ISSN 22773061

Predilection of Reusability over Maintainability in Aspect-Oriented Systems

Ritika Chaudhary, Ram Chatterjee
M.Tech. Student, Manav Rachna College of Engineering/ Department of Computer science & engineering, Faridabad, India
ritzchaudhary19@gmail.com
Assistant Professor, Manav Rachna College of Engineering/ Department of Computer science & engineering, Faridabad, India
ramchatterjee.mrce@mrei.ac.in

ABSTRACT

Maintenance is the important phase in software development lifecycle which initiates after the software has been deployed for use. Reusability is an important area of concern which depicts the extent to which a module can be used again in different applications with slight or no modification. Maintainability is one of the contributing factors for assessing Reusability. So, assessment of Reusability is preferred over Maintainability.

This paper has been split into Introduction, Role of Maintainability, Role of reusability and conclusion. In the introduction section the concept that how the software evolves has been discussed. The second section focuses on the Role of Maintainability. The third second emphasizes on the Role of Reusability within the domains of Object-Oriented Programming and Aspect-Oriented Programming. In the last section we have concluded that assessment of Reusability must be given more preference over the assessment of Maintainability.

Indexing terms/Keywords

Module-Oriented Approach, Object-Oriented Approach, Aspect-Oriented Software development, Quality Parameters.

Council for Innovative Research
Peer Review Research Publishing System
Journal: INTERNATIONAL JOURNAL OF COMPUTERS & TECHNOLOGY
Vol 6, No 3
editor@cirworld.com
www.cirworld.com, member.cirworld.com

INTRODUCTION

In this internet era it has been noticed that society has become software dependent. Software can be seen being used in everyday life. With the help of software one can get connected to his friends and families within seconds anywhere in the world. Software has been found as part of every technology such as net banking, trading etc.

Being dependent on the services of software it is important to keep it running without problems. Errors may occur in software which needs to be corrected so that its performance is not affected. In other words, software maintenance should be regularly monitored which can enhance software's lifespan. As long as software services are available, maintenance remains as an integral activity driven by the changes in the business rules and environment.

The maintenance effort required for software might be greater than the effort required for its initial implementation. That is why; there is a heavy financial load for maintaining software.

The lifecycle of the software begins when the product is conceived till it becomes useless. The software lifecycle may consist of following phases: specification, design, implementation, testing, installation, and maintenance.

In Figure 1, Waterfall model is presented with all the phases.

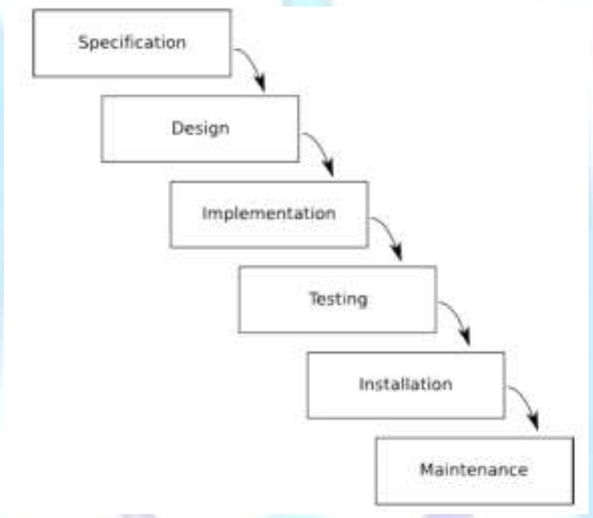

Figure 1. The Waterfall Model

During the lifespan of the software, changes occurring in operational environment have to be met by making changes in the software. These changes are necessary to keep the software acceptable and functional. These changes are called software evolution. In figure 2 an iterative model for software evolution is shown.

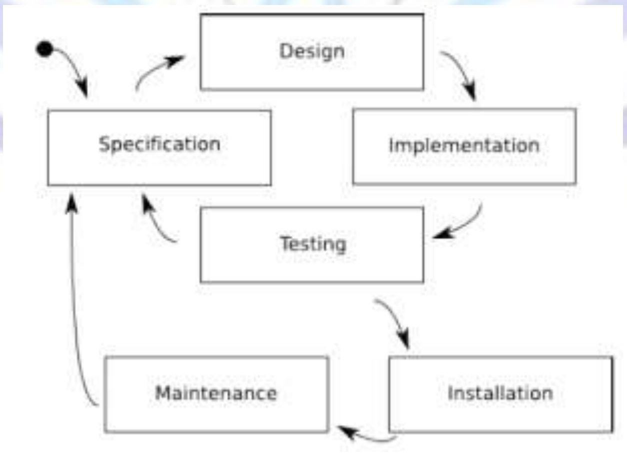

Figure 2. An iterative model for software evolution

Software evolution shows that even if the implementations of the software is correct and fulfill the requirements, the software has to be updated on regular basis which also justifies the need for maintenance.

In the waterfall model the last phase of software's lifecycle is the maintenance which starts after the product is implemented and tested and continues until the software is useless. In the waterfall, maintenance is actions to correct errors in software that has occurred in the testing phase. According to the Lehman's laws, maintenance is not just corrective actions but it also includes the effort to accommodate new requirements in the software by adding or improving features. The aspect of maintenance is making changes that aim to increase maintainability, performance or other metrics.

Software quality is a major point of concern and it is therefore important to assess the maintainability of software. According to IEEE, maintainability can be defined as “The ease with which a software system or Component can be modified to correct faults, improve performance or other attributes, or adapt to a changed environment” [9].The changes occurring due to modification have effect on interfaces, components and features.

Role of maintainability

Maintenance is a core part of the software lifecycle, which accounts for more than half of the cost of software development. Therefore, it requires prompt attention. The primary concern of the software maintenance is to make the software functional and up to date.

Software maintenance is classified into four types [8]:

1) **Corrective**- Corrective maintenance refers to fixing a program. This is estimated to cover approximately 17% of maintenance costs.

2) **Adaptive**- Adaptive maintenance refers to modifications that adapt to changes in the data environment such as new product codes or new file organization or changes in hardware of software environment. It covers about 18% of the costs.

3) **Perfective**- Perfective maintenance refers to enhancements: making the product better, faster smaller, better documented; clear structured with more functions or reports. It covers most of the costs with approximately 60% share

4) **Preventive**- Preventive maintenance is defined as the work that is done in order to try to prevent malfunctions or improve maintainability.

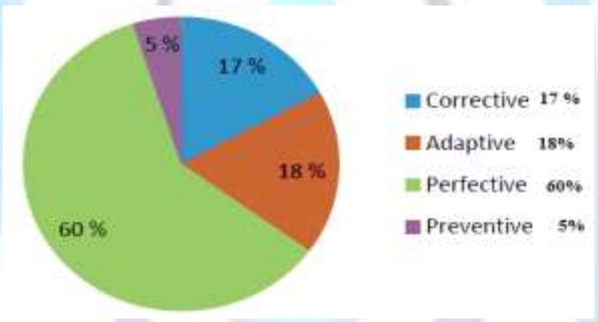

Figure 3: Distribution of percentage of effort during software maintenance, categorized by type

It is estimated that approximately 85% of all maintenance costs come from adaptive and perfective maintenance efforts. In developing a software system High maintainability can lower the cost for maintenance activities.

Although software maintenance is important, but little work has been done to assess software maintainability. Thus, it is necessary to analyze software quality by considering maintainability as one of the assessment factor.

To minimize maintenance costs people have found ways by introducing better development methodologies that can minimize the effects of change, simplify the understanding of programs, and can facilitate the early detection of faults. The “understandability” is directly related to the maintainability, as it is one of the dominant factors that affect software maintainability. For example, if a code does not have any logical errors, but it is difficult to understand, then an increase in costs of maintenance is inevitable. Understandability can be considered as prerequisite for maintainability.

Maintainability of software can be determined by several factors, but, the factor that constitutes the most in determining how maintainable a program is that how well it is organized into separate modules that can be developed and maintained independently.

A concern is a feature that a system should implement which includes all the functional, nonfunctional requirements and the design constraints in the system. Each concern (purpose, concept, and goal) of the program should be localized in a single module, so that if the requirements related to that concern change, only that module needs to be modified. Similarly, each module should address only one concern, so that the developer of that module can focus exclusively on that concern. This separation of concerns is also important to software reusability. In the presence of crosscutting concerns, the separation of concerns is very difficult to achieve, as aspects of a program cannot be confined to a single program component.

In this regard methodology such as the object-oriented approach has played an important role which tends to improve the maintainability of software by employing the concepts of object and encapsulation. The OO minimizes the impact of change by providing some constructs. However, some concerns may crosscut among a number of objects whose modification may be difficult as they are scattered across many objects. E.g. code implementing a security concern may be scattered throughout login, logout and network-socket modules. Thus code segments implementing a security concern may crosscut through other functional concerns of the program. So it becomes difficult to update security code scattered throughout the application.

Aspect oriented programming (AOP) has been introduced to address this issue, as a way of improving the modularity of code there by facilitating maintenance. Concerns that crosscut across different components are represented by the construct aspect.

AOP has introduced different constructs which are as follows.

(i)**Aspects**- Aspects are similar in nature to classes in OO and equivalent to the other OO constructs such as interface. Aspects are used to package advices, pointcuts, and intertype declarations in a modular unit in the same way as classes and interfaces package declarations and methods. Aspects along with classes can be referred to as **modules** and they both represent the basic unit of modularity in AOP.

(ii)**Advices**-Advices are similar in nature to methods in OO. Aspects contain the body of the code that is executed at a given point which is captured by a join point. The advices along with the methods are referred to as **operations**.

(iii)**Intertype Declarations**- These are used to add new fields or methods to a class.

The main focus of AOP is to improve the implementation of crosscutting concerns (i.e., scattering and tangling).

Aspect Oriented Programming (AOP) makes maintenance easy and promotes reuse of software components by separating core concerns from crosscutting concerns. Core concerns are the functional requirements of the system. Some concerns such as logging, security etc. are scattered throughout the code, which affects reusability and maintainability. Such concerns are called crosscutting concerns In AOP languages concern is encapsulated in an aspect and is connected to main classes by using pointcuts. This would remove extraneous code from the classes of the program and allow them to focus on their core purpose thereby making them more maintainable and reusable. The implementation of each crosscutting concern is centralized in a single aspect.

The factors that affect software maintainability are analyzability, changeability, stability, and testability.

1) **Analyzability** is attributes of software that bear on the effort needed to diagnose of deficiencies or causes of failure and to identify parts to be modified.

2) **changeability** is some attributes of software that bear on the effort needed to make modifications, eliminate faults or change the system in response to environmental change.

3) **Stability** can be represented by attributes of software that bear on the risks associated with unexpected effects of modifications.

4) **Testability** refers to attributes of software that bear on the effort needed to validate modifications.

Internal and external quality of software can be easily distinguished. In [9] maintainability model has been proposed. The objective is to establish a relationship between maintainability (external quality) and internal quality attributes such as size, complexity, coupling and inheritance.

Size- Size is used to evaluate the case of understandability of the code by the developers and maintainers. It can be measured in many ways such as by counting lines of code, the number of statements etc [8].

Complexity- By software complexity we mean the difficulty to preserve, modify and comprehend the software. It is the measure of how difficult a software system is and it is really desirable to achieve low complexity in software system [8].

Coupling- Coupling means the interdependence between different components or functions. It is the measure of interconnections among the modules in a software structure. It is the degree to which each program module depends on the other and it is required to achieve low coupling in software systems [8].

Inheritance- Inheritance is defined as classes having same methods and operations based on hierarchy. It is a mechanism by which one object inherits the characteristics from one or more objects [8].

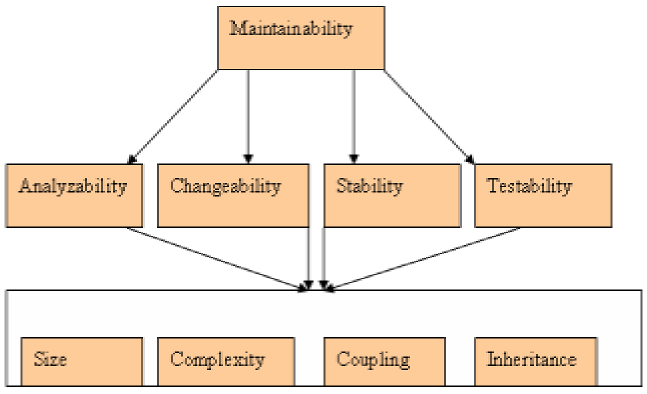

Figure 4: Maintainability Model[9]

Role of Reusability

Software quality is the degree to which a system, component, or process meet specified requirements. Several metrics have been proposed to measure the quality of software**[2]**. They are focused on the following four system attributes:

• Reusability

• Maintainability

• Understandability

• Testability

Reusability is the ability of software elements to serve for the construction of different elements in the same software system or across different ones. Reusability in case of an object-oriented C++ program is illustrated through Figures 5 and 6. While building an object-oriented C++ program, it is mandatory to define reusable source code (such as a class) in a file that by convention has a .h filename extension, and commonly known as a header file. Programmers use preprocessor directives (#include) to include header files and take advantage of reusable software components. For example the type string provided in the C++ Standard Library and user-defined types like class Book.

```
//Book.h
//Book class definition in a separate file from main
#include<iostream>
using std::cout;
using std::end1;

#include<string>//class Book uses c++ standard string class
using std::string;
//Book class definition
class Book
{
public:
            book(string name)
          {
                  setCourseName(name); //call set function to initialize courseName
```

```
        }//end Book constructor
                //function to set the course name
                void setCourseName(string name)
                {
                        courseName=name;
                }//end fuction setcourseName
                //function to get the course name
                string getCourseName()
                {
                        return courseName; //return object's courseName
                }
        //display a welcome message to the Book user
                void displayMessage()
                {
        //call getcourseName to get the courseName
                        cout<<"welcome\n"<<getCourseName()<<"!"<<endl;
                }
                //end function displayMessage
private:
string courseName; //course name for this Book
};//end class Book
```

Figure 5. Book class definition

In our example, we separated the code into two files Book.h and TextFile2.cpp. The main function that uses class Book is defined in the source-code file TextFile2.cpp. In C++, a header file such as Book.h (Figure 5) cannot be used to begin program execution, because it does not contain a function named main. If you try to compile and link Book.h by itself to create an executable application, Microsoft Visual C++ .NET will produce the linker error message:

```
error LNK2019: unresolved external symbol _main
```

This error indicates that the linker could not locate the program's main function. To test class Book (defined in Figure 5), you must write a separate source-code file containing a main function (such as Figure 6) that instantiates and uses objects of the class.

```
// TextFile2.cpp
#include<iostream>
using std::cout;
using std::endl;

#include "Book.h" //include definition of class Book

//function main begins program execution

int main()
```

```
{
// create two Book objects
Book Book1("CS101 Introduction to c");
Book Book2("CS102 Data Structures");
// display initial value of courseName for each Book
cout<<"Book1createdforcourse:"<<Book1.getCourseName()<<"\n        Book2        created        for
course:"<<Book2.getCourseName()
<<endl;
return 0; // indicate successful termination
}// end main
```

Figure 6. Including class Book from file Book.h for use in main

Maintenance**[6]** is the activity of modifying a software system after initial delivery. Maintenance activities are classified into four categories: corrective, perfective, adaptive maintenance and evolution. Since the main goal of AOSD is to improve the evolution of OO systems, our focus is on the evolution facet of aspect-oriented systems.

Flexibility and understandability are the core factors for promoting reuse and maintainability. Both require software abstractions to support understandability and flexibility. Understandability indicates the level of difficulty for studying and understanding a system's design and code. Flexibility indicates the level of difficulty for making drastic changes to one component in a system without a need to change others. An understandable system enhances its own maintainability and reusability, because most maintenance and reuse activities require that software engineers first try to understand the system's components.

Furthermore a software system must be flexible enough to support the addition and removal of functionalities and the reuse of its components with a minimum amount of effort. The understandability factor is related to the following attributes : (i) size, (ii) coupling, (iii) cohesion and (iv) separation of concerns.

Coupling and cohesion affect understandability because a component of the system cannot be understood without reference to the other components to which it is related. The size of design and code may indicate the amount of effort needed for understanding the software components. The separation of concerns criterion is a predictor of understandability because the more localized the concerns, easier it is to understand them. The flexibility factor is influenced by the following attributes: (i) coupling, (ii) cohesion and (iii) separation of concerns.

Coupling is the degree to which the elements or modules in a design are connected. The coupling degree impacts on system quality such as maintainability (modifying a given module may require the modification of some of its connected modules), understandability (a very connected module is very hard to understand), reusability (the more independent a module is, the easier it is to be reused in another application), testability (a fault in a module may cause failures in its connected modules), modularity (low coupling between modules improves software modularity), and efficiency (strong coupling between modules complicates a system). Thus a good programming principle is to minimize the coupling. Cohesion is the degree to which elements within a module are related to one another. High cohesion indicates good module subdivision. Low cohesion increases complexity thereby increasing the likelihood of errors during the development process. High cohesion, low coupling and separation of concerns constitute desired attributes because they mean that a component represents a single part of the system and the system components are independent or almost independent. Furthermore, the system's concerns are not scattered and tangled. In figure 7, Relationship of reusability with external and internal characteristics is depicted.

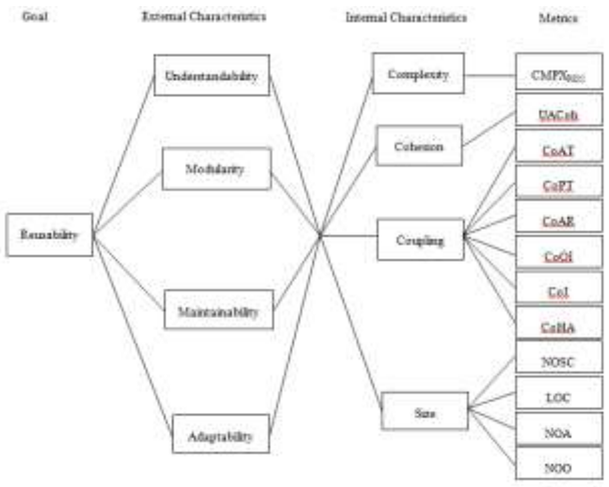

Figure 7. Relationship of reusability with external and internal characteristics [3].

"Reusability is the cost of transferring a module or program to another application". Reusability has become an important element due to rapid software development and increasing complexity. Software reusability enhances the quality of software by reducing development time, effort and cost. It is the most important criteria for the evaluation of software system. A reusable component will help in better understandability and low maintenance efforts for the application[6]. Therefore, it is necessary to estimate reusability of the component, before integrating reusable code into the system.

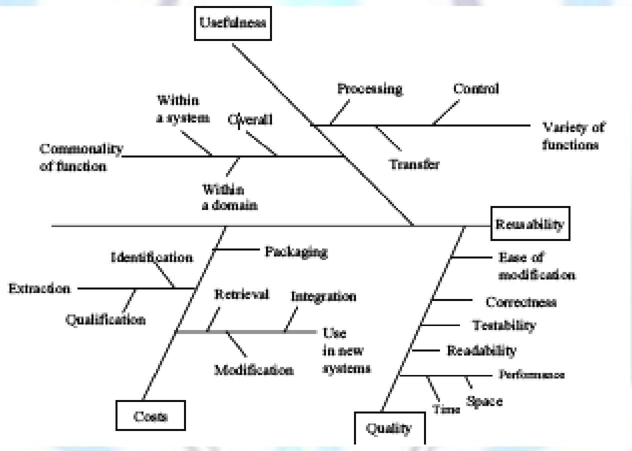

Figure 8. Factors affecting reusability

In figure 8, Factors affecting reusability are depicted.

AOP complements Object-Oriented paradigm by providing a better means of addressing the well-known problems of the separation of concerns. The programming codes which cannot be encapsulated in modules/objects but remain scattered throughout the code are the limitation of OOA. Examples of such codes are exception handling, security, synchronization etc. To overcome these limitations the concept of aspect has been introduced. AOSD approach has significant features[3]:

1) In AOSD, crosscutting concerns are removed from the modules and implemented separately as aspects, which are modular units designed to implement concerns.
2) AOSD provides a mechanism to weave the aspects with core modules by using aspect weaver.
3) This approach implements reusability, as aspects can be reused.
4) By reducing code tangling (scattered code being repeated in many places), it makes it easier to understand the core functionality of a module.

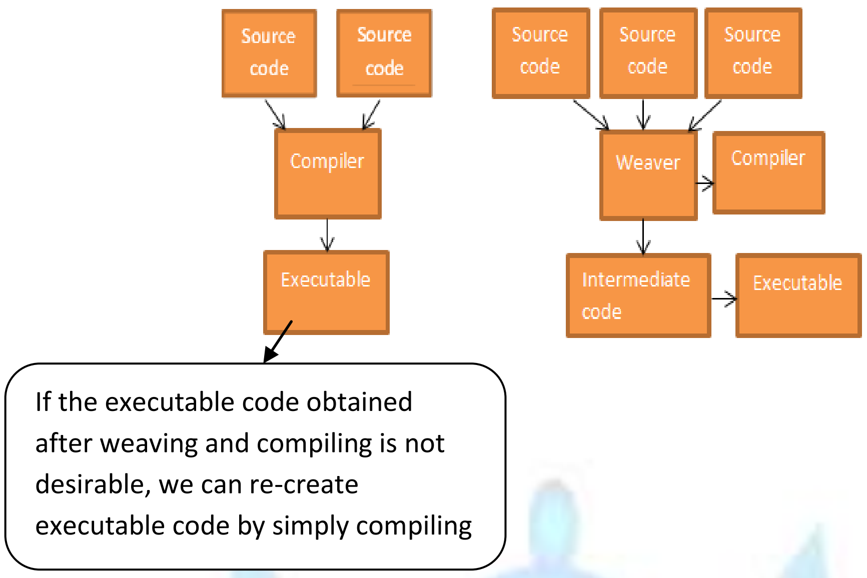
Source code
Source code
Compiler
Executable
Source code
Source code
Source code
Weaver
Compiler
Intermediate code
Executable
If the executable code obtained after weaving and compiling is not desirable, we can re-create executable code by simply compiling

Figure 9. Aspect Weaver[5]

The aspect description languages that define aspect modules uses new language construct which cannot be processed by traditional programming language compilers. Before the compiler can produce a binary executable file some pre-processing is needed. A special language processor called an aspect weaver is used to coordinate the co-composition of the aspects and components. The aspect weaver can then weave the aspects into the components and generates an intermediate source program which can be processed by a regular compiler. In Figure 10 Aspects and components are considered from the problem domain, thereafter an aspect weaver merges them into an application .Weaving is required as the aspects are defined with an aspect description language that cannot be processed by standard compilers.

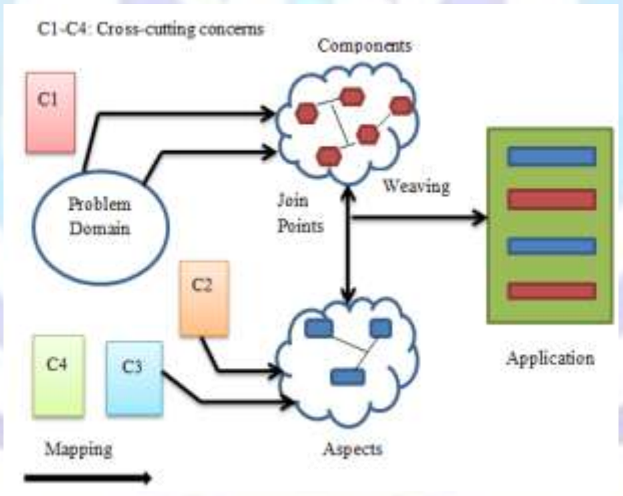

Figure 10. Aspect-oriented software development [7]

ISSN 22773061

Modularity: AOP addresses each concern separately with minimal coupling resulting in modularized implementations even in the presence of crosscutting concerns. Such an implementation produces a system with less duplicated code.

Understandability: Modularized implementations results in a system will increase understandability.

Maintainability: In AOP, crosscutting concerns are modularized in one unit called aspect. It is easier to modify code written in one module than code scattered in multiple modules such as in OOP.

Adaptability: AOP is not replacement of MOA and OOA but complement these approaches, so AOP is easily adaptable.

AOP provides some of OOP's advantages, such as functionality encapsulation, inheritance, and modularity but AOP can overcome an important disadvantage of OOP. In OOP, objects do not deals with crosscutting issues that are not confined to a single class, module. AOP plays a very crucial role in this regard. AOP implements reusability by providing a mechanism, which programmers can use to write code representing crosscutting concerns once. Programmers can write references to aspects at join points. These references then call the required aspects. The compiler reads the references and weaves the aspects into the application. The compiler also combines the separate aspect descriptions into an executable form.

Aspects eliminate scattered lines of code that otherwise would have demanded from programmers considerable amount of time in writing, changing and maintaining code for applications. This makes changing and updating applications more accurate.

In aspect-oriented approach, code related to crosscutting concerns is modularized in an aspect, which earlier was scattered in primary concerns (classes). Later this code can be integrated with the classes that must support this policy by a process known as weaving. Weaving injects the code of an aspect into well-defined locations (joinpoints) in to the syntactic structure of a primary concern. This is called reusability of aspect code.

AOP also facilitates reusability because it is modular. That is, it enables programmers to use aspect modules wherever necessary in an application without rewriting code.

OOP allows for the encapsulation of data and methods specific to the goal of a specific object. The goal of the class is to fully encapsulate the code needed for the concern. Unfortunately, this isn't always possible. Consider the following two concerns:

Concern 1: The system should keep a price related to the wholesale value of all products.

Concern 2: Any changes to the price should be recorded for historical purposes.

The first concern dictates that all products in the system must have a wholesale price. In the object-oriented world, a Product class can be created as an abstract class to handle common functionality of all products in the system

```
public abstract class Product
{
private real price;
Logger loggerObject;
        Product()
        {
                price=0.0;
```

```
            loggerObject= new Logger();
        }
        public void putPrice(real p)
        {
            loggerObject.writeLog("Changed Price from"+price+"to"+p);
            price=p;
        }
        public int getPrice()
        {
            return price;
        }
}
```

The Product class satisfies the requirement in concern 1.Now let's consider concern 2, which requires that all operations involved in changing the price be logged. This concern does not conflict with the first concern and is easy to implement. The following class defines a logging mechanism:

```
public class Logger
{
        private OutputStream ostream;
}
        Logger()
        {
        }
        void writeLog(String value)
        {
        }
```

As a result, a new Product class emerges:

```
public abstract class Product
{
        private real price;
        Logger loggerObject;
        Proct()
        {
             price=0.0;
             loggerObject = new Logger ();
        }
```

```
public void putPrice(real p)
{
        loggerObject.writeLog("Changed Price   from"+price+"to"+p);
}
public int getPrice()
{
        return price;
}
public void persistIt()
{
}
```

This results in tangled code. AOP provides the solution to this problem by seperating concern into separate module called Aspect.

```
define aspect Logger
{

Logger loggingObject= new Logger();
        when calling set*(taking one parameter)
        {
        }
loggingObject.writelog("called set method");
}
```

The aspect can be compiled along with the component Product class using a compiler provided by the AOP system.

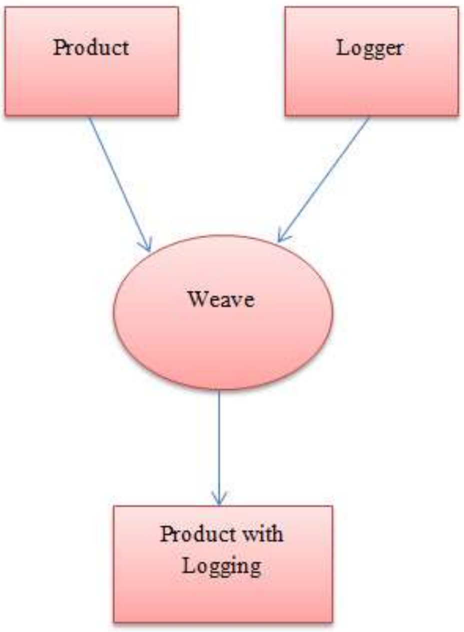

Figure 11. Graphical illustration of weave[1]

Conclusion

Maintenance is the important phase in software development lifecycle which comes after it has been delivered to the user as it contributes significantly to development effort. It has been seen that more than three-quarters of the total time spent on a particular program is on evolution and rest of the time on modifications, such as fixing bugs. It is therefore necessary to write the program in such a way that it not only works, but also become easier to modify to produce new, improved versions. It may be possible that the person who wrote the program may not be maintaining it. So it may be coded in such a way that someone else can easily understand and modify it. In figure 8, it has been observed that maintainability is one of the sub factors influencing reusability. Hence assessment of reusability is more preferable as compared to assessment of maintainability as a sole factor. In context with Aspect-Oriented system as depicted in figure 7, assessment of maintainability is confined to one of assessment of external characteristics which is insufficient to assess reusability. As maintainability comes into picture as the software has been deployed, better design & development of the software is an activity of greater priority than focusing only on assessing maintainability, to ease and enhance the development process reusability of component is an essential factor to be considered. Amongst several components available to be reused, the decision of choosing the right component for reusability is subjected to its empirical assessment, in which maintainability is one of the contributing factors. Hence assessment of reusability is highly preferable over assessment of maintainability.

REFERENCES

[1] J.Gradecki and N.Lesiecki. *Mastering AspectJ: Aspect-Oriented Programming in Java,* Wiley, 2003

[2] K.Arora, A.Singhal and A.Bansal, Correlation between Various Quality Characteristics for Aspect-Oriented Systems, *International Journal of Computer Applications,* pp: 11-21,2012

[3] P.Grover, R.kumar and A..Kumar, Assessment of Reusability in Aspect-Oriented Systems using Fuzzy Logic, *ACM SIGSOFT Software Engineering Notes* 35(5), pp: 1-5, 2010

[4] R.Fitzpatrick, Software Quality:Definitions and Strategic Issues, Staffordshire University, School of Computing Report,1996

[5] P.Grover, R.Kumar,and A.Kumar, A Survey of Current Popular Software Development Methodologies: Module-Oriented, Object-Oriented and Aspect-Oriented, *International Conference of Software Engineering Research and Practice,*2006

[6] C.Sant'anna, A.Garcia, C.Chavez, C.Lucena, and A.Staa , On the Reuse and Maintenance of Aspect-Oriented Software: An Assessment Framework In: *Proceedings XVII Brazilian Symposium on Software Engineering,* 2003

[7] J.Herder, Aspect-Oriented Programming with AspectJ,2002

[8] J. Chen and X. Liu, “Software Maintainability Metrics Based on the Index System and Fuzzy Method”, *1st International Conference on Information Science and Engineering(ICISE2009)*

[9] IEEE, “IEEE Standard Glossary of Software Engineering Terminology,” IEEE Std 610.12-1990, 1990.

Author’ biography with Photo

Ritika Chaudhary has received her B.Tech degree from Kurukshetra University in 2010. Currently she is pursuing M.Tech from Manav Rachna College of Engineering affiliated to MDU Rohtak. Her area of interest includes aspect systems and software testing.

Ram Chatterjee is working as an assistant professor in MRCE(CSE Department).He has obtained MCA, M.Tech. (CSE) from CDAC, Noida. His Area of Specializations is Computer Science and Engineering. He has published more than 7 research papers in international journals and conferences of repute. He can be reached by e-mail at: ramchatterjee.mrce@mrei.ac.in