A Comparative Study on the Effect of Multiple Inheritance Mechanism in Java, C++, and Python on Complexity and Reusability of Code

Fawzi Albalooshi
Department of Computer Science
University of Bahrain
Kingdom of Bahrain

Amjad Mahmood
Department of Computer Science
University of Bahrain
Kingdom of Bahrain

***Abstract*—Two of the fundamental uses of generalization in object-oriented software development are the reusability of code and better structuring of the description of objects. Multiple inheritance is one of the important features of object-oriented methodologies which enables developers to combine concepts and increase the reusability of the resulting software. However, multiple inheritance is implemented differently in commonly used programming languages. In this paper, we use Chidamber and Kemerer (CK) metrics to study the complexity and reusability of multiple inheritance as implemented in Python, Java, and C++. The analysis of results suggests that out of the three languages investigated Python and C++ offer better reusability of software when using multiple inheritance, whereas Java has major deficiencies when implementing multiple inheritance resulting in poor structure of objects.**

***Keywords*—Reusability; complexity; Python; Java; C++; CK metrics; multiple inheritance; software metrics**

I. Introduction

Inheritance is one of the fundamental concepts of object-oriented (OO) software development. There are 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, on the other hand, 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. Stroustrup [1], [2] 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.

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 [1], [2] 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 [3] asserts that it is good to have inheritance 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 object-orientation. The ability to incorporate multiple inheritance in system design and implementation will better structure the description of objects modeling, their natural status and enabling further code reuse as compared to single inheritance.

Java, C++, and Python are three widely used OO programming languages in academia and industry. 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 known as Java APIs (Application Programming Interface). Nevertheless, Java has a limitation when it comes to implementing multiple inheritance. C++ is another widely used programming language and is considered to be the most comprehensive due to its support to a variety of programming styles such as procedural, modular, data abstraction, object-oriented and generic programming [1], [2]. It supports single and multiple inheritance in which a child class can inherit the properties of a single parent class and multiple parents. Python is a powerful object-oriented general-purpose programming language created by Guido van Rossum [6]. It has wide range of applications from Web development to scientific and mathematical computing to desktop graphical user Interfaces. It is a simple language; open source, portable across platforms, extensible and embeddable, interpreted, and has large standard libraries to solve common tasks. Similar to C++ single and multiple inheritance is supported by Python. An empirical study on the use of inheritance in Python systems was carried out by Orru *et al.* [7]. More details about the implementation of multiple inheritance in these languages are discussed in Section 2.

To the best of our knowledge there has been no studies comparing the complexity and reusability of commonly used object-oriented programming languages. In this paper, we present implementation of multiple inheritance and use CK

(Chidamber and Kemerer) [8] metrics to study the complexity and reusability of multiple inheritance as implemented in Python, Java, and C++. For this purpose, we used a sample design and code from real-life systems involving multi-level multiple inheritance and its implementation.

The rest of the paper is organized as: Section 2 presents the implementation of multiple inheritance in Java, C++, and Python. Section 3 details the complexity and reusability analysis for the three languages. It discusses software metrics and how they are applied in the measurement of the complexity and reusability followed by a discussion of the results. In Section 4 we address the current use of multiple inheritance in open source software and the impact of such practice on its complexity and reusability and Section 5 concludes the paper.

II. Multiple Inheritance Implementation in Java, C++, and Python

In Java, a class can singly inherit the properties of another class. Java does not support multiple inheritance of classes, but it supports multiple inheritance of interfaces [9]. 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 to as the 'diamond problem' [10]. This is a situation that occurs when implementing multiple inheritance in which a class inheriting from two or more super classes with a common ancestor. The super classes inherit the common ancestor method(s) and/or attribute(s). This results to their child class to inherit multiple versions of the same method(s) and/or attribute(s) (one from each super class). Thus, a conflict arises during program execution involving the child class on which version of the same inherited method/attribute to use. Java interfaces do not have a state, thus do not pose such a threat. The more recent Java 8 compiler resolves the issue of which default method a particular class uses, however this solution has its limitations. To overcome Java's shortcoming in implementing multiple inheritance, researchers investigated compromised solutions. Two of the most commonly used approaches are termed as approximation [11] and delegation [12] of multiple inheritance. C++ overcomes the diamond problem with the use of virtual inheritance. Program 1(b) shows the implementation of multiple inheritance in C++ for the Java example shown in Program 1(a). In Python the diamond problem is nicely resolved using the "Method Resolution Order" approach which is based on the "C3 superclass linearization" algorithm. Program 1(c) shows the implementation of multiple inheritance in Python.

```
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";}
}
```

Program 1(a): Approximating multiple inheritance in Java.

```
class A { // The primary class to be inherited
        public string a() { return a1();}
        protected string a1() {return "A";}
}

class B { // Implementation class for the interface IB
        public string b() {return self.b1(); }
        protected string b1() {return "B";}
}

class C extends A, B {
        protected string b1() {return "C";}
        protected string a1() {return "C";}
}
```

Program 1(b): Multiple inheritance in C++.

```
class A:
   def  a(self): (return a1();)
   def a1(): (return "A")

class B(A):
   def  b(self): (return b1();)
   def b1(): (return "B")
```

```
class C(A,B):
   def b1(): (return "C")
   def a1(): (return "C")
```

Program 1(c): Multiple inheritance in Python.

Thirunarayan *et al*. [11] proposed 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 Program 1(a) 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. This approach however suffers from a number of shortcomings such as, limited code reuse, limited support for polymorphism and difficult implementation of overriding. Polymorphism could not be fully supported due to the fact that class C may not support all methods in B. Any change in class B will require changes to the interface IB and to the class C. Overriding cannot easily be implemented with inner classes such as B and may require the modification of the parent class.

Tempro and Biddle [12] suggest that delegation can be used to simulate multiple inheritance in Java. Their solution is similar to that presented by Thirunarayan *et al.* [11] as shown in Program 1(a) in which the class B is incorporated as an inner class within C and declaring an object b to implement it. They demonstrate that protocol conformance can be achieved by single inheritance and the use of Java's capability that allows 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. There are a number of drawbacks of this approach. 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. Thirdly, protected fields and methods of the delegation object are only accessible to extending classes and, fourthly, the programmer does not have control over class libraries such as Java Core API thus creating interfaces for such classes is not possible; and finally, 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. It is important to note that the main use of interface classes in Java is to define uniform interfaces. An interface class can only have signatures of 'public' operations with no data members. When used for the purpose of inheritance all operations must be defined in the class that implements the interface and so do the attributes. This limitation results to repeated coding of the interface operations and the definition of necessary attributes whenever an interface is used. This act is the inverse advantage of code reuse the primary advantage of inheritance.

III. Complexity and Reusability Analysis of Python, Java, and C++

A number of software metrics have been proposed to analyze the complexity and reusability of object-oriented programming languages. In this section we review the metrics and then we analyze the complexity and reusability of Python, Java, and C++.

A. Software Metrics

A software metric measures or quantifies a software characteristic such as number of classes or lines of code or the number of operations, etc. They help software developers and managers to track the status of software specification or implementation [13]. Metrics for OO software have been a major research topic for more than two decades. A survey carried out by Genero *et al.* [14] presented nine different initiatives to establish metrics for OO software such as CK [8], Li and Henry [15], MOOD (Metrics for Object Oriented Design) [16], Lorenz and Kidd[17], Briand *et al.* [18], Marchesi [19], Harrison *et al.* [20], Bansiya and Davis [21], and Genero *et al.* [22]. More recently other researchers such as Amalarethinam and Hameed [23], Ibrahim *et al*. [24] and Abu Bakar [25] have also reviewed metrics for OO software. The CK [8] set of metrics has gained wide acceptance due to the fact that it was empirically tested by many researchers such as that reported in [26]-[29]. The originators of the CK [8] 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). WMC is the number of methods defined in a class including methods, constructors 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 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. The more NOC a class has the more important it is and therefore must carefully be designed and tested due its high impact on others. CBO 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 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 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. Chidamber *et al.* [30] 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 [31]. 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.

B. *The Sample Application*

To determine the exact difference in implementing multiple inheritance in Python, Java and C++, we devised a sample application as shown in Fig. 1. 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 methods. 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 methods. 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. 1. StudentEmpolyee class is at the second level and ParentStudentEmployee is in the third with an attribute each and set and get methods for each of the attributes.

Fig. 2 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 their associated (set and get) methods for the same classes in the C++ implementation. However, 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. 2. For each inner-object an additional data member and a set and a get method had to be declared to access its attribute.

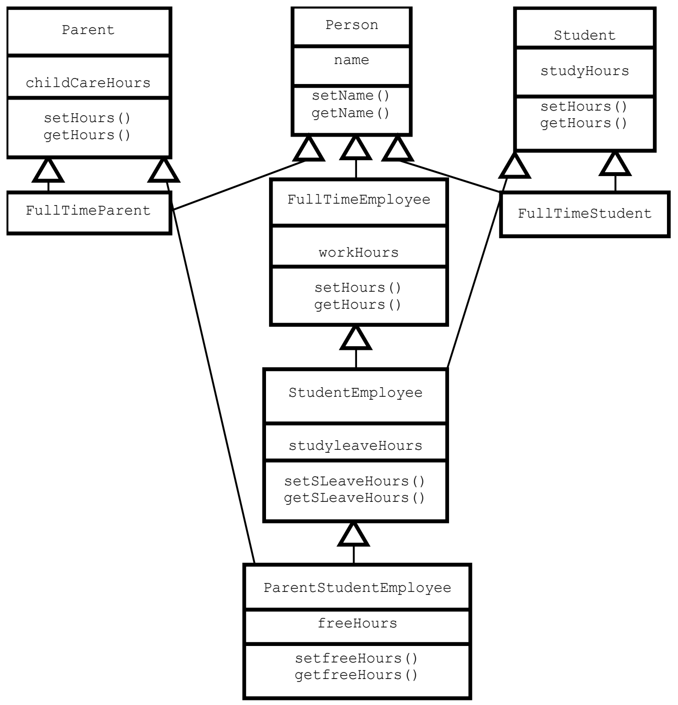

Fig. 1. C++ class diagram.

Thus each of the four classes had an additional attribute (inner-object) and two additional methods (for the single attribute in the inner-object) each Using the approach recommended by Thirunarayan *et al.* [11] and Tempro and Biddle [12] 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. Fig. 3 shows the Python class diagram for the same implementation classes presented in Fig. 1 and 2. The sample design used to measure the difference in implementing multiple inheritance can easily be implemented in the three languages and has four situations of multiple inheritance to enable us to precisely calculate the associated metrics in the different implementations.

C. *Applying the Metrics*

To compare the three implementations, we used the six CK metrics [8] as discussed in Section 3.1. Table 1 shows the values for CK set of metrics for the Python, Java and C++ implementations. The classes that inherit from more than one super class are underlined. Details on how the tabulation values are calculated are presented in the following two paragraphs:

For Java implementation WMC is 2 for the classes Person, Student, Parent, FullTimeEmployee, FullTimeStudent and FullTimeParent whereas WMC is 4 for StudentEmployee and ParentStudentEmployee. DIT is 0 for Person, Student and Parent classes. It is 1 for FullTimeEmployee, FullTimeStudent and FullTimeParent, 2 for StudentEmployee and 3 for ParentStudentEmployee. NOC is 3 for Person, 0 for Student, Parent, FullTimeStudent, FullTimeParent and ParentStudentEmployee. Its 1 for FullTimeEmployee and StudentEmployee. CBO is 0 for Person, Student, Parent, and FullTimeEmployee.

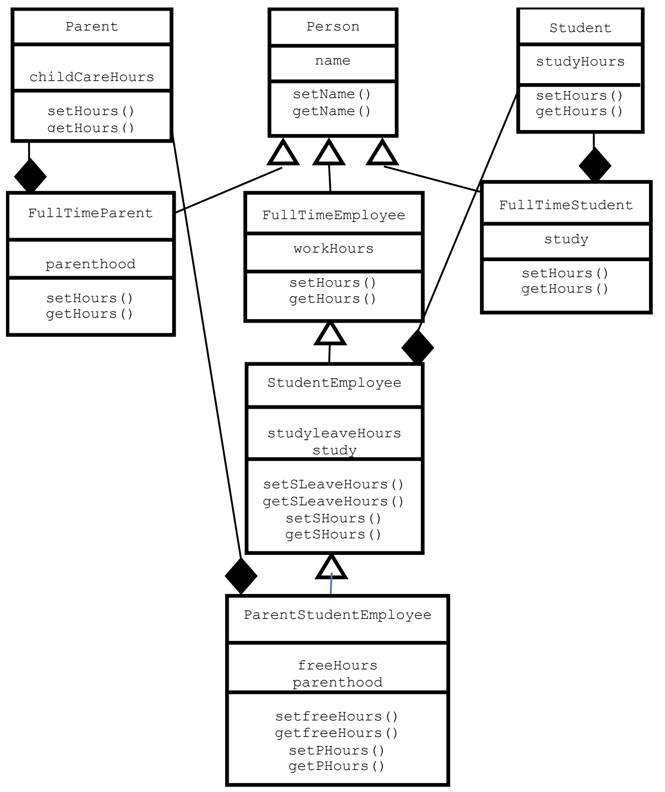

Fig. 2. Java class diagram.

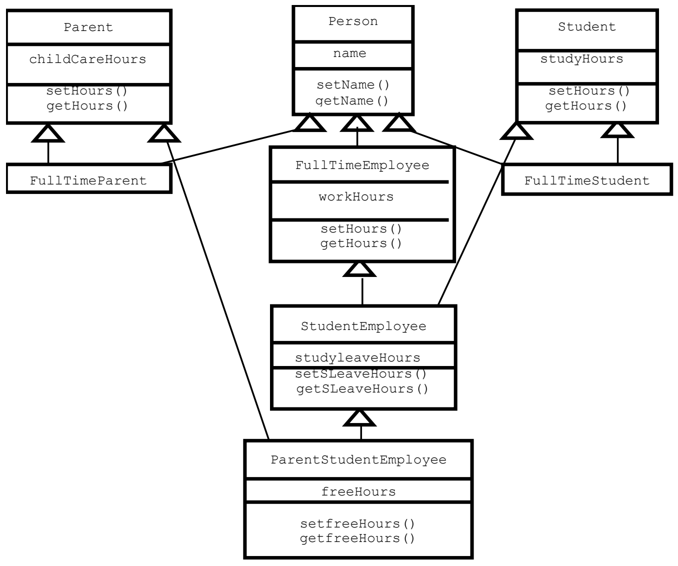

Fig. 3. Python class diagram.

It is one 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 have an inner object of type Parent each. RFC and LCOM measure for the classes is the same as WMC due to the simplicity of our sample programme.

TABLE. I. CK METRICS FOR JAVA, C++ AND PYTHON CLASSES

Class	WMC			DIT			NOC			CBO			RFC			LCOM		
	Java	C++	Python	Java	C++	Python	Java	C++	Python	Java	C++	Python	Java	C++	Python	Java	C++	Python
Person	2	2	2	0	0	0	3	3	3	0	0	0	2	2	2	2	2	2
Student	2	2	2	0	0	0	0	2	2	0	0	0	2	2	2	2	2	2
Parent	2	2	2	0	0	0	0	2	2	0	0	0	2	2	2	2	2	2
FullTimeEmployee	2	2	2	1	1	1	1	1	1	0	0	0	2	2	2	2	2	2
FullTimeStudent	2	0	0	1	1	1	0	0	0	1	0	0	2	0	0	2	0	0
FullTimeParent	2	0	0	1	1	1	0	0	0	1	0	0	2	0	0	2	0	0
StudentEmployee	4	2	2	2	2	2	1	1	1	1	0	0	4	2	2	4	2	2
ParentStudentEmolyee	4	2	2	3	3	3	0	0	0	1	0	0	4	2	2	4	2	2
Total:	20	12	12	8	8	8	5	9	9	4	0	0	20	12	12	20	12	12

As it is primarily designed to investigate the difference in implementing multiple inheritance between the three languages. For the Python and C++ implementation, WMC is 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. In the same way StudentEmployee and ParentStudentEmployee inherit from more than one class and require to define less new methods than 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. NOC for Student and Parent classes differ than that in 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 Pyhton and 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. We used the combined metrics to investigate the reusability of classes as proposed by Goel and Bhatia [32] and the results are given in Table 2.

TABLE. II. CK REUSABILITY METRICS FOR JAVA, C++ AND PYTHON CLASSES

Class	DIT+NOC			CBO+LCOM			WMC + RFC		
	Java	C++	Python	Java	C++	Python	Java	C++	Python
Person	3	3	3	2	2	2	4	4	4
Student	0	2	2	2	2	2	4	4	4
Parent	0	2	2	2	2	2	4	4	4
FullTimeEmployee	2	2	2	2	2	2	4	4	4
FullTimeStudent	1	1	1	3	0	0	4	0	0
FullTimeParent	1	1	1	3	0	0	4	0	0
StudentEmployee	3	3	3	5	2	2	8	4	4
ParentStudentEmolyee	3	3	3	5	2	2	8	4	4
Total:	13	17	17	24	12	12	40	24	24

D. Discussion

The metrics' values presented in Table 1 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 Section 3.1 resulting to the Python and C++ implementations to be more desirable than Java. DIT remained unchanged in all implementations. However, NOC in the Python and C++ implementations is higher which is a desirable characteristic due to the fact that classes could have more than one child.

Reusability is the most fundamental benefit achieved with the use of inheritance. According to Booch [3] 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. More recent researchers such as Gupta and Dashore [33] and Goel and Bhatia [32] have also appreciated the importance of OO software reusability. The first developed a tool to measure reusability and the latter investigated the measurement of the reusability of a class and in particular the use of the CK metrics for this purpose. Goel and Bhatia [32] 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 is (WMC) the more is the 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 [8]. These set of metrics' values for our implementations are presented in Table 2. The classes that inherit from more than one class (thus implementing multiple inheritance) are underlined.

Analysis of the results based on the combined metrics approach proposed by Goel and Bhatia [32] clarifies the differences between the three implementations further. Table 2 shows that the Python and C++ implementations have major advantages. The DIT metric's values for all implementations are identical, but the NOC's are different. The Python and C++ implementations have 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. 1 and 3. In the Java implementation the same two classes are declared as inner-objects for the same four classes. Therefore, the Python and C++ implementations have a positive measure over Java for this combined metric. For the second combined metric, the CBO value for the Java implementation is 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 inner object. Therefore, CBO+LCOM values for the Java implementation double the Python and C++ by 12 counts as shown in Table 2. 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 Python and C++ is shown in Table 2. 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.

IV. Multiple Inheritance in Open Source Software

In this section we present our investigation of the use of multiple inheritance in big open source software. For this purpose, we selected JRE and Eclipse, two of the largest open source systems that were analyzed by Tempero *et al.* [34]. The authors found substantial unnecessary overriding present in all applications. Their results assert that in the applications they examined the number of classes that inherit something in addition to number of classes that override something are roughly equivalent to the number of classes in the application as a whole. Two of the biggest applications they presented were JRE and Eclipse. Their empirical study was based on the Qualitas Corpus [35] open source code repository.

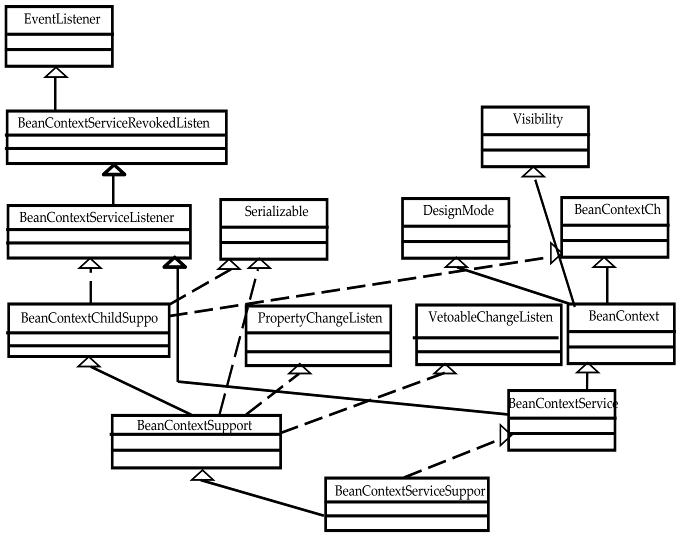

Fig. 4. JRE class diagram.

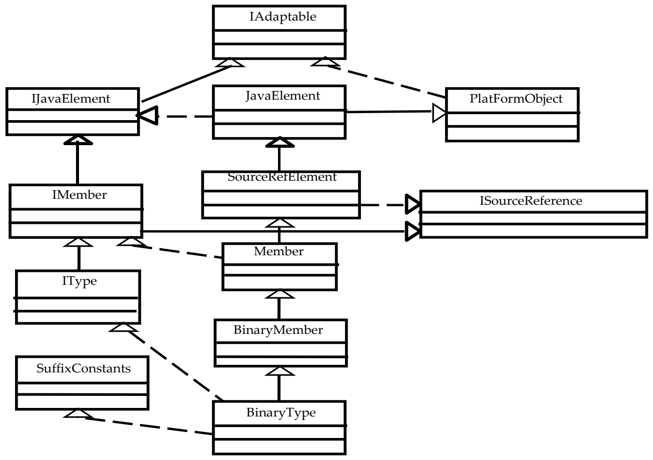

Fig. 5. Eclipse example design.

For our study, the source code of both the applications was downloaded from SourceForge [36]. We used StarUML [37] to reverse engineer the code to UML designs. Fig. 4 shows the UML design reverse engineered from parts of Java code for the Java beans context from JRE. Fig. 5 shows the UML design reverse engineered from parts of open source code for the Eclipse JDT. Both applications follow a similar approach to implement multiple inheritance in which a class inherits from another and delegates from one or more interfaces to simulate multiple inheritance. The process of delegation requires the inheriting class to implement the interface class(es).

A critical analysis of both implementations shows high DIT reaching to six levels in Fig. 5 with a low number of direct children-NOC for each subclass. Furthermore, delegation necessitates interface class operations' definition in inheriting/implementation classes which increases coupling and reduces the class's cohesion thus increasing CBO, LCOM, and WMC metrics as we discuss in Section 3.4. The increase in WMC has a relative impact on the increase in RFC as shown in our experimental results in Table 1. Both implementations show high DIT thus an increase in design complexity. The object-oriented programming community does not recommend more than three levels due to the complexity it invites when maintaining the code. On the contrary, high number of children breadthwise is recommended and increases the importance of the parent class, but due to the inheritance limitation imposed by Java NOC is low resulting to a negative impact on reuse. The increase in CBO and LCOM further negatively effects reuse so does the increase in WMC and RFC. Both implementations further suffer from the diamond problem. The first implementation is in BeanContextSupport and BeanContextServicesSupport classes and the second is in the JavaElement, Member and BinaryType classes.

To further demonstrate the difference between Java and C++/Phython implementations of multiple inheritance we developed the class diagram shown in Fig. 6 as a possible implementation in C++ of the design shown in Fig. 5 without reducing the number of classes as they are part of a bigger system. The new design improves the original implementation in a number of ways. Firstly, the diamond problem is not present anymore and the number of relationships dropped from 15 to 11.

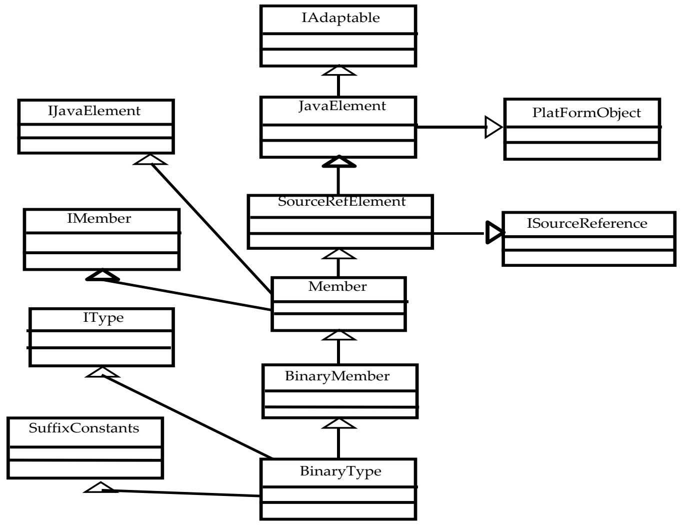

Fig. 6. Redesign of the Eclipse example.

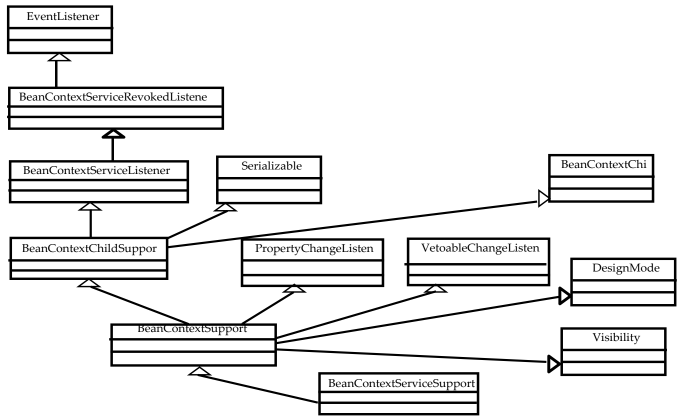

Fig. 7. Redesign of the JRE example.

Secondly, the operations of interface classes in Fig. 5 and associated attributes in the implementation classes need to be defined only once in Fig. 6. In affect coupling is reduced and the classes are more cohesive thus, CBO and LCOM metrics values are reduced. Furthermore, the number of operations defined in the classes is reduced because they are now fully inherited (with their implementations) which greatly reduces the values of WMC and RFC metrics and finally DIT is optimized. These major improvements to the partial code will reduce its complexity and increase its reusability indicating that much more benefits is gained if the whole application is to be redesigned for it to be implemented in a language such as C++ or Python. An analysis of the redesign of the JRE example shown in Fig. 7 presents a similar picture. The diamond problem present in a number of classes in the design in Fig. 4 disappeared; the number of classes and relationships between them is less; and operations and attributes need to be defined only once resulting to less values for CBO, LCOM, WMC and RFC for the design.

V. Conclusion

The paper presents an important issue faced by OO software developers. Using Java, Python and C++, we presented the effect of the programming language on the resultant software. The case discussed in this paper in multiple inheritance for which a program was designed to determine the difference in the implementations. For a fair comparison, the diamond problem was avoided in order not to advantage the C++ and Python implementations. We used the CK metrics to measure the complexity and the combined metrics proposed by Goel and Bhatia to measure reusability. The results clearly affirm that the Java implementation is less reusable. The Python and C++ implementations have a higher NOC indicating the ability of the classes to become better parents for multiple classes, which is considered as positive measure of reusability. CBO and LCOM in the java implementation doubled the Python and C++ clearly suggesting that the latter two implementations have better reusability. The higher count of WMC in combination with RFC for the Java implementation further asserts that the Python and C++ implementations are 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. We also demonstrated the negative effect of the use of simulated multiple inheritance in big open source industrial software.

References

[1] Stroustrup, B. Multiple inheritance for C++. The C/C++ Users Journal, 1999.

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

[3] Booch, G. Object-oriented analysis and design with applications, 2nd Edition, Addison-Wesley, 1998.

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

[5] Taft, S. T.; Duff, R. A.; Brukardt, R. L.; Ploedereder, E.; Leroy, P. Eds. Ada 2005 reference manual: language and standard libraries. In LNCS 4348 ,Springer, 2006.

[6] Lutz, M. Learning Python, 5th Edition. Published by O'Reilly Media, Inc. (June), CA, USA, 2013.

[7] Orru M.; Tempro E.; Marchesi M. (2015) How Do Python Programs Use Inheritance? In A Replication Study. Software Engineering Conference (APSEC), Asia-Pacific. 2015, 1-4 Dec.

[8] Chidamber, S. R.; Kemerer, C. F. A Metrics Suite for Object Oriented Design, IEEE Transactions on Software Engineering, 1994, Vol 20, No. 6.

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

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

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

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

[13] Vogelsang, A.; Fehnker, A.; Huuck, R.; Reif, W. Software Metrics in Static Program Analysis. In the International Conference on Formal Engineering Methods. Springer Berlin Heidelberg, 2010, pp 485-500.

[14] Genero, M.; Piattini, M.; Calero, C. A Survey of Metrics for UML Class Diagrams, Journal of Object Technology, 2005, Vol. 4., No. 9, pp 59-92, http://www.jot.fm/issues/issue_2005_11/article1, (ETH Zurich).

[15] Li, W.; Henry, S. Object-Oriented Metrics that Predict Maintainability, Journal of Systems and Software, 1993, Vol. 23, No. 2, pp. 111-122.

[16] Harrison, R.; Counsell, S. J.; Nithi, R. V. An Evaluation of the MOOD Set of Object-Oriented Software Metrics, IEEE Transactions on Software Engineering, 1998, Vol. 24, No. 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++. In 19th International Conference on Software Engineering (ICSE 97), Boston, USA, 1997, pp. 412-421.

[19] Marchesi, M. OOA Metrics for the United Modeling Language. In 2nd Euromicro Conference on Software Maintenance and Reengineering, 1998, pp. 67-73.

[20] Harrison, R.; Counsell, S.; R. Nithi, R. Coupling Metrics for Object-Oriented Design. In 5th International Software Metrics Symposium Metrics, 1998, pp. 150-156.

[21] Bansiya, J.; Davis, C. A. Hierarchical Model for Object-Oriented Design Quality Assessment, IEEE Transactions on Software Engineering, 2002, Vol. 28, No. 1, pp. 4-17.

[22] Genero, M.; Piattini, M.; Calero, C. Early Measures for UML Class Diagrams, L'Object, 2001, Vol. 6, No. 4, (Hermes Science Publications), pp. 489-515.

[23] Amalarethinam D. I. George; Hameed P.H. Maitheen Shahul. Analysis of Object Oriented Metrics on a Java Application. International Journal of Computer Applications, 2015, Volume 123, Number 1.

[24] Ibrahim Ahmed Abd ElHalim; Kamal Amr; Hassan Hesham. Object Oriented Metrics and Quality Attributes: A Survey. In INFOS'16, Giza, Egypt, May 09-11 2016, published by ACM.

[25] Abu Bakar N. S. A. The Analysis of Object-Oriented Metrics in C++ Programs. Lecture Notes on Software Engineering, 2016, Volume 4, Number 1.

[26] Basili, V. R.; Briad, L. C.; Melo, W. L. A Validation of Object-Oriented Design Metrics as Quality Indicators, IEEE Transactions Software Engineering, 1996, vol. 22, pp. 751-761.

[27] Cartwright, M.; Shepperd, M. An Empirical Investigation of Object-Oriented Software System, IEEE Transactions on Software Engineering, 2000, Volume 26, Issue 8, Page 786-796.

[28] Pant, Y.; Henderson-Sellers, B.; Verner, J. M. Generalization of Object-Oriented Components for Reuse: Measurement of Effort and Size Change, J. Object-Oriented Programming, 1996, vol. 9, pp. 19-41.

[29] Hirama Kechi. Software Complexity Analysis Based on Shannon Entropy Theory and C&K Metrics. IEEE Latin America Transactions, 2016, Volume 14, Issue 5.

[30] Chidamber, S. R.; Darcy, D. P.; Kemerer, C. F. Managerial Use of Metrics for Object-Oriented Software: An Exploratory Analysis, IEEE Transactions on Software Engineering, 1998, Vol. 24, No. 8.

[31] Abreu, F. B.,; Cuche, J. S. Collecting and Analyzing the MOOD2 Metrics. In workshop on Object-Oriented Product Metrics for Software Quality Assessment (ECOOP'98), Brussels, Belgium, pages 258-260, July 21st 1998.

[32] Goel, B. M.; Bhatia, P. K Analysis of Reusability of Object-Oriented System using CK Metrics. International Journal of Computer Applications, 2012, Vol. 60, No. 10, pp 32-36.

[33] Gupta A.; Dashore P. An Approach to Analyse Software Reusability of Object Oriented Code. International Journal of Research in Science & Engineering, 2017, Volume 3, Issue 1.

[34] Tempero, E.; Counsell, S.; Noble, J. An Empirical Study of Overriding in Open Source Java. In ACSC'10 proceedings of the 33rd Australasian Computer Science Conference, Brisbane, Australia, January 2010, pp 3-12.

[35] Tempero, E.; Anslow, C.; Dietrich, J. The Qualitas Corpus: A Curated Collection of Java Code for Empirical Studies. In Software Engineering Conference (APSEC), 2010 17th Asia Pacific, January 2011, pp 336-345.

[36] Sourceforge. Available online: https://sourceforge.net/ (12/4/2017).

[37] StarUml 5.0. Available online: http://staruml.software.informer.com/5.0/ (12/4/2017)