

1

2 **Original Research Article**

3 **Software Design Concerns Associated with**

4 **Simulating Multiple Inheritance in Java for**

5 **Implementation Purposes**

6

7

8 ***Dr. Fawzi Albalooshi****

9

10 ***Department of Computer Science, College of Information Technology,***

11 ***University of Bahrain***

12 ***fabblooshi@uob.edu.bh***

13 ***Fawzi Albalooshi***

14 ***fabblooshi@uob.edu.bh***

15 ***fabblooshi@uob.edu.bh***

16

17

18 **ABSTRACT**

OO software development has become the dominant development approach with Java as the common implementation language. A well-known drawback in Java is its limitation in implementing multiple inheritance which is considered by many researchers a fundamental concept in OO. Approaches in simulating multiple inheritance in Java have been thought of and implemented. In this paper some of these approaches are presented and their negative side effects on the developed software are highlighted. The paper addresses important aspects related to implementing multiple inheritance in Java that may be neglected by developers, and proposes two additional steps in the development life cycle when implementing a system with multiple inheritance relationship(s) in Java. This proposed solution as illustrated with examples ensures proper software development practice throughout the development stages even if there are specific requirements to implement multiple inheritance in Java.

19

20 ***Keywords:*** *Software Engineering, Object-Oriented Software Development, Java*

21 *Programming, Multiple Inheritance.*

22

23 **1. INTRODUCTION**

24

25 According to Booch [1] "inheritance is a relationship among classes wherein one class

26 shares the structure and/or behavior defined in one (single inheritance) or more (multiple

27 inheritance) other classes". Inheritance is a fundamental mechanism that distinguishes

28 object-oriented (OO) method of software development than more traditional ones. The

29 benefits of inheritance include information sharing between a subclass and its super

30 class(es) and software reuse that ultimately results to reduced development time and effort.

31 There are two types of inheritance single and multiple. Single inheritance is the ability of a

32 class to inherit the features of a single super class with more than a single inheritance level

33 i.e. the super class could also be a subclass inheriting from a third class and so on. Multiple

34 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 figure 1. Stroustrup [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 worthwhile considering it though some researchers such as Stroustrup [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 [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; the first, how to deal with name collisions from super classes? And the second, how to handle repeated inheritance? He presents solutions to these two problems. Other researchers [3] 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 [4].

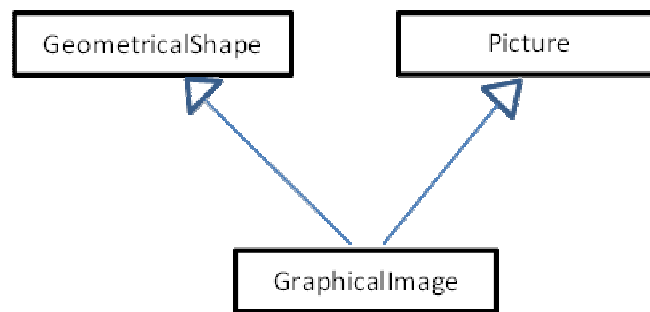


Fig. 1. Multiple Inheritance Example.

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. Java is currently 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) [5]. Nevertheless, it has a major limitation when it comes to implementing multiple inheritance which motivated researchers to think of ways to overcome as discussed in section 2 in this paper. Section 3 addresses the drawback of the solutions discussed in section 2 from a software engineering prospective. Section 4 addresses the stages of development with emphasis on multiple inheritance and presents an approach to extend the development activities to overcome the gap discussed in section 3. Concluding remarks are presented in section 5.

2. MULTIPLE INHERITANCE AND JAVA

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

79 multiple inheritance, however the language supports multiple inheritance of interfaces [6].
80 According to [7] a strong reason that prevents Java from extending more than one class is to
81 avoid issues related to multiple inheritance of attributes from more than one level which is
82 referred at as the 'diamond problem'. In which a sub-class inherits from two or more super
83 classes that share the same ancestor resulting to more than one instance of the same
84 ancestor state (attribute) present in the child class at the lower level of the inheritance
85 hierarchy thus raising the issue of which instance of the ancestor state is valid and should be
86 accessed? On the other hand, interfaces do not have state, thus do not pose such a threat,
87 and the more recent Java 8 compiler resolves the issue of which default method a particular
88 class uses. To overcome this difficulty, researchers investigated compromised solutions.
89 Two of the reported work in the literature have a similar approach with minor differences are
90 discussed in the following two paragraphs.

91 Thirunarayan et al [8] investigated approximating multiple inheritance in Java by enabling a
92 subclass *C* to inherit from a single superclass *A* and to implement an interface *IB* that is
93 implemented by a class *B* in an effort to simulate multiple inheritance in Java. The example
94 in figure 2 outlines the authors' solution to approximating multiple inheritance in Java. The
95 class *B* is then incorporated as an inner class (with composition relationship) in the class *C*.
96 The authors initially present three main difficulties with their solution. The first is that code
97 reuse would be limited, but it is possible. The second is polymorphism and the third is
98 overriding. Polymorphism could not be fully supported due to the fact that class *C* may not
99 support all methods in *B*. Amendments to class *B* will require changes to the interface *IB* and
100 to the class *C*. Overriding is a fundamental concept of inheritance but cannot easily be
101 implemented with inner classes such as *B* and may require the modification of the parent
102 class. The authors conclude that multiple inheritance can be simulated by the use of
103 forwarding to achieve code reuse, interfaces to achieve polymorphism, and back-referencing
104 to approximate overriding.

105 Temprow and Biddle [9] highlight the two main benefits of inheritance as code reuse and
106 protocol conformance. Code defined in the parent class is reused by the child class and the
107 child class responds to the message similarly to the parent class and can substitute it, thus
108 achieving protocol conformance. The authors suggest that delegation can be used to
109 simulate multiple inheritance in Java, but there are two main setbacks. The first is that in
110 some cases the amount of code needed to achieve reuse is almost as much as the code
111 being reused. The second is the difficulty in accessing objects imposed by the solution which
112 renders classes to be highly coupled and with low cohesion. Their solution is similar to that
113 presented by Thirunarayan et al [8] as shown in figure 2 in which the class *B* is incorporated
114 as an inner class within *C* and declaring an object *b* to implement it. In their paper they
115 demonstrate that protocol conformance can be achieved by single inheritance and the use of
116 Java's capability which allows the multiple implementation of Java interface classes. The
117 technique they use is called 'interface-delegation' which require a child class to inherit from a
118 single parent class and implements and delegates to as many interface classes resulting to
119 the child class reusing all the parent classes. In addition to the two main drawbacks
120 highlighted above the solution suffers from the following: first, protected fields and methods
121 of the delegation object are only accessible to extending classes; second, the programmer
122 does not have control over class libraries such as Java Core API thus creating interfaces for
123 such classes is not possible; and third, delegation can be problematic in the presence of
124 self-calls. The authors recommend that every class intended for reuse by inheritance (such
125 as Java Core API library of classes) should also have a matching interface to enable such
126 an approach in simulating multiple inheritance to be applicable.

127 The above two approaches in simulating multiple inheritance in Java proposed by the
128 researchers is adopted and recommended by many Java developers as it is evident in online

129 Java forums and posts. An approach recommended by Venners [10] uses composition (also
130 referred at as inner class/object) instead of inheritance especially if code reuse is the goal.
131 On the other hand, Lagorio et al [11] completely replace inheritance with composition as
132 presented in their framework titled FeatherJigsaw.

```
133
134     class A { // The primary class to be inherited
135         public string a() { return a1();}
136         protected string a1() {return "A";}
137     }
138
139     interface IB { // Second class to be inherited declared as an interface
140         public string b(IB self);
141         public string b1();
142     }
143
144     class B implements IB { // Implementation class for the interface IB
145         public string b(IB self) {return self.b1(); }
146         protected string b1() {return "B";}
147     }
148
149     class C extends A implements IB { // Subclass inheriting from A and
150                                     // implementing IB's interface
151         B b; // Innerclass as composition relationship
152         public string b(IB self) {return b.b(this); }
153         protected string b1() {return "C";}
154         protected string a1() {return "C";}
155     }
```

156
157 **Fig. 2. Approximating Multiple Inheritance in Java.**

158

159 **3. THE PROBLEM - OO DESIGN VS JAVA-ORIENTED DESIGN**

160

161 One of the fundamental concepts in software engineering is that implementation must be
162 based on design, but when attempting to implement a design that uses multiple inheritance
163 as part of the solution we are faced with a dilemma when using Java as the programming
164 language. The design is violated to enable a compromised solution to implement multiple
165 inheritance which does not only affect the classes associated with the multiple inheritance
166 relationship, but may affect other classes in the design. In effect it requires a redesign; an
167 extended design; or more precisely a Java-oriented design. To clarify this claim let us
168 consider the problem raised by Temporo and Biddle [9]. The class diagram shown in figure
169 3.1 represents a graph composed of vertices (nodes in the graph) represented as the class
170 *Vertex*. *VisualVertex* (is a visually displayed graphical icon representing a *Vertex*) inherits
171 from the classes *Vertex* and *Component* (a class borrowed from *java.awt* to provide
172 graphical representation capabilities). If we wish to allow the graph edges to be observers of
173 vertices they are attached to, so that when a vertex changes position all edges attached to it
174 are notified in order to react to the change. The standard design pattern *Observer* [12-13]
175 would be used. It requires the *VisualEdge* class (a visually displayed graphical icon
176 representing an edge (it is not shown in figure 3.1 in order not to complicate the figure)) to
177 implement the *Observer* interface and the *VisualVertex* class to extend the class
178 *Observable*. Implementing the *Observer* interface is possible since Java allows multiple
179 interface implementations, but extending (directly inheriting) the *Observable* class by
180 *VisualVertex* is not possible. Because *VisualVertex* will now need to inherit from three

181 classes: *Vertex*, *Component*, and *Observable*. It is possible to inherit from *Vertex* (since it is
182 a user-defined class) using interface-delegation as discussed in section two, but *Component*
183 and *Observable* are pre-defined classes and can only be extended to be used.
184 Nevertheless, one of them must to be inherited using interface-delegation which is the heart
185 of the problem. If we attempt to apply this technique on the *Observable* class we are faced
186 with a difficulty. The implementation of the *Observer* interface would expect an argument
187 that conforms to *Observable* and if *VisualVertex* implements *Observable* as an interface it
188 would not conform and could not be passed as an argument. The *Observable* class and
189 *Observer* interface depend on each other and must be used as specified by the pattern.
190 Furthermore, an attempt to use interface-delegation to inherit from the *Component* class in
191 order to extend (directly inherit from) the *Observer* to overcome the obstacle will cause
192 nonconformance with the use of the *Component* class and its related AWT classes. As a
193 solution to this problem we are forced to write our own version of the *Observer* pattern
194 considering the fact it can easily be written, but this act raises a serious issue of rewriting
195 code already available for reuse.

196

197 **4. DESIGN ISSUES FOR MULTIPLE INHERITANCE**

198

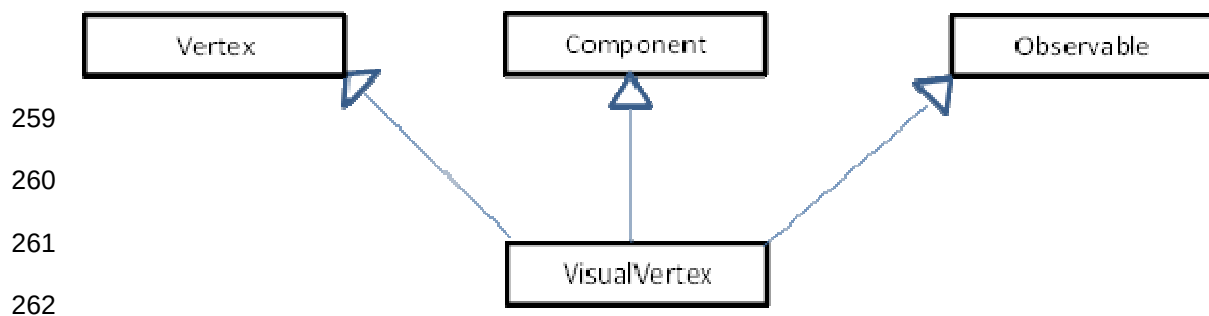
199 Inheritance is mostly recognized in the analysis stage as part of the system in the real world
200 and designers make use of such a situation to benefit from it for software reuse purposes
201 and the mapping of information and functionality according to the system domain. Typically,
202 designers build on such a relationship in the design documents without constraining
203 themselves with implementation issues. Therefore, class diagrams for the inheritance related
204 classes are drawn and attributes, operations, and relationships are decided based on it.
205 According to Blaha and Rumbaugh [14] inheritance has three purposes, firstly to support
206 polymorphism, secondly to structure the description of objects, and thirdly to enable code
207 reuse. Overriding a super class feature by a subclass may also be necessary in some cases
208 in order to specify a behavior specific to the sub-class to tighten the specification or improve
209 performance. As class design progresses adjustments may be made to increase inheritance
210 by rearranging classes and operations, abstracting common behavior for a group of classes,
211 and using delegation to share behavior. In the late detailed design stage prior to
212 implementation it is considered good practice to fine tune classes with inheritance
213 relationships to ensure proper implementation.

214 Multiple inheritance is a fundamental OO concept that is applied in early software
215 development stages and developers should not constrain themselves with the limitations of
216 the programming language to be used in early stages and should freely apply multiple
217 inheritance concepts in order to design a proper OO system. In other words, regardless of
218 the programming language capabilities to support multiple inheritance or not, ideally the
219 analysis and design documents should clearly and freely include specifications for multiple
220 inheritance concepts if found suitable for the system under development. However, if such
221 design concepts are not supported by the programming language special implementation
222 classes could be defined to overcome such difficulties. We recommend that implementation
223 decisions related to the programming language to be used be carefully assessed and
224 separate implementation language specific design documents be created that clearly
225 indicate their purpose. Such classes and associated code will most likely need special
226 testing arrangements to ensure multiple inheritance issues are properly implemented such
227 as code reuse, polymorphism, dynamic binding, and overriding. Whether to use single
228 inheritance, interface-delegation, inner class, or a combination of the three it is important that
229 a proper OO software development approach is followed in order to analyse and design the
230 software system independent of the implementation language especially the classes and
231 their relationships such documents should be left intact showing the exact OO nature of the

232 system in case future enhancements become necessary or for it to be implemented in an
 233 alternative programming language. Figure 4 below shows our two proposed amendments to
 234 the development process. The first is a special design stage to cater for a system that uses
 235 multiple inheritance and it is to be implemented in a programming language that does not
 236 support such a mechanism such as Java. The second is related to amendments to existing
 237 systems. Systems undergo continues developments for improvements as it is clearly evident
 238 in a study carried out by Nasseri et al [15] in which they observed that the number of classes
 239 in four live Java systems of different sizes have continually increased (in some cases more
 240 than four times) as improved versions of the software were released. Classes within systems
 241 were continuously moved across the same inheritance hierarchy. Modifications to a system
 242 are inevitable, therefore, we strongly believe that future modifications to a system must
 243 properly be analysed to ensure that design documents are kept up-to-date, the new
 244 modifications are properly integrated, and a proper software engineering approach is
 245 followed.

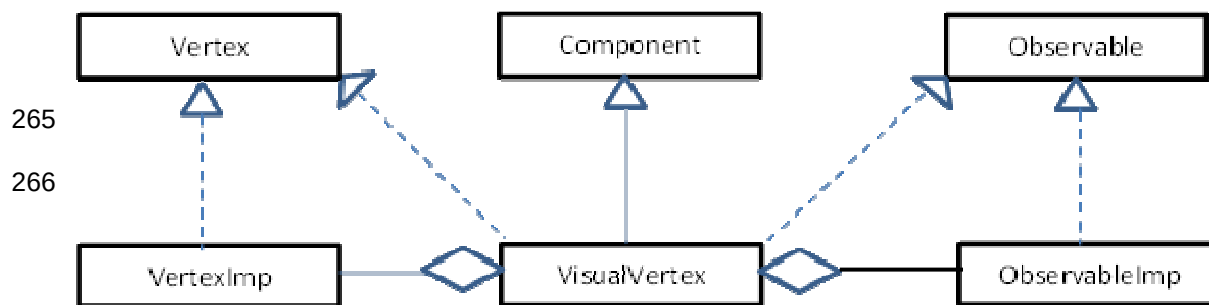
246 For example, if it is required to implement the system discussed in section 3 and shown in
 247 figure 3 in C++ the classes and their relationships shown in figure 3.1 would be implemented
 248 as they are. However, if it is required to implement it in Java we would have to implement the
 249 classes shown in figure 3.2 which were especially modified to suit Java but at the same time
 250 simulate (as much as possible) the design shown in 3.1 and were developed to
 251 accommodate the difficulties in implementing multiple inheritance in Java. Therefore, the
 252 class model shown in figure 3.1 is the base and is a more accurate OO representation and
 253 design for the problem. We would face OO design shortcomings if the design was specially
 254 developed for Java implementation, thus rendering the more accurate version never to be
 255 thought of and therefore future modifications and enhancements would be carried-out on an
 256 inaccurate OO design which may result to unseen design and implementation difficulties and
 257 flaws.

258



263 **Fig. 3.1. Typical OO Design**

264



267

268

Fig. 3.2. Java-Oriented Design

269

Fig. 3. Typical OO Design vs Java-Oriented Design.

270

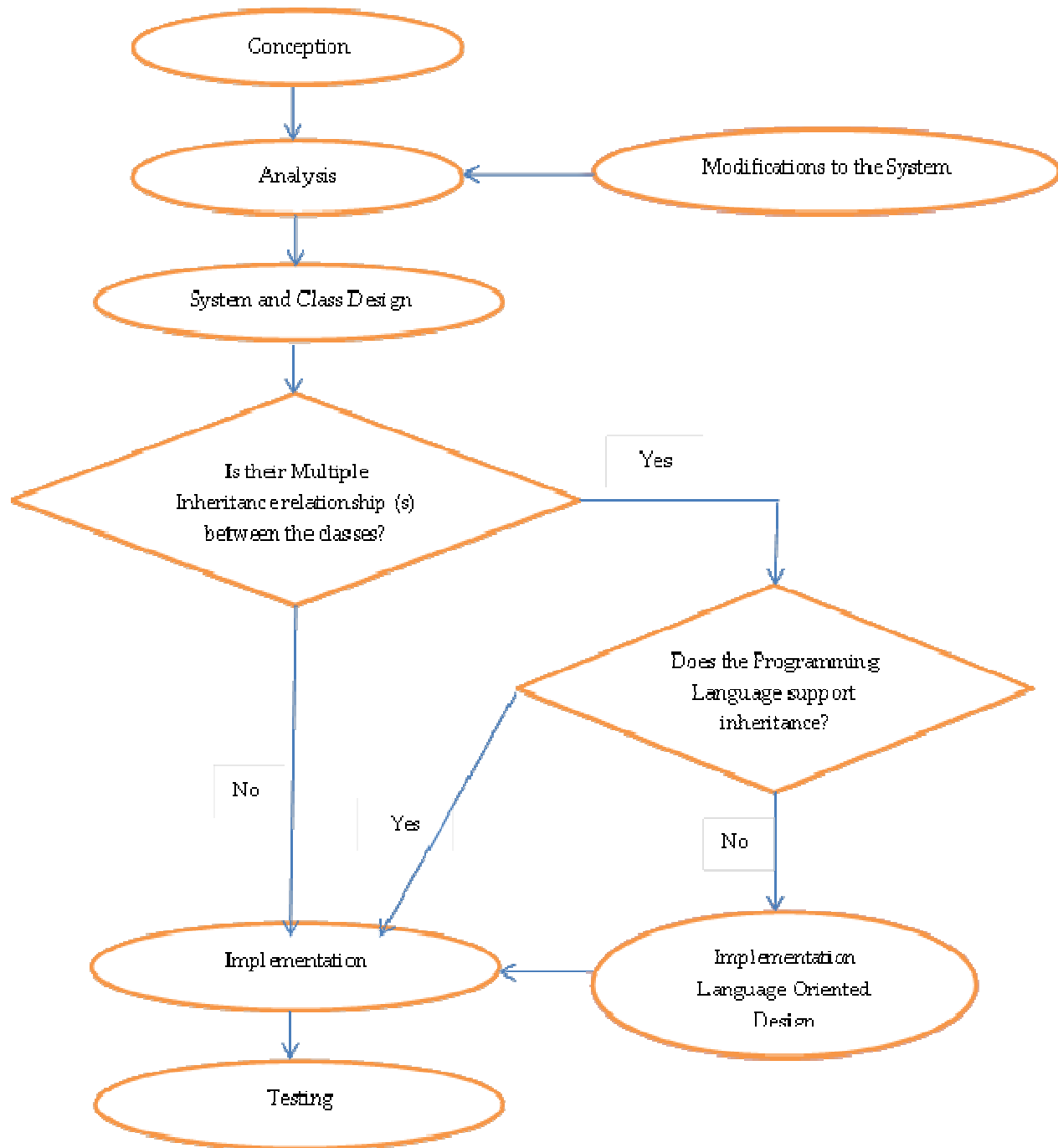


Fig. 4. Extending the Software Development Stages.

As another example consider the programme shown in figure 2 if we assume that Java supports multiple inheritance the code for the same programme would be written as shown in figure 5 below. The code in figure 2 was actually written based on the multiple inheritance concept as shown in figure 5, thus strongly supporting the suggestions presented in figure 4

277 in that a system must first be analysed and designed with multiple inheritance in mind and
 278 then if the implementation language does not support multiple inheritance necessary
 279 simulation modifications must be planned and designed before being implemented. Keeping
 280 both designs eases the implementation of future modifications to the system. For example, if
 281 we introduce a minor change to the code in figure 2 such as to the name of the function
 282 '*public string b(IB self)*' in the class *B*. The interface *IB* and the class *C* will need to be
 283 updated for the change. However, the same change in the same function in the
 284 implementation shown in figure 5 will require no change in the class *C*. Modifications to the
 285 system after implementation must be addressed in early stages as shown in figure 4.

286

```
287      class A { // The primary class to be inherited
288
289          public string a() { return a1();}
290
291          protected string a1() {return "A";}
292      }
293
294      class B { // The primary class to be inherited
295
296          public string b() { return b1();}
297
298          protected string b1() {return "B";}
299      }
300
301      Class C extends A, B {//
302
303          protected string b1() {return "C";}
304
305          protected string a1() {return "C";}
306      }
```

299 **Fig. 5. Java Classes with Multiple Inheritance.**

300

301 **5. CONCLUSION**

302

303 OO software development methodology has become the most popular development
 304 paradigm in use nowadays. Many systems are being developed using Java due to its rich
 305 collection of readily available well-designed set of classes known as Java APIs and many
 306 other benefits such as the availability of a common network-centric platform, its security
 307 features, dynamic-ability, and extensibility. A major deficiency in Java its limitation to
 308 implement multiple inheritance which motivated many researchers to suggest solutions to
 309 overcome them as reviewed in section 2. Though such solutions are possible there is a cost
 310 on the developed software that developers must endure as discussed in section 3. A higher
 311 cost a Java system will endure would be on the design side if special care is not practiced,
 312 that may have negative effects on implementation as explained in section 4 in which the

313 authors suggest a slightly modified software development life-cycle by which such costs and
314 difficulties are reduced to the minimum.

315

316 REFERENCES

317

318 1. Booch Grady. Object-oriented analysis and design with applications. 2nd Edition.
319 Published by Addison-Wesley in December 1998.

320 2. Stroustrup Bjarne. Multiple inheritance for C++. The C/C++ Users Journal. May 1999.

321 3. Ducournau R., Morandat F., and Privat J.. Empirical assessment of object-oriented
322 implementations with multiple inheritance and static Typing. OOPSLA 2009, October 25-29,
323 2009, Orlando, Florida, USA. ACM, 2009.

324 4. Taft S. T., Duff R. A., Brukardt R. L., Ploedereder E., and Leroy P., editors. Ada 2005
325 reference manual: language and standard libraries. LNCS 4348. Springer, 2006.

326 5. Flanagan David. Java in a NUTSHELL. 3rd Edition. Published by O'Reilly & Associates,
327 Inc.; November 1999.

328

329 6. Gosling James, Joy Bill, Steele Guy, Bracha Gilad, and Buckley Alex. The Java language
330 specification – Java SE. 7th Edition. Oracle America, Inc.; 2013.

331 7. Oracle 2014. Multiple Inheritance of State, Implementation, and Type. Accessed 15th
332 December 2014. Available:
333 <http://docs.oracle.com/javase/tutorial/java/landl/multipleinheritance.html>.

334 8. Thirunarayan Krishnaprasad, Kniesel Gunter, and Hampapuram Haripriyan. Simulating
335 multiple inheritance and generics in Java. Computer Languages, Volume 25, Issue 4,
336 December 1999, Pages 189-210, Published by Elsevier Science Ltd.

337 9. Temprow Ewan and Biddle Robert. Simulating multiple inheritance in Java. The Journal of
338 Systems and Software. 2000; 55, pp. 87-100. Published by Elsevier Science Inc.

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

342 11. Lagorio Giovanni, Servetto Marco, and Zucca Elena. Featherweight Jigsaw – Replacing
343 inheritance by composition in Java-like languages. Information and Computation, 2012;
344 volume 214, pages 86-111. Published by Elsevier Inc.

345 12. Gamma E., Helm R., Johnson R., Vlissides J. Design patterns: elements of reusable
346 object-oriented software, Addison-Wesley Professional Computing Series. Addison-Wesley,
347 Reading, MA; 1995.

348 13. JavaWorld, Inc. 2014. Observer and Observable: an introduction to the Observer
349 interface and Observable class using the Model/View/Controller architecture as a guide.
350 Accessed 23rd July 2014. Available: [http://www.javaworld.com/article/2077258/learn-](http://www.javaworld.com/article/2077258/learn-java/observer-and-observable.html)
351 [java/observer-and-observable.html](http://www.javaworld.com/article/2077258/learn-java/observer-and-observable.html).

352 14. Blaha Micheal and Rumbaugh James. Object-oriented modeling and design with UML.
353 2nd Edition. Pearson Education Inc. United States of America; 2005.

354 15. Nasser E., Counsell S., and Shepperd M. Class movement and re-location: An empirical
355 study of Java inheritance evolution. The Journal of Systems and Software, 2010: volume 83,
356 pages 303-315. Published by Elsevier Inc.

357