DOI: https://doi.org/10.53555/nncse.v2i3.496 **Publication URL: http://nnpub.org/index.php/CSE/article/view/496**

PROGRAMMING USING ASPECTS

Saurabh Saxena[1]*, Meena Kumar[2]
*[*1,2]Assistant Professor, DIRD Delhi*
*[*1]saurabhsaxenait@gmail.com, [2]meenarana.96@rediffmail.com*

Corresponding Author: -
E-mail: saurabhsaxenait@gmail.com

Abstract: -
We can define an 'Aspect oriented programming' as executing the code whenever a program shows certain behaviors. Aspects are an additional unit of modularity. Aspects can be reused. By reducing code tangling it makes it easier to understand what the core functionality of a module is. An "aspect weaver" takes the aspects and the core modules and composes the final system.

INTRODUCTION

In the evolution age of computers where everyone is almost dependent on computers for various tasks, there is a big challenge for software developers to provide the tools that meet the requirement of the layman and of course, the security to his valuable data. From the beginning of this changing era, there are many paradigms in programming according to the technological changes. These programming paradigms can be classified as: -

- Procedural programming
- Executing a set of commands in a given sequence
- Fortran, C, Cobol

- Functional programming
- Evaluating a function defined in terms of other functions
- Lisp

- Logic programming
- Proving a theorem by finding values for the free variables
- Prolog, Haskell

- Object-oriented programming (OOP)
- Organizing a set of objects, each with its own set of responsibilities
- Smalltalk, Java, C++ (to some extent)

- Aspect-oriented programming (AOP)
- Executing code whenever a program shows certain behaviors
- AspectJ (a Java extension)
- Does not *replace* O-O programming, but rather *complements* it

Programming using Aspects

OOPS is considered as a security tool for data as it provide the 'data hiding' from the external users. Everything in OOPS is organized in objects and each object has its own data and functions to work on. Programming languages like JAVA, C++ etc provide a very smart way to do the things but there are certain anomalies to
OOPS as:-

a) Some programming tasks cannot be neatly encapsulated in objects, but must be scattered throughout the code. Some of the tasks are :-
   - Logging (tracking program behavior to a file)
   - Profiling (determining where a program spends its time)
   - Tracing (determining what methods are called when)
   - Session tracking, session expiration
   - Special security management

b) The result is crosscutting code-the necessary code "cuts across" many different classes and methods The cross cutting has the following consequences: -

--- Redundant code
--- Difficult to reason about
--- Non-explicit structure
--- The big picture of the tangling isn't clear
--- Difficult to change
--- Have to find all the code involved...
--- and be sure to change it consistently
--- and be sure not to break it by accident
--- Inefficient when crosscutting code is not needed

There are various crosscutting concerns as:-

• AOP addresses behaviors that span many, often unrelated, modules.
• Core Concerns:
– Primary core functionality.
– Central functionality of a module.
• Crosscutting Concerns:
– System wide concerns that span multiple modules.
– Cuts across the typical division of responsibility.
• OOP creates a coupling between core and crosscutting concerns.
• AOP aims to modularize crosscutting concerns.

The cross cutting concerns can be explained using an example as:- Let consider the UML for a simple figure editor in which there are two concrete classes of figure element, points and lines. The concern that the screen manager should be notified whenever a figure element moves. This requires every method that moves a figure element to do the notification. Now we can define the core of aspect oriented programming i.e 'aspects'. An **'aspect'** can be understood as:- It is a modular units that cross-cut the structure of other modular units. It is defined in terms of partial information from other units. It exist in both design and implementation phases.
Also we can define an 'Aspect oriented programming' as

- In AOP crosscutting concerns are implemented in aspects instead of fusing them into core modules.
- Aspects are an additional unit of modularity.
- Aspects can be reused.
- By reducing code tangling it makes it easier to understand what the core functionality of a module is.
- An "aspect weaver" takes the aspects and the core modules and composes the final system.

The 'weaving' is a set of rules that specify how to integrate the final system. It can be implemented in various ways:

– Source to source translation.
– Byte code enhancement, first compile source with original compiler, then weave aspects into class files.
– Just-in-time weaving done by a class loader.
– By the language compiler. The JAsCo language supports runtime weaving and unweaving.

Terminology used for 'Aspects'

The terminology used for AOP may contain the following:-

1) Join point
2) Point cut
3) Advice
4) Introduction
5) Aspect

The brief introduction of all these is:-
---- A join point is a well-defined point in the program flow
---- A point cut is a group of join points
---- Advice is code that is executed at a point cut
---- Introduction modifies the members of a class and the relationships between classes
---- An aspect is a module for handling crosscutting concerns
---- Aspects are defined in terms of point cuts, advice, and introduction
---- Aspects are reusable and inheritable
---- Each of these terms will be discussed in greater detail

Let us consider an example to understand the terminology:- A pointcut named "move' that chooses various method calls:

```
pointcut move():   call(void FigureElement.setXY(int,int))
 || call(void Point.setX(int))
 || call(void Point.setY(int))
 || call(void Line.setP1(Point))
 || call(void Line.setP2(Point));
Advice (code)
 //that runs before the move pointcut: before (): move()
{
 System.out.println("About to move");
}
// Advice that runs after the move pointcut: after(): move()
 {
   System.out.println("Just successfully  moved");
}
```

Join Point

- ■ A join point is a well-defined point in the program flow
- ■ We want to execute some code ("advice") each time a join point is reached
- ■ We do *not* want to clutter up the code with explicit indicators saying *"This is a join point"*
- ■ AspectJ provides a syntax for indicating these join points "from outside" the actual code
- ■ A join point is a point in the program flow "where something happens"
- ■ Examples:
- ■ When a method is called
- ■ When an exception is thrown
- ■ When a variable is accessed

Pointcuts

- Point cut definitions consist of a left-hand side and a right-hand side, separated by a colon
- The left-hand side consists of the point cut name and the point cut parameters (i.e. the data available when the events happen)
- The right-hand side consists of the point cut itself Example point cut:
  point cut setter(): call(void setX(int));
- The name of this point cut is setter
- The point cut has no parameters
- The point cut itself is call(void setX(int))
- The point cut refers to any time the void setX(int) method is called Example point cut designators I
- When a particular method body executes:
- execution(void Point.setX(int))
- When a method is called:
- call(void Point.setX(int))
- When an exception handler executes:
- handler(ArrayOutOfBoun dsException)
- When the object currently executing (i.e. this) is of type Some Type:
- this(SomeType)

Example point cut designators II

- When the target object is of type SomeType
- target(SomeType)
- When the executing code belongs to class MyClass
- within(MyClass)
- When the join point is in the control flow of a call to a Test's no-argument main method
- cflow(call(void Test.main()))

Point cut designator wildcards

- It is possible to use wildcards to declare point cuts:
- execution (* *(..))
- Chooses the execution of any method regardless of return or parameter types
- call(* set(..))
- Chooses the call to any method named set regardless of return or parameter type
- In case of overloading there may be more than one such set method; this pointcut picks out calls to all of them

Point cut designators based on types

- You can select elements based on types. For example,
- execution(int *())
- Chooses the execution of any method with no parameters that returns an int
- call(* setY(long))
- Chooses the call to any setY method that takes a long as an argument, regardless of return type or declaring type
- call(* Point.setY(int))
- Chooses the call to any of Point’s setY methods that take an int as an argument, regardless of return type
- call(*.new(int, int))
- Chooses the call to any classes’ constructor, so long as it takes exactly two ints as arguments

Point cut designator composition

- Point cuts compose through the operations **or** (“||”), **and** (“&&”) and **not** (“!”) Examples:
- target(Point) && call(int *())
- Chooses any call to an int method with no arguments on an instance of Point, regardless of its name
- call (* *(..)) && (within(Line) || within(Point))
- Chooses any call to any method where the call is made from the code in Point’s or Line’s type declaration
- within(*) && execution(*.new(int))
- Chooses the execution of any constructor taking exactly one int argument, regardless of where the call is made from
- !this (Point) && call (int *(...))
- Chooses any method call to an int method when the executing object is any type except Point

Pointcut designators based on modifiers

- call (public * *(..))
- Chooses any call to a public method
- execution (!static * *(..))

- Chooses any execution of a non-static method
- execution (public !static * *(..))
- Chooses any execution of a public, non-static method
- Point cut designators can be based on interfaces as well as on classes

Kinds of advice

- AspectJ has several kinds of advice; here are some of them:
- Before advice runs as a join point is reached, before the program proceeds with the join point
- After advice on a particular join point runs after the program proceeds with that join point
- after returning advice is executed after a method returns normally
- after throwing advice is executed after a method returns by throwing an exception
- after advice is executed after a method returns, regardless of whether it returns normally or by throwing an exception
- Around advice on a join point runs as the join point is reached, and has explicit control over whether the program proceeds with the join point
- Advice is a method like construct that expresses the action to be taken at the join points that are captured by a pointcut.
- Before advice

– executes prior to the join point
- After advice

– executes following the join point
- Around advice – surrounds the join point's execution

– can continue original execution, bypass execution or cause execution with an altered context

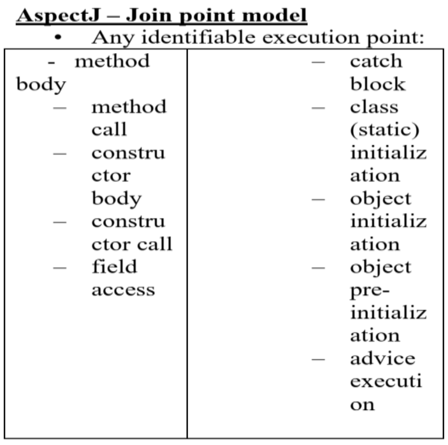
– can cause execution of the join point multiple times

Introduction

- An introduction is a member of an aspect, but it defines or modifies a member of another type (class). With introduction we can
- add methods to an existing class
- add fields to an existing class
- extend an existing class with another
- implement an interface in an existing class
- convert checked exceptions into unchecked exceptions

AspectJ: -

- Aspectj is: a general-purpose Ao extension to Java Java platform compatible easy to learn and use freely available under an Open Source license
- Aspectj enables the modular implementation of a wide range of crosscutting concerns
- When written as an aspect the structure of a crosscutting concern is explicit and easy to reason about □ Aspects are modular □ AspectJ enables: Name-based crosscutting (tend to affect a small number of other classes) Property-based crosscutting (range from small to large scale)
- Adoption of it into an existing project can be a straightforward and incremental task:
- to begin with development aspects
- other paths are possible, depending on the needs of the projects
- The goals of the AspectJ project are to make AOP technology available to a wide range of programmers, to build and support an AspectJ user community



AspectJ – Join point model

- Any identifiable execution point:

- method body	– catch block
– method call	– class (static) initialization
– constructor body	– object initialization
– constructor call	– object pre-initialization
– field access	– advice execution

- All join points have a context. Certain point cuts can capture the context and pass it to the advice.

What about encapsulation?

- AOP breaks encapsulation in a controlled manner.

- Encapsulation is broken between classes and aspects.
- Aspects marked as privileged have access to the private members of a class.
- Encapsulation is preserved between classes.
- An aspect encapsulates a crosscutting concern.

AspectJ – Aspects •
Aspects can:
– Include data members and methods. – Be declared abstract (won't be weaved).
– Have access specifiers.
– Extend classes or aspects. – Implement interfaces.

- Aspects are not the same as classes:

– Cannot be directly instantiated.
– Cannot inherit from concrete aspects.
– Can be marked as privileged.

Static crosscutting
- Dynamic crosscutting modifies the execution behavior of the program.
- Static crosscutting modifies the structure of the program.

– Member introduction.
– Type-hierarchy modification.
– Compile-time warning declaration.
– Exception softening.
- Member introduction adds data members and methods to classes

Modifying the class hierarchy
- Existing classes can be declared to implement an interface or extend a super class.
- Works as long as Java inheritance rules are not violated (no multiple inheritances). **declare parents : [Type] implements [Interface List]; declare parents : [Type] extends [Class];**
- Aspects can be made dependant only on a base type or interface. This makes aspects more reusable.

AspectJ – Exception softening
- Converts a checked exception into a runtime exception.
- Sometimes it can be inconvenient to have to deal with checked exceptions. Involves a proliferation of try/catch blocks and throws clauses. Example: SQLException in JDBC API **declare soft : SQLException : within(DatabaseAccess);**
- Exception is automatically rethrown as a org.aspectj.lang.SoftException

Policy Enforcement
- Mechanism for ensuring that system components follow certain programming practices.
- For example; enforce that public access to instance variables is prohibited.
- Avoid incorrect usages of an API.
- Ensures better quality.
- In AOP policy enforcement concerns can be implemented in aspects.
- Aspect based policy enforcement is reusable.
- AOP can be used for "Design by Contract".

Policy Enforcement in AspectJ
- Compile time and runtime enforcement.
- In AspectJ it is possible to specify custom compile time warnings and errors using pointcuts.

declare warning: get (* System.out) || get (* System.err)
: "consider using Logger.log () instead"; declare error: set (public * *) || get (public * *)
: "nonpublic access is not allowed";
- Runtime enforcement can be achieved through advice that detects policy violations.

Aspect precedence
- It is possible for more than one piece of advice to apply to a join point.
- It may be important to control the order in which advice is applied. **declare precedence: Aspect1, Aspect2, ...;**
- In the absence of any specified precedence control, the order in which the advice is applied is non-deterministic (at compile time).
- If more than one piece of advice in the same aspect applies to a point cut, then the advice that comes first in the file has precedence.

Aspect association

- Typically an aspect represents a singleton object (aspect instance) that is instantiated by the system.
- It may be useful to have more than one instance of an aspect in the system. Can create associations between aspect instances and other objects.
- Can associate a new aspect instance with a target object by using a point cut.

Conclusion

- AOP has many potential benefits.
- Higher modularization.
- Cleaner responsibilities for individual modules.
- Improved separation of concerns.
- Easier system evolution. Much easier to add crosscutting functionality.
- More code reuse.
- Can make code easier or harder to understand.
- Already starting to become mainstream (JBoss, JDO, Spring).
- Like any new technology, AOP can be abused.
- Aspect-oriented programming (AOP) is a new paradigm--a new way to think about programming
- AOP is somewhat similar to event handling, where the “events” are defined outside the code itself
- AspectJ is not itself a complete programming language, but an adjunct to Java
- AspectJ does not add new capabilities to what Java can do, but adds new ways of modularizing the code
- AspectJ is free, open source software
- Like all new technologies, AOP may--or may not--catch on in a big way.

References :-

[1].dictionary.reference.com/browse/**a spect-oriented+programming**
[2].ftp://ftp.inf.ufrgs.br/pub/pod/artigos /article.ps.gz
[3].sit.iitkgp.ernet.in/research/aut05vo l/topic12.ppt
[4].www.ijetae.com/files/Volume2Issu e4/IJETAE_0412_81.pdf