Using Aspect Oriented Techniques to Build-in Software Quality

Obeten O. Ekabua
Department of Computer Science
North-West University Private Bag X2046
Mmabatho 2735, South Africa.

Abstract - Today's software systems are growing rapidly in number, size, complexity, amount of distribution and number of users with the evolving technologies being geared towards improving their quality. Aspect oriented software development is a new paradigm that claims to improve the quality of software using separation of concerns. In this paper we show how the aspect oriented paradigm has evolved from the object oriented paradigm, giving definitions of key aspect oriented terms to aid comprehension and clarification. We then show how techniques involved in aspect oriented design can help to improve software quality.

Keywords: Aspect, Weavers, Crosscutting, Object Oriented Paradigm, Point_Cuts, Advice

1. Introduction

Programming languages and systems evolutions from the crude assembly and machine codes of the earliest computers, through concepts such as formulae translation, procedural, structured, functional, logic programming and programming with abstract data types has been experienced in computer Science. These steps in programming technology has helped to achieve clear separation of concerns at the source code level and introduce some form of quality to the next level of programming. In recent times, the focus has been on object-oriented programming: where software systems are built by decomposing a problem into objects. A clever idea, but with limitations which affect quality measurement [1].

One of the latest concepts in programming today is Aspect-Oriented Programming (AOP) which is used primarily in academia and research and development organisations, but making waves into mainstream development. Its use is an evolutionary way of developing software that improves upon object-oriented programming (OOP), in the same way that OOP improved upon procedural programming. OOP introduced the concepts of encapsulation, inheritance, and polymorphism for creating a hierarchy of objects that model a common set of behaviors. Although OOP has become so relevant today, yet it has failed in handling common behaviours that extends across unrelated objects. That means OOP enhances a vertical relationships but not horizontal relationships. As an example, logging code is often scattered horizontally across object hierarchies but has nothing to do with core functions of the objects that it is scattered across. The same is true for other types of code such as security and exception handling. This scattered and unrelated code is known as crosscutting code and is the reason for AOP's existence: a typical quality issue, implying that AOP intends to introduce some form of quality to OOP concepts. AOP provides a solution for abstracting crosscutting code that spans object hierarchies without functional relevance to the code it spans. AOP is a tool that allows you to abstract the crosscutting code into a separate module known as an aspect, rather than embedding crosscutting code in classes and then apply the code dynamically where it is needed. Applying the crosscutting code dynamically is achieved by defining specific places, known as pointcuts in the object model where crosscutting code should be applied. Depending on the intended AOP framework, crosscutting code is injected at the specified pointcuts at runtime, or compile time. Ideally, AOP introduces a very powerful concept which allows the introduction of new functionality into objects without the objects needing to have any knowledge of that introduction. [2]. One of the basic truth that we must accept while considering AOP as an option is the fact that, whenever the original source code of a programmer is modified, his intent and assumptions degrade [3].
Defects and deterioration of software are cause by changes in source code and a lot of these changes cannot be avoided, but can be minimized. When changes are made to software, in most cases, the entire program is reengineered [4].

Today's software developers are greatly attracted to the idea of AOP and have recognized the concept of crosscutting concerns and problems associated with the implementation of such concerns. They also have been pondering on how to adopt AOP into their development processes. They have shown interest in knowing how aspects can be applied in an existing code and what kind of benefits applying this aspect would yield? They have also been interested in knowing how the performance slope for AOP would look like. Answers to these interest is express in AspectJ which is a general-purpose extension of Java [5]. The most interesting concern here is that, AOP technology makes it easier to write and change certain concerns [6].

In this paper, section 1.0 briefly introduced the basic idea of AOP as it relates to software development and quality while section 2.0 explains the key terminology used in AOP concept. This is followed by section 3.0 which discusses various design approaches that are used by AOP to differentiate it from its OOP counterpart. Section 4.0 explains how AOP can be used to build quality into software by improving its modularity and points out problems associated with scattered code. The section also explains how AOP introduces a new module called aspect to handle this code. Finally, section 5.0 concludes by saying that AOP promises to be a powerful tool with tangible benefits and also say where further work is expected.

2. AOP key Terminologies

Understanding AOP terminology is essential to appreciating the ideas behind AOP. The fundamental concepts are described and explained in this section

Aspects – an aspect is defined as a special kind of concern or non-functional element of code arising from a post object implementation. An aspect is a concern whose functionality is triggered by other concerns usually in multiple forms. It packages advice and pointcuts into functional units in much the same way that OOP uses classes to package fields and methods into cohesive units. A concern is any code related to a goal, feature, concept, or "kind" of functionality. For instance, there may be a logging aspect that contains advice and pointcuts for applying logging code to all setter and getter methods on objects. Aspects are not just a neat trick or careful technique for adding logging or synchronization or other simple functionality [7] to source code. Aspects are a natural evolution of the object-oriented paradigm which provides a solution to some difficulties encountered while modularizing object-oriented code. In most cases, functionality does not work, and the same lines of source code are repeated in many different object-oriented classes because those classes each need that functionality. It cannot easily be wrapped up in a single place. Instances of this kind of code are found in audit trails, transaction handling, and concurrency management, such code can now be modularized using aspects.

Pointcuts: **-** Point_cuts typically define the points in a model where advice will be applied. For example, point_cuts define where in a class, code should be introduced or which methods should be intercepted before they are executed [7]. Point_cuts are also known as join_points. A method call join point is the point in the flow when a method is called, and when that method call returns. Each method that calls itself is a join_point. The lifetime of the join_point is the entire time from when the call begins to when it returns (normally or abruptly), but execution is at the join_point only at the moment the call begins and the moment it returns. A pointcut therefore describes a point in the execution of a program where crosscutting behaviour is required [8].

Advice: **-** Advice is code that crosscuts or is applied to the existing model and is commonly referred to as a mix-in. Advice code [7] modifies the behaviour or properties of an existing object. In AspectJ, advice is the implementation of behaviour that crosscuts the set of execution points defined by the poincut. This makes advice an obvious construct to use to implement the sequence of behaviors defined in sequence diagrams in the crosscutting themes [7]. Advice can be defined to execute before, after, or around the execution points defined by pointcuts. It is possible for the programmer to decide to add advice before a join_point runs or after it finishes running, and can also force advice to run instead of the join_point. Whenever the point at which an aspect adds a behaviour is defined, the behaviour to be added must also be defined [9].

Weavers:- The aspect weaver accepts the component and aspect programs as input, and emits in some cases, a C program as output. The weavers job is integration, rather than inspiration [10]. Aspect weavers work by generating a join_point representation of the component program, and then executing (or compiling) the aspect programs with respect to it.

3. Aspect Oriented Quality Design Approaches

Aspect Oriented Quality Design approaches is requirement engineering techniques that recognise the importance of addressing both functional and non-functional crosscutting concerns and the non-crosscutting ones. The need for the development of a new technology, composition and traceability prompted the emergence of this approaches. Therefore, AO quality design approaches focus on systematically and modularly treating, reasoning about, composing and subsequently tracing crosscutting functional and non-functional concerns via suitable abstraction, representation and composition mechanisms tailored to the requirements engineering domain.[11]. There are different kinds of approaches that are used in aspect oriented design which are intended to bring out the salient differences of AOP and OOP and to explain quality design approach in building AO software. Amongst these approaches [12] are:

Aspect-Oriented Design Modelling- AODM extends standard UML with aspect oriented concepts. For AspectJ implementation language, AODM-UML extention is originally defined to support aspect oriented concepts. However, AODM has evolved to become more generic and now supports other asymmetric AO Programming approaches (such as composition filters and ad aptive programming).Symmetric and asymmetric approaches are levels of concern separation supported by existing AOD approach. In general, the distinction relates to whether an approach includes a means to separate all kinds of concerns, both crosscutting and non-crosscutting (symmetric) or includes a means to just separate crosscutting concerns from the rest of the system (asymmetric) [11]. Yet, symmetric AO implementation techniques are not naturally supported by AODM. This AODM has various artefacts such as:

AOD Language – which was originally designed to model AspectJ-style AOP, its design elements are still heavily related to AspectJ. When discussing design methods that target various AOP platforms, AspectJ terminology is a major interest.

Aspects Specification – This represents aspects as classes with the <<aspect>> stereotype and was adopted because of the structural similarities between aspects and classes. Like classes, aspect act as containers and namespaces for attributes, operations, pointcuts, advice and intertype declarations. Aspects can also engage in the same association and generalization relationships as classes. Aspects differ from classes in their instantiation and inheritance mechanisms. In AspectJ, aspect declaration can contain instantiation clauses that specify the way in which the aspect should be instantiated. Child aspects inherit all features from their parents' aspects but only abstract pointcuts and java operations can be overridden. This stereotype augments the Meta class with some additional meta-attributes to hold the instantiation clause, and a boolean expression which specify whether the aspect is privileged or not [11].

Crosscutting Specification – Aspect Oriented Design Method supports the specification of behavioural and structural crosscutting. Crosscutting structure is expressed in class diagrams within a parameterized template diagrams and captured in a parameterized class and partial sequence diagrams. Parameterisation is used to represent crosscutting. The type(s) to be crosscut are represented as parameter(s) to the template. The crosscutting structure is applied to this parameter. The concrete structural elements to be crosscut are applied to the template as arguments. Class and sequence diagrams dictate the manner in which integration occurs [11].

Integration Specification – Aspect Oriented Design Methods support the specification of structural and behavioural integration through integration of crosscutting design models.The type structure of a given design model is affected by structural crosscutting and these can occur at some location in a target class hierarchy. Behavioural crosscutting affects the model's behavioural specifications and occurs at some joinpoint in the execution specification [11].

Composition Semantics – Due to the close ties of composition semantics with AspectJ, details are not provided and it can be inferred that, the composition semantics followed by AODM are very similar to AspectJ.

Themes/UML Approaches Theme defined:
Theme is an approach used to support the separation of concerns during the analysis and design phases of software lifecycle and provides a UML based AOD language called Theme/UML which extends the UML meta-model. The Theme approach expresses concerns in conceptual and design constructs called themes which are more general than aspects and more closely encompass concerns with relation to the symmetric separation. Any concern, whether crosscutting or not, may be encapsulated in a theme. The theme approach involves identifying and designing separate *themes*, which are then combined to make a whole system. Themes can be thought of as analogous to features or concerns. So, in a sense, the Theme approach lets you design each feature of your system separately, and then offers a way to combine them. Themes are useful in Analysis, Design and Composition. Themes retangled after composition and they "look" different as you move from requirements to design [7]. Themes are participant and actions described in a set of requirement and are fairly abstract. This part is more concrete at the design level and consists of classes, methods and the relationship between them and is both structural and behavioural: the "thing" and "actions" that work together to provide a feature of your system.

Relationship between Themes
Understanding the idea of themes in building quality software would be more useful if the relationship that exist between themes are clearly explained.
Themes are two-way related: by *sharing concepts* and by *crosscutting.* At the same time and in rare cases, it is possible for themes to be independent – that means relating in no way to the system.

Sharing Concepts: **-** Different themes may have design elements that represent the same core concepts in the domain. A design element might be a particular class, method, or attribute and a concept relates to something from the domain. Considering the banking system for example, the core concept might be a bank account and the concept of a bank account can be described by classes of varying names, such as *Account*, or *BankAccount.* Each theme contains specifications for those same concepts designed from the perspective of the theme. Another way of looking at concept sharing is to think of it, at some level, as "structure" sharing. Strictly speaking, though, we don't consider that themes actually "share" structure because each team will have its own version as appropriate to the feature under design. Encapsulation in this manner has the benefit of locality, where only and all relevant functionality for a concern is present in a theme. Take, for example, the **transfer funds**, **apply charges**, and **apply interest** features in a banking system. Each of these three features works with Account and all three work with *Checking* and *Savings* account, while two of them also work with Loan accounts [7].

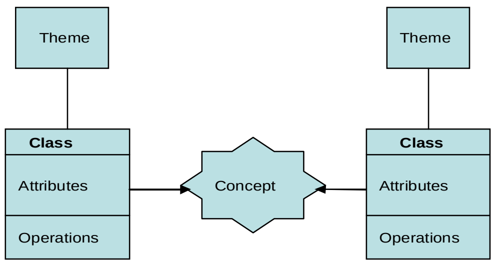

Figure 1: Themes related by concept sharing Source: Aspect Oriented Analysis and Design [7]

Crosscutting: **-** This type of relationship is called asymmetrical crosscutting**-** where behaviour in one theme can be triggered by behaviours in other themes. Here there are *base* themes, *crosscutting* themes and *aspect* themes. The last two are used synonymously and are always themes that have their behaviours triggered in tandem with behaviours in other themes. Aspects in the Theme approach are the same as in the asymmetric separation approach. Base Themes are those that trigger aspect themes. They might be themes that share concepts with other themes, and they might be aspects themselves and have their own base [7]. Sometimes there is a base theme which is the result of the composition of other themes to which an aspect can be applied (fig 2):

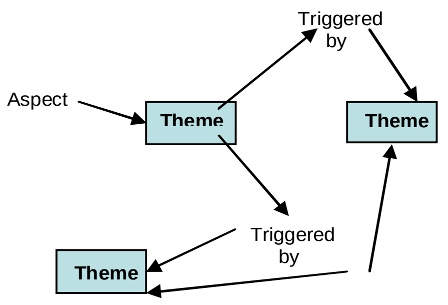

Figure 2: Themes related by crosscutting: The base theme triggers the aspect themes. Source: Aspect Oriented Analysis and Design [7]

4. Using Aspect Oriented Programming to Build in Quality

Software engineers are interested in looking for ways to improve modularity in software and these makes software easier to develop and maintain as it improves its overall quality. Recently, aspect-oriented programming emerged as a new technique with the idea to cleanly separate the implementation of crosscutting concerns (requirements and design elements that affect multiple modules). AOP is an evolutionary step that improves the implementation's comprehensibility and simplifies incorporating new requirements as well as changes to existing ones. This systematic approach promises direct mapping of requirements and design intentions to the implementation. Such a mapping, traces the reasoning behind a piece of code's existence. Efficiently implementing crosscutting concerns also has an important indirect benefit: It frees resources to concentrate on the quality of the core implementation. All these factors combine to improve software quality [13].

4.1 AOP's direct applications to quality improvement

AOP's idea goes beyond just modularizing crosscutting concerns from the design and implementation perspectives, but also enhances software quality through the following methods:

General system policy enforcement. This is achieved by creating reusable aspects that would help to enforce different contracts and provide guidance in following "best" practice [14]. As an example, the enterprise JavaBeans specification contains 600 pages long and specifies many restrictions for programmers to comply. The required enforcement level cannot be achieved with diligence.

Enabling logging functionality. A better QA process can be enabled while the QA person can augment the bug report with the associated log thereby giving developers a better chance of reproducing the behaviour. Being able to reproduce a bug is a major frustration for many software engineers and the understanding of system behaviour, nonintrusive logging and tracing functionality serves as a useful tool. Systematic implementation rarely occurs since the conventional implementation of the equivalent functionality is cumbersome that [13].

The use of virtual mock objects. Virtual mock objects are interesting useful for testing software quality. Complex scenarios are not always tested due the amount of effort required. If mock objects are combined with AOP, it serves as an interesting act to introduce the mock objects without making any changes to the core system. Fault injection into the system becomes an easy task to check the response of the system. Such testing is made easy and practical without compromising the core design for the testability. [13]

Consideration of conditional decision. Deciding on whether functionality is necessary or not is a time and space trade-off which depends on running the pooling and catching optimization. AOP provides a nonintrusive way to perform such analysis without worrying about accidental changes to core behaviour [13].

The deployed systems don't require the use of AOP as it is enough to use an AOP-enabled system during development and testing. The features can be removed by excluding them from the build system. It has become very important to know that AOP has come as a systematic approach to handle the presence of crosscutting concerns and their negative implications for software quality in traditional implementations. While maximising its benefit, more insights in the practice and design patterns of AOP are being gained as a common technique for modularizing crosscutting concerns. However, AOP cannot solve all software quality issues and therefore, traditional quality-management techniques would continue to be useful. AOP is seen as a powerful tool which has tangible benefits with regards to implementing crosscutting concerns. Within the nearest future, the implications of using AOP would be realized since it has such great benefits than its potential risks [13].

4.2 Quality Characteristic of Aspect-Oriented Programming

Aspect-Oriented Programming [15] is an improved programming paradigm from procedural programming and Object Oriented Programming. OOP allows an object to implements part of system's functionality by interacting with each other via messages (or method calls) to achieve the system's goal. Although procedural and object-oriented languages are extremely useful, modularizing crosscutting concerns such as logging and synchronization becomes a major problem. Why program code scatter among objects is because, concerns are implemented as an interaction of related objects. The scattered code would normally cause the following problems:

- As a specification of a crosscutting concern is changed, programmers must modify all related objects and this issue can occur recursively.
- Difficulties are encountered in reusing an object independently since objects are connected to each other with a crosscutting concern.
- Difficulties are encountered while reusing implementation of a concern independently of objects. If objects interact in the same way, the concern must be re-implemented by the programmers

The encapsulation of a crosscutting concern by a new module unit named ‘aspect’ is a fresh idea of AOP. This is to mean that, one concern can be written in a single aspect. An aspect consist of some advices and an advice is a method-like unit consisting of a procedure and a condition used to execute the procedure. A condition to execute an advice is specified by a pointcut. A pointcut is defined by a subset of join points, which are well-defined events during program execution, such as method calls and field accesses. Using join points, a programmer can separate crosscutting concerns from objects [15]. Modularized crosscutting concerns have good maintainability and reusability.
Some join points examples are:

- A method call to an object,
- A method execution of an object after dynamic binding,
- A field access of an object, and
- Exception handling.

Advices are linked to objects by three types of forms: *before* (immediately before join points), *after* (immediately after), and *around* (replacement of join points). Advices can access runtime context information, for example, a called object, a caller object, and parameters of a method call.

A sample code of aspects is shown in figure 3 below:

```
Class someclass {
        Public void doSomething (int x)  { … }
                }
Aspect loggingAspect {
                before ( ): call ( void
SomeClass.doSomething ( … ) )  {
                } }
                Aspect ParameterValidationAspect  {
                    before (int x)
args (x)  && call (void *.doSomething ( … ) )   {
if ( (x < 0) || ( x >
constants.x_MAX_FOR_SOMETHING ) ) {
   throw new RuntimeException ( “invalid
parameter!”) ;
} } }
```

Figure 3: Aspect examples: Logging and parameter checking

Source: Debugging Support for AOP based on Program Slicing and Call Graph [15]

Considering a sample code of aspect in figure 3, *LoggingAspect* logs a method call to *SomeClass.doSomething*. Whenever the method name is *doSomething*, *ParameterValidationAspect* would validate all method calls and if the validation fails it throws an exception. For this particular example, whenever the specification of the parameter validation is changed, programmers would change only the aspect instead of all callers of *doSomething*. Also, both aspects are executed when *SomeClass.doSomething* is called. During such a case, a compiler (or an interpreter) serializes the advices being executed. For AspectJ, programmers normally write the precedence of aspects to adjust the execution sequence of advices.
Different AOP applications are in use. For OOP, design patterns are design components which describe the interaction of objects. [16]. Interaction of objects is said to be a kind of crosscutting concern; therefore programmers can afford to write a pattern as an aspect. Aspects which implement design patterns are reusable components [17]. More so, it is useful for applications to support debugging and to write crosscutting concerns in a distributed software environment [18, 19].

5. Conclusion and Further Work

Presented in this paper is how to build quality in software systems using Aspect Oriented Programming techniques. The question of what are aspects and how they can be useful in separation of concerns is answered by this new technology. AOP is an evolutionary move for building more reliable software and encapsulate all sorts of robustness features that were previously difficult to handle. A revolution in quality is needed and now is the time to start the battle. The tool and technique to manage quality is now within grasp as offered by AOP.
To ease clear understanding, this research paper has defined key terminology of AOP and explained quality design approaches of aspect oriented programming.
Not only modularizing crosscutting concern from the design and implementation perspectives as discussed by the paper, but it also identifies specific applications that improve software quality and gives a sample code that shows the characteristics of aspect oriented programming.
For future work, more is expected in the area of impact analysis, security, ripple effect computation, testing and the reusability of aspects.

6. References

1 Elrad T, Filman R, and Bader A: Aspect Oriented Programming. Communication of the ACM, pp.29-32, October 2001/vol.44, No 10
2. Holmes J: “Taking Abstraction a step further”.

3 Alexander, R.: The Real Cost of Aspect Oriented Programming. IEEE Software Society, pp 91,93. Nov/Dec. 2003

4. Fayad M and Adam A.: An Introduction to Software Stability. Communications of the ACM, pp 95-98, Sept. 2001/Vol44, No 9.

5. Gregor K, Erik H, Jim H, Mik K, Jeffrey P and William G.: Getting Started With AspectJ. Communications of the ACM, pp59-65, October 2001/Vol. 44, No. 10.

6. Andres Diaz Pace J and Marcelo R.: Analysing the Role of Aspects in Software Design. Communications of the ACM, October 2001/Vol. 44, No. 10.

7. Clark S and Baniassad E: Aspect-Oriented Analysis and Design – The Theme Approach. Addison-Wesley, march, 2005.

8. Karl L, Doug O. and Johan O.: Aspect Oriented Programming with Adaptive Methods. Communications of the ACM, October 2001/Vol. 44, No. 10

9. John V. and Jeffrey V.: Can Aspect Oriented Programming Lead to More Reliable Software?. IEEE Software Society, pp 19-21, Nov/Dec 2000.

10. Henry S and Kafura D: Software Structure Metrics Based on Information flow in IEEE Transaction on Software Engineering, Vol. SE-7: pp509-518, 1981.

11. Soares S, Laureano E and Borba P: Implementing Distribution and Persistence Aspects with AspectJ. In Proc. of OOPSLA 2002, pp.174-190, November 2002.

12. Stein D, Hanenberg S and Unland R: “A UML-based Aspect Oriented Design Notation for AspectJ”, AOSD 2002, Erischede, The Netherlands, 2002.

13 Ramnivas L: Aspect-Oriented Programming will Improve Quality. IEEE software computer society, pp. 90,92 2003.

14. Laddad R: AspectJ in Action-Practical Aspect-Oriented Programming, Manning Publications, 2003.

15. Ishio T, Kusumoto S and Inoue K: “Debugging Support for Aspect-Oriented Program Based on Program Slicing and Call Graph”, Proceedings of the 20th IEEE International Conference on Software Maintenance ICSM 2004.

16. Gamma E, Helm R, Johnson R and Vlissides J: “Design Patterns: Elements of Reusable Object-Oriented Software, Addison Wesley 1995.

17. Hannermann J and Kiczales G: Design Pattern Implementation in Java and AspectJ.
In Proc. of OOPSLA 2002, pp. 161-173, November 2002.

18. Ishio T, Kusumoto S and Inoue K: Program slicing Tool for Effective Software Evolution Using Aspect Oriented Technique. In Proc. of IWPSE 2003, pp. 3-12, September 2003.