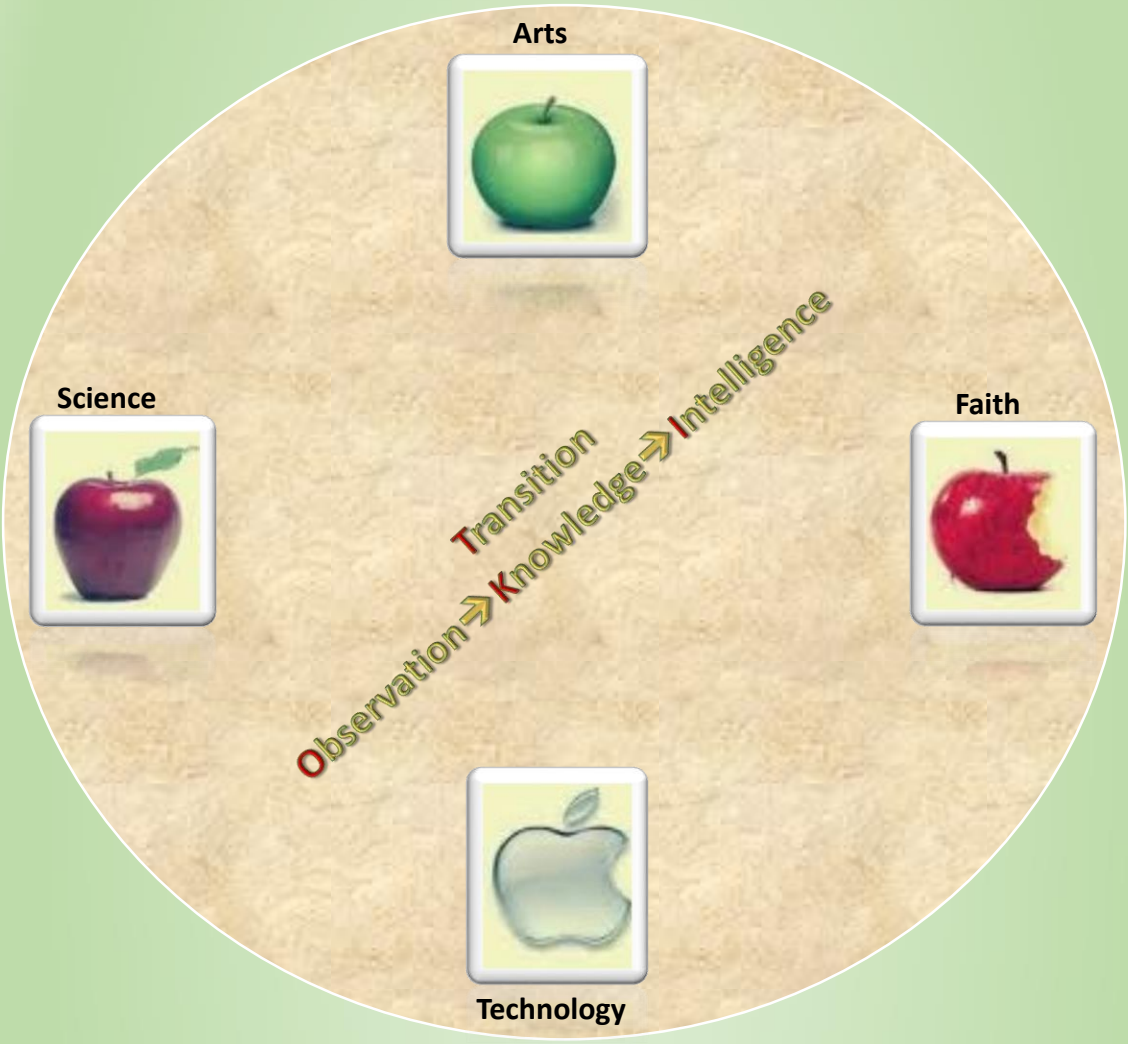

Editors – Prof. Amos DAVID & Prof. Charles UWADIA

Transition from Observation to Knowledge to Intelligence

25-26 August 2016
University of Lagos, Nigeria

Editors

Prof. Amos DAVID
Prof. Charles UWADIA

Cross Cutter Design Pattern for implementing Separation of Concerns in a Distributed System

RUFAI Adewole Usman, FASINA Ebun Philip, UWADIA Charles Onuwa

RUFAI Adewole Usman, FASINA Ebun Philip, UWADIA Charles Onuwa

Department of Computer Sciences

University of Lagos, Akoka, Lagos, Nigeria

Abstract: Achieving a better separation of concerns has been the preoccupation of the software engineering research community. The main goal includes the attainment of modularity and reasoning about software systems amongst others. Many paradigms (aspect oriented programming inclusive) had been proposed to actualize the separation of concerns. Aspect Oriented Programming is characterized by its twin attributes of obliviousness and quantification. It also solves the twin problems of scattering and tangling of codes. However, these attributes according to a section of the research community, sacrifice program understanding and modularity. In this paper we present the Cross Cutter design pattern, based on the existing object-oriented programming technology, as an approach at separating the concerns that crosscut software systems. The pattern was implemented on a distributed system. The two approaches were evaluated using standard paradigm-independent metrics. The metrics used are the separation of concern (SoC) metrics of Concern Diffusion over Components (CDC), Concern Diffusion over Operations (CDO), and Concern Diffusion over Length of Codes (CDLOC). The results obtained suggest that CCDP offers a better separation of concern than the Aspect-Oriented Programming paradigm. The pattern preserves the notion of modularity and a concise way of reasoning about the software amongst other attributes.

Keywords: obliviousness, quantification, scattering, tangling, design patterns, aspect oriented programming.

1. Introduction

Cross cutting concerns are aspects of a program which crosscuts other concerns (Kiczales *et al.* 1997). Kiczales *et al.* (1997) in their inspirational work, proposed the Aspect Oriented Programming paradigm to solve crosscutting concerns in software system. It is based on the construction of aspects as a new mechanism for the compartmentalization of crosscutting concerns. However, while AOP provides lexical separation of concerns (Przybylek, 2010) that solves the twin problems of scattering and tangling of codes, it is fraught with the problems of quantification and obliviousness. These problems actually break the encapsulation principle that is pivotal to separation of concerns. This work is motivated by the views of Steimann (2006) and Przybylek (2010). To illustrate the effect of quantification and obliviousness we consider the logging aspect below as suggested by Laddad (2003).

```
1 public aspect TraceAspectV1 {
2    pointcut traceMethods()
3      : (execution(* *.*(..))
4      || execution(*.new(..))) && !within(TraceAspectV1);
5    before() : traceMethods() {
6      Signature sig = thisJoinPointStaticPart.getSignature();
7      System.out.println("Entering ["
8                + sig.getDeclaringType().getName() + "."
9                + sig.getName() + "]");
10   }
11 }
```

The aspect is made up of the pointcut descriptor, lines 2 to 4 and the advice, lines 5 to 11.The traceMethods() pointcut captures the calls to all methods in the system. The signature **(* *.*(..)** of the **execution** pointcut type quantifies all the methods of the system. This typically takes the control over the modules from the programmer. It is equally impossible to determine by code inspection where aspect codes will execute on the base codes. In this paper we intend to achieve the following:

- The construction of a new design pattern for solving the problem of crosscutting concerns within the framework of the existing object-oriented programming paradigm. This pattern removes the issues of quantification and obliviousness that characterizes the aspect oriented programming technology (Section 2).
- The evaluation of the Cross Cutter Design Pattern and Aspect Oriented Programming implementation of a distributed system using well known software engineering metrics (Section 3).

2. The Proposed Cross Cutter Design Pattern

This paper proposed a new design pattern called Cross Cutter Design Pattern(CCDP). The Cross Cutter Design pattern is a structural pattern. Structural patterns are focused on the organization of the program. The methodology for actualizing the CCDP includes the use of the Formal Concept Analysis (FCA) to assist in the detection of common concerns in software. The concerns are then isolated. The FCA allows the discovering of common concerns on the basis of a rough description of the class. Second, an appropriate class diagram for the domain is constructed. Third, the problem is implemented using the proposed Cross-Cutter Design Pattern (CCDP). The Cross Cutter Design Pattern encapsulates these concerns without the attendant loss in program understanding and modularity. The pattern can be used for logging, tracing, security and other applications.

Gamma *et al.* (1995) in their seminal work opined that a design pattern should possess four essential elements viz:

1. A name that is meaningful to the pattern.
2. A description of the problem area that explains when the pattern may be applied.
3. A solution description of the parts of the design solution, their relationships and their responsibilities.
4. The statement of the consequences of applying the pattern.

Based on these elements, the Cross-Cutter Design Pattern is presented below:

*Pattern name***:** Cross-Cutter

Description: It comprises of the CrossCutterDaemon class that aggregates crosscutters into a list and adds crosscutter objects to the list of crosscutters. It possesses interfaces that connect to the concreteContexts.

Problem description: Many applications are fraught with the presence of concerns that crosscut many modules which often leads to the scattering and tangling of codes in various modules. Several attempts have been made in solving this problem. The most common is the AOP technology which is characterised with the notion of quantification and obliviousness. The notions have been shown to impair modularity and program understanding. The Cross-Cutter design pattern is based on the OOP technology and preserves the notions of modularity and program understanding.

Solution description: The class diagram in Figure 1 shows the UML (Unified Modelling Language) model of the Cross-Cutter design pattern. The pattern is made up of the crosscutter class, crosscutterDaemon and interfaces: *ICrossCutterable*, *ICrossCutter* and *ICrossCutterData.*. The interface *ICrossCuttable* is realized by *Ticks* class. The interface *ICrossCutterData* is realized by the class *ConcreteContext.* The interface *ICrossCutter* realized the class *CrossCutter*. The *ConcreteContext* class is a subclass of the *CrossCutter* class. The *CrossCutter* class depends on the *ICrossCuttable* interface and the *CrossCutterDaemon* class depends on the *ICrossCutter* interface

Consequences: The pattern promotes reuse by hiding implementation. It provides for modularity thus allows developers to reason about their applications. However, it is verbose for some applications.

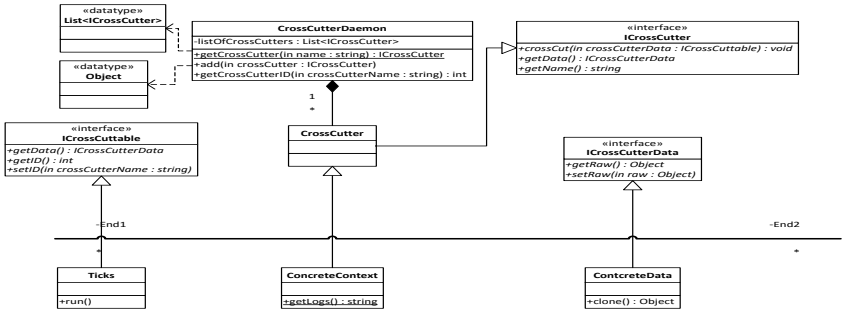

Figure 1. The Generic Class Diagram for the Cross Cutter Design Pattern

2.1. Experimentation

The experimentation is carried out as follows: Given a problem, the first task is to identify the crosscutting concerns in a domain of various concerns. The FCA helps in this regard. Second, an appropriate class diagram for the domain is constructed. Third, the problem is implemented using the proposed Cross-Cutter Design Pattern and the AOP technology for comparison using software engineering attributes of separation of Concerns (SoC), coupling, cohesion and size metrics.

2.2. Cross Cutter Design Pattern on Distributed System

In distributed environment it is desirable to invoke methods on remote objects. The Remote Method Invocation (RMI) provides a platform-independent means of invoking methods on remote objects. With RMI the networking details required by explicit programming of streams and sockets is done away with the fact that an object located remotely is almost transparent to the programmer (Graba, 2007). Once a reference to the remote object has been obtained, the methods of that objects may be invoked in exactly the same way as those of local objects. We illustrate, the use of the Cross-Cutter Design Pattern in a distributed environment using a motivating example. We consider a distributed application that calculates the final value of an investment

from remote machines (clients).Each user is logged on the environment for each transaction. Implementing the RMI is made up of the interface definitions for the remote services; implementation of the remote services; stub and skeleton files; a server to host the remote services; a RMI naming service that allows clients to find the remote services and a client program that needs the remote serves.

2.2.1 The RMI server process and the Cross-Cutter Design Pattern

The class diagram in Figure 2 below shows that, the package *ccdp* is made up of the crosscutter class, crosscutterDaemon and interfaces: *ICrossCutterable, ICrossCutter* and *ICrossCutterData.*. The interface *ICrossCutterable* is realized by the classes *Server* and *Ticks.*

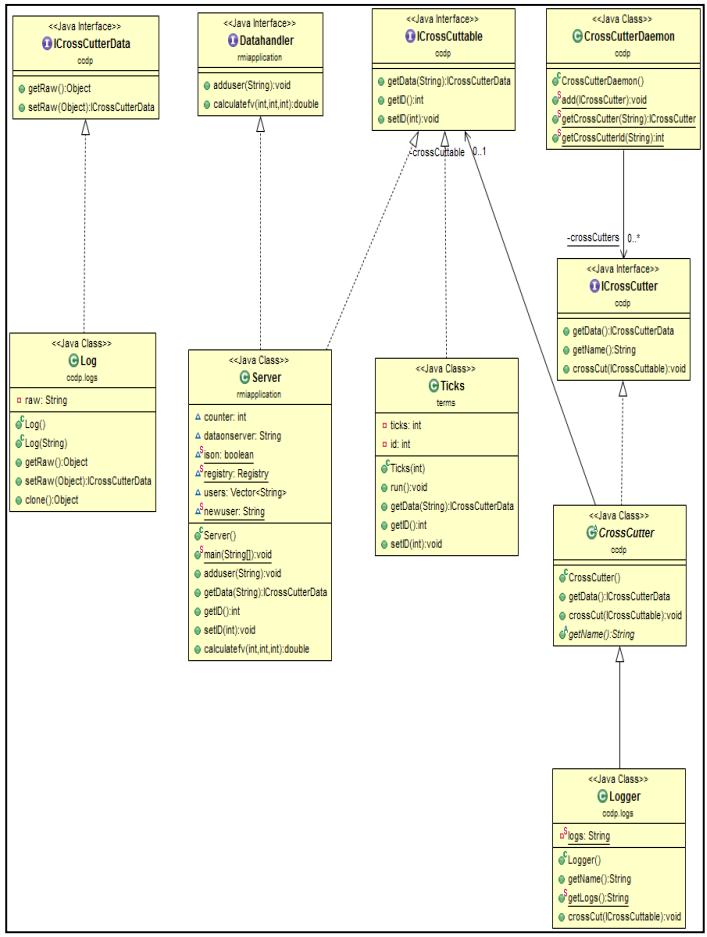

Figure 2: Class Diagram for the server process of The RMI

The interface Datahandler is realized by the class *Server*, the interface *ICrossCutterData* is realized by the class *Log.* The interface *ICrossCutter* realized the class CrossCutter. The *Logge*r class is a subclass of the *CrossCutter* class. The CrossCutter class depends on the *ICrossCuttabke* interface and the *CrosCutterDaemon* class depends on the *ICrossCutter* interface.

This interface should import package *java.rmi* and must extend interface *Remote.* The interface definition for this example must include the signature for the methods *adduser* and *calculate*, which are made available to clients.

The RMI client process and the Cross-Cutter Design Pattern

The client obtains a reference to the remote object from the registry. This is done by the use of method *lookup* of the class *Naming*. This supplies as argument to the method the same URL that the server did when binding the object reference to the object's name in the registry.

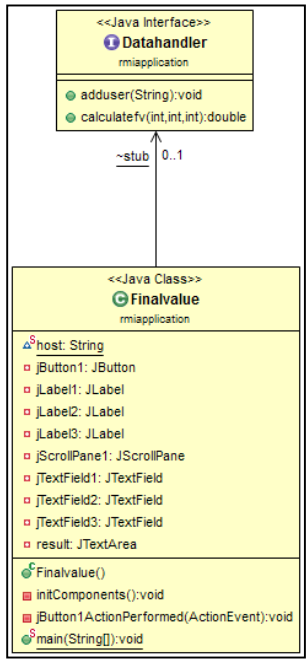

Figure 3: The Class diagram for the client program

2.2.3 The Aspect Oriented Programming Implementation

The AOP implementation of cross cutting concerns is characterised by the localization of those concerns in an AOP construct called the aspect.

2.2.3.1 Server process

The class diagram below depicts the relationship between aspect and the server. The listing below shows the aspect called *Logging* that logs user's transactions. The *pointcut logg()* captures executions to method *adduser()* in the Server class. Once the aspect is compiled along with the Server class, a log message will print to *System.out* after the method is executed.

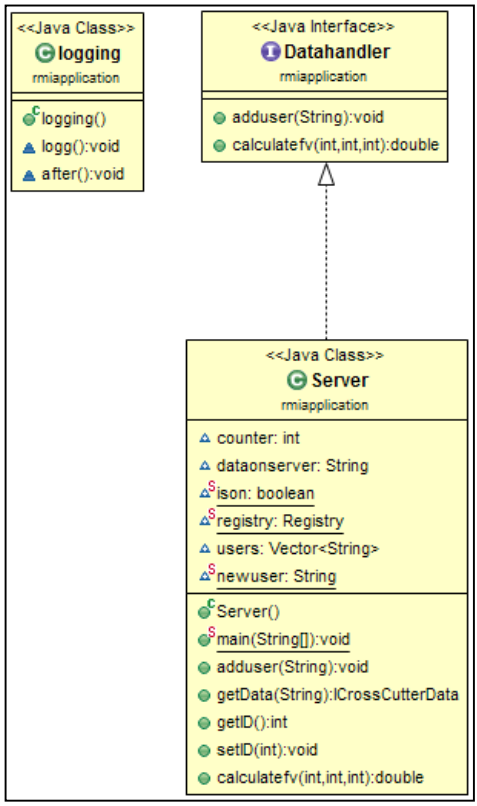

Figure 4: Class diagram for the Server aspect

Listing 1: The logging aspect

```
Package rmiapplication;


Public aspect Logging {
        pointcut logg():  execution(void adduser(String));


        after(): logg(){
                System.out.println("\nmachine " + Server.newuser
                                + " has connected \n");
        }}
```

2.2.3.2 Client process

The class diagram below depicts the relationship between aspect and the server. The listing below shows the aspect called *Hoster* that provides hostname. The pointcut hosting () captures executions to main in the Finalvalue class. Once we compile the aspect along with the *Finalvalue* class, a dialogue box for the hostname is displayed before the method is executed.

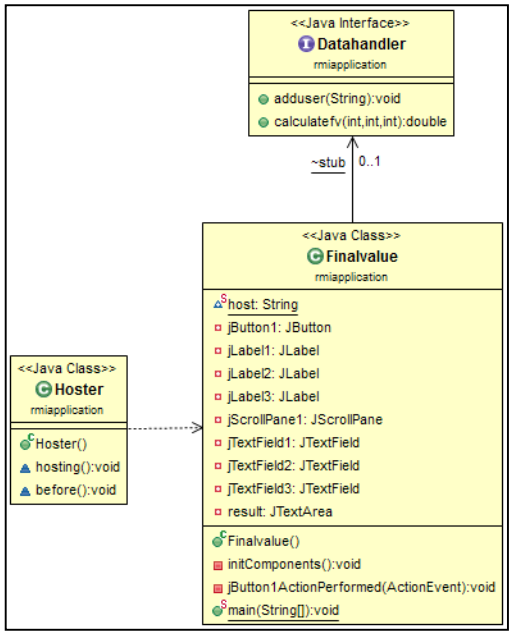

Figure 5: The class diagram for the client

3. 3. Evaluation of the CCDP and AOP approaches for a Distributed System

In the study, Chidamber and Kemerer, (1994) (CK) metrics for separation of concerns was selected to evaluate the (CCDP) implementation and the AOP implementation. These metrics have been used in several studies to measure software. The metrics chosen are paradigm-independent, thus allowing the comparison between the CCDP and AOP implementations. The separation of concerns metrics measure the degree to which a single concern in the software system maps to the design components at varying levels of granularity and the line of codes (Gracia *et al.,* 2005). The concern measures of interest are defined by Sant'Anna *et al.* (2003) are three concern measures which quantify the diffusion of a concern over components, operations, and line of code. Concern Diffusion over Components (CDC) and Concern Diffusion over Operations (CDO) measure the degree of concern scattering at different levels of granularity. The CDC counts the number of classes and aspects related to a concern while CDO counts the number of methods and advices (Figueiredo *et al.,* 2008).

The Concern Diffusion over Lines of Code (CDLOC) measures the degree of concern tangling. This metric counts the number of concern switches for each concern through the lines of code. The table below shows the values of the various CK metrics obtained by code inspection. In the distributed/parallel application the server and the client side were considered.

Table 1 CK metrics for distributed/parallel application.

Metric	CDC	CDO	CDLOC	CBC
CCDP	6	8	4	4
AOP	4	17	4	6

Metric	DIT	LCOM	WOC
CCDP	2	0	4
AOP	0	0	1

The logging concern in both the CCDP and AOP implementation were evaluated. In the SoC metrics the CCDP provided a better result with a lower CDO metrics, which indicates a lower degree of concern scattering at the level of internal component members. However, the AOP presents a better result with the CDC metrics. It is noteworthy here that the effect of quantification on other components by AOP is not taken into consideration.

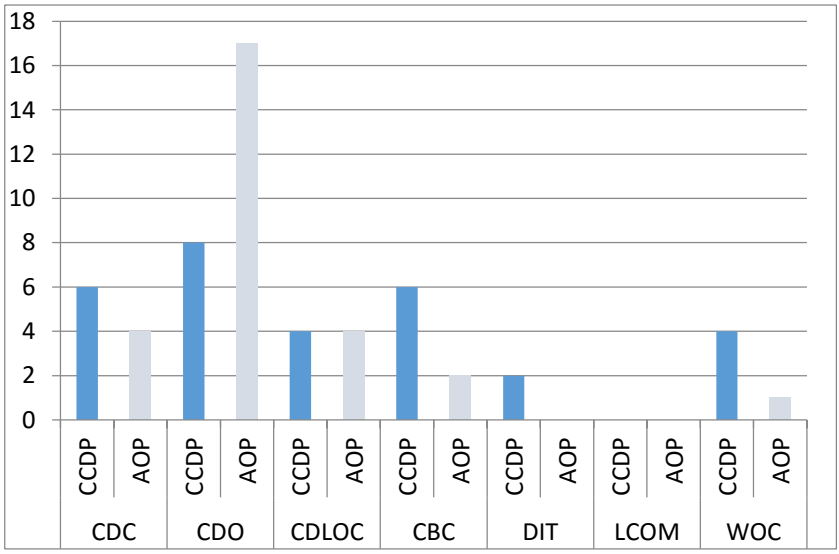

Figure 6: The CK metrics for the Distributed/Parallel application

Both technologies record a tie of 4 in the CDLOC metrics. In the coupling metrics of CBC the AOP provides a better result than the CCDP. The DIT metrics reveals that the CCDP outperforms the AOP. This is a tradeoff scenario in which a higher DIT provides a greater potential for reuse on the one hand, while it constitutes a greater design complexity on the other. The LCOM for both the CCDP and AOP is zero; this indicates that both technologies are cohesive. The AOP gives a better for the WOC metrics.

The results so far reaffirm the CCDP superiority over the AOP in the separation of concerns in distributed/parallel applications.

4. Related Work

Kiczales *et al.* (1997) first proposed the AOP paradigm to solve crosscutting concerns in software system. It is based on the construction of aspects as a new mechanism for the compartmentalization of crosscutting concerns. Filman and Friedman (2000) gave an exposition on AOP being equal to quantification and obliviousness. The study concluded that AOP allows programming by making programming assertions over programs written by programmers oblivious to such assertion. In Kizcales *et al.* (2001), AspectJ -a new extension to the OO java for encapsulating crosscutting concern – was proposed. It solves the problem of tangling and scattering of codes in software systems.

Grisworld *et al.* (2006) introduced crosscutting programming interfaces (XPIs) which help insulate aspects from the details of code they advice and constrain that code to expose behaviors in specified ways. Although to design XPIs no explicit reference to aspects is required, however the problem of quantification and obliviousness persists. There have been several critics of the AOP paradigm; chief among the critics is Steimann (2006) who concludes that the success of AOP is paradoxical.

Various instances where OOP can achieve AOP objectives were itemed, however no example was shown to demonstrate the assertions made. Forster and Steimann (2006) came up with a simple language extension that equips aspectJ with static type check. This leads to the prevention of infinite recursion and nonsensical expressions. However, it possesses the general AOP weaknesses of obliviousness and quantification. Przybylek (2010) opined that AOP provides lexical SoC both does not make software modular.

Przybylek (2011) found that programming abstractions proposed by AOP actually harm software modularity, instead of improving it. Cacho *et al.* (2014) shows that the blending of design patterns results depends on the patterns involved, the composition intricacies, and the application requirements. A key drawback of this approach is that blending of the patterns leads to tangling.

Panunzio and Vardanega (2014) proposed a component –based process with separation of concerns for the development of embedded

real-time systems. The separation of concerns is achieved with the allocation of distinct concerns to software entities of the approach. The strength of this study lies in the fact that the component model supports the separation of concern between functional and non-functional concerns. However, the major drawback is the low level of adoption of the approach. The aim of our work is to separate cross cutting concerns using the OOP technology.

5. Conclusion and Future Work

The first goal of the paper is to design a novel structural design pattern to resolve structural cross-cutting concerns like business rules, logging, information security and so on, in distributed system. The second goal of the paper is on evaluating the design pattern with the view of establishing that it removes obliviousness and quantification by using the CK metrics. These two goals are largely achieved. Our findings suggest that CCDP offers a better separation of concern than the Aspect-Oriented Programming paradigm. The pattern preserves the notion of modularity and a concise way of reasoning about the software amongst other attributes. The future work includes the use of the design pattern in web services, CORBA (Common Object Request Broker Architecture), and Map Reduce platforms.

List of references

Cacho, N., Sant'Anna, C., Figueirodo, E., Dantas, F., Gracia, A., and Batista, T (2014) Blending design patterns with aspects: A quantitative study. The Journal of Systems and Software.

Chidamber, S.R. and Kemerer, C.F., (1994). *A Metrics Suite for Object Oriented Design.*IEEE Transactions on Software Engineering, Vol . 20, No. 6.pp.476-492.

Figueiredo, E.,Silva, B., Sant'Anna,C., Gracia, A.,Whittle,J. and Nunes , D. (2008). Crosscutting Patterns and Design Stability: An Exploring Analysis.

Filman, R.E. and Friedman,D.P.(2000) Aspect Oriented Programming is Quantification and Obliviousness. NASA, USA.

Forster, F. and Steimann, F., (2006). AOP and the Antimony of the Liar. FOAL, Bonn, Germany.

Gamma, E, Helm, R, Johnson R & Vlissides, J (1995), *Design patterns: elements of reusable object oriented software*. Addison-Wesley, New York, NY

Graba, J. (2007), *An introduction to network programming with Java*. Addison-Wesley, New York, NY.

Gracia,A., Sant'Anna, C., Figueirido,E,Kulesza, U., Lucana, C., and von Staa , A.(2005) Modularizing Design Patterns with Aspects: A Quantitative Study. SEL, PUC-Rio.

Griswold,W.G. (2006), Modular software design with crosscutting interfaces. IEEE Software.

Kiczales, G, Lamping, J, Mendhekar, A, Maeda, C, Lopes, CV, Loingtier, J & Irwin, J (1997), 'Aspect-oriented programming', *Proc.European Conf. on Object-Oriented Programming (ECOOP),* Finland, Springer-Velag LNCS 1241.

Kiczales, G, Hilsdale, E , Hugunin, J , Kersten, M , Palm, J, & Grisword, W (2001), 'An overview of AspectJ', *In Knudsen, J.L. , editor, ECOOP 2001- Object-Oriented Programming 15th European Conference*, Budapest Hungary, volume 2072 of Lecture Notes in Computer Science, pp.327-353. Springer-Verlag, Berlin.

Laddad, R (2003), *AspectJ in action: practical aspect-oriented programming*. Greenwich: Manning Publications.

Panunzio, M. and Vardanega, T., (2014). A component-based process with separation of concerns for the development of embedded real-time software systems. Department of Mathematics, University of Padova, via Trieste 63, 35121 Padova, Italy.

Przybylek, A (2010), What is wrong with AOP? *In proc.International joint conference on software technologies.* Athens, Greece.

Przybylek, A (2011), Impact of aspect-oriented programming on software modularity. CSMR.

Sant'Anna, C., Gracia, A., Chavez,C, Lucena, C and von Staa,A.(2003) On the reuse and maintanence of aspect-oriented software: an assessment framework. Technical reports PUC-Rio.

Steimann, F (2006), 'The paradoxical success of aspect oriented language.' *Proc .OOPSLA, U.S.A.*

Transition from Observation to Knowledge to Intelligence (**TOKI**) Conference is a forum that allows researchers from the fields of Competitive Intelligence, Internet of Things (**IoT**), Cloud Computing, Big Data and Territorial Intelligence to present their novel research findings and results. Common to all these fields are the concepts of information, information systems, knowledge, intelligence, decision-support systems, ubiquities, etc. The relevance of research findings, results obtained, systems developed and techniques adopted in these research fields for both the industries and government cannot be overemphasized.

Therefore, the Conference welcomes contributions in the following areas:

- Smart Cities: With focus on Intelligent Transportation Systems, Observatory Systems, Smart Electricity Grids, building automation, assisted living and e-health management systems. Areas such as application of Geographical Information Systems, Territorial Intelligence and Sensors are also considered.
- Big Data Analytics: This includes Big Data, Information Visualization, Data Analysis and related applications.
- Semantic Web: Standardized formats and exchange protocols for web based data.
- IoT Analytics: These center around innovative algorithms and data analysis techniques for extracting meaning and value from the Internet of Things.
- Resource Management: This includes energy saving techniques, effective and efficient utilization of resources, intelligent data processing, mining, fusion, storage, and management, context awareness and ambient intelligence.
- IoT Enabling Technologies: These center around technologies that drive pervasive / ubiquitous systems some of which include but not limited to IPv6, NFC, RFID and Microprocessors.
- Interoperable and Adaptive Information Systems: These include but are not limited to Decision Support Systems, collaborative and co-operative systems and other forms of systems that support interfacing of multiple elements and entities.
- Mobile IoT: Smart phone applications for generating and consuming data, crowd sourced data, e-commerce, mobile advertising, B2B, B2C and C2C connectedness.

Cloud Computing: Including security, storage and access to data stored in the cloud; service provisioning and resource utilization; cloud communication protocols; interoperability among users and devices with respect to linked data.

Editors

Prof. Amos DAVID & Prof. Charles UWADIA

9782954676036