Periodicals of Engineering and Natural Sciences ISSN 2303-4521
Vol. 7, No. 4, December 2019, pp.2022-2033

Aspect oriented programming: Concepts, characteristics and implementation

Omar Anwer Abdulhameed, Ahmed Younus Yousuf, Rasha Hassan Abbas
Computer Technology Engineering Department
Al-Mansour University College, Iraq

ABSTRACT

Programming techniques have been passed through many development stages in their progressing path to cope with the increasing complexity of systems requirements. So, one of the main goals of the programming languages designers is how to develop programming language that can handle and manage the spread and overlapping of different functionality concerns. Because unmanageable and uncontrollable scattering of concerns inside the system may cause many problems during system running in present or/and during applying maintenance and developing the system in future. One of the most recent and powerful solutions to overcome these problems is via using Aspect-Oriented Programming (AOP) approach. This research is demonstrates the features and the problems with implying AOP techniques in the software development process.

Keywords: Aspect-oriented programming, software Engineering, Object Oriented Programming, System Concerns, Code Weaving, Code Tangling and Scattering

Corresponding Author:
Omar Anwer Abdulhameed,
Department of Computer Technology Engineering,
Al Mansour University College, Iraq
Email: omar.abdulhameed@muc.edu.iq

1. History

Gregor Kiczales and his team at *Xerox PARC* originated concept of Aspect-Oriented Programming (AOP). They examined Object Oriented Programming (OOP) limitations in the premature of 1990s . They investigated the knowledge of innovative programming languages techniques that could support programmer efficiency by easing code modularization. Subsequently, at *Northeastern University* in the USA, doctorate student *Cristina Lopes* and others had been pursued analogous philosophy about this concern. Finally, *USA's Defense Advanced Research Projects Agency (DARPA)* perceived the work, injected finance and stimulated association amid *PARC* and Northeastern, who joined to design the 1st and highly typical AOP language of AspectJ.

IBM's research group gave special importance to the training continuity of modularizing issues at preceding programming application, and presented the enormous *HyperJ* and Concern Manipulation Environment that haven't sighted vast employment.

Another widespread AOP tool is *AspectWerkz* that created by dual *BEA* teams that published under the minor *GNU* public license.

,AspectJ and *AspectWerkz* in January 2005, accepted to join their determinations to make a solitary tool that takes the finest from every scheme.

Marketable software suppliers like *JBoss* as well presents AOP products. The *JBoss* AOP company has been issued in late 2004 and exhibits an aspect-oriented agenda and aspects library.

Microsoft in January 2005 issued an Enterprise Library with seven various "application blocks" that are possibly considered aspects and retrieved by means of attributes of *Visual Studio .Net. Loom* is another project.

2. Introduction

Any system developing process depends on its core module-level requirements and system-level requirements. Numerous system-level necessities have a tendency to be orthogonal (conjointly autonomous) to each other and to the module-level requirements (business problem requirements). System-level requirements have a tendency to crosscut numerous core modules too. For instance, crosscutting concerns are comprised by a distinctive enterprise relevance like logging, authentication, pooling, resource, management, performance and storage management. Every one crosscuts multiple subsystems. For instance, an every stateful business object is affected by a storage-management concern [1].

Implementing these crosscutting concerns is considered to be a challenging issue that conventional programming approaches, such as Object-Oriented Programming (OOP) and Procedural-Oriented Programming (POP), cannot modularize very effectively. Lack of code modularity usually results in a tangled and complex code [2].

In software engineering, there are three basic programming paradigms attempt to support programmers to apply the concerns' separation, especially cross-cutting concerns, to achieve better modularization. These paradigms are: **Aspect-Oriented Programming (AOP)**, **Aspect-Oriented Software Development (AOSD)** and **Post Object-Oriented Programming (POOP)**. The main difference between AOP and AOSD is that the first paradigm applies via using firstly language changes, whereas the second one employs a combination of methodology, environment, and language. While most of the POOP initiatives do not aim to replace OOP; they seek to refine or improve or reinvigorate it.

The bottom line, AOP paradigm has developed a protruding software development technology these days, it aims to improve software modularity [3] [4]. AOP's core idea stands for "separating the business logic in an application from the common services (crosscutting concerns) that support it" [5]. AOP is adopted as a complement to the OOP rather than as a replacement to it [6].

3. AOP basic concepts and terminologies

In this section, the main concepts and terminologies associated with AOP will be demonstrated.

3.1. Concerns

Concern can be defined as any area of interest or focus [7]. A complex software system can be viewed as a joint application of numerous concerns. A conventional system can comprise several concern types (see Figure 1) like logic, business, data persistence, performance, logging and debugging, security, authentication, error investigation, multithread safety etc.. Moreover a system may consist of development-process concerns like maintainability, traceability, comprehensibility, along with evolution easiness [8].

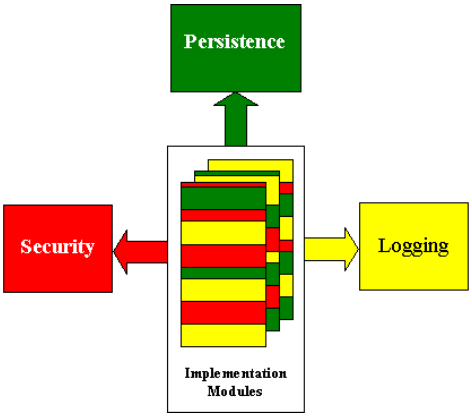

Figure 1. A system as a concerns group fulfilled by diverse modules

Separation of concerns means dividing a program into distinctive sections, in such away which attempt to achieve minimum possible overlapping in functionality of these parts [9]. All programming procedures, such as routine programming and object-oriented programming, aid implementing several parting and encapsulating concerns into solitary entities, by providing many structures (like procedures, packages, classes, and methods). However, some concerns are called *crosscutting concerns* because they resist these encapsulation's ways to cut across numerous modules in a program [10]. In an AOP language, there are a few crosscutting expressions which can enclose the concern in one place. The power, safety, and usability of the language are the features which determine the difference between languages (e.g., a limit form of crosscutting is expressed by interceptors that specify the methods in the absence of considerable support for debugging or type-safety).

3.2. Join point

It is a clear-cut point during a program execution, such as the method invoking or the exception handling.

3.3. Code tangling and scattering

As a consequence of crosscutting concern, some code is *scattered* or *tangled*. This will increase the complexity of understanding and maintaining it. Scattered code refers to a concern such as logging, distributed over a number of modules (for instance, methods and classes). This indicates that altering logging involves adjusting all affected modules. Modules conclude with several concerns (for instance, account administering, auditing, security and logging). This indicates that altering one module involves realizing all the tangled concerns.

Code tangling and code scattering can influence together software enterprise and expansions in numerous methods [11], [12]:

- **Poor traceability:** The correspondence amid a concern and its application can be obscured if several concerns are simultaneously implemented. This leads to a poor mapping between the concern and its implementation.
- **Lower productivity:** The developer's concentricity is shifted from the major concern to the peripheral concerns if multiple concerns are simultaneously implemented. This leads to reduce productivity.
- **Less code reuse:** It can be tricky to reprocess the code if a module implements multiple concerns.
- **Poor code quality:** A code with invisible problems is generated by code tangling. So, if too many concerns are targeting at once, enough attention will not be received to one or more of those concerns.
- **More difficult evolution:** A design that just addresses existing concerns is often produced by a restricted view and reserved resources. Reworking the implementation is frequently needed to address the future requirements. Thus, many modules will be touched because the implementation is unfragmented. As a result of these changes, inconsistencies *can be generated by* modifying each subsystem. To guarantee that these implementation changes have not caused bugs, considerable testing effort is required.

For instance, consider a transferring an amount in an application of the business services offered by a bank from one account to another with a conceptually so easy method (the code written in a Java-like syntax):

```
void transfer(Account fromAccount, Account toAccount, int amount)
{
   if (fromAccount.getBalance() < amount) {
   throw new InsufficientFundsException();
   }
   fromAccount.withdraw(amount);
   toAccount.deposit(amount);
}
```

In an application of real-world banking, the transfer process is more complicated with compare with the above method. It involves safety checks to make sure that the current user is authorized to carry out this operation.

Thus, the operation must be entered in the system log in a database transaction to prohibit incidental data loss. Those new concerns will be in a comprehensible version like this:

```
void transfer(Account fromAccount, Account toAccount, int amount)
{
  if (!getCurrentUser().canPerform(OP_TRANSFER)) {
    throw new SecurityException();
  }
  if (amount < 0) {
    throw new NegativeTransferException();
  }
  if (fromAccount.getBalance() < amount) {
    throw new InsufficientFundsException();
  }
  Transaction tx = database.newTransaction();
  try {
     fromAccount.withdraw(amount);
     toAccount.deposit(amount);
     tx.commit();
     systemLog.logOperation(OP_TRANSFER, fromAccount, toAccount,amount);
  }
  catch(Exception e) {
     tx.rollback();
  }
}
```

Transactions, security, and logging are various new concerns (*cross-cutting concerns*) which have begun to be *tangled* with the essential functionality (occasionally termed the *business logic concern*) of the code. Because of these cross-cutting concerns, the code has missed its style and simplicity. Also, these cross-cutting concerns appear *scattered* across numerous methods.

Therefore, any modifications need to be done on any of these concerns would require huge work.

3.4. Aspects

While the core concerns of the system correctly encapsulated in their own modules, the cross-cutting concerns does not so. This rises the system intricacy and creates continuance very complicated. AOP tries to fix this matter by providing forms enable the IT worker to represent cross-cutting concerns in independent modules termed *aspects*. In another words, AOP develops schemes using loosely coupled, fragmented applications of *crosscutting* concerns. Whereas OOP develops systems by means of insecurely coupled, fragmented applications of popular concerns. The modularization element in AOP is termed as an *aspect*, while the modularization element in OOP is termed as a *class.*

So an *aspect* is a concern which cross-cuts numerous classes and/or approaches [13]. Aspects can contain:

- *Advice*, which is the logic that is triggered by a certain event [14], also can be refer to as the code combined with set points in a program.
- Lastly, *inter-type declarations* (also known as *open classes*), which is the structural members joined to more classes to represent crosscutting concerns affecting the structure of these classess [15], [16]. This gives the ability to programmers to declare members or parents of another class in one place, so as to join each code associated to a concern in a solitary aspect.

For instance, before accessing a bank account, a security check is performed using advice of a security module. The times needed to access a bank account is defined by the pointcut and how the security check can be implemented is defined by the code in the advice body. Thereby, the check in addition to the places are feasibly preserved together in single position. Moreover, later program changes can be anticipated by a good pointcut. So, the advice will be applied to the new method created by another developer to access the bank account when it executes.

3.5. Pointcut

It is a group of join points that determine where the associated advice will be invoked.

4. Implementation

A traditional way in designing an application starts by defining its business functions (all the tasks that must be performed by the application). However, developers may discover during their works in developing an application that there is some implied stuff (i.e. crosscutting concerns), which never has been written in the specifications, needs to be write in different places and at many times.

"AOP means they finally have an opportunity to ask how to apply plumbing across the board so you write it once to serve the whole application". And consequently AOP projects begin with designers indicating the orthogonal utilities which each object will demand [5].

4.1. AOP Development Process

AOP consists of three basic development stages (see Figure 2) [12], [17]:

1. Aspectual decomposition: Refer to the operation of decomposing the requirements to distinguish between crosscutting and popular concerns. This can be accomplished by separating concerns of module-level from crosscutting concerns of system-level. For instance, three concerns in a credit card module can be specified as: core credit card processing, logging, and authentication.

2. Concern implementation: Refer to the operation of implementing every concern individually. For processing example of credit card, logging unit, the core credit card processing unit and authentication unit; can be implemented.

3. Aspectual recomposition: Refer to the operation that applied by an aspect integrator to specify recomposition procedures by producing aspects of modularization elements. The recomposition process (***weaving*** or integrating) can be used to form the ending system. For processing example of credit card, we can state, in a language supplied using AOP application, operation's start and completion be logged. Moreover, we can assign that any operation should remove authentication before it steps for the logic of business.

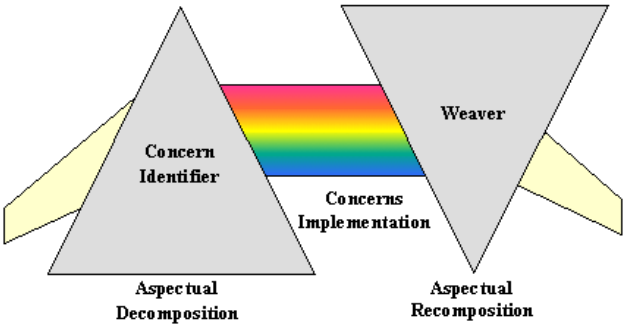

Figure 2. Steps of AOP development

According to the underlying languages structures and environments, there are dual dissimilar means of AOP programs can have an effect on other programs [18]:

1) A combination program is resulted that is proper in the unique language and undifferentiated from a usual program to the ultimate interpreter.

2) The ultimate environment or interpreter is adaptively changed to be able to identify with and apply AOP features.

4.2. Implementing weaving technique

Due to the complicatedness of variable environments, the majority of implementations generate suitable combined programs over a method of weaving. According to some criteria supplied to the weaver, it gathers an individual concern in a weaving process. The equivalent AOP language can be executed throughout

using different weaving procedures. A weaving technique eventually impacts only the facility of an implementation, in turn it can mightily affect the adoption of a language. The following figure illustrates how to deploy AOP program with comparing to other programming paradigms (see Figure 3):

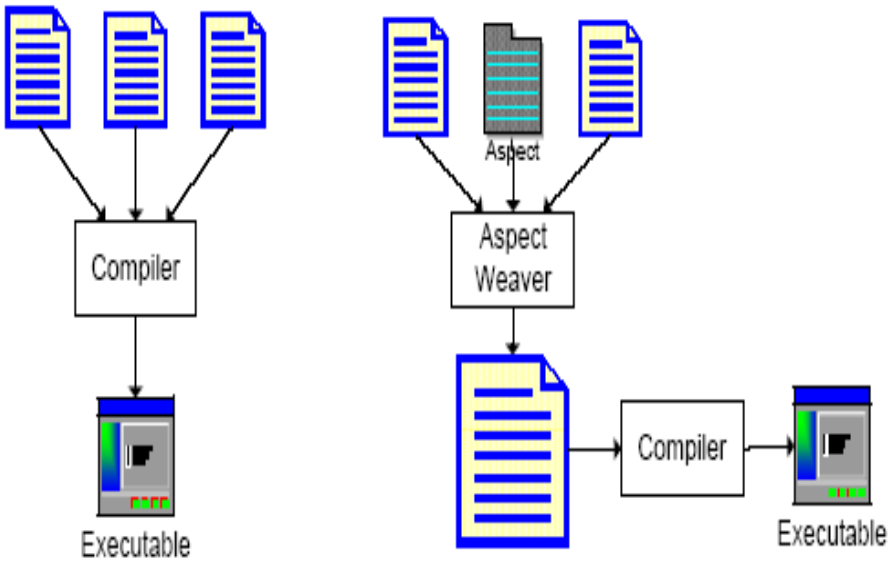

Figure 3. How to deploy AOP program with comparing to other programming paradigms

Some different ways are used to implement weaving, such as:

- ***Source-level weaving***, carried out by employing preprocessors (as C++ was carried out primarily in CFront) that entail right to use files of the source of program. In 2001, AspectJ launched with source-level weaving.
- ***Bytecode weavers***, can be organized throughout the construct process. Namely, a weave model is per-class, throughout class loading. Java's distinct binary form qualifies bytecode weavers to run with all Java programs in [.class-file] usage. In 2002, AspectJ provided a per-class bytecode weaver, and provided advanced load-time provision after the incorporation of *AspectWerkz* in 2005. All ways which join programs at runtime must provide mechanisms to separate them correctly to preserve the segregated model of programmer. Java's bytecode supporting in connection with several source files allow any debugger to progress over a correctly woven [.class file] in a source editor. Nonetheless, some third-party decompilers can't manage woven code since they only capable of handling code created via Javac instead of all supported bytecode forms.
- ***Deploy-time weaving***, implemented by applying post-processing, but instead of patching the created code, it *subclassing* current modules in order to introduce the modifications as a result of method-overriding.

The current classes stay unchanged, even at runtime, and all current tools (debuggers, profilers, etc.) are feasibly employed over expansion. In plentiful *J2EE* application servers implementation, such as *IBM*'s *WebSphere*, a similar approach has already established itself.

In .NET, there are essentially dual methods to modify a system performance as in *weave* a .NET program [13]:

- ***Compile-Time Weaving***: before deployment, program has adapted throughout the process of building on the development machine.
- ***Runtime Weaving***: after deployment, the program is reformed through its implementation.

The example about credit card processing system can be used to illustrate code weaving process. Only two operations will be considered for briefness purpose: credit and debit, also considering that the system contains an appropriate suitable logger.

Consider the following credit card processing module:

```
public class CreditCardProcessor {
    public void debit(CreditCard card, Currency amount)
       throws InvalidCardException, NotEnoughAmountException,
              CardExpiredException {
        // Debiting logic
    }
    public void credit(CreditCard card, Currency amount)
        throws InvalidCardException {
        // Crediting logic
    }
}
```

Also, consider the following logging interface:

```
public interface Logger {
    public void log(String message);
}
```

The required composition needs to apply the subsequent weaving rules (which stated here in natural language):

- "Log each public operation's starting"
- "Log each public operation's completion"
- "Log any exception thrown by each public operation"

The weaver could adopt these weaving procedures and concern implementations to create the corresponding composed code as below:

```
public class CreditCardProcessorWithLogging {
    Logger _logger;
    public void debit(CreditCard card, Money amount)
        throws InvalidCardException, NotEnoughAmountException,
               CardExpiredException {
        _logger.log("Starting CreditCardProcessor.credit(CreditCard,
                    Money) " + "Card: " + card + " Amount: " + amount);
        // Debiting logic
        _logger.log("Completing CreditCardProcessor.credit(CreditCard,
                    Money) " + "Card: " + card + " Amount: " + amount);
    }
    public void credit(CreditCard card, Money amount)
        throws InvalidCardException {
        System.out.println("Debiting");
        _logger.log("Starting CreditCardProcessor.debit(CreditCard,
                    Money) " + "Card: " + card + " Amount: " + amount);
        // Crediting logic
        _logger.log("Completing CreditCardProcessor.credit(CreditCard,
                    Money) " + "Card: " + card + " Amount: " + amount);
    }
}
```

5. Framework of AOP languages

As some other sorts of implementation of programming approach, an AOP carrying out composed of dual basic divisions:

(1) The AOP language specification:

At a greater level, the language specification depicts language constructs and syntax. Hence, an AOP language identifies dual mechanisms [19]:

- ***Implementation of concerns:*** Refer to the process of mapping each distinct requirement into codes in order that a compiler has a possibility to change them into runnable code. As it takes the form of identifying processes, traditional languages such as C, C++, or Java can be used with AOP.
- ***Weaving rules specification:*** Refer to the process of how to form autonomously applied concerns to develop the latter system. As a result, an application involves employing or building a language for giving

rules to form various implementation pieces to construct the latter system. The language for specifying weaving rules could be an extension of the implementation language, or could be completely different from this language.

(2) The AOP language implementation:

The code's correctness verification is applied by the language implementation depending on language specification and translates it into another code form that is able to execute by the target machine. Thus, compilers of AOP language perform two basic logical phases [1]:

1. Add the distinct concerns
2. Transform the consequential information into executable code

The weaver is feasibly implemented by an AOP implementation in different ways, as mentioned before.

6. AOP and other programming paradigms

Concepts of OOP and computational reflection lead to the appearance of aspect concept. Aspects associate widely to programming concepts such as delegation, mixins, and subjects. The functionality of AOP languages is similar to the meta-object protocols, but more restricted. Composition Filters and the hyperslices approach are other ways of applying aspect-oriented programming paradigms.

Start from the 1970's, interception types and dispatch-patching which are identical to sseveral AOP operation methods have been being applied by developers, but these not ever had the semantics that the crosscutting qualifications have been recorded in the same place.

To implement separation of code, designers have concerned on other alternative approaches like C#'s partial types. But, such approaches need a quantification mechanism that enable programmers to access many join points of the code with a statement in the form of a declaration.

7. AOP vs OOP

Presently, OOP considers as the most adopted programming methodology for developing new software systems. OOP power emerges when it comes to modeling common behavior. But OOP weakness appears when trying to implement concerns that extend over many modules, here the AOP role come out.

AOP is complementary to OOP. OOP uses to design and develop objects, while AOP uses to explore objects. In AOP, composition flow runs from crosscutting issues to the major concern, while in OOP methods the flow runs in the reverse direction. OOP still has difficulty separating the expressions of multiple concerns because OOP doesn't sufficiently address performances that cross over plentiful (frequently unrelated) modules. Inheritance in OOP may lead to «conceptual explosion». Also class hierarchy itself does not help to its appropriate usage; design templates and join rules needed. Therefore, Complexity of the task to completely locate a crosscutting concern may leads to poor maintainability, reliability and reuse. While AOP methodology fills this void [14], [20]. Finally, AOP relies on tools more than OOP [21], [22].

In short words, AOP is used to Fragment Crosscutting Structure while OOP is used to fragment Hierarchical Structure.

8. Problems & attainments when adopting AOP

The main aim that programmers attempt to achieve during the process of developing their programs is to keep their codes as possible as simple, readable and conceptual understandable.

Adopting AOP to support applications implementation may have following disadvantages:

- Widespread program failure may be caused by a programmer if he made a legal error in presenting crosscutting.

- On the contrary, a second programmer can alter the joined points in the program (for instance, by moving methods or renaming) so that the aspect writer cannot expect them, this will lead to unintended consequences.

On other hand, there are several benefits of using AOP approach as follows:

- **Fragmented implementation of crosscutting concerns**, the main advantage of modularizing crosscutting concerns is given the ability to a single programmer to implement modification that affect the whole system simply, this termed as consistency [23].
- **More code reuse**, AOP provides structures those enable an application to employ the functions of aspects in a simple way without requiring to reconstruct or code again anything under no conditions.
- **Afford a powerful form to handle system's requirements**, provide the ability of describing each application's necessity for a service like logging. Namely, the services are superior-described, superior -coded and richer in functionality.
- **Better exploitation of development team capabilities**, provide the ability of creating discrete aspects enables a developing group to give the professional a task, so that the superior specialists for the task job can employ their skillfulness and practice.
- **Late binding of design decisions**, since requirements can be implemented as separate aspects, a system analyst and designer can postpone making design decisions for future requirements.
- **Easier-to-evolve a system**, by creating new aspects it's easy to append new functionality to the system because aspected modules may be uninformed of crosscutting issues. Further, the existing aspects will crosscut any new modules added to a system, aiding produce a consistent evolution.

9. Discussion and conclusions

AOP's main goal is to restore concern modularity through applying the following processes:

- Identify crosscutting concerns
- Extract crosscutting concerns as separate modules, called Aspects
- Develop and maintain crosscutting concerns independently
- Building the system from the aspects (weaving), by weave the modules together during the following two stages:
  - Build - to compose application
  - Runtime - to dynamically change behavior
- Debugging aspects

AOP basic concepts and terms can be abridged as in Table 1:

Table 1. Summary AOP basic concepts

Term	Description
Concern	A special goal, concept, or area of interest.
Crosscutting Concerns	Concerns tangled and scattered across many modules in a program.
Aspect	A concern modularization which cuts across several modules.
Core (Base Code)	The non-aspect program section, the business program part.
Advice	Action taken by an aspect at a special join point.
Join Point	A point through the program implementation as in the method execution or the management of an *exception*.
Pointcut	A predicate which fits the points of join.
Scattered Code	A code regarding one concern spread in several units of the system.
Tangled Code	A code with different concerns interlacing to each other.
Coupling	Dependency between the modules.
Weaving	The procedure of interpolating aspect code into other code, can be accomplished at build time, load time, and run time.

The following points can adopted to compare between the AOP languages:

- The join points that are exposed,
- The language used to identify the points of join clearly,
- The operations allowed at the points of join, and
- The structure's improvements which can be expressed.

Code optimization will be achieved as an outcome of these advantages, because they will lead to the following gains:

- Less number of code lines required for writing,
- Saving time,
- Minimalizing expenses
- Releasing assets for more praiseworthy activities and processes than writing code like a logging program.

There are many realities, facts and conclusions that associated with AOP as follows:

- There are many applications include crosscutting concerns. It can be with a lot of time for programmers to study how to develop it, it correspondingly diverts them from resolving the business problem that stands for the basic concern.
- AOP is suitable for modularizing many crosscutting concerns, such as: tracing, logging, security, error handling and so on.
- Aspectize transaction semantics: not possible, because trying applying it often lead to irreversible actions and deadlock [24].
- Aspectize transaction interfaces, possible, but artificial [24].
- Aspectize transaction mechanisms; only syntactical separation [24].
- AOP is not a solution to unsolved problems, but a better solution for solved problems [25].
- Design patterns can introduce complexity not found in AOP [25].
- AOP provides a simpler solution than dynamic proxies [25].
- Frameworks introduce limitations not found in AOP [25].
- Annotations and AOP are highly complementary [25].
- AOP increases the degree of abstraction.
- The process of debugging crosscutting functionality can be done by using proper tools.
- Aspects those weakly designed can break as classes evolve.
- Aspects and classes are both can be tested easily.
- Although AOP is complex, but still simpler than the alternatives
- Applying AOP needs to be applied by experience system developers.
- Designers can apply AOP incrementally.
- AOP makes code execution slower [25].

References

[1] J. Zhuk, "Integration-Ready Architecture and Design Software Engineering with XML, Java, .NET, Wireless, Speech. And Knowledge Technologies", Cambridge University Press, 2004. https://www.cambridge.org/9780521525831.

[2] H. A. Kurdi, “Review on Aspect Oriented Programming”, *International Journal of Advanced Computer Science and Applications (IJACSA)*, Vol. (4), No. (9), 2013.

[3] G. A. S. Sheela, and A. Aloysius, ”Aspect Oriented Programming - Cognitive Complexity Metric Analysis Tool”, *International Journal of Scientific Research in Computer Science, Engineering and Information Technology (IJSRCSEIT)*, Vol. (3), Issue (1), Jan. – Feb. 2018.

[4] A.Santosa, P. Alvesa, E. Figueiredoa, and F. Ferrarib, “Avoiding code pitfalls in Aspect-Oriented Programming”, *Science of Computer Programming*, Vol. (119), April 2016.

[5] S. Sharwood, "A new aspect to programming?", http://builderau.com.au, April 2005.

[6] I. Boticki, M. Katic, S. Martin, “Exploring the Educational Benefits of Introducing Aspect-Oriented Programming Into a Programming Course”, *IEEE Transactions on Education*, Vol. (56), No. (2), pp.217-226, 2013.

[7] J. Rossberg ,"Pro Visual Studio Team System Application Lifecycle Management", Springer-Verlag New York, Inc., 2008.

[8] Q. H. Mahmoud, "Middleware for Communications", John Wiley & Sons Ltd, 2004. www.wileyeurope.com.

[9] S. Puri**,** " Ember.js Web Development with Ember CLI: Build ambitious single-page web applications using the power of Ember.js and Ember CLI ", Packt Publishing Ltd., 2015.

[10] S.R Raheman., et al., "Aspect oriented programs: Issues and perspective", *Journal of Electrical Systems and InformationTechnology*, Vol.5, pp.562–575, 2017. https://dx.doi.org/10.1016/j.jesit.2017.06.003.

[11] M. Dessì," Spring 2.5 Aspect-Oriented Programming", Published by Packt Publishing Ltd., 2009.

[12] N. Påhlsson, " Aspect-Oriented Programming: An Introduction to Aspect-Oriented Programming and AspectJ ", Topic Report for Software Engineering, 2002.

[13] Gael Fraiteur and the postsharp.org community Website, "AOP for .NET",

http://www.postsharp.org/aop.net, 2007.

[14] A. Grigoriev, "Aspect Oriented Programming: Aspect – one side or part, appearance, viewpoint", December 2003.

[15] M. Singh, et. al., "Mutant Generation for Aspect Oriented Programs", *Indian Journal of Computer Science and Engineering*, Vol. 1, No. 4, pp. 409-415, 2010.

[16] A. S. Bist, "A Review - Aspect-Oriented Programming ", International Journal of Engineering Sciences & Research Technology (IJESRT), Vol. 16, pp. 357-360, Aug. 2012.

[17] M. Chibani, B. Belattar, and A. Bourouis, "The Use of the Aspect Oriented Programming (AOP) Paradigm in Discrete Event Simulation Domain: Overview and Perspectives", Conference: The Third International Conference on Digital Information Processing and Communications (ICDIPC), 2013. https://www.researchgate.net/publication/265794449.

[18] A. Assaf, "A Common Aspect Languages Interpreter", thesis, Software Engineering [cs.SE], University of Nantes, 2011.

[19] R. Laddad, R. Johnson, " Aspectj in Action: Enterprise AOP with Spring Applications", 2nd Edition, Manning Publications Co., 2010.

[20] V. O. Safonov and A. N. Safonova, "Aspect Oriented Programming and Aspect.NET", MSR Rotor Workshop, Cambridge UK, July 2002.

[21] D. Wampler, "The Future of Aspect Oriented Programming", Aspect Programming – LLP, 2003.

[22] V. García Díaz, Juan Manuel Cueva Lovelle, B. Cristina Pelayo García Bustelo ,and Oscar Sanjuán Martinez, " Advances and Applications in Model Driven Engineering", Information Science Reference (an imprint of IGI Global), 2014.

[23] J. Paquet and S. A. Mokhov (Eds.), "Comparative Studies of Programming Languages, COMP6411 Lecture Notes, Revision 1.9", 5 August, 2010.

[24] S. Aregger and A. Schmidig, "AOP: Does It Make Sense? The Case of Concurrency and Failures", 2004.

[25] A. Pashazadeh, "Aspect Oriented Programming: Separating Software Concerns with AOP", Payeshgaran MT, 2007.