Journal of Fundamental and Applied Sciences

ISSN 1112-9867

Available online at http://www.jfas.info

Research Article

ASPECT ORIENTED IMPLEMENTATION OF DESIGN PATTERNS USING METADATA

G. Alipour*[1], A. B. Sangar[2], M. H. Mogaddam[3]

[1]Department of Computer Engineering, Hashtrood Branch, Islamic Azad University, Hashtrood, Iran

[2]Department of Computer Engineering, Urmia Branch, Islamic Azad University, Urmia, Iran

[3]Department of Computer Engineering, Hashtrood Branch, Islamic Azad University, Hashtrood, Iran

Published online: 18 June 2016

ABSTRACT

Computer programming paradigms aim to provide better separation of concerns. Aspect oriented programming extends object oriented programming by managing crosscutting concerns using aspects. Two of the most important critics of aspect oriented programming are the “tyranny of the dominant signature” and lack of visibility of program’s flow. Metadata, in form of Java annotations, is a solution to both problems. Design patterns are assumed as the general solutions for Object-Oriented matters. They assist in software complexity management and serve as a bridge among software designers as well. These properties have led the patterns to be introduced as a choice in order to prove new technologies. Successful implementations share a generic solution: the usage of annotation to configure and mark the participants, while the pattern's code is encapsulated in aspects. This loses the coupling between aspects and type signatures and between the code base and a specific AOP framework. Also, it increases the developer's awareness of the program's flow.

Author Correspondence, e-mail: gh.alipour@iaut.ac.ir

doi: http://dx.doi.org/10.4314/jfas.v8i2s.133

In the present article, aspect oriented programming and design patterns are introduced and also taking the benefit of annotation equipment in java language is proposed as a solution to reduce tight coupling and increase program flow observation rate for aspect oriented programming.

Keywords: aspect oriented programming, design patterns, object oriented programming, metadata.

1. INTRODUCTION

Software system development raised as an engineering matter in 1968. Since then, numerous works have been devoted to solve the problems it is going to face. The activities have mainly been focused on the better separating of software systems as well as component its sections into independent and straightforward units such as classes and objects so that there is the least overlapping and interference amongst the units. This programming approach operates properly in program logic analysis. However, it turned out in practical researches that it does not work to take the benefit of object oriented software development in quantizing some concerns. These concerns, which are called crosscutting concerns, include aspects of a program that overshadow other concerns. These concerns are often unable to be clearly separated from other parts of system either in design or application which leads to departure and complexity problems. In fact, preparing some solutions for crosscutting concerns problems, aspect oriented programming completes object oriented one and is not assumed as an alternative for it. Design pattern is a reusable solution for the problems rising in the software design domain. First the patterns were utilized as an architecture concept in C++ language and then the book GOF was published in 1997.Patterns provided connection among software designers through introducing novel solutions. In this article we are about to have a review on aspect oriented programming. Then, after assessing some well-known patterns, a solution is proposed to connect the patterns and aspect oriented programming.

Aspect oriented programming

The emergence of the Aspect Oriented Programming (AOP) paradigm is driven by the need for better ways of describing and encapsulating concerns in a software application. Object Oriented Programming (OOP) provides a good way for this by using objects that encapsulate state and actions; however this is limited to the problem domain of an application. The so-called crosscutting concerns could not be fitted. Among the usual crosscutting concerns are logging, authentication and transaction management. These aspects are not related with the problem domain of the application but rather they "cut through" it. The current crosscutting concerns management is to interleave them with the core logic code. Unfortunately this breaks the modularization of the system. To solve this situation, research explored how crosscutting concerns can be isolated from the business logic and be applied in a non-intrusive manner. AOP was coined by G. Kiczales and his team at Xerox PARC in the early 1990's. Also, they developed one of the first and most popular AOP languages, AspectJ, as an extension to Java. AOP gained notoriety among software developers and architects, as systems have become more complex and old paradigms have been unable to keep pace. AOP does not replace OOP but extends it by providing further separation of concerns.

Concerns separation

Including a specific set of necessary behaviors of a plan, concern is a demand of a system which is a priority for benefit takers[2]. And it can influence on different software units. Concern may include responsibility or irresponsibility demands such as events documentary, efficiency, etc. besides, concern can be raised in either high or low levels like security issue or hidden memory.

Parameter-based development is a widely used method to generate complex systems. In fact, demands are devoted to parameters belong to the type of class, object and service. Although there are some necessities that are not able to concentrate on a single parameter and might influence on numerous parameters. This type of cut across necessities is called crosscutting concerns. In computer sciences crosscutting concerns refer to the aspects of plan which overshadow other concerns of plan [3]. These concerns are often hard to be clearly separated from other parts of the system either in design or applying and lead to dispersion and complication.

Introduction of aspect oriented programming

A software system insists of sorts of concerns. Crosscutting concerns separation tries to decline the existing dependencies among these concerns and separate them in order to gain the quantizing goal in system. This act of separation is applicable by many planning languages such as object oriented programming. However, the existing planning languages cannot separate crosscutting concerns. In order to support crosscutting concerns separation aspect oriented programming was introduced. So aspect oriented programming is a paradigm for separating concerns and quantizing crosscutting concerns inside existing the so called aspects, well-defined concerns.

Providing some solutions for crosscutting concerns, aspect oriented programming completes object oriented one. Connection point, cut point, suggestion code and aspect are concepts defined by means of a new layer introduced by this method. It facilitates the ability to cohere crosscutting concerns into aspects and prevents the corresponding code to get scattered. Aspect oriented programming gathers all the data, methods and classes related to concern and provides the ability to put the system into separate unites in the best way.

Aspect oriented programming was defined as a project in PARC in mid-1990. It roots in works aimed at code quantizing and facilitating reuse and preservation [4]. In 1997 Mr. grigure kikzals and colleagues in PARC company managed to define aspect concept and introduced it in OOPSIA conference [5]. The group efforts not only led to introducing an aspect oriented programming, but also an applying of a language called AspectJ with the purpose of introducing an aspect oriented programming methodology available for a large number of developers. Olauph spinch [6] introduced AspectC++ 3aspect oriented programming language that was based on C++ language. Adding an annexed8 on the C++ 3 language he created Aspect C++ planning language.

The concept of aspect

Aspect is a planning unit designed to perform an application which shortcuts practical plan. It is originally a unit preventing dispersion of crosscutting operation code and is often explained as a crosscutting structure. Practical plans which are made of classes and aspects by using aspect oriented programming language. Through the presence of classes and aspects in the plan it is illustrated that quantizing of a plan occur in to aspects: (1) basic operations performed by means of classes. (2) Crosscutting concerns operation performed by means of aspects. Therefore, in a practical plan, since aspect performs a crosscutting concern it differs from class.

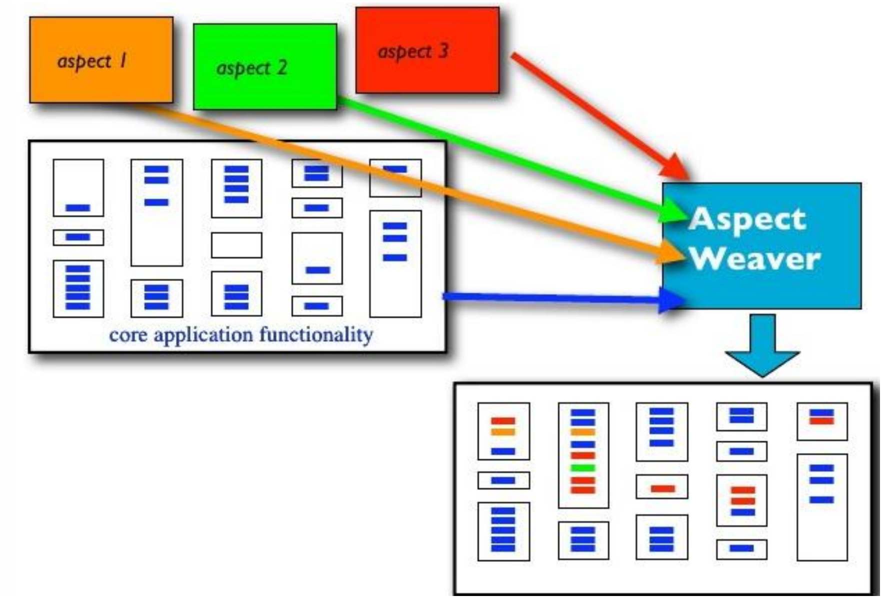

Fig.1. The structure of aspect oriented programming

In aspect oriented programming language aspects are codding separately, and then are weaved by other system members' codes so that the intended system is made (see fig.1). Structurally, an aspect is composed of two parts: advice code and cutpoint. The advice includes a code piece related to a concern that must be applied. A joint point is a position in the plan where an aspect can be added. A set of joint points in the plan is called cut point. In fact, the cut point determines the points where the advice point must be applied.

There are two significant points about aspect; first, an aspect does not directly apply a crosscutting operation, but instead uses an exclusive API. Second, like object, aspect is an abstract concept that is capable of being used in various planning languages.

Design Patterns

There is a common misconception about design patterns, spread among people newly introduced to them, namely that they are fundamental building blocks of software systems. Design patterns are the embodiment of OOP design principles applied to recurrent software design problems. A system is not a sum of patterns but rather patterns provide help in solving problems in system's design. There is a big mistake in trying to take into consideration all changes the system has to accommodate. To allow the evolution of the system, one has to create such a design that would facilitate changes. This is accomplished by encapsulating the variance and separating it from the aspects that do not vary. Variance will only cause limited damage when it happens.

Singleton pattern

The Singleton class is an implementation of the Singleton pattern. Usually, a Singleton has only one instance, but it is not mandatory. It contains a static member of type Singleton Class, named unique Instance, referencing the single instance. The global access point is the static method Instance. The method checks whether an instance has been created, creates one if not, and returns a reference to the instance.

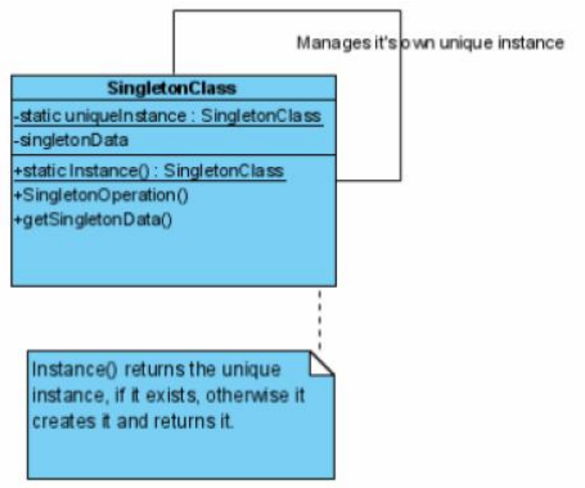
Manages it's own unique instance
SingletonClass
-static uniqueInstance : SingletonClass
-singletonData
+static Instance() : SingletonClass
+SingletonOperation()
+getSingletonData()
Instance() returns the unique instance, if it exists, otherwise it creates it and returns it.

Fig.2. Singleton pattern

Observer pattern

Observer describes a one to many publish/subscribe relationship between objects, one object notifying the others when its state changes. Observer is the interface implemented by all the objects that subscribe for notifications. It contains an update method, called by the publisher when it changes its state. Subject is the interface implemented by the publishers. It contains methods for attaching and detaching subscribers.

When notifying its observers, Concrete Subject, a Subject implementation, sends itself as a parameter to the update method. Hence Concrete Observer, an Observer implementation, uses the Subject parameter of its update method to synchronize its state with the new state of the Subject.

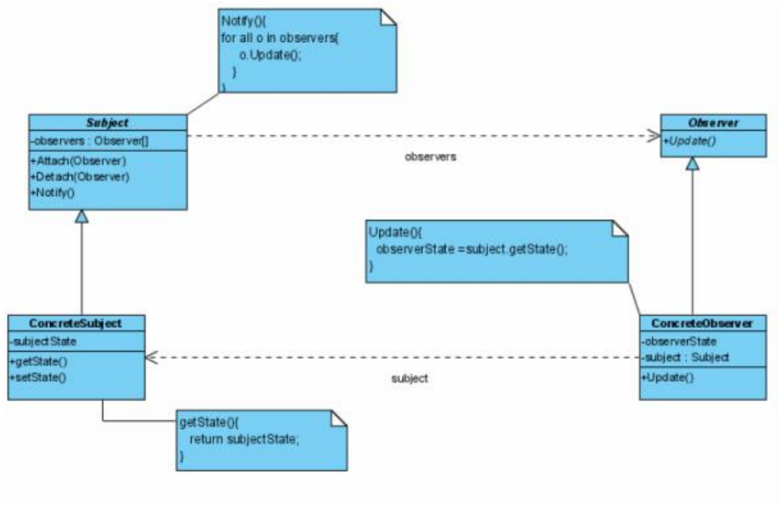

Fig.3. Observer pattern

State pattern

State shows how an object can change its behavior when its state changes. Context is the object changing its behavior. This is achieved by encapsulating the behavior in several objects, each defining a state of the Context and only one being active at a time. The Context delegates to the current state object all the received requests. The State interface exposes a set of operations common to all states and is implemented by concrete state objects.

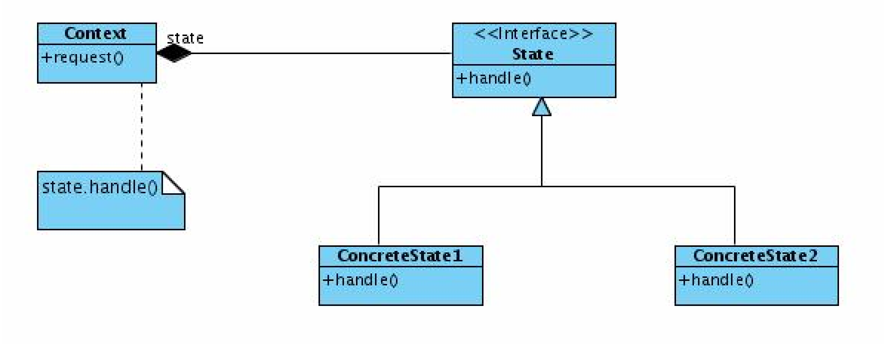

Fig.4. State pattern

Proxy pattern

The Proxy pattern shows how an object can be hidden behind a placeholder or surrogate that exhibits the same interface as the original object. The proxy pattern is used to accomplish different goals, though it has more or less the same structure. As the Iterator, this pattern becomes ubiquitous in almost all modern development platforms, in the form of a dynamic proxy. Proxy pattern implementations are heavily used in the development of run time weaving AOP frameworks. All the objects to be advised are hidden behind proxies, in which the advices' code resides. Dynamic proxies are general solutions for creating proxies for any class type. This flexibility comes with the price of complexity, decreased speed and verbosity; hence developers need sometimes to write their own proxy pattern implementations.

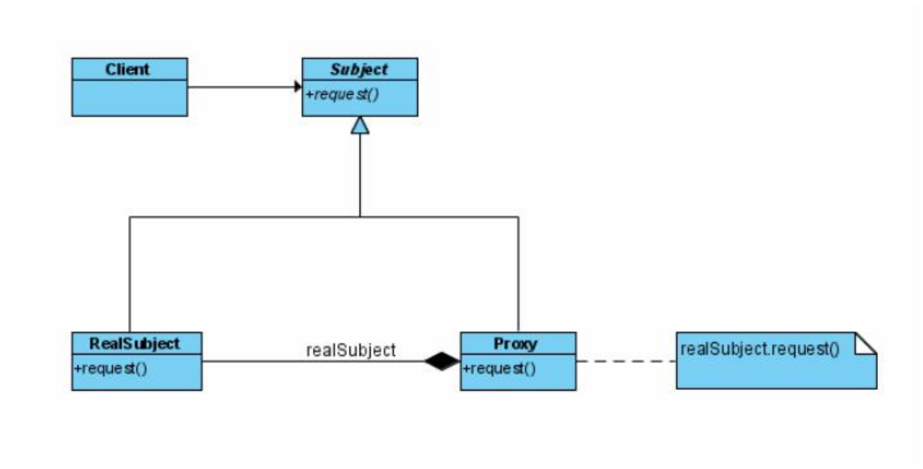

Fig.5. proxy pattern

Aspect oriented programming, metadata, and design patterns

There are two issues to be remarked in AOP's criticism: lack of visibility of program flow and difficult debugging; and tight coupling of aspects to the names of language constructs composing the point cuts, known as "tyranny of the dominant signature"[7]. A common solution, as shown in [7], to both issues is to use the metadata facility of the Java platform introduced in version 1.5, namely annotations, to mark language constructs to be advised. Annotations are a way to decorate Java language constructs with the purpose of providing information in a declarative manner. AspectJ, starting with version 1.5, offers the possibility of using annotations in the point cuts. This approach increases the visibility of the program flow and frees the developer from the burden of the "tyranny of dominant signature". A library of aspects can come with its own set of annotations to be applied on the language constructs to be advised.

The Java language, starting with version 1.5, accepts annotations on several language constructs, like classes, methods, method arguments, class attributes and variables. The limitation is that annotations on local variables are not accessible in the source, class file or runtime. Hence, AspectJ cannot intercept annotated local variables.

2. CONCLUSION

AOP is a programming paradigm which comes as an extension to OOP to allow the encapsulation of crosscutting concerns. As OOP brought the concepts of class, method and attribute, AOP comes with its own set of concepts: pointcut, advice, introduction, aspect. Due to the fact that AOP comes not as a programming language, but as frameworks, in order to apply aspects a compiler-like entity is needed. This entity bears the name aspect weaver and the process is called weaving. The weaver is just one component of an AOP framework. The other one is the specific language used to express AOP specific constructs. Hence, in order to classify AOP frameworks, those two components have to be analyzed. Depending on when the weaving occurs, there are compile time, load time and run time frameworks. As for the specific language, there is a plethora of solutions, ranging from XML files to language extensions. AspectJ is the most successful AOP framework to date. It offers the possibility of compile time or load time weaving. Its specific language is an extension to the Java programming language.

Design patterns are generic solutions to recurrent problems in object oriented design. The "Gof" patterns have the status of classics due to their generality and iniquitousness. One of the most important achievements of these patterns is the creation of a common vocabulary between software engineers. These facts concur to make the "Gof" patterns a choice for proving new technologies. AOP aims to extend OOP making this choice even more evident. AspectJ was chosen to provide the aspect oriented implementation of the 5 "Gof" patterns.

The goal of this article is to use design patterns, AspectJ and metadata, in form of Java annotations, as proof for a solution to overcome two of the most important critics of AOP, namely the "tyranny of the dominant signature" and flow hiding. The tyranny of the dominant signature is the tight coupling of method or type signature to the weaving of aspects. Flow hiding is the lack of information for the developer on where and how aspects are woven. Annotations are used to mark join points to be advised by aspects incorporating the pattern's logic. The results yield the following conclusion: in order to have a beneficial AOP implementation, pattern related code should crosscut the code performing the logic of the participants in the pattern. A significant number of the "Gof" pattern are either generic solutions (Facade, Interpreter) or pure object oriented solutions. The following four patterns offer the most beneficial implementations using AspectJ and annotations: Singleton, Observer, State and Proxy. There is a recurring theme in the design of these patterns: annotations are used to mark and configure the participants, while the aspects hold the patterns' logic. By using annotations, the types involved in the pattern are loose coupled with the aspects. Also, plugging/unplugging the pattern resumes to marking/not marking types with annotations. All pattern related code is separated from the participants and has a higher degree of generality. Another important achievement is the lack of coupling to a specific AOP framework.

3. REFERENCES

[1] S. Vijay, and A. Shetty, "A Study on Different Approaches Towards Aspect-oriented Requirements Engineering," Indian Journal of Computer Science and Engineering, 2011, 2(4), 407- 419.

[2] Sampaio, N. Loughran, A. Rashid, and P. Rayson, "Mining Aspects in Requirements," Proc, IEEE Int'l Workshop on Aspect-oriented Requirements Engineering and Architecture Design, 2005, 62-66.

[3] Schauerhuber, W. Schwinger, E. Kapsammer, W. Retschitzegger, M. Wimmer, and G. Kappel, A Survey on Aspect-oriented Modeling Approaches, Technical Report, Vienna University of Technology, Wien, Austria, 2007.

[4] Lionel Seinturier Renaud Pawlak, "Foundations of AOP for J2EE Development," Springer-Verlag New York, no. ISBN: 1-59059-507-6, 2005.

[5] Olaf Spinczyk, Daniel Lohmann, and Matthias Urban, "Advances in AOP with AspectC++," in Software Methodologies Tools and Techniques (SoMeT '05), Tokyo, Japan, 2005, 33-53.

[6] J.L., Sanchez, F., Toro,M Herrero, "Fault tolerance as an aspect using JReplica," in Proceedings of the Eighth IEEE Workshop on Future trends of Distributed Computing Systems, Los Alamitos, 2001, pp. 201–207.

[7] Hannemann J., Kiczales G, "Design Pattern Implementation in Java and AspectJ, " Proceedings of the 17th Annual ACM conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA), pages 161-173, 2002.

[8] Laddad R.: “AOP and metadata: A perfect match”, pt 2, (http://www.ibm.com/developerworks/java/library/j-aopwork4/index.html).

How to cite this article:
Alipour G, Sangar AB, Mogaddam MH. Aspect oriented implementation of design patterns using metadata. J. Fundam. Appl. Sci., 2016, *8(2S), 816-825.*