An Analysis of Development Speed of Crosscutting Code using Aspect Oriented Programming

Ayushi Agrawal[1] Ganpat Singh Chauhan[2]
[1,2]Department of Computer Science & Engineering
[1,2]Poornima College of Engineering ISI-6, RIICO Institutional Area Sitapura, Jaipur – 302 022 (India)

Abstract— In Object Oriented Programming language there exists crosscutting concerns (e.g. logging, validation, transaction etc.) across multiple modules causing the code scattering, code confusion and making the system difficult to maintain and to extend. Aspect-Oriented Programming is a new technology and can provide more ideal modularized structure for programming where Aspects can be defined to modularize such concerns. Although aspect orientation claims to permit a better modularization of crosscutting concerns, it is still not clear whether the development time for such crosscutting concerns is increased or decreased by the application of aspect oriented techniques. The study addresses this issue by comparing the development times of crosscutting concerns in OOP techniques and aspect-oriented composition techniques using the Java and AspectJ. In that way, the experiment reveals opportunities and risks caused by aspect-oriented programming techniques in comparison to object oriented ones. After studying the impact of Aspect-Oriented Programming on the development speed for crosscutting code, due to the transparency between aspect-oriented program and the base program it is observed that AOP is suitable for parallel programming, and yet achieves parallelization based on Java multi-thread mechanism. A mechanism can be proposed to achieve automatic parallelization among aspects and base program.

Key words: OOP, Aspect oriented programming, Crosscutting Concerns

I. INTRODUCTION

Aspect Oriented Programming (AOP) is a software development approach addressing certain problems that software developers face when dealing with Separation of Concerns. Separation of concerns is an important thing in any software development process where semantically similar parts of software should be modularized and organized in a proper way to achieve well-built designs. During the last decades, object oriented programming introduced advanced guidelines, terms, patterns and best practices to achieve this goal. Aspect oriented programming on the other hand is a way to address problems which cannot be efficiently solved with object oriented programming, thus AOP complements OOP. Such problems are generally caused by so-called Cross-Cutting Concerns like logging, security, transaction management, caching or debugging, which spread over the entire scope of a set of software modules containing the core concerns, as the following pseudo code represents:

```
Logger logger = Logger.getLogger(...);
TransactionManager tm = tmservice.
getTrascactionManager();
public void addAccount(Account account) {
logger.info("Creating (" + account + ")    Account");try {
                tm.startTransaction(...);
                erp.add(account);
                db.add(account);
                crm.add(account);
                tm.commit();
        } catch (Exception){
                tm.rollback();
                logger.error("Account creation failed");
        }}
```

As we can see, the business code is surrounded with a lot of resource work, including open transaction, logging and exception handling. Thus, the core concerns become "polluted" by code segments which do not really represent the core business logic. The following diagram represents the concerns in a system:

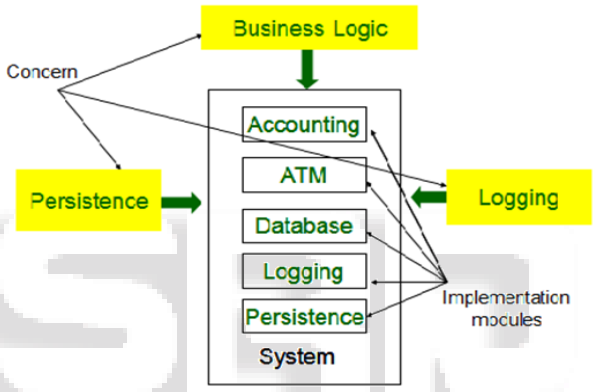

Fig. 1: Concerns of a System

II. CONCEQUENCES OF CROSSCUTTING CODE

- Redundant code: Same fragment of code in many places
- Difficult to reason about: Non-explicit structure
- Difficult to change: Have to find all the code involved.
- Inefficient when crosscutting code is not needed
- Poor traceability: simultaneous coding of many concerns in a module breaks linkage between the requirement and its implementation.
- Lower productivity: developer is paying too much attention to peripheral issues rather than the business logic.
- Less code reuse: cut-and-paste code between modules is the lowest form of reuse.
- Harder refactoring: changing requirements means touching many modules for a single concern.

III. LITERATURE REVIEWS

A. Application Scenario:

The application scenario consists of a bank, its customers, the bank accounts the customers own and the transactions on these accounts. Figure 2 shows how the domain model.

All classes are implemented as Plain Old Java Objects (POJOs):

```
public class Bank {
private ArrayList<Account> accounts;
        public Bank() {
                accounts = new ArrayList<Account>();
        }
        public void addAccount(Account account) {
                accounts.add(account);
        }
        public void removeAccount(Account account) {
                accounts.remove(account);
        }
        }
  public class Account {
        private double balance;
        private Customer owner;
        private String currency;
        public Account(double balance, Customer owner)
   {
                this.balance = balance;
                this.owner = owner;
        }
        public Account(Customer owner) {
                this.owner = owner;
                this.balance = 0;
        }
        public Customer getOwner() {
                return this.owner;
        }
        public void setBalance(double newbalance) {
                this.balance = newbalance;
}
        public double getBalance() {
                return this.balance;
        }
        public String getCurrency() {
                return currency;
        }
        public void setCurrency(String currency) {
                this.currency = currency;
        }
}
public class Customer {
        private String name;
        private String address;
        private ArrayList<Account> accounts;
        public Customer(String name, String address) {
                this.name = name;
                this.address = address;
                accounts = new ArrayList<Account>();
        }
}
```

B. Join Point:

Join points are simply well-defined points in the execution of a program. A join point defines the locations where cross-cutting behavior (advice) may be added. There are different join points proposed by different AOP implementations. AspectJ for example defines execution and invocation of methods or constructors, as well as initialization of objects, classes or fields as join points. Contrarily loops or multiple statements are not recognized as join points in AspectJ.

C. Pointcut:

A pointcut can simply be considered as a set of join points. The real advantage of pointcuts comes into play when they can be defined by combining join points to create semantically useful locations to add cross-cutting behaviour. Considering the Bank.addAccount() method in our application scenario it is possible to create a pointcut which is declared in AspectJ as follows:

```
pointcut addingAccount(Account account):
call(void addAccount(Account)) && args(account);
```

This pointcut will pick up each call to a method named addAccount that takes one parameter of type Account and does not have a return value. Instead of giving the exact definition of the method names and the parameters we can also use wildcards that AspectJ supports to define pointcuts. This allows for defining dynamic pointcuts. Below is another example:

```
pointcut settersToLog():
                execution(void set*(double)) ||
                execution(void set*(String));
```

This pointcut will pick up each execution of a method that has a name starting with "set", does not have a return value and accepts a double or a String as argument. Pointcut definitions can be elaborated and extended by combining join points with boolean operators like & or || but they can also be constrained by specifying the type of the object the pointcut will pick up (by using the keyword target in AspectJ) or by adding parameters and arguments.

D. Aspect:

An aspect encapsulates a cross-cutting concern as a whole and may contain pointcuts, advices plain Java methods or additional field declarations. In the application scenario the cross-cutting concern "logging" can be modeled as an aspect. AspectJ allows for creating aspect implementations very similar to plain Java classes. ALoggingAspect could be defined as follows:

```
  public aspect LoggingAspect {
        // ... aspect specific fields
        // pointcut definitions
   pointcut addingAccount(Account account)
   call(void addAccount(Account)) &&args(account);
   pointcut settersToLog():
   execution(void set*(double)) ||
   execution(void set*(String));
        // an advice implementation
        after(Account account) returning:
addingAccount(account) {
        Logger logger =  Logger.getLogger("trace");
        logger.info("New account added: " + account);
        }
        after() returning: settersToLog(){
        // ... another advice implementation
        }
        // ... other pointcuts, advices or introductions
}
```

E. Weaving:

Weaving basically means processing aspect and non-aspect components of a program in order to produce the desired

output. There are different strategies among AOP systems about how to produce the woven code. The main weaving strategies can be divided into compile-time weaving and run-time weaving. Compile-time weaving involves using a pre-processor to generate woven Java byte code. Run-time weaving on the other hand is the process of weaving and unweaving aspects while the application is running. The main issue in the weaving process is to produce executable code which cannot be distinguished from an ordinary program from the perspective of the underlying interpreter ensure compatibility.

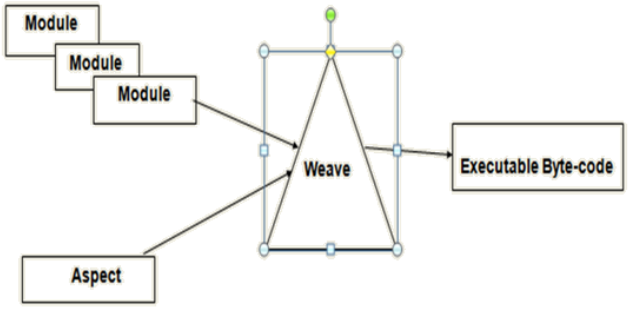

Fig. 2: Weaving in AOP

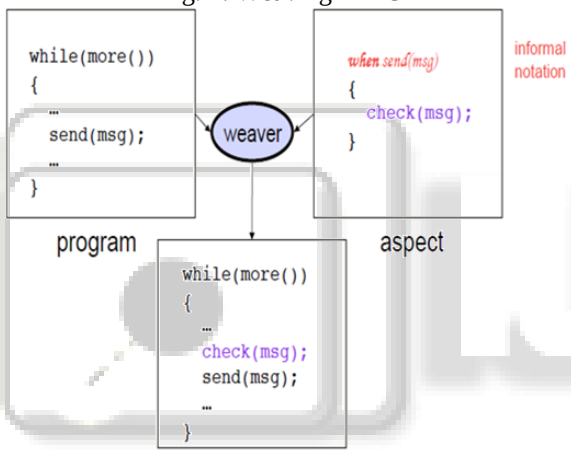

Fig. 3: Simplified view of AOP

IV. Advantages of Aspect Oriented Programming

- Aspect Oriented Programming methodologies provide a convenient means of easily separating crosscutting concerns which is not provided by other programming methodologies and designs.
- The application of aspect-oriented techniques turns out to be time saving for a large number of code clones caused by a crosscutting concern. For a rather small number of code clones we expect that the development speed decreases.
- Aspect Oriented Programming helps overcome system level coding i.e. Logging, Transaction or Security management problem by centralizing these cross-cutting concerns.
- Aspect Oriented Programming addresses each aspect separately in a modular fashion with minimal coupling and duplication of code. This modular approach also promotes code reuse by using a business logic concern with a separate logger aspect.
- Make easier to add newer functionality's by adding new aspects and weaving rules and subsequently regenerating the final code. This ability to add newer functionality as separate aspects enable application designers to delay or defer some design decisions without the dilemma of over designing the application.
- Rapid development of evolutionary prototypes using OOP by focusing only on the business logic by omitting cross-cutting concerns such as security, transaction, logging etc. Once the prototype is accepted, additional concerns like security, logging, auditing etc can be weaved into the prototype code to transfer it into a production standard application.
- Developers can concentrate on one aspect at a time rather than having to think simultaneously about business logic, security, logging, performance, multithread safety etc. Different aspects can be developed by different developers based on their key strengths.

V. Conclusion

Aspect-oriented techniques have been applied widely today, but the efficiency of the woven-code is still unsatisfied. This limits applications of aspect-oriented technology in software monitoring, software fault-tolerant and so on, especially in the field of time-sensitive application. With the extensive use of multi-core platform, it is a good solution to use this platform to reduce the impaction on efficiency of aspect-oriented code. As the relations between core modules and cross-cutting modules are orthogonal, it makes the parallelization of these two types of codes possible. But it is not always orthogonal in vertical way and the data exchange often exists between the base program and the aspect code, which makes automatic parallelization compiling complicated. In this paper, we point out and explore some issues in the parallelization of aspect-oriented program, and give the corresponding solution in a respective manner. Our further work is divided in two directions: First, based on current job, find out a more general solution of this problem, and implement the automatic tool for parallel compiling. Second, we want to find a way to connect multicore platform and aspect-oriented programs. Multi-core hardware platform is a major trend, the problem of how to make better use of the platform to run aspect-oriented program, and get the best performance of the program, so as to solve the current efficiency problem which now widely exist will be our focus in the future.

VI. Acknowledgment

I feel greatly indebtedness to my teachers, who are constant source of inspiration and for their moral support, invaluable suggestion during my seminar study.

The preparation for this report was made possible by the cooperation of my many colleagues and friends without which the study might never have been completed. There is not enough space to list all but their support and encouragement is in my heart.

We also want to present our gratitude to Mr. Amol Sexena, HOD, Department of Computer Science and Engineering, who gave us moral support and guided us in different manner regarding the topic. He had been very kind and patient while suggesting us the outlines of this paper

and correcting our doubts. We thank him for his overall support.

We are also grateful to our respected Campus Director Dr. O. P. Sharma who gave us the opportunity to implement our ideas and guided our team to put our efforts in right direction and all those who have helped us directly or indirectly in our endeavors.

REFERENCES

[1] Aspect-Oriented Programming, the European Conference on Object-Oriented Programming (ECOOP), Finland. Springer-Verlag LNCS 1241. June 1997.
[2] Third International Symposium on Empirical Software Engineering and Measurement, Stefan Hanenberg, Sebastian Kleinschmager, Manuel Josupeit-Walter, University of Duisburg-Essen
[3] Laddad, R.: AspectJ in Action: Practical Aspectoriented Programming, Manning, 2003
[4] www.elsevier.com/locate/infsof
[5] Stefan Hanenberg, Sebastian Kleinschmager, Manuel Josupeit-Walter- "Does Aspect-Oriented Programming Increase the Development Speed for Crosscutting Code? An Empirical Study".
[6] Bartsch, M.; Harrison, R.: "An exploratory study of the effect of aspect-oriented programming on maintainability, Software Quality".
[7] He Tengfei Mao Xiaoguang – "Research on Parallelization of Aspect-Oriented Program".
[8] Lu Yang et al. "A Case Study for Monitoring-Oriented Programming in Multi-core Architecture".

