inheriting from Person) which will otherwise occur if one or more child classes inherit from one of them and at the same time inherit from Person (or another class that already inherits from it) such as StudentEmployee and ParentStudentEmployee as shown in Fig. 3. StudentEmpolyee class sets at the third level and ParentStudentEmployee at the fourth with an attribute each and set and get functions for each of the attributes.

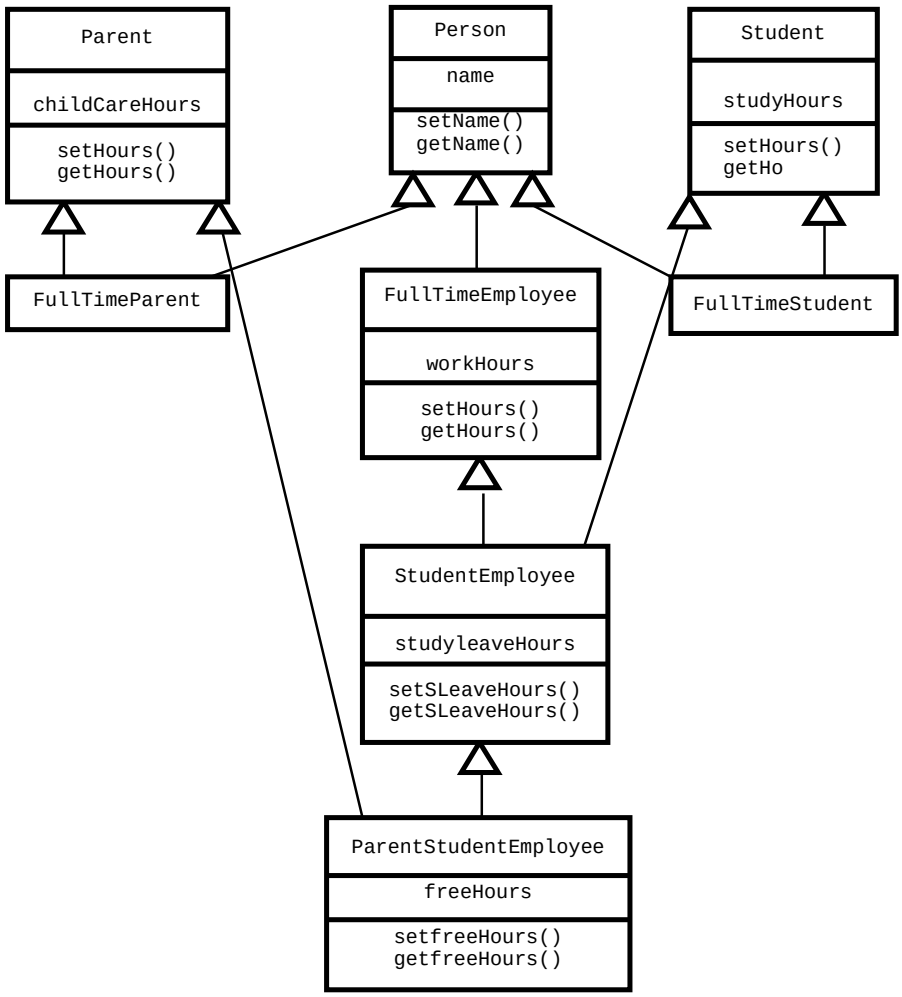

Fig. 3. C++ class diagram

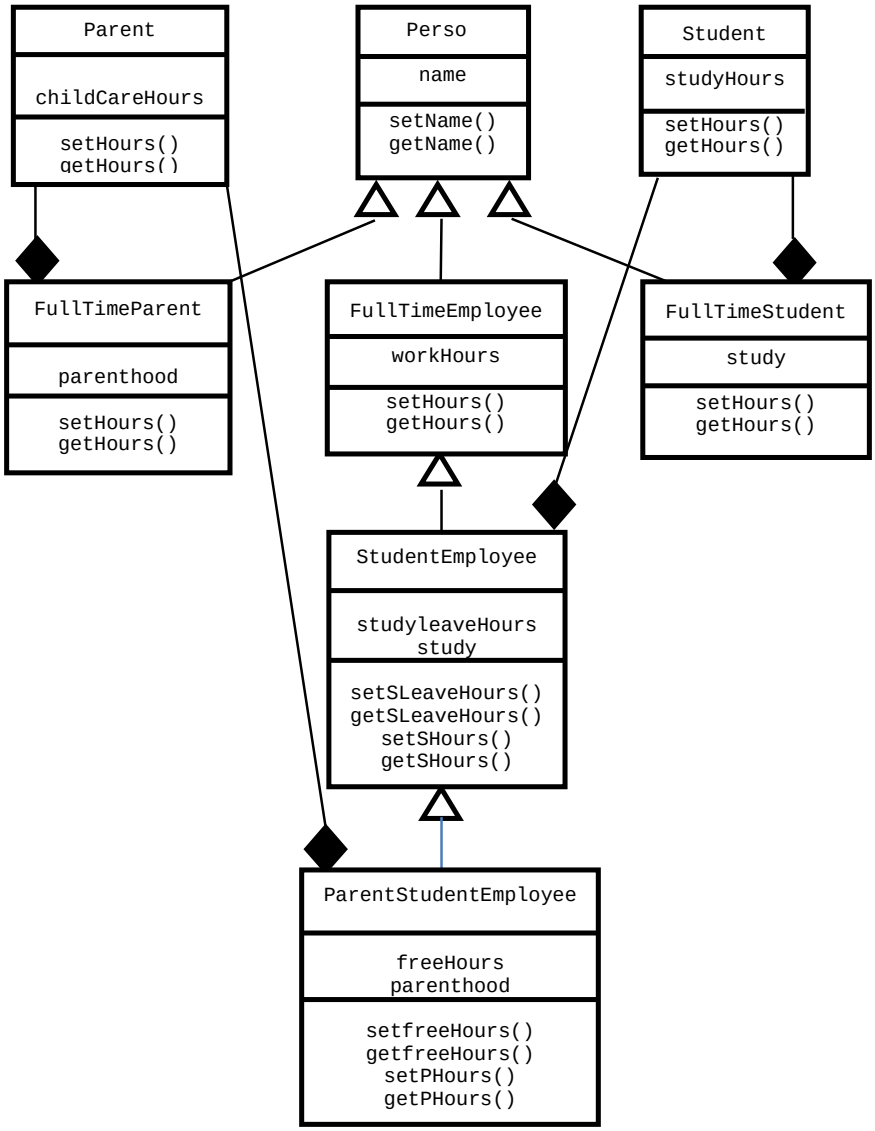

Fig. 4. Java class diagram

Fig. 4 shows the Java implementation for the same set of classes and similarly to the C++ implementation the "diamond problem" between the classes is avoided. All Java classes have the same set of attributes and set and get functions for the same classes in the C++ implementation. But to achieve multiple inheritance in the FullTimeStudent, FullTimeParent, StudentEmployee, and ParentStudentEmployee classes the inner-object approach was used. Each of these classes would inherit from one and contain an object of type the other class as shown in Fig. 4. For each inner-object an additional data member and a set and a get function had to be declared to access its attribute, thus each of the four classes had an additional attribute (inner-object) and two additional functions (for the single attribute in the inner-object) each. Using the approach recommended by Thirunarayan et al. [9] and Tempro and Biddle [10] will require the declaration of additional interface classes which for the purpose of our study will increase the number of declared classes. We therefore chose to minimize classes so that the comparison is more precise. The example system used to measure the difference in implementing multiple inheritance is simple and can easily be implemented in both

languages at the same time has four instances of multiple inheritance to enable us to precisely calculate their metrics in the two languages.

5.2 Applying the metrics

To compare the two implementations we used the six CK metrics [14], and these are, WMC, DIT, NOC, CBO, RFC, and LCOM. WMC is the number of methods defined in a class including functions, constructor and destructor. The larger the number of methods in a class the greater the impact on children this is due to the fact that the methods will be inherited by the children. Classes with large number of methods are application specific which limits their reuse. DIT is the depth of the inheritance tree calculated as the max path from root to node. Deeper trees present greater design complexities as more classes are inherited. The potential reuse of inherited methods is increased but there is a risk in predicting their behavior. NOC is the number of children (immediate subclasses) to a class. The more children a class has the more important it is and therefore must carefully be designed and tested due its high impact on others. CBO is coupling between classes' objects and is calculated as the number of classes to which each class is coupled. The more coupling the less a class becomes reusable due to its dependability on other classes. RFC is response for a class and is calculated as the number of methods in the class in addition to the number of methods called by methods in the same class. The larger the number of methods invoked as a response to a message the more complex becomes a class in addition to increasing the complexity of testing and debugging. LCOM is lack of cohesion in methods and is calculated as the count of the number of methods pairs whose similarity is 0 minus the count of methods pairs whose similarity is not 0, or more precisely (number of pair of methods that have no common attribute)-(number of pair of methods that have common attribute). Cohesiveness of a method is desirable since it promotes encapsulation. Tables 1 and 2 show the calculation of the CK set of metrics for the Java and C++ implementations respectively. The classes that inherit from more than one class are underlined. Details on how the tabulation values are calculated are presented in the following two paragraphs.

Table 1 shows the metrics calculations for the Java implementation. As explained above WMC is simply the number of methods defined in a class. It set to 2 for the classes Person, Student, Parent, FullTimeEmployee, FullTimeStudent and FullTimeParent. StudentEmployee and ParentStudentEmployee has 4 methods each. DIT for a class is calculated as the longest path from root to the class and its 0 for Person, Student and Parent classes. It is 1 for FullTimeEmployee, FullTimeStudent and FullTimeParent. It is 2 for StudentEmployee and 3 for ParentStudentEmployee. The number of children or NOC is 3 for Person, 0 for Student, Parent, FullTimeStudent, FullTimeParent and ParentStudentEmployee. Its 1 for FullTimeEmployee and StudentEmployee. CBO is coupling between classes and its 0 for Person, Student, Parent, and FullTimeEmployee. It is 1 for FullTimeStudent, FullTimeParent, StudentEmployee and ParentStudentEmployee. This is because FullTimeStudent and StudentEmployee have an inner object of type Student each. So does FullTimeParent and ParentStudentEmployee they have an inner object of type Parent each. RFC and LCOM measure for the classes is the same as WMC due the simplicity of our sample programme as it is primarily designed to investigate the difference in implementing multiple inheritance between Java and C++.

Table 2 shows the metrics calculations for the C++ implementation. WMC is set to 2 for Person, Student, Parent, FullTimeEmployee, StudentEmployee and ParentStudentEmployee. In addition to inheriting from Person, FullTimeStudent and FullTimeParent inherit methods from Student and Parent classes respectively therefore have no methods of their own and WMC for them is 0. Similarly, StudentEmployee and ParentStudentEmployee inherit from more than one class and require to declare less methods than in the Java implementation. DIT measure remained the same as the Java implementation. Its 0 for Person, Student and Parent classes; 1 for FullTimeEmployee, FullTimeStudent and FullTimeParent; 2 for StudentEmployee; and 3 for ParentStudentEmployee. The number of children or NOC for Student and Parent classes differ than the Java implementation the rest of the classes have the same measure. It is 3 for Person; 2 for Student and Parent; 1 for FullTimeEmployee and StudentEmployee; and 0 for FullTimeStudent, FullTimeParent and ParentStudentEmployee. The C++ implementation has 0 coupling resulting to a 0 CBO measure for all classes. Similarly to the Java classes RFC and LCOM measure for the C++ classes is the same as WMC, but

the classes FullTimeStudent, FullTimeParent, StudentEmployee and ParentStudentEmployee measured less than the Java implementation due to their ability to inherit from more than one class without the need for extra methods.

Table 1. CK metrics for java classes

Class	WMC	DIT	NOC	CBO	RFC	LCOM
Person	2	0	3	0	2	2
Student	2	0	0	0	2	2
Parent	2	0	0	0	2	2
FullTimeEmployee	2	1	1	0	2	2
FullTimeStudent	2	1	0	1	2	2
FullTimeParent	2	1	0	1	2	2
StudentEmployee	4	2	1	1	4	4
ParentStudentEmolyee	4	3	0	1	4	4
Total:	20	8	5	4	20	20

Table 2. CK metrics for C++ classes

Class	WMC	DIT	NOC	CBO	RFC	LCOM
Person	2	0	3	0	2	2
Student	2	0	2	0	2	2
Parent	2	0	2	0	2	2
FullTimeEmployee	2	1	1	0	2	2
FullTimeStudent	0	1	0	0	0	0
FullTimeParent	0	1	0	0	0	0
StudentEmployee	2	2	1	0	2	2
ParentStudentEmolyee	2	3	0	0	2	2
Total:	12	8	9	0	12	12

Reusability is the most fundamental benefit achieved with the use of inheritance and according to Booch [1] any artefact of software development can be reused, including code, design, scenarios, and documentation, but classes serve as the primary linguistic vehicle for reuse. Classes when properly designed and implemented can be used again (reused) in new development projects reaching up to 70% in some projects. Thus the more classes are efficiently developed to be reusable the more time and effort can be saved in new projects. Goel and Bhatia [29] investigated the measurement of the reusability of a class and in particular the use of the CK metrics for this purpose. They combined the six metrics with each other and came up with three new metrics to measure the reusability of a class. The first combined metric was the DIT and NOC. They believe that the deeper the depth of a class the more potential for reuse, thus DIT has a positive effect on reusability. Also a particular value of NOC has a positive impact on reuse. Therefore the increase in DIT in combination with NOC has a positive effect on reusability. The second combined metric is CBO and LCOM. Coupling has negative impact on reusability so does the lack of cohesion which increases complexity and has negative effect on reusability. Therefore, these two metrics have an inverse effect on reusability, the higher CBO+LCOM the less reusable is the class. The third was the combination of WMC and RFC metrics. The higher the number of methods (WMC) the more impact on children. Such classes tend to be application specific thus limiting their reuse. The higher RFC the more complex a class is thus having negative effect on its reusability. The higher WMC+RFC the less reusable a class is. Their observations on the indications of the CK metrics of a software system were formerly highlighted by the metrics originators [14]. These set of metrics' values for our system are presented in Tables 3 and 4. The classes that inherit from more than one class (thus implementing multiple inheritance) are underlined.

Table 3. CK reusability metrics for java classes

Class	DIT + NOC	CBO + LCOM	WMC + RFC
Person	3	2	4
Student	0	2	4
Parent	0	2	4
FullTimeEmployee	2	2	4
FullTimeStudent	1	3	4
FullTimeParent	1	3	4
StudentEmployee	3	5	8
ParentStudentEmolyee	3	5	8
Total:	13	24	40

Table 4. CK reusability metrics for C++ classes

Class	DIT + NOC	CBO + LCOM	WMC + RFC
Person	3	2	4
Student	2	2	4
Parent	2	2	4
FullTimeEmployee	2	2	4
FullTimeStudent	1	0	0
FullTimeParent	1	0	0
StudentEmployee	3	2	4
ParentStudentEmolyee	3	2	4
Total:	17	12	24

6 Results and Discussion

The metrics' values presented in Tables 1 and 2 show that the Java implementation has higher values for WMC, CBO, RFC, and LCOM for all four classes inheriting from two parents. The higher the value of each of these metrics the less desirable is the code as discussed in the previous section resulting to the C++ implementation to be more desirable than the Java. DIT remained unchanged in both implementations, but NOC in the C++ implementation is higher which is a desirable characteristic due to the fact that classes could have more than one child.

Analysis of the results based on the combined metrics approach proposed by Goel and Bhatia [29] clarifies the differences between the two implementations further. Tables 3 and 4 show that the C++ implementation has major advantages. The DIT metric's values for both implementations are identical, but the NOC's are different. The C++ implementation has higher NOC value by 4 counts this is because the Student and Parent classes have two children each as a result of inheritance by the FullTimeStudent, FullTimeParent, StudentEmployee and ParentStudentEmployee classes as shown in Fig. 3. Where in the Java implementation the same two classes are declared as inner-objects for the same four classes. Therefore, the C++ implementation has a positive measure over Java for this combined metric. For the second metric CBO, Table 1 shows 1 for each of the four classes inheriting from two. Due to the fact that each inherits from one and incorporates the other as an inner-object. LCOM in the Java implementation as shown in Table 1 is also higher by 8 due to the need for methods to access the data members of the inner objects in the multiple inheriting four classes, two for each. Therefore, CBO+LCOM values for the Java implementation double the C++ with 12 counts extra as shown in Tables 3 and 4. As a result the Java implementation is less reusable as discussed in the previous section. The third metric is the combination of WMC and RFC. They both have higher values in the Java implementation by 8 counts each for the same reason LCOM increased. Resulting to the two metrics having 16 counts extra in the Java implementation than in C++ as shown in Tables 3 and 4. All four multiple inheriting classes increased by 4 each in the Java implementation thus resulting for them to be considered less reusable as discussed in the previous section.

7 Conclusions

The paper measures using CK metrics the effect of implementing multiple inheritance in two widely used programming languages namely Java and C++. The case study used is especially designed to have a number of multiple inheritance relationships between its classes and at different levels. At the same time the 'diamond problem' present in multilevel multiple inheritance cases is avoided to ensure a fair comparison. There are four multiple inheritance relationships at three different levels. The CK set of metrics has gained wide acceptance by the software engineering community and was empirically tested by many researchers as discussed in section 4. As discussed in the introduction section one of the fundamental benefits of multiple inheritance is to better structure the description of objects modeling their natural status and enabling further code reuse to that benefited from single inheritance. This led as to use a combination of the CK metrics as set by Goel and Bhatia [29] to measure the reusability of the two implementations. Analysis of the results based on the combined metrics approach as discussed in section 5.3 clearly affirms that the Java implementation is less reusable. The C++ implementation has a higher NOC indicating the ability of a C++ class to become a better parent for multiple classes which is considered as positive measure of reusability. CBO and LCOM have an inverse effect on reusability the more the less reusable a class is and the java implementation doubled the C++ in this combined metric clearly suggesting that the C++ implementation is more reusable. The higher count of WMC in combination with RFC for the Java implementation further asserts that the C++ implementation is more reusable. The outcome of the experiment presented in this paper confirms the concerns raised by a number of researchers about the Java implementation (or simulation) of multiple inheritance as highlighted in section 3. Thirunarayan et al. [9] cautioned that code reuse would be limited, polymorphism could not be fully supported, and overriding cannot easily be implemented with inner classes. Tempro and Biddle [10] raised two main drawbacks. The first, is that in some cases the amount of code needed to achieve reuse is almost as much as the code being reused. The second, is the difficulty in accessing objects imposed by the solution which renders classes to be highly coupled with low cohesion. This paper provides clear evidence using software metrics that implementing (or simulating) multiple inheritance in Java will result to undesirable effects on the produced software such as, increased coupling, lack of cohesion and increased software complexity leading to major negative effects on the reusability of the produced software. Due to the fact that Java is a popular programming language in wide use, it is important that developers realize its limitations in implementing a very useful object oriented mechanism such as multiple inheritance. Developers may be encouraged to use some of the published and practiced approaches to simulate multiple inheritance, but they must be aware of the impact of such implementations on the developed software and especially on the reusability of its classes. The impact of which is compounded with the increase in the number of multiple inheritance opportunities present in the developed software.

Competing Interests

Author has declared that no competing interests exist.

References

[1] Booch G. Object-oriented analysis and design with applications. 2nd Edition (Addison-Wesley in December); 1998.

[2] Stroustrup B. Multiple inheritance for C++. The C/C++ Users Journal; 1999.

[3] Stroustrup B. The C++ Programming Language, Fourth Edition. Addison-Wesley; 2013.

[4] Ducournau R, Morandat F, Privat J. Emprical assessment of object-oriented implementations with multiple inheritance and static Typing, in OOPSLA 2009, October 25-29, 2009, Orlando, Florida, USA. ACM; 2009.

[5] Taft ST, Duff RA, Brukardt RL, Ploedereder E, Leroy P, editors. Ada 2005 reference manual: Language and standard libraries. LNCS 4348 (Springer); 2006.

[6] Flanagan D. Java in a NUTSHELL. 3rd Edition (O'Reilly & Associates, Inc.; November); 1999.

[7] Gosling J, Joy B, Steele G, Bracha G, Buckley A. The Java language specification – Java SE, 7th Edition, (Oracle America, Inc.); 2013.

[8] Oracle. Multiple Inheritance of State, Implementation, and Type; 2014.
(Accessed 15th December 2014)
Available: http://docs.oracle.com/javase/tutorial/java/landl/multipleinheritance.html

[9] Thirunarayan K, Kniesel G, Hampapuram H. Simulating multiple inheritance and generics in Java. Computer Languages, (Elsevier Science Ltd). 1999;25(4):189-210.

[10] Tempro E, Biddle R. Simulating multiple inheritance in Java. The Journal of Systems and Software, (Elsevier Science Inc.). 2000;55:87-100.

[11] Venners B. Inheritance versus composition: Which one should you choose? Java World, Inc; 2014. (Accessed 23rd July 2014)
Available:http://www.javaworld.com/article/2076814/core-java/inheritance-versus-composition--which-one-should-you-choose-.html

[12] Lagorio G, Servetto M, Zucca E. Featherweight Jigsaw – Replacing inheritance by composition in Java-like languages. Information and Computation, (Elsevier Inc.). 2012;214:86-111.

[13] Marcela G, Marion P, Coral C. A survey of metrics for UML class diagrams. Journal of Object Technology. 2005;4(9):59-92.
Available: http://www.jot.fm/issues/issue_2005_11/article1 (ETH Zurich)

[14] Chidamber SR, Kemerer CF. A metrics suite for object oriented design. IEEE Transactions on Software Engineering. 1994;20:6.

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

[16] Harrison R, Counsell SJ, Nithi RV. An evaluation of the MOOD set of object-oriented software metrics. IEEE Transactions on Software Engineering. 1998;24:6.

[17] Lorenz M, Kidd J. Object-oriented software metrics: A practical guide. (Prentice Hall, Englewood Cliffs, New Jersey); 1994.

[18] Briand L, Devanbu W, Melo W. An investigation into coupling measures for C++. 19th International Conference on Software Engineering (ICSE 97), Boston, USA. 1997;412-421.

[19] Marchesi M. OOA metrics for the united modeling language. 2nd Euromicro Conference on Software Maintenance and Reengineering. 1998;67-73.

[20] Harrison R, Counsell S, Nithi R. Coupling metrics for object-oriented design, 5th International Software Metrics Symposium Metrics. 1998;150-156.

[21] Bansiya J, Davis C. A hierarchical model for object-oriented design quality assessment. IEEE Transactions on Software Engineering. 2002;28(1):4-17.

[22] Genero M, Piattini M, Calero C. Early measures for UML class diagrams. L'Object, (Hermes Science Publications). 2001;6(4):489-515.

[23] Basili VR, Briad LC, Melo WL. A validation of object-oriented design metrics as quality indicators. IEEE Transactions Software Engineering. 1996;22:751-761.

[24] Cartwright M, Shepperd M. An empirical investigation of object-oriented software in industry. Technical Report TR 96/01, Department of Computing, Talbot Campus, Bournemouth University; 1996.

[25] Nielsen S. Personal communication, June 18; 1996.

[26] Pant Y, Henderson-Sellers B, Verner JM. Generalization of object-oriented components for reuse: Measurement of effort and size change. J. Object-Oriented Programming. 1996;9:19-41.

[27] Chidamber SR, Darcy DP, Kemerer CF. Managerial use of metrics for object-oriented software: An exploratory analysis. IEEE Transactions on Software Engineering. 1998;24:8.

[28] Abreu FB, Cuche JS. Collecting and analyzing the MOOD2 metrics. Workshop on Object-Oriented Product Metrics for Software Quality Assessment (ECOOP'98), Brussels, Belgium. 1998;258-260.

[29] Goel BM, Bhatia PK. Analysis of reusability of object-oriented system using CK metrics. International Journal of Computer Applications. 2012;60(10):32-36.

Peer-review history:
The peer review history for this paper can be accessed here (Please copy paste the total link in your browser address bar)
http://sciencedomain.org/review-history/14573