Impact of Aspect Orientation on Object Oriented Software Metrics

Arti Chhikara
Maharaja Agrasen College, Delhi, India.
Email id: phd.aarti@gmail.com

R.S.Chhillar
Deptt. Of Computer Sc. And Applications, Rohtak, India.

Abstract: Object Oriented Paradigm has become quite popular over recent years and has completely replaced the Procedural Oriented Paradigm. Object-Oriented Programming (OOP) encourages software reuse by providing design and language constructs for modularity, encapsulation, inheritance and polymorphism. Although OOP has met great success in modeling and implementing complex software systems, it has its limitations. The main limitation of this approach is that there are some system functionalities that cannot be assigned to a single module in the system decomposition. Aspect Oriented Paradigm (AOP) is one of the emerging paradigms that promise to enhance software design and promotes reuse. This research paper focuses on several object oriented metrics and how the introduction of Aspects affects these metrics.

Keywords: Aspect, Aspect Oriented Software Development, Object Oriented Paradigm

1. Introduction:

Object Oriented Paradigm has become quite popular over recent years and has completely replaced the Procedural Oriented Paradigm. Object-Oriented Programming (OOP) encourages software reuse by providing design and language constructs for modularity, encapsulation, inheritance and polymorphism. However, Object Oriented abstractions currently are recognized as being unable to capture all concerns of interest in software system [1,2]. Many important concerns often crosscut several objects and classes of object-oriented software system.

The development of software using Aspect Oriented Paradigm is a relatively new software engineering paradigm. Aspect Orientation technique is built on Object-Orientation and addresses some of the pitfalls that were not addressed by OO. Aspect Oriented Programming (AOP) provides explicit constructs to develop software systems whose crosscutting concerns are better modularized because they are no longer tangled together and they are clearly separated from the principal decomposition. This research paper determines the effect of AO on software metrics.

The rest of the paper is organized as follows. Section 2 presents basic literature research of AO metrics. Section 3 presents the Morris metrics suit, explaining the meaning of these metrics and the rationale behind them. Section 4 explains the impact of aspect orientation on Morris metric suit. Section 5 presents the concluding remarks and future scope.

2. Literature Research

2.1 Aspect Oriented Paradigm

Aspect Oriented Software Development (AOSD) is a programming paradigm in which crosscutting concerns that pervade a system are isolated and extracted into separate modules called aspects. AOP aids a better handling of crosscutting concerns as, compared to object-orientation [3]. Concurrency, security, logging, debugging and fault tolerance are examples of such concerns. These concerns are called crosscutting concerns. Crosscutting concerns violate the modularization goal that a system is decomposed into small independent parts: their main characteristic is that they are transversal with respect to the units in the principal decomposition, i.e., their implementation consists of a set of code fragments distributed over a number of units.

Reasoning about crosscutting concerns can be quite difficult, because it requires to deal with a lot of modules at the same time, in that there is no modularization support for them[6,7]. For example, to deal with the persistence functionality, we have to understand all the pieces of code that perform persistent storage and retrieval. Crosscutting concerns are sources of problems, because their modification requires that [6,7]:

- all code portions where such a functionality is implemented must be located (problem: scattering);

- all ripple effects associated with the changes must be determined (problem: tangling).

AO technology attempt to modularize these scattered implementations in order to avoid the code-tangling phenomena associated with it [4,5]. This approach implements reusability, as aspects can be reused and by reducing code tangling, it makes it easier to understand what the core functionality of a module is.

2.2 Aspect Oriented Software Metrics:

Software Metrics are measurement tools for measuring some properties of a piece of software or its specifications. Metrics for aspect-orientation is needed to ensure that AO really accomplishes its objective of enhancing software design and providing better software designs. Several object oriented metrics have been proposed but these metrics are not adequate to capture all the features of aspect-oriented software as mentioned in many papers [7,8,9]. AOSD introduces new abstractions and complexity dimensions to software engineering. As a consequence, new metrics that can account aspects features must be developed to assess aspect system functionalities.

Although AOP is a relatively new paradigm, some research touching metrics and quality has already been conducted. For instance Ceccato and Tonella[10] have proposed a metric suit for aspect oriented paradigm. They revised the popular metric suit proposed by Chidamber and Kemerer[11] and proposed 10 new metrics:

1. Weighted Operations in Module(WOM)
2. Depth of Inheritance Tree(DIT)
3. Number OF Children(NOC)
4. Crosscutting Degree of an Aspect(CDA)
5. Coupling on Advice Execution(CAE)
6. Coupling on Method Call(CMC)
7. Coupling on Field Access(CFA)
8. Coupling Between Modules(CBM)
9. Response For a Module(RFM)
10. Lack of Cohesion in Operations(LCO)

Sant'Anna et al[12] also proposed a coupling metric suit for Aspect Oriented Systems. Zhao[13] proposed a metric suit that is based on dependency model for aspect oriented software. He used the dependency graph to measure the some aspect oriented mechanisms that are not measured individually.

3. Metrics for OO Software Development Environments

The metrics suit that is used in this paper is the one that is proposed by Morris [14] in his master's thesis.

In his master's thesis Morris [14] made some important observations on OO code and proposed candidate metrics for productivity measurement:

Methods per Class

Average number of methods per object class = Total number of methods / Total number of object classes

- larger number of methods per object class complicates testing due to the increased object size and complexity
- if the number of methods per object class gets too large extensibility will be hard
- A large number of methods per object class may be desirable because subclasses tend to inherit a larger number of methods from superclasses and this increases code reuse.

Inheritance Dependencies

Inheritance tree depth = max (inheritance tree path length)

- Inheritance tree depth is likely to be more favorable than breadth in terms of reusability via inheritance. Deeper inheritance trees would seem to promote greater method sharing than would broad trees
- A deep inheritance tree may be more difficult to test than a broad one
- Comprehensibility may be diminished with a large number inheritance layers

Degree of Coupling between Objects

Average number of uses dependencies per object = total number of arcs / total number of objects

arcs = max (number of uses arcs) - in an object uses network

arcs - attached to any single object in a uses network

- A higher degree of coupling between objects complicates application maintenance because object interconnections and interactions are more complex.

- The higher the degree of uncoupled object the more objects will suitable for reuse within the same applications and within other applications.
- Uncoupled objects should be easier to augment than those with a high degree of 'uses' dependencies, due to the lower degree of interaction.
- Testability is likely to degrade with a more highly coupled system of objects.
- Object interaction complexity associated with coupling can lead to increased error generation during development.

Degree of Cohesion of Objects
Degree of Cohesion of Objects = Total Fan-in for All Objects / Total No. of Objects

- Low cohesion is likely to produce a higher degree of errors in the development process. Low cohesion adds complexity which can translate into a reduction in application reliability.
- Objects which are less dependent on other objects for data are likely to be more reusable.

Object Library Effectiveness
Average number = Total Number of Object Reuses / Total Number of Library Objects

- If objects are actually being designed to be reusable beyond a single application, then the effects should appear in object library usage statistics.

Factoring Effectiveness
Factoring Effectiveness = Number of Unique Methods / Total Number of Methods

- Highly factored applications are more reliable for reasons similar to those which argue that such applications are more maintainable. The smaller the number of implementation locations for the average task, the less likely that errors were made during coding
- The more highly factored an inheritance hierarchy is the greater degree to which method reuse occurs
- The more highly factored an application is, the smaller the number of implementation locations for the average method

Degree of Reuse of Inheritance Methods
Percent of Potential Method Uses Actually Reused (PP):
PP = (Total Number of Actual Method Uses / Total Number of Potential Method Uses) x 100
Percent of Potential Method Uses Overridden (PM):
PM = (Total Number of Methods overridden / Total Number of Potential Method Uses) x 100

- Defining methods in such a way that they can be reused via inheritance does not guarantee that those methods are actually reused.

Average Method Complexity
Average method complexity = Sum of the cyclomatic complexity of all Methods / Total number of application methods

- More complex methods are likely to be more difficult to maintain.
- Greater method complexity is likely to lead to a lower degree of overall application comprehensibility.
- Greater method complexity is likely to adversely affect application reliability.
- More complex methods are likely to be more difficult to test.

4. Impact of Aspect Orientation on Object Oriented Metrics

In this section an analysis of Impact of Aspect Orientation on Object Oriented Metrics is provided.

1. Method per Class: Aspects combine crosscutting functionalities in modular, encapsulated units. These crosscutting functionalities would be tangled in core class, if there is no aspect oriented design. Applying aspects in programs will filter out aspectual functions and as a result number of tangled methods will get reduced. This will reduce the value of Method per Class metric.

2. Inheritance Dependencies: Subclasses that might be defined only for the purpose of applying their own implementation of aspectual behavior will not exist in systems designed using AO paradigm, because aspects will be responsible for that. As a result the value of this metric will also get reduced.

3. **Degree of Coupling between Objects:** Aspects are new entities on which core classes depend. However, it should be noted that unlike aspects core classes are more likely to be reused. So, the presence of aspects is likely to decrease the coupling between core classes, yet increase the coupling between core classes and aspect classes.

4. **Degree of Cohesion of Objects**: The presence of aspects will filter out crosscutting behavior, and therefore increases cohesion. Figure 1 is an example of this

```
Class Lock
  {
   public:
   virtual ~Lock( );  // checks if a function is locked or not
   private bool Movable()
   public void Move( );
  };
```

Figure 1: Class Lock

The function Movable( ) is likely to contain synchronization checking that determine if the function Move can be invoked on an object of type Lock. This can be seen as a synchronization aspect, which uses its own flags to determine synchronization. Such a crosscutting function reduces the cohesion of the class lock.

5. **Factoring Effectiveness**: As aspects are introduced in programs, number of total methods would also increase. As a result the value of this metric will also get increase.

6. **Degree of Reuse of Inheritance Methods:** When the aspects are introduced in systems, subclasses that might be defined only for the purpose of applying their own implementation of aspectual behavior will not exist in the system. As a result reuse of inheritance methods will also get reduced.

7. **Average Method Complexity:** With the introduction of aspects in a program, aspectual functions will get filter out and therefore the number of tangled methods will get reduced. This will reduce Average Method Complexity.

5. Conclusion and Future Work:

The problem of separation of concerns led to the emergence of aspect oriented paradigm.Aspect oriented software development is an emerging paradigm that provides new abstractions and mechanisms to support the modularization of crosscutting concerns through the software development. In this paper we presented the impact of aspect orientation on object oriented software metrics. We found that results are comparatively better with the introduction of aspects in software system. But the introduction of aspects to a system complexes the system design. So future work would be to determine the best design practices for specific aspect oriented metrics.

References:

[1] Kiczales, G. et al. "Aspect Oriented Programming". European conference on Object-Oriented Programming (ECOOP), LNCS (1241), Springer-Verlag, Finland, June 1997.

[2] Tarr, P. et al. "N degree of Separation: Multi-Dimensional Separation of Concerns". Proceedings of the 21st International Conference on Software Engineering May 29, 1999.

[3] Kiczales, G., Lamping ,J., Mendhekar, A., Maeda, C., Videira Lopes, C., Loingtier, Jean-Marc, Irwin, J.: Aspect-Oriented Programming, ECOOP, Finland, Springer-Verlang LNCS vol. 1241, pp.220-242, 1997

[4] Aldawud, Omar, Tzilla Elrad, and Atef Bader."A UML Profile for Aspect Oriented Modeling". OOPSLA 2001 Workshop on Aspect Oriented Programming.

[5] Zakaria, Aida, Hoda Hosny, and Amir Zeid. "A UML Extension for modeling Aspect-Oriented Systems". UML 2002 Workshop on Aspect Oriented Modeling.

[6] Mariano Ceccato and Paolo Tonella. Codebender: Remote software protection using orthogonal replacement. IEEE Software, 28(2):28-34, 2011.

[7] Mariano Ceccato. Migrating Object Oriented code to Aspect Oriented Programming. PhD thesis, University of Trento, Italy, December 2006.

[8] Noda, Natsuko and Tomoji Kishi. On Aspect-Oriented Design- An Approach to Designing

[9] Quality Attributes". Software Design Laboratories. NEC Corporation.

[10] J. Zhao, Coupling Measurement in Aspect-Oriented Systems, Technical Report SE-142-6, Information Processing Society of Japan (IPSJ), July 2003.

[11] Ceccato, M., Tonella P.: Measuring the Effects of Software Aspectization. WARE cd-rom (2004)

[12] Chidamber, S., Kemerer, C.: A Metrics Suite for OO Design. IEEE Trans. Soft. Eng. 20(6) (1994) 476-493

[13] Sant'Anna, C. et al.: On the Reuse and Maintenance of Aspect-Oriented Software: An

[14] Assessment Framework. In Proc. SBES (2003) 19-34

[15] Zhao, J.: Measuring Coupling in Aspect-Oriented Systems. Int. Soft. Metrics Symp. (2004)

[16] [Morr89] K. Morris, "Metrics for Object-oriented Software Development Environments," Masters Thesis, MIT, 1989.