British Journal of Mathematics & Computer Science

British Journal of Mathematics & Computer Science
6(5): 435-443, 2015, Article no.BJMCS.2015.089
ISSN: 2231-0851

SCIENCEDOMAIN *international*
www.sciencedomain.org

SCIENCEDOMAIN

Software Design Concerns Associated with Simulating Multiple Inheritance in Java for Implementation Purposes

Fawzi Albalooshi[1*]

[1]*Department of Computer Science, College of Information Technology, University of Bahrain, Bahrain.*

Article Information

DOI: 10.9734/BJMCS/2015/15476
Editor(s):
(1) Ke-Lin Du, Chief Scientist, Enjoyor Labs, Hangzhou, China and Department of Electrical and Computer Engineering, Concordia University, Canada.
Reviewers:
(1) M. Bhanu Sridhar, Department of CSE & IT, GVP College of Engineering for Women, India.
(2) James A. Rodger, Indiana University of Pennsylvania, USA.
(3) Anonymous, India.
Complete Peer review History: http://www.sciencedomain.org/review-history.php?iid=735&id=6&aid=7757

Original Research Article

Received: 27 November 2014
Accepted: 03 January 2015
Published: 13 January 2015

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.

Keywords: Software engineering, object-oriented software development, java programming, multiple inheritance.

1 Introduction

According to Booch [1] "inheritance is a relationship among classes wherein one class shares the structure and/or behavior defined in one (single inheritance) or more (multiple inheritance) other classes". Inheritance is a fundamental mechanism that distinguishes object-oriented (OO) method

*Corresponding author: falblooshi@uob.edu.bh;

of software development than more traditional ones. 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.

There are two types of inheritance 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] 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.

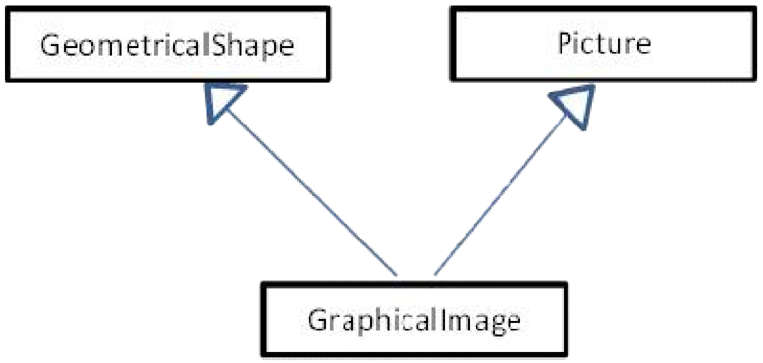

Fig. 1. Multiple inheritance example

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].

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 multiple inheritance, however the language supports multiple inheritance of interfaces [6]. According to [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’. 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 difficulty, 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. [8] 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*. 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 [9] 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 with low cohesion. Their solution is similar to that presented by Thirunarayan et al. [8] 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 [10] 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. [11] completely replace inheritance with composition as presented in their framework titled Feather Jigsaw.

```
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

3The Problem - OO Design Vs Java-Oriented Design

One of the fundamental concepts in software engineering is that implementation must be based on design, but when attempting to implement a design that uses multiple inheritance as part of the solution we are faced with a dilemma when using Java as the programming language. The design is violated to enable a compromised solution to implement multiple inheritance which does not only affect the classes associated with the multiple inheritance relationship, but may affect other classes in the design. In effect it requires a redesign; an extended design; or more precisely a Java-oriented design. To clarify this claim let us consider the problem raised by Tempro and Biddle [9]. The class diagram shown in Fig. 3.1 represents a graph composed of vertices (nodes in the graph) represented as the class *Vertex*. *VisualVertex* (is a visually displayed graphical icon representing a Vertex) inherits from the classes *Vertex* and *Component* (a class borrowed from *java.awt* to provide graphical representation capabilities). If we wish to allow the graph edges to be observers of vertices they are attached to, so that when a vertex changes position all edges attached to it are notified in order to react to the change. The standard design pattern *Observer* [12-13] would be used. It requires the *VisualEdge* class (a visually displayed graphical icon representing an edge (it is not shown in Fig. 3.1 in order not to complicate the figure)) to implement the *Observer* interface and the *VisualVertex* class to extend the class *Observable*. Implementing the Observer interface is possible since Java allows multiple interface implementations, but extending (directly inheriting) the *Observable* class by *VisualVertex* is not possible. Because *VisualVertex* will now need to inherit from three classes: *Vertex*, *Component*, and *Observable*. It is possible to inherit from *Vertex* (since it is a user-defined class) using interface-delegation as discussed in section two, but *Component* and *Observable* are pre-defined classes and can only be extended to be used. Nevertheless, one of them must to be inherited using interface-delegation which is the heart of the problem. If we attempt to apply this technique on the *Observable* class we are faced with a difficulty. The implementation of the *Observer* interface would expect an argument that conforms to *Observable* and if *VisualVertex* implements *Observable* as an interface it would not conform and could not be passed as an argument. The *Observable* class and *Observer* interface depend on each other and must be used as specified by the pattern. Furthermore, an attempt to use interface-delegation to inherit from the *Component* class in order to extend (directly inherit from) the *Observer* to overcome the obstacle will cause nonconformance with the use of the *Component* class and its related *AWT* classes. As a solution to this problem we are forced to write our own version of the *Observer* pattern considering the fact it can easily be written, but this act raises a serious issue of rewriting code already available for reuse.

4 Design Issues for Multiple Inheritance

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

Multiple inheritance is a fundamental OO concept that is applied in early software development stages and developers should not constrain themselves with the limitations of the programming language to be used in early stages and should freely apply multiple inheritance concepts in order to design a proper OO system. In other words, regardless of the programming language capabilities to support multiple inheritance or not, ideally the analysis and design documents should clearly and freely include specifications for multiple inheritance concepts if found suitable for the system under development. However, if such design concepts are not supported by the programming language special implementation classes could be defined to overcome such difficulties. We recommend that implementation decisions related to the programming language to be used be carefully assessed and separate implementation language specific design documents be created that clearly indicate their purpose. Such classes and associated code will most likely need special testing arrangements to ensure multiple inheritance issues are properly implemented such as code reuse, polymorphism, dynamic binding, and overriding. Whether to use single inheritance, interface-delegation, inner class, or a combination of the three it is important that a proper OO software development approach is followed in order to analyse and design the software system independent of the implementation language especially the classes and their relationships, such documents should be left intact showing the exact OO nature of the system in case future enhancements become necessary or for it to be implemented in an alternative programming language. Fig. 4 below shows our two proposed amendments to the development process. The first is a special design stage to cater for a system that uses multiple inheritance and it is to be implemented in a programming language that does not support such a mechanism such as Java. The second is related to amendments to existing systems. Systems undergo continues developments for improvements as it is clearly evident in a study carried out by Nasseri et al. [15] in which they observed that the number of classes in four live Java systems of different sizes have continually increased (in some cases more than four times) as improved versions of the software were released. Classes within systems were continuously moved across the same inheritance hierarchy. Modifications to a system are inevitable, therefore, we strongly believe that future modifications to a system must properly be analysed to ensure that design documents are kept up-to-date, the new modifications are properly integrated, and a proper software engineering approach is followed.

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

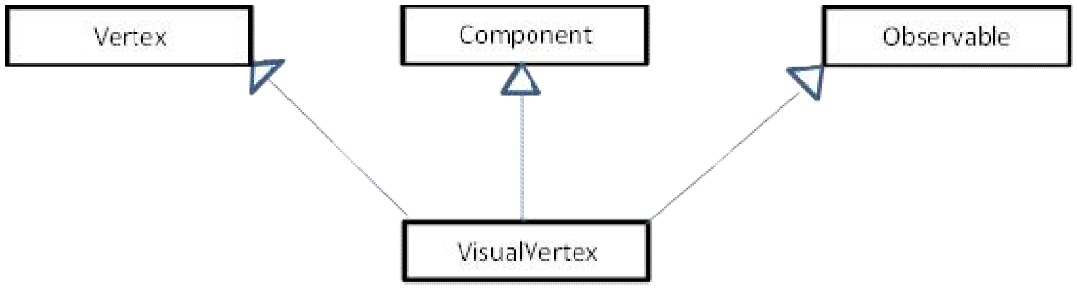

Fig. 3.1. Typical OO design

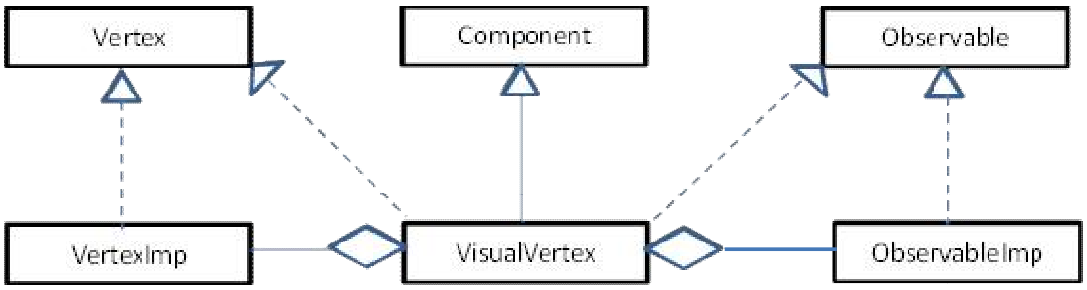

Fig. 3.2. Java-oriented design

Fig. 3. Typical OO design vs Java-oriented design

As another example consider the programme shown in Fig. 2 if we assume that Java supports multiple inheritance the code for the same programme would be written as shown in Fig. 5 below. The code in Fig. 2 was actually written based on the multiple inheritance concept as shown in Fig. 5, thus strongly supporting the suggestions presented in Fig. 4 in that a system must first be analysed and designed with multiple inheritance in mind and then if the implementation language does not support multiple inheritance necessary simulation modifications must be planned and designed before being implemented. Keeping both designs eases the implementation of future modifications to the system. For example, if we introduce a minor change to the code in Fig. 2 such as to the name of the function '*public string b(IB self)*' in the class *B*. The interface *IB* and the class *C* will need to be updated for the change. However, the same change in the same function in the implementation shown in Fig. 5 will require no change in the class *C*. Modifications to the system after implementation must be addressed in early stages as shown in Fig. 4.

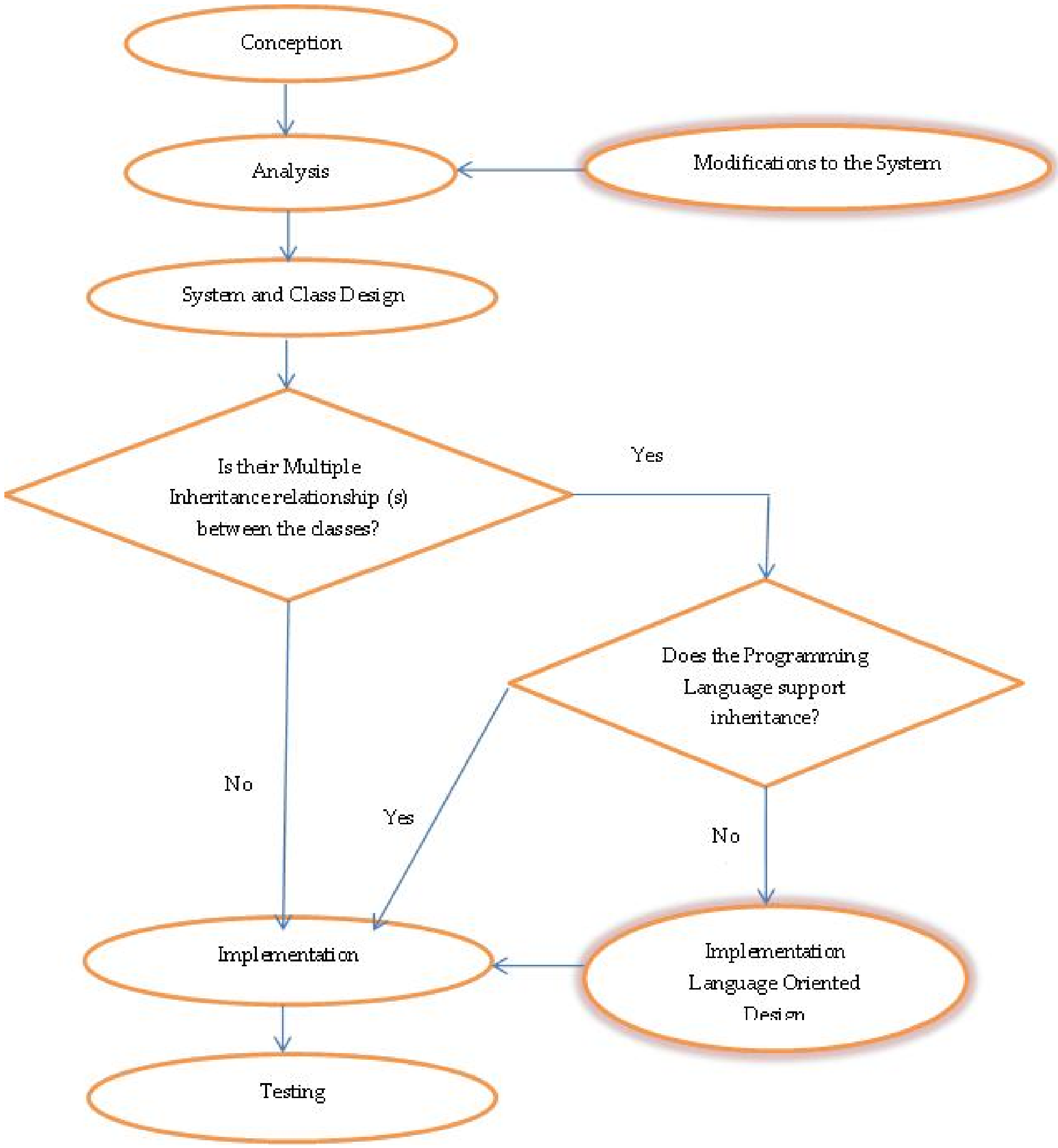
Conception
Analysis
Modifications to the System
System and Class Design
Is their Multiple Inheritance relationship (s) between the classes?
Yes
Does the Programming Language support inheritance?
No
Yes
No
Implementation
Implementation Language Oriented Design
Testing

Fig. 4. Extending the software development stages

```
class A { // The primary class to be inherited
          public string a() { return a1();}
          protected string a1() {return "A";}
}
class B { // The primary class to be inherited
          public string b() { return b1();}
          protected string b1() {return "B";}
}
Class C extends A, B {//
          protected string b1() {return "C"};
          protected string a1() {return "C"};
}
```

Fig. 5. Java classes with multiple inheritance

5 Conclusion

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

Competing Interests

Author has declared that no competing interests exist.

References

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

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

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

[4] Taft ST, Duff RA, Brukardt RL, Ploedereder E, Leroy P, editors. Ada 2005 reference manual: language and standard libraries. LNCS 4348. Springer, 2006.

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

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

[7] Oracle 2014. Multiple inheritance of state, implementation, and type. Accessed 15th December 2014.
Available: http://docs.oracle.com/javase/tutorial/java/IandI/multipleinheritance.html

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

[9] Tempro Ewan, Biddle Robert. Simulating multiple inheritance in Java. The Journal of Systems and Software. 2000;55:87-100. Published by Elsevier Science Inc.

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

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

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

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

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

[15] Nasseri E, Counsell S, Shepperd M. Class movement and re-location: An empirical study of Java inheritance evolution. The Journal of Systems and Software. 2010;83:303-315. Published by Elsevier Inc.

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)
www.sciencedomain.org/review-history.php?iid=735&id=6&aid=7757