

FOURTH TURKISH ASPECT-ORIENTED SOFTWARE DEVELOPMENT WORKSHOP PROCEEDINGS



December 29, 2009

Ankara, Turkey

Bedir Tekinerdoğan (ed.)
Department of Computer Engineering
Bilkent University
06800 Bilkent, Ankara, Turkey



Bilkent University

Table of Contents

Preface.....	1
• <i>Aspect-Oriented Implementation of an ATM System</i> Ahmet Çağrı Şimşek, Melihcan Türk	3
• <i>Refactoring a Remote Password Management System</i> Abdullah Atmaca, Muhammet Kabukcu, Cem Mergenci	12
• <i>AOP Approach for Modeling Crosscutting Concerns in Gaming Applications</i> Selçuk Onur Sümer, Alper Karaçelik	21
• <i>Aspect-Oriented Application of Command Control System</i> İskender Yakın, Bahar Pamuk	29
• <i>Aspect-Oriented Banking System</i> Buğra Mehmet Yıldız, Çağrı Toraman, Uğur Bilen	37
• <i>Aspect-Oriented Refactoring of a J2EE Framework for Security and Validation Concerns</i> Başak Çakar, Elif Demirli, Şadiye Kaptanoğlu	45
• <i>Aspect-Oriented Refactoring of Vector Image Drawing Tool Framework: Joodle</i> Kemal Eroğlu, Aytuğ Murat Aydın, Mehmet Ali Abbasoğlu	57
• <i>Object-Oriented Refactoring vs. Aspectual Refactoring of Legacy Code</i> Duygu Sarıkaya, Can Ufuk Hantaş, Dilek Demirbaş.....	68
• <i>Aspect-Oriented Wireless Network Graph Simulator</i> Abdullah Bülbül, Ömer Faruk Uzar, Utku Ozan Yılmaz	79

Preface

During the last years several international conferences and workshops have been organized around the topic of aspect-oriented software development. In Turkey, we have started the organization of the First National workshop on Aspect-Oriented Software Development Workshop (TAOSD) in June 2003. The second TAOSD workshop has been organized in September 2007. The Third TAOSD workshop has been organized at Bilkent University in December 2008. This is the report of the fourth workshop that has again been organized at Bilkent University in Ankara.

The workshop has been organized within the AOSD (graduate) course that is given at Bilkent University. The course provides an in-depth analysis of AOSD and teaches the state-of-the-art AOSD techniques. One of the basic requirements for fulfilling the requirements of the course was an aspect-oriented software development project. Students had to form groups of three and select complex cases from an industrial project or ongoing academic projects. These cases were analyzed for crosscutting concerns and refactored to an aspect-oriented implementation. This resulted in a unique collection of valuable aspect-oriented case studies which shows the strengths and weaknesses of AOP. The project results have been presented as workshop papers in this proceeding of the TAOSD workshop. Because most students in the AOSD course wrote a workshop paper for the first time they got a course on how to write workshop papers and got extensive feedback on their papers. The primary goals of this workshop were (1) sharing knowledge and improve consciousness on AOSD, and (2) Stimulate research and education on AOSD in Turkey.

About 50 participants both from industry and academic institutes have registered for this workshop. The program included four presentations sessions including 9 paper presentations. Each of them took 30 minutes including a demonstration of the aspect-oriented implemented systems, and questions from the audience. The workshop was concluded with a plenary session in which we summarized the workshop results.

After organizing four national workshops on AOSD we have reflected on the aspects that have been identified, implemented and presented during the workshops. The result is Aspect Catalog – a catalog with a list of all the aspects in the TAOSD workshops. The Aspect Catalog provides a unique opportunity for researchers and developers to analyze the aspects in the corresponding papers and where possible reuse these in their own context.

The workshop had the same format as the workshop in 2003 and 2008. Again, the unique character of this workshop was based on two factors. First of all, the presenters of the workshop were students and not, as in a usual workshop setting, experienced researchers. Second, the audience consisted mainly of persons who were interested in AOSD but had not used it before. This event would not have been possible without the help and support of many people. First of all I would like to thank the Department of Computer Engineering, Bilkent University for actively supporting this event. Further, I would like to thank all the participants to the workshop for sharing this unique event with us. Of course, the workshop would not have been successful without the enthusiasm and the hard work of the CS586 Aspect-Oriented Software Development Course students. I would like to thank them one by one: Başak Çakar, Elif Demirli, ŞadiyeKaptanoğlu, Uğur Bilen, Çağrı Toraman, Buğra Mehmet Yıldız, Ahmet ÇağrıŞimşek, Melihcan

Türk, Alper Karaçelik, Selçuk Onur Sümer, Mehmet Ali Abbasoğlu, Aytuğ Murat Aydın, Kemal Eroğlu, Muhammed Abdullah Bülbül, Ömer Faruk Uzar, Utku Ozan Yılmaz, Abdullah Atmaca, Muhammet Kabukçu, Cem Mergenci, Dilek Demirbaş, Can Ufuk Hantaş, Duygu Sarıkaya, Bahar Pamuk, İskender Yakın.

The workshop has been organized as an event of Bilsen – Bilkent Software Engineering Group, which has been recently founded at Bilkent University. The main goal of the group is to foster research and education on software engineering and support ongoing activities in these directions. This was our first event. More events will follow in the future.

Bedir Tekinerdogan
Bilkent University, Ankara, Turkey
December 31, 2009

Aspect-Oriented Development of an ATM System

Ahmet Çağrı Şimşek, Melihcan Türk
Department of Computer Engineering
Bilkent University
{asimsek, mturk}@cs.bilkent.edu.tr

Abstract

Automated teller machine (ATM) software provides a simple and useful interface which facilitates the remote banking transactions. The development of an ATM system is a complex process involving several concerns like security, transactions, session handling, logging and constraint checking. It appears that those concerns cannot be easily localized in single module and crosscut multiple modules. This includes maintenance, reuse and understanding of the system. An emerging software design paradigm called Aspect-Oriented Programming offers a reasonable solution in order to separate concerns in the design process. In this work, we try to apply Aspect-Oriented Programming approach to the ATM Simulation software.

1. Introduction

ATM (automated teller machine) is a device that is connected to the bank network which enables the clients of a bank with access to banking transactions in a public space without the need for a cashier, human clerk or bank teller. On most modern ATMs, the identification of customers performed by inserting a plastic ATM card with a magnetic stripe or a plastic smart-card with a chip, that contains a unique card number and some security information, such as an expiration date. Authentication is provided by the customer entering a personal identification number (PIN). Using an ATM, customers can access their bank accounts in order to make cash withdrawals and check their account balances as well as depositing money to their accounts and performing money transfers.

ATMs typically connect directly to their ATM Controller via either a dial-up modem over a telephone line or directly via a leased line.

A typical ATM consists of mechanical, hardware and software components like cash dispenser, customer console, receipt printer, card reader and a computer to run the software. Previously, several different programs and operating systems were developed for specific ATM hardware platforms. Nowadays with the common use of commodity software, standard "off-the-shelf" operating systems and programming environments are used such as Microsoft Windows XP, various Linux distributions and Java Environment.

In this paper we analyzed software that is designed for simulating an ATM System using Object Oriented Design techniques. What really matters in a good software design is that the developed software must be understandable, maintainable, extensible, reusable and adaptable. To achieve these we make use of principles of software engineering such as abstraction, decomposition, encapsulation and separation of concerns [3] for low coupling and high cohesion. Software engineers and designers used various techniques and methodologies like Object Oriented Design and Design Patterns. Even though they helped a lot, there are still some issues that cannot be solved by this technique. One of them is the difficulty of the modularization of crosscutting concerns. In this point Aspect Oriented Software Development comes into play and provides solutions for the problem of separation of concerns.

The rest of this paper is organized as follows: Section 2 gives a detailed background about the ATM domain and the requirements. Section 3 gives an overview of the object oriented design of the system and the used design patterns. Section 4 identifies the aspects in the system. Section 5 provides an Aspect Oriented solution for the problem of separation of concerns. Further discussion is placed in section 6. We finally provide our conclusion in section 7.

2. ATM Software

The ATM Simulation [2] software is developed to control a simulated automated teller machine (ATM) which has various components for handling different kinds of operations. The first component of the machine is a magnetic stripe reader which is used for reading an ATM card. The machine also has a customer console which is used in order to interact with the customer. Moreover, there is a slot on the machine for depositing envelopes and a dispenser for cash. There is also a printer which is used for printing customer receipts on the machine. The last component is a key-operated switch for allowing an operator to start or stop the machine. The ATM communicates with the bank's computer over an appropriate communication link.

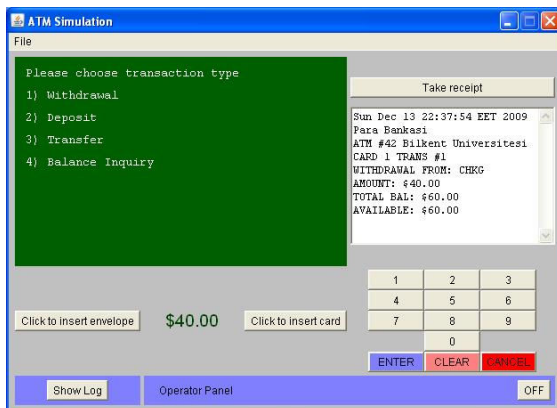


Figure 1: ATM Simulation

The ATM serves one customer at a time. In order to use the ATM, a customer is firstly required to insert an ATM card and enter a personal identification number (PIN). After that, the customer is able to perform the transactions. The card is retained in the machine until the customer specifies that he/she desires to do no more transactions.

Before making each transaction, the ATM tries to communicate to the bank. If the connection is not enabled, the software prevents the transaction and displays a warning message. If the connection to the bank is working, the ATM gets verification that the specified transaction is allowed by the bank. The transaction is prevented if the card is not valid or the PIN is incorrect. If the bank determines that the customer's PIN is invalid, the customer will be required to enter the PIN again in order to be allowed to make the transaction. If the customer cannot successfully enter the PIN after three tries, the card will be permanently retained by the machine, and the customer will have to contact the bank to get it back. The controls which are mentioned are the security concerns of the system.

The bank also looks at the available balance before performing the transactions such as withdrawal and transfer. If there is not enough balance, it prevents these operations. Moreover, the daily limit of the customer is checked before doing the withdrawal operation. The bank also checks whether the selected account is available for the inserted card number. All of these controls are related with the constraint checking concerns.

The ATM provides the customer with a printed receipt for each successful transaction, showing the date, time, machine location, type of transaction, account(s), amount, and ending and available balance(s) of the affected account. After each transaction, it also asks the customer whether he/she wants to do another transaction.

The ATM has a key-operated switch that will allow an operator to start and stop the machine. After turning the switch to the "on" position, the operator will be required to verify and enter the total cash on hand. The machine can only be turned off when it is not servicing a customer. When the switch is moved to the "off" position, the machine will shut down, so that the operator may remove deposit envelopes and reload the machine with cash, blank receipts, etc.

The ATM also keeps a log of transactions to solve the ambiguities arising from a hardware failure in the middle of a transaction. Entries are written in the log for each message sent to the Bank, for each response coming

from the Bank, for the dispensing of cash, and for the receiving of an envelope. Log entries contain card numbers and dollar amounts, but for security never contain a PIN.

3. Object-Oriented Design Overview

3.1. Object-Oriented Design

The existing ATM simulation system has an object oriented design and implementation representing typical properties of object oriented design paradigm. The simplified packages (Figure 2), use cases (Figure 3), class diagram (Figure 4) and UML diagrams of the design patterns are given in the figures.

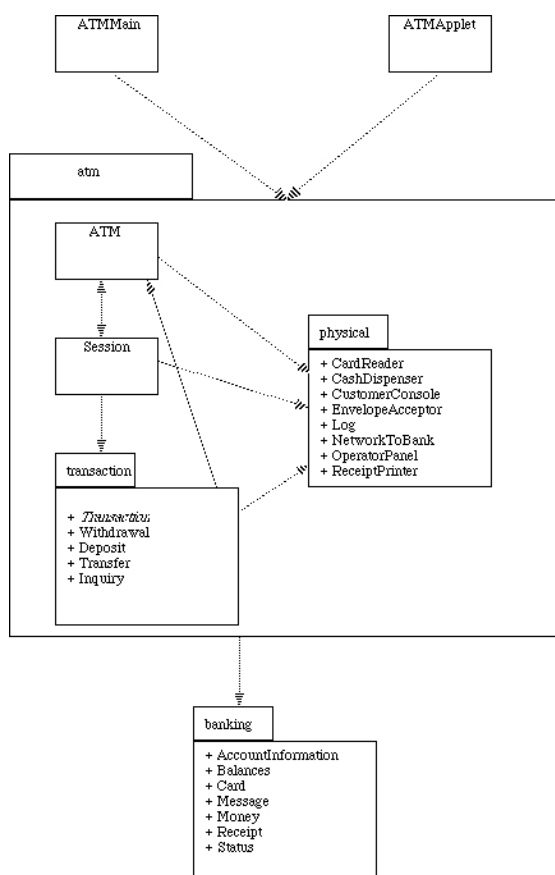


Figure 2: Package Diagram

The system mainly comprises of packages like atm which is the core component, session which provides a session per customer, transaction consisting of allowed transaction types, physical for the physical components like receipt printer and cash dispenser, banking providing banking related things like money and account, simulation to simulate the actual remote bank center.

We have three types of actors in the use cases. Customer being the client consuming the services provided by the system like transactions through the atm session, operator responsible for maintenance of the atm machine by putting cash into the cash dispenser and switching on and off the machine, simulated bank providing the remote end.

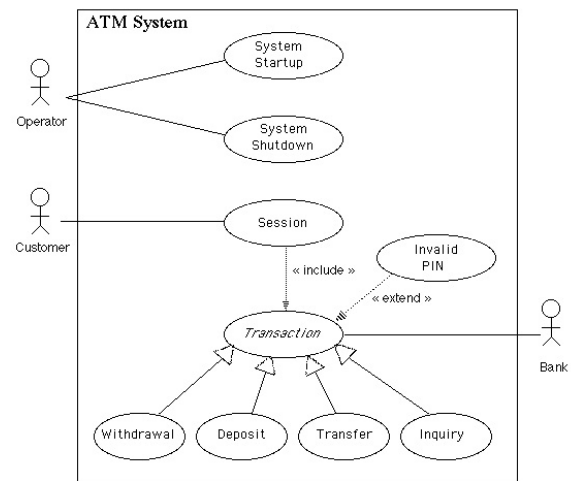


Figure 3: Use Cases

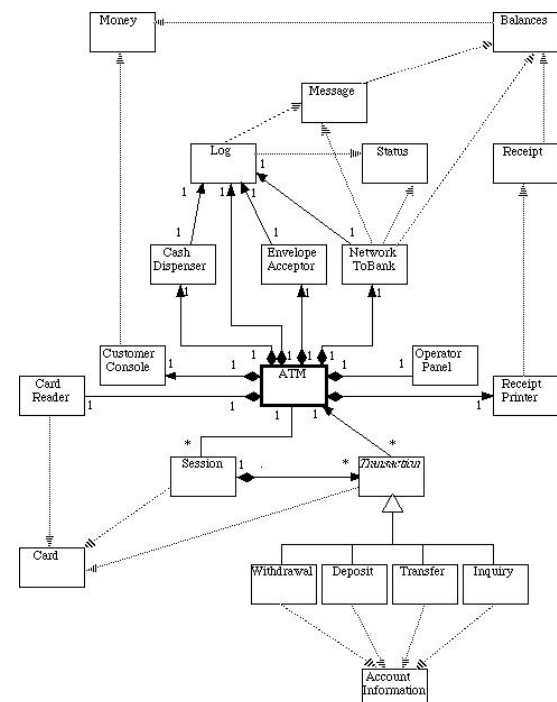


Figure 4: Class Diagram

ATM the main class of the system, accommodates most of the classes in the system and runs a main thread for handling customer sessions and transactions. Session

class provides a session for each customer when they insert their cards by handling the states of their session. Base class Transaction and its subclasses provide different banking transactions. NetworkToBank class handles the communication between the atm and the simulated bank. SimulatedBank class responses to the messages of the atm in order to complete client transactions. Log class logs the events occurred and their consequences which are important there for needs to be logged.

3.2. Design Patterns

Design patterns identify, name, and abstract common themes in object-oriented design. They capture the intent behind a design by identifying objects, their collaborations, and the distribution of responsibilities. Design patterns play many roles in the object-oriented development process: they provide a common vocabulary for design, they reduce system complexity by naming and defining abstractions, they constitute a base of experience for building reusable software, and they act as building blocks from which more complex designs can be built. [1]

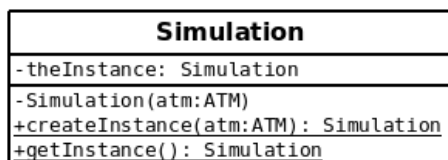
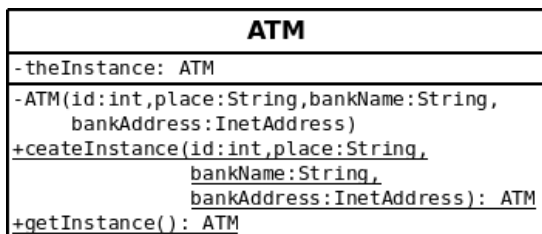


Figure 5: Singleton Pattern

3.2.1. Singleton Pattern

The singleton pattern is a design pattern that is used to restrict instantiation of a class to one object. [4] Here we have two classes

which need to have only one instance and they are the ATM and the Simulation. ATM class must have only one instance because it consists most of the objects, runs the main thread and manages resources. Again the Simulation class simulates most of the remote banking components and there must be only one bank center.

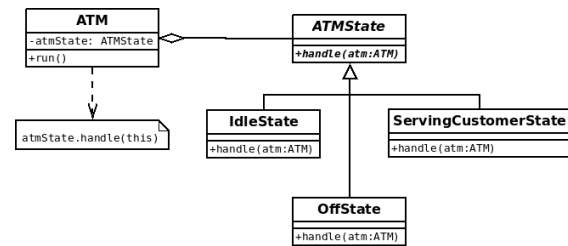


Figure 6: State Pattern

3.2.2. State Pattern

The state pattern is a behavioral software design pattern, also known as the objects for states pattern. This pattern is used in computer programming to represent the state of an object. This is a clean way for an object to partially change its type at runtime. [5] In the system in several places like main atm thread and performing session we have different states and there are different things to do in each different state. For example atm has off state in which atm is not available untill an operator puts money in it and switches it on, it has idle state in which atm is switched on but there is no customer using it and atm has serving customer state in which an actual customer is using it. Also when performing a session we have states such as reading card, reading pin, choosing transaction, performing transaction and ejecting card.

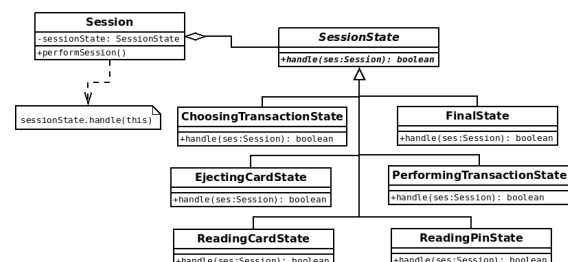


Figure 7: State Pattern

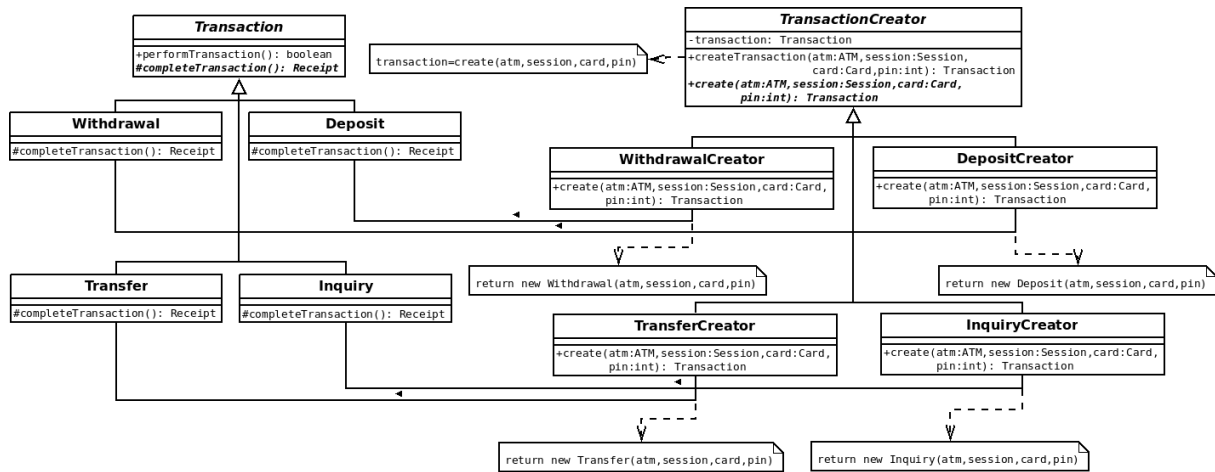


Figure 8: Factory Method Pattern

3.2.3. Factory Method Pattern

The factory method pattern is an object-oriented design pattern. Like other creational patterns, it deals with the problem of creating objects (products) without specifying the exact class of object that will be created. [6] In the customer session, customers choose the type of transaction they want to perform. The appropriate type of transaction which handles the specifics of that type must be created and returned to customer. In order to free the Session class from creating transactions and make it not to worry about what kind of transaction it is creating Factory Method Pattern is applied.

4. Aspect Identification

Although one concern should be handled by one object, more than one objects can be used to deal with one concern in the Object-Oriented Design. In the ATM software, these security concerns are handled by the NetworkToBank and the Simulated-Bank classes. So, this situation causes scattering of the code.

4.1. Constraint checking

The ATM software is also responsible for checking some constraints before doing the transactions. The first one of these constraints is the amount of the money in the relevant account. The withdrawal and the transfer transactions should be blocked if the balance of the specified account is less than the amount which is selected by the customer.

In section 2, we identified the concerns of the ATM software. These concerns crosscut over multiple modules and need to be modularized in order to increase the cohesion and decrease coupling of the components. In this section, we observe these concerns in detailed way.

4.2. Security Handling

Before each transaction, the ATM checks whether the connection to the bank is enabled. If the connection is disabled, it shows a warning and blocks the transaction. If it is working, the bank checks whether the card is valid and the PIN is correct. If the card number is not valid or the PIN is incorrect, the transaction will be prevented.

The other constraint which should be checked is the daily limit of the account. If the customer exceeds this limit, the ATM software should not allow the customer to withdraw and transfer money from any account.

The last constraint which should be considered before making the transaction is checking the availability of the account. In the system, there are three types of account which are checking, savings and money market. The customers do not need to have all of these accounts. So the system should check whether the customer has the account which is selected in the ATM.

All of these concerns are handled in the SimulatedBank class which is also responsible for updating the account balances after each transaction. So, it causes the

tangling of the code because one class handles more than one concern.

4.3. Transaction Handling

As we mentioned before, the ATM should provide a printed receipt after each successful transaction. The receipt includes the date and time of the transaction, the location of the ATM machine, the type of the transaction, the relevant account(s), the amount entered by the customer, and ending and available balance(s) of the affected account.

After each transaction, the ATM software should ask the customer whether he/she wants to do another transaction. According to the answer, it returns to the asking transaction state or ejects the card of the customer.

Although we try to separate the concerns into different objects, more than one concern can be handled by one class. In the Object-Oriented Design of the ATM software, the concerns of making transaction, printing receipt and asking to do another transaction are handled in the Transaction class. This situation causes low cohesion and tangling of the code.

4.4. Logging

The ATM also maintains an internal log of transactions. The operations are written in the log after each transaction, cash dispense, and receiving envelope.

The logging concern is handled by three classes which are NetworkToBank, CashDispenser, and Envelope-Acceptor. These classes are also responsible for performing the relevant transactions. Tangling and scattering of the code occurs because of these conditions.

5. Aspect Oriented Programming

In this section, we solve the problem of crosscutting concerns by using Aspect-Oriented Programming approach.

5.1. Security Handling

In order to modularize the security concerns, we firstly define two pointcuts. The first pointcut captures the joinpoint where the ATM sends message to the bank. The advice

of this pointcut controls the availability of the connection to the bank.

The second pointcut (Figure 9) captures the operations of `withdrawal()`, `initiateDeposit()`, `completeDeposit()`, `transfer()`, and `inquiry()` of the `SimulatedBank` class. `around()` advice of the aspect compares the PIN which is entered by the customer with the PIN of the account number. If the PINs are different, it returns failure message. Otherwise, it proceeds the relevant operation.

```
pointcut securityCheck( SimulatedBank sb,
                        Message message ):
    call(private Status SimulatedBank.*(..))
    && args(message, ..)
    && target(sb);

Status around( SimulatedBank sb,
               Message message):
securityCheck( sb, message )
{
    Status result = sb.checkCard(
        message.getCard().getNumber()
    );
    if(result.isSuccess())
    {
        Status tmp = proceed(sb,message);
        if(tmp != null)
        {
            result = tmp;
        }
    }
    return result;
}
```

Figure 9: Security Handling Aspect

5.2. Constraint Checking

In order to increase cohesion of the code, we define an aspect for constraint checking. We define three pointcuts for checking the availability of the account, the account balance, and daily limit of the account. The first pointcut deals with checking the availability of the account. So it captures each transaction function in the `SimulatedBank` class. In the advice part of this pointcut, whether the account is available for inserted card is determined.

The joinpoint of the withdrawal function in the `SimulatedBank` class is captured by the second pointcut. The advice of this pointcut checks the daily limit of the customer. If the amount which is wanted to be withdrawn exceeds the daily limit, the advice returns a failure message. Otherwise, the function proceeds.

The third pointcut (Figure 10) of the aspect captures the `Withdrawal` and `Transfer`

functions of the `SimulatedBank` class. In the advice of the pointcut, the amount entered by the customer is compared with the available balance. If the amount is more than balance, the `around()` advice returns a failure message. Otherwise, it allow the function to proceed.

```
pointcut confirmBalance( SimulatedBank sb,
                        Message message ):
(call(private Status transfer( Message,
                               Balances ))
 || call(private Status withdrawal( Message,
                                   Balances )))
&& args(message, ..)
&& target(sb);

Status around( SimulatedBank sb,
               Message message):
confirmBalance(sb, message)
{
    Status result = sb.confirmBalance(
        message.getAmount(),
        sb.getAccountNumber(
            message.getCard().getNumber(),
            message.getFromAccount()
        ));
    if(result.isSuccess())
    {
        Status tmp = proceed(sb,message);
        if(tmp != null)
        {
            result = tmp;
        }
    }
    return result;
}
```

Figure 10: Constraint Checking Aspect

5.3. Transaction Handling

In the `TransactionHandling` aspect, we define two pointcuts. Both of them capture the `performTransaction()` method of the `Transaction` class. However, they are responsible for performing two operations. The advice of the first one is created for asking the customer whether he/she wants to do another transaction.

```
pointcut transaction(Transaction transaction):
call(* Transaction.performTransaction(..)
&& target(transaction);

after(Transaction transaction):
transaction(transaction)
{
    if(transaction.completed)
    {
        Receipt receipt = null;
        try
        {
            receipt =
                transaction.completeTransaction();
        }
        catch(CustomerConsole.Cancelled e){ }
        transaction.getATM().getReceiptPrinter()
            .printReceipt(receipt);
    }
};
```

Figure 11: Transaction Handling Aspect

The second advice (Figure 11) is defined to providing receipt to the customer. After the operation of `performTransaction()`, it generates a receipt by using the transaction object which is the parameter of the pointcut.

5.4. Logging

We define three pointcut in the `Logging` aspect. The joinpoints of these pointcuts are the `sendMessage()` method of the `NetworkToBank` class, `dispenseCash()` method of the `CashDispenser` class, and the `acceptEnvelope()` method of the `EnvelopeAcceptor` class. There are two advices of the last two joinpoints which calls the relevant methods of the `Log` class in order write entries to the log.

The joinpoint of the first pointcut (Figure 12) send message to the `NetworkToBank`. We defined a `before()` advice in order to keep the log of sending the message to the bank. The `sendMessage()` method also gets a response from the bank which should also be logged. We handle this by creating an `after()` advice which get the result coming from the bank as a parameter.

```
pointcut message( Message message,
                  Balances balances):
call(* NetworkToBank.sendMessage(..)
&& args(message, balances);

before(Message message, Balances balances):
message(message, balances)
{ log.logSend(message); }

after(Message message, Balances balances)
returning(Status result):
message(message, balances)
{ log.logResponse(result); }
```

Figure 12: Logging Aspect

6. Alternative Aspect Oriented Implementations

In order to implement aspects in our project, we have used `AspectJ` [7] which is a seamless `Aspect-Oriented` extension to the `Java` programming language. However, we could also use another tools such as `JBoss` [8] whose implementation is a bit different.

In `JBoss`, the pointcuts are defined in the `XML` files. The related advice of the defined pointcut should be added to the `Java` class which must implement the `Interceptor` interface of the `JBoss` API.

In order to give an example about the usage of aspects in JBoss, we implemented the transaction pointcut of the Transaction Handling aspect. The definition of the pointcut and the associated advice can be seen in the Figure 13. The detailed comparison between AspectJ and JBoss is made in the discussion section of the paper.

```

Bind pointcut="* Transaction->
performTransaction(..)">
<interceptor class="TransactionHandling"/>
</bind>

Public class TransactionHandling
implements Interceptor
{
    public String getName() {
        return TransactionHandling;
    }
    public Object invoke(Invocation invocation)
    throws Throwable
    {
        Transaction transaction =
        (Transaction)invocation.getTargetObject();
        Object rsp = invocation.invokeNext();
        if (transaction.completed)
        {
            Receipt receipt = null;
            try
            {
                receipt =
                transaction.completeTransaction();
            }
            catch (CustomerConsole.Cancelled e) { }
            transaction.getATM().getReceiptPrinter()
            .printReceipt(receipt);
        }
        Return rsp;
    }
}

```

Figure 13: JBoss Alternative of the Transaction Handling Aspect

7. Discussion

Aspect-Oriented Programming helps designers better modularize the system and eliminate cross-cutting of concerns. With better modularization, software change scenarios are easier, the existing code is more reusable and flexible. On the other hand refactoring a system previously designed with Object Oriented techniques is difficult. But it does not mean that using Object Oriented approach together with the Aspect Oriented approach is bad. If you design a system from scratch by using OO and AO techniques together, then you will have a better design than the previous one.

One of the reasons why a system which is designed by OO approach is difficult to convert into the AO design is that designers did not need strict naming conventions in OO and they were partially right. So in AO design, one should know about the detailed uses of every aspect in the system to be able to not to

conflict with pointcut definitions and to conform with aspect conventions. This obligation requires strict and clear conventions for coding.

In this work, we used AspectJ. There are alternative frameworks for AOSD such as JBoss and others. We used AspectJ plugin of Eclipse [9] development environment, which is very easy to use when compared to JBoss.

The cross-cut references and the advice markers are very useful in checking the code segments captured by advices. The AspectJ extension in Eclipse increased the testing and debugging opportunity for aspect oriented programming. Since AspectJ treats the pointcuts and advices as first level abstractions, we got any possible syntax error at the time of compilation. This functionality surely shortens the implementation phase of the software development cycle especially for the big projects. On the other hand, for JBoss AOP, you have to execute the code in order to get any possible syntax error related with your point-cuts, since you write them either in a separate XML file or as string parameters within Java code.

8. Conclusion

In this paper, we have developed an ATM Simulation System and studied the application of aspect oriented software design. We first gave the object-oriented design and then worked on identifying, specifying and implementing aspects to handle cross-cutting concerns and enhanced the design quality of ATM System.

As a result, we have seen that aspect-oriented software design paradigm addressed the problem of scattering and tangling in the object oriented design of applications by expressing the cross-cutting concerns. Despite the advantages, in the development stage, we have seen that Aspect Oriented Programming requires awareness of the aspect joinpoints, otherwise pointcuts might easily result in difficult to find bugs in the program.

In conclusion, many core requirements of an application could be addressed by using AOSD to enhance modularization by separating cross-cutting concerns and to

provide a design that supports software quality factors.

9. Acknowledgements

We would like to thank to Ass. Prof. Bedir Tekinerdoğan, for teaching us the steps of the Aspect-oriented Software development in the CS 586 course and organizing the 4th Turkish Aspect-Oriented Software Development Workshop.

References

- [1] E. Gamma, R. Helm, R. Johnson, J. Vlissides. Design Patterns: Abstractions and Reuse of Object-Oriented Design, European Conference on Object-Oriented Programming, Conference Proceedings, Springer-Verlag, Lecture Notes in Computer Science, 1993.
- [2] R. C. Bjork, Professor of Computer Science, Gordon College, <http://www.math-cs.gordon.edu/courses/cs211/ATMExample/>
- [3] W. Hursch and C. Lopes. Separation of Concerns, technical report, College of Computer Science, Northeastern University, 1995.
- [4] http://en.wikipedia.org/wiki/Singleton_pattern
- [5] http://en.wikipedia.org/wiki/State_pattern
- [6] http://en.wikipedia.org/wiki/Factory_method_pattern
- [7] <http://www.eclipse.org/aspectj/>
- [8] <http://www.jboss.org/jbossaop/>
- [9] <http://www.eclipse.org/>
- [10] G. Kiczales, J. Lamping, A. Mendhekar, C. Lopes, J. Loingtier, J. Irwin. Aspect-Oriented Programming, European Conference on Object-Oriented Programming (ECOOP), Springer-Verlag, LNCS 1241, June 1997.

Aspect-Oriented Refactoring of a Remote Password Management System

Abdullah Atmaca, Muhammet Kabukçu, Cem Mergenci
Bilkent University Department of Computer Engineering
06800, Ankara, Turkey
{aatmaca,muhammet,mergenci}@cs.bilkent.edu.tr

Abstract

Remote Keychain (RKC) is a password management utility that helps users to easily access their account information over a network; therefore, alleviating the problem of forgotten passwords. Such a system has many components responsible for managing the communication, authentication, security, persistence etc. Although applying Object-Oriented Design Patterns brings a more elegant design, it does not solve the problem of crosscutting concerns. Aspect-Oriented Software Development (AOSD), on the other hand, addresses crosscutting concerns by capturing them as first class entities. In this paper, we apply OOD Patterns to an existing design, use AOSD concepts to separate concerns, implement new features and enforce design decisions.

1. Introduction

Password management utilities built into operating systems[1] or other softwares (browser[7, 4], mail client[2]) takes an important role in account management for users. Users are not overwhelmed with different requirements for usernames and passwords of each website, rather by remembering a single master password, they are able to use all of their accounts. Furthermore, one can choose different strong passwords for different websites to enhance security. However, the service comes in its own cost: security of all passwords depends on the master password. User is expected to choose this master password wisely, so that it is not predictable.

However, these password management utilities are local to the computer they are installed. Users who needs to access their Web accounts from different places are supposed to remember each of those passwords. Remembering a password of an account, maybe even username in some cases, is especially difficult after getting used to the password management software.

RKC aims to solve this problem. This password management system runs over the network; therefore, users can easily access their stored account information remotely and securely.

RKC is a relatively complex application incorporating different problems such as: network communication and encoding, protocol implementation, cryptography operations and persistence. Managing network communication requires using network input and output streams at the necessary places. Encoding ensures both parties in communication can understand messages of each other, even if they run on different software/hardware platforms. A protocol is crucial to a network application defining the order and type of messages exchanged. Cryptography operations are also performed between these tasks, since the information being sent is confidential. Furthermore, server application keeps record of its users and their accounts.

The concerns described above are not easy to capture using a conventional Object-Oriented Design. Although, the application of design patterns leads to a better separation, resulting code is still scattered and tangled in terms of presented concerns. This problem is due to the fact that shortcoming of Object-Oriented Design in capturing crosscutting concerns. [13]

Aspect-Oriented Programming (AOP) paradigm emerged as a result of this problem. It captures crosscutting concerns in first class entities “aspects”; therefore, provides a high cohesion and low coupling design.

In this work, we are going to study the development process of RKC. Firstly, we are going to present an object-oriented design including design patterns for the system. Then, we are going to identify crosscutting concerns and their effects on maintenance and further development. Lastly, we apply AOSD techniques to address the problem of crosscutting concerns and to produce a better design.

The rest of the paper is organized as follows. Section 2 presents functional and non-functional requirements analysis of the system. Section 3 describes the object-oriented design and applied design patterns. Section 4 identifies the

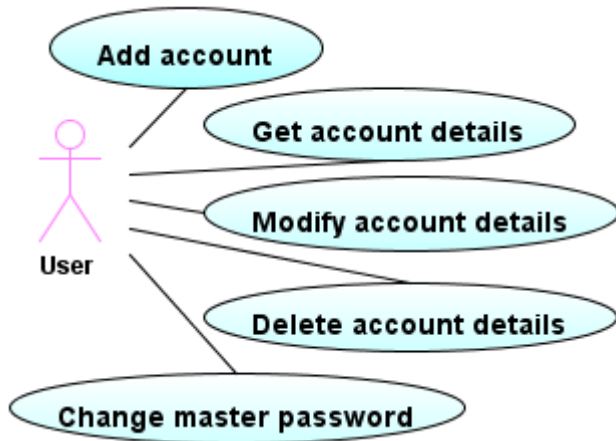


Figure 1. Use cases for a user.

problems with object-oriented design and introduces cross-cutting concerns. Section 5 defines aspects as a solution to crosscutting concerns problem. Lastly, Section 6 provides a discussion of the whole work and concludes the paper.

2. Requirements Analysis

RKC consists of two main parts: client and server. Although, they are generally used on different systems, they share mostly the same application logic; therefore, requirements and design of these two entities are considered at the same time.

This section presents functional and non-functional requirements of a remote password management utility. Figure 1 shows use cases for a RKC user.

2.1. Functional Requirements

The RKC system has following features:

- RKC Server supports multiple concurrent users.
- New users can use the service through RKC Client by registering to RKC Server with their usernames and passwords.
- Users are authenticated with their master username and master password.
- Users can add, delete, modify or retrieve username, password and personal notes for different domains in the RKC Server.
- Users can change their master password any time by providing their old password.

- Users cannot recover their master password by any means; for the sake of simplicity and to increase security.

2.2. Non-Functional Requirements

Non-functional requirements are related to security. The system has following cryptographic properties to ensure security:

- Before authentication, client and server use public key cryptography to establish a secure connection. After key agreement[5] and authentication, symmetric key cryptography is used for its speed and convenience.
- Clients have public key of servers they want to connect to. Anybody can operate a RKC Server and clients can connect to servers they want. Server operators should announce their public keys in order for clients to connect them.
- Authentication process is a challenge-response protocol [14]. Client initiates authorization by sending username. Server responds with a random challenge r . Client then sends $\{n, r, W1\}_s$ where n is random padding, $W1 = H(p)$ is password in hashed form and s denotes public key encryption. As a last step, Server compares $W1$ to database entry to authenticate the user.
- After authorization session key for symmetric key encryption is computed to be $K = H(n, r, W1)$.
- On the server side, user account data are stored as $K(W2(data))$ where K is server master password and $W2 = H(1 || p)$. If database is compromised, attacker is not able to recover all data at once.
- RSA[6] algorithm with PKCS1 random padding is used for public key cryptography.
- AES[11] with CTR[16] mode of operation is used for symmetric key cryptography.
- Message integrity and authenticity is ensured by HMAC-MD5[10] algorithm with a different key than symmetric key cryptography to increase security.
- Replay attacks are prevented using nonces (number used once)[9].

3. Object-Oriented Design

The object-oriented design of core RKC functionality is shown in figure 2. Server and client applications are omitted for the sake of simplicity.

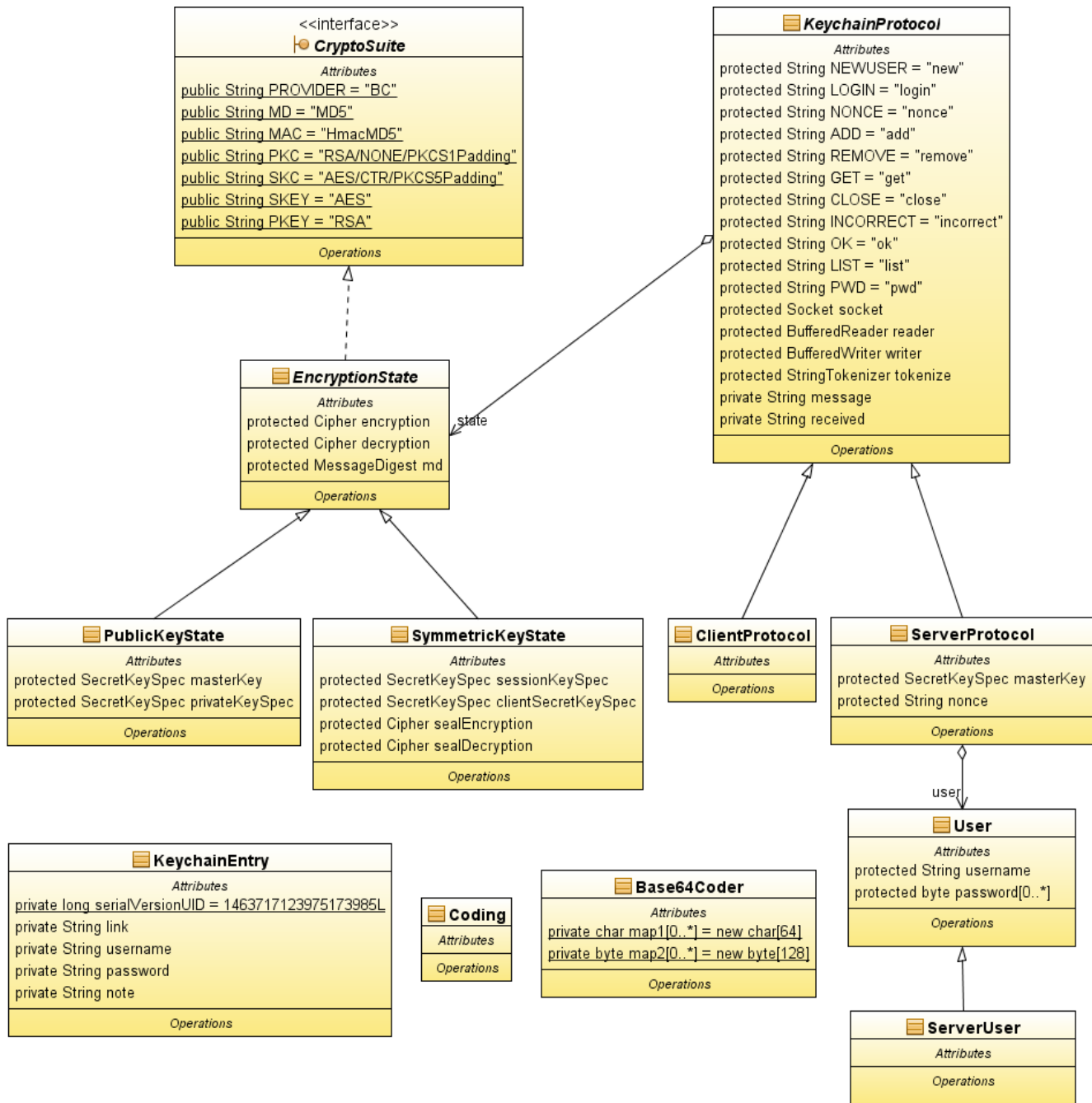


Figure 2. UML Class Diagram showing Object-Oriented Design of RKC

The components are briefly explained below:

KeychainProtocol: Abstract superclass defining the interface subclasses should implement providing basic mechanisms involving message sending and receiving. Communication protocol specific operations are managed under this entity.

ServerProtocol: Concrete subclass of KeychainProtocol responsible from implementing server-side functional re-

quirements.

ClientProtocol: Concrete subclass of KeychainProtocol responsible from implementing client-side functional requirements.

EncryptionState: Abstract superclass for common cryptography related functions such as encryption, decryption, message digest. Encoding/decoding of encrypted/decrypted messages are also handled in this class.

PublicKeyState: Concrete subclass of EncryptionState. It implements public key cryptography functions for client and server that are used during the authentication phase.

SymmetricKeyState: Concrete subclass of EncryptionState. It implements symmetric key cryptography functions for client and server that are used after the initial authentication phase.

CryptoSuite: Interface containing algorithm and algorithm provider information. By using this interface other packages can write compliant code with RKC.

User and ServerUser: User class defines general operations required to manage a user in the client application. ServerUser extends its functionality to provide Keychain management and persistence for client users.

Base64Coder and Coding: Base64Coder provides static methods used to encode and decode data in Base64 format as described in RFC 1521. Coding defines convenience methods for accessing Base64Coder.

KeychainEntry: Data structure for a single Keychain element. It consists of a domain, username, password and personal notes for that domain.

3.1. Design Patterns

The initial design of RKC was poor, combining all of the encryption and communication work mainly inside KeychainProtocol class; therefore, resulting in a tangled code that is hard to maintain. ServerProtocol and ClientProtocol classes were responsible for very little action on communication, but had many cryptography tasks. Furthermore, they did not share a common protocol interface, which may introduce communication problems on both sides due to loose definition.

We applied following patterns to achieve a better design.

Strategy Pattern

We applied strategy pattern [12] to protocol related classes: KeychainProtocol, ServerProtocol and ClientProtocol. Figure 3 shows the design as a UML class diagram. KeychainProtocol class defines a common protocol interface for its subclasses to implement. By this way, each side can communicate easily by adhering to the interface while using their own implementation.

The application should choose a communication strategy depending on its role, client or server. Rest of the application is not affected from the selection of a strategy.

State Pattern

Cryptography tasks are inherently a different concern than the communication itself. Therefore, they should be defined in a different class and used over its interface. EncryptionState class serves this purpose.

When the application flow is examined, two key states can be identified. First, the public key cryptography phase

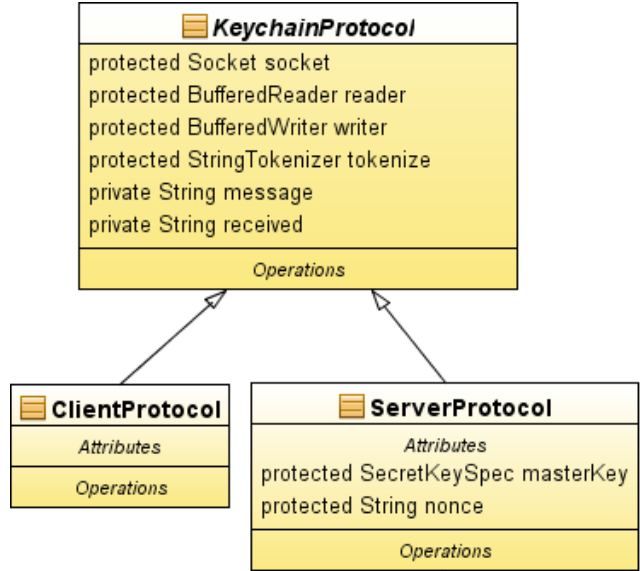


Figure 3. Strategy pattern in UML class diagram.

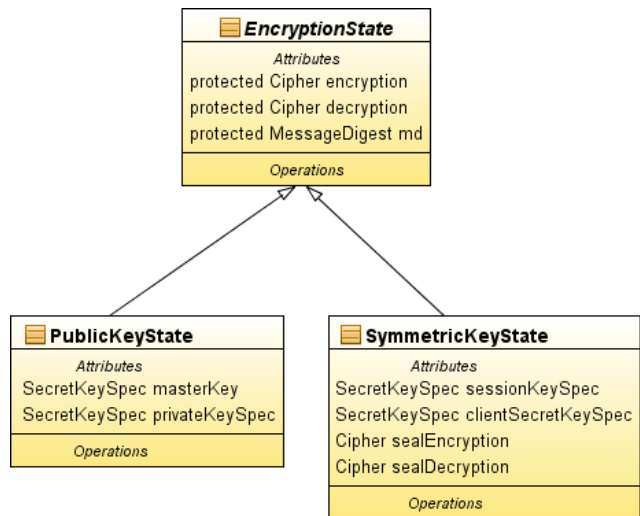


Figure 4. State pattern in UML class diagram.

used during the authentication process. Second, the symmetric key cryptography phase used for the rest of the application after the authentication is completed. Capturing details of these two states in PublicKeyState and SymmetricKeyState classes eliminates state checking if conditions and results in a more organized code. The design is presented in Figure 4.

Although state pattern separates states, it does not define how transition should occur between states. [12] Since state transition is affected by the result of the communication, protocol classes determine which state to use.

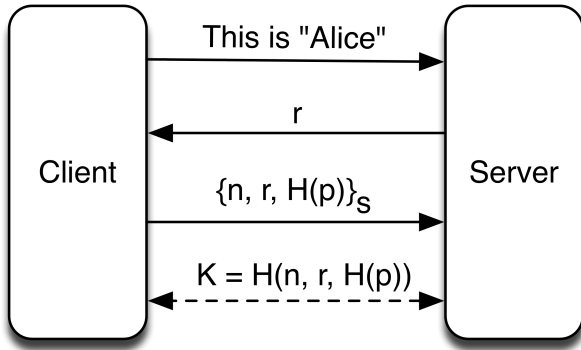


Figure 6. Authentication scheme.

4. Crosscutting Concerns

In the previous section, we have presented improvements over the existing design of RKC. We achieved a better design by applying strategy and state patterns in relevant places, by separating communication and cryptography concerns and by trying to write a less scattered and tangled code as much as possible.

We now sufficiently enhanced the modularity of the design and grounded it on well established object-oriented design principles and patterns. However, we can still identify some problems in this design because of the shortcomings of object-oriented design paradigm. These problems are crosscutting and cannot be captured inside first class elements, objects.

By applying AOSD principles we are aiming to capture following crosscutting concerns in aspects; therefore, solving the problem of crosscutting: User Authentication, Decryption and Message Authentication. Moreover, we are going to define two enforcement aspects to ensure proper usage of functions: Crypto Responsibility and Network Stream Access.

Figure 5 is a visualization of three identified aspects. They span a total of six classes, Decryption aspect being the most scattered one appearing in four classes.

4.1. Authentication

Authentication, not to be confused with message authentication, is the process of confirming each sides of the communication are actual client and server. In RKC, this is achieved by well established mechanisms using public key cryptography. Authentication should be performed on both sides of a communication; therefore, it is implemented in ClientProtocol and ServerProtocol classes separately, which means that authentication is a crosscutting concern.

Current authentication scheme is depicted in Figure 6.

Refer to Section 2.2 for a detailed and technical explanation. A modification may be required for authentication phase. A simpler and faster approach that does not use public keys can be preferred in future. Or a completely different method can be devised to meet custom needs. Impact of such a change is to modify corresponding methods in two classes. The modification requires careful attention, since both parts should comply with each other.

Capturing authentication phase in an aspect resolves this problem by bringing all authentication related code together.

4.2. Decryption

Decryption related tasks are handled under Encryption-State and its subclasses. However, initialization of a decryption module requires outside information called initialization vector (IV), which is supplied from protocol classes. Furthermore, every decryption operation should be preceded by a decoding operation of the message.

When implementing a new feature involving decryption, above steps have to be performed correctly. The programmer is overwhelmed by remembering and applying the correct procedure in decryption initialization and decoding.

We are aiming to organize decryption tasks by defining an aspect; so that, programmer can write related tasks in a single place.

4.3. Message Authentication

Message authentication is an optional process that guarantees the received message is coming from supposed sender and not spoofed in symmetric key cryptography. A message authentication code (MAC) is sent along the message and checked at the receiving side whether it gives the expected result. In the original RKC, message authentication was not implemented. Assuming that we may not be able to directly use or extend the functionality of existing code; we want to implement message authentication mechanism using an aspect intercepting message send and receive methods.

Since aspects are loosely bound to the code, we can turn off message authentication any time, possibly due to performance issues in a restricted platform or because it is not necessary in a trusted environment.

5. Aspect-Oriented Programming

This section will provide implementation details regarding aspects defined in previous section. AspectJ is used as AOSD language. [15] We omit method implementations as they are not the primary concern.

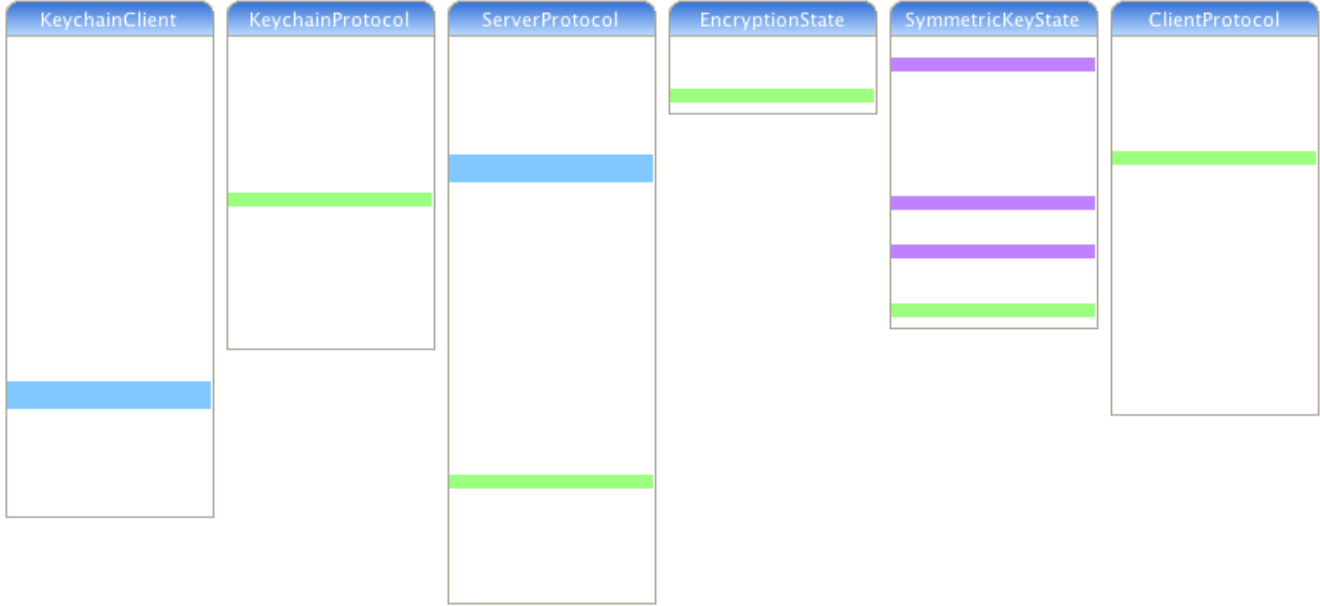


Figure 5. Visualization of aspects in RKC classes. Blue: Authentication aspect, Green: Decryption aspect, Purple: Message Authentication aspect

AspectJ Development Tools (AJDT) [3] is an Eclipse plugin, integrating AspectJ to Eclipse framework. It is easy to use and has many helpful development aids. We used AJDT to implement required aspects for RKC.

5.1. Authentication Aspect

Authentication aspect implements methods that will be used during key agreement mechanism. Authentication takes place as a result of a client sending either a login or new user request to the server. Therefore, pointcuts shown in Listing 1 are defined to capture necessary points.

The pointcut defined in lines 2 - 6 catches client side login and new user functions with application context, where lines 8 - 12 do the same for server side.

Listing 1. Authentication Aspect Pointcuts

```

1 public aspect Authentication {
2     pointcut clientSide( ClientProtocol p,
3         String username, String password ) :
4         (call( * ClientProtocol.login( String, String ) ) ||
5          call( * ClientProtocol.newUser( String, String ) ))
6         && args( username, password ) && target( p );
7
8     pointcut serverSide( ServerProtocol p,
9         String username, String password ) :
10        (call( * ServerProtocol.login( String, String ) ) ||
11         call( * ServerProtocol.newUser( String, String ) ))
12        && args( username, password ) && target( p );
13 }

```

Corresponding following advise structures, presented in

Listing 2, execute necessary application logic and cryptography algorithms to complete authentication.

Listing 2. Authentication Aspect Advises

```

1 public aspect Authentication {
2     User around( ClientProtocol p, String username,
3         String password ) : clientSide(p, username, password)
4     { // client side authentication logic }
5
6     User around( ServerProtocol p, String username,
7         String password ) : serverSide(p, username, password)
8     { // server side authentication logic }
9 }

```

Authentication aspect unifies the authentication work performed at different sections of two classes into a single aspect. In case a modification is required in authentication process, it can be performed easily by modifying this single entity. Advise defined in lines 2 - 4 executes authentication logic for client side according to given context. Lines 6 - 8 provide similar advise for server side authentication.

5.2. Decryption Aspect

Decryption aspect eases the initialization and IV specification stages of decryption. Rather than declaring pointcuts and advises, this aspect simply introduces a new constructor to IvParameterSpec class and a new decryption method to Cipher class both accepting a single String parameter and doing required decoding process before actual decryption takes place. Listing 3 shows the inter-type declarations.

Listing 3. Decryption Aspect Introductions

```
1 public aspect Decryption
2 {
3     public IvParameterSpec.new( String s )
4     { // decode string and return IV spec }
5
6     public byte [] Cipher.doFinal( String s )
7     { // decode string and decrypt }
8 }
```

Normally, Java does not allow introducing new members, properties or methods, to classes. By using aspects we can benefit from the concept of open classes in Java as in other OO languages like Objective-C, Python, Smalltalk. [8]

While implementing the Decryption aspect, we encountered problems with AJDT. Building process sometimes became problematic and a clean build was required to have a stable working product. Inter-type declarations were also not working properly, as far as we experimented, but we managed to find some workarounds.

5.3. Message Authentication Aspect

Aspects help us in situations where modification or extension of a code is not possible. Assuming such a case in RKC, we want to implement message authentication feature in an aspect. Listing 4 shows outline of the implementation.

Listing 4. Message Authentication Aspect

```
1 public aspect MessageAuthentication
2 {
3     private Mac mac;
4
5     /* Pointcuts */
6
7     pointcut appendMac( String message ) :
8     execution(* SymmetricKeyState+.prepareMessage(..)
9     && args( message ));
10
11    pointcut checkMac( BufferedReader reader ) :
12    execution(* SymmetricKeyState+.receive(..)
13    && args( reader ));
14
15    pointcut initMac( byte [] sessionKey ) :
16    execution(SymmetricKeyState.new(..)
17    && args( sessionKey ));
18
19    /* Advises */
20
21    String around( String message ) : appendMac( message )
22    { // append mac to prepared message }
23
24    String around( BufferedReader reader ) :
25    checkMac( reader )
26    { // check integrity of received message }
27
28    after( byte [] sessionKey ) : initMac( sessionKey )
29    { initMac( sessionKey ); }
30
31    /* Helper methods*/
32
33    private void initMac( byte [] sessionKey )
34    { // initialize mac }
35
36    private String getEncodedMac( String s )
37    { // compute mac and encode }
38
39    private byte [] deriveMacKey( byte [] sessionKey )
```

```
40 { // derive mac key }
41 }
```

First we define a private property `mac` to perform cryptographic calculations. Message authentication is a two way process: a MAC should be calculated and appended to outgoing messages, and MACs of received messages should be checked to understand if they are authentic. One extra step is required to initialize the algorithm. Since MACs are only used in symmetric key cryptography we declare pointcuts for corresponding `SymmetricKeyState` methods and constructor for initialization part. Required computations are performed in `advise` parts and MAC added and MAC checked results are returned. Helper methods are private to the aspect and are used to ease some of the work.

5.4. Enforcement Aspects

So far we have discussed production aspects that are used in implementation of new features for an application. On the other hand, enforcement aspects are useful during the development process. One of the problems in software engineering is to deal with constantly changing requirements in a project. Due to time restrictions design decisions may be compromised unintentionally to meet the deadline. Enforcement aspects help developer to remember design decisions and modify the design accordingly. Important design decisions that should not be compromised are ensured to remain with enforcement aspects.

We discuss two enforcement aspects in RKC, crypto responsibility and stream access, in the following parts.

Crypto Responsibility Enforcement Aspect

By design, RKC classes are supposed to use `EncryptionState` and its subclasses for cryptography tasks. They should not perform any custom encryption/decryption, because it would result in a scattered code. Listing 5 defines `CryptoResponsibility` enforcement aspect.

Lines 3 - 5 specify a pointcut for initializing a `Cipher` instance that is used in every cryptographic operation. Lines 7 - 9 describe another pointcut for initializing a `MessageDigest` instance that is used in challenge-response and integrity check operations. Both pointcuts also determine authorized scope of these operations. A compile time error is defined in lines 11 - 14 forcing programmer to adhere design specifications and alerting in case of a violation.

Listing 5. Crypto Responsibility Enforcement Aspect

```
1 public aspect CryptoResponsibility
2 {
3     pointcut cipherInstance() :
4     call(* Cipher.getInstance(..)) &&
5     within( rkc.* ) && !within( EncryptionState+ );
6
7     pointcut messageDigestInstance() :
8     call(* MessageDigest.getInstance(..)) &&
9     within( rkc.* ) && !within( EncryptionState+ );
```

```

10 |
11 | declare error :
12 |     cipherInstance() || messageDigestInstance() :
13 |         "Crypto related tasks should be performed in a
14 |         subclass of EncryptionState.";
15 | }

```

Stream Access Enforcement Aspect

A similar restriction apply on network communication. Listing 6 defines StreamAccess enforcement aspect for this purpose. Only KeychainProtocol and some parts of its subclasses can use network connections. Reading from a network connection is allowed only on client login method, defined in lines 3 - 7. Writing to a network connection is allowed only on server login and newUser methods, specified in lines 9 - 14. These methods are privileged, because they need direct access to connection streams in order to send unencrypted initial requests. In order to prevent misuse of streams two compile time errors, one for reader stream and one for writer stream, are defined in lines 16 - 20.

Listing 6. Stream Access Enforcement Aspect

```

1 | public aspect StreamAccessEnforcement
2 | {
3 |     pointcut ReaderStreamAccessEnforcement() :
4 |         get(BufferedReader KeychainProtocol.reader) &&
5 |         !( within( KeychainProtocol ) ||
6 |           within( Authentication ) ||
7 |           withincode(* ClientProtocol+.login(..) )
8 |         );
9 |
10 |    pointcut WriterStreamAccessEnforcement() :
11 |        get(BufferedWriter KeychainProtocol.writer) &&
12 |        !( within( KeychainProtocol ) ||
13 |          within( Authentication ) ||
14 |          withincode(* ServerProtocol+.login(..) ) ||
15 |          withincode(* ServerProtocol+.newUser(..) )
16 |        );
17 |
18 |    declare error : ReaderStreamAccessEnforcement() :
19 |        "Reader stream should not be accessed from here.";
20 |
21 |    declare error : WriterStreamAccessEnforcement() :
22 |        "Writer stream should not be accessed from here.";
23 | }

```

6. Discussion and Conclusion

The application of design patterns to an existing design results in a more elegant solution and readily solves some of scattering and tangling problems. Well studied design patterns also ensures expected results from the code, preventing faulty designs.

State and Strategy patterns can be easily applied to network applications, due to very similar application logics, as in the form we used: strategy being the place defining state transition.

Design patterns solve crosscutting concerns to some extent, but fail to capture them all because of the limitation of object-oriented programming paradigm. Aspects capture crosscutting concerns as first class elements. However,

there is a trade-off in capturing crosscutting concerns in aspects. Having too many aspects brings many pointcuts and advises, which may in turn make maintenance more difficult than an object-oriented approach.

By using aspects, we can extend the functionality of a code that cannot be modified. This is particularly useful in Java. New features can be implemented without subclassing even if source code does not exist.

Enforcement aspects ensure that best practices are followed and important design decisions are not compromised during software development.

In this paper, we have made object-oriented and aspect-oriented improvements over an existing remote password management application, RKC. First, we applied two design patterns, strategy and state. Second, we identified crosscutting concerns in the final design, authentication, decryption and message authentication. Finally, we implemented aspects to address crosscutting concerns. We also specified two enforcement aspects, crypto responsibility and stream access, to ensure design decisions are met at each stage of development.

As a result, we observed that aspect-oriented software development paradigm solves the scattering and tangling problems, which cannot be fully solved by object-oriented approach; therefore, simplifying the software development process.

7. Acknowledgments

We would like to thank Asst. Prof. Dr. Bedir Tekin-erdoğan, for his efforts and for organizing the Turkish Aspect-Oriented Software Development (TAOSD) Workshop 2009.

References

- [1] Apple keychain access. <http://www.apple.com/macosx/what-is-macosx/apps-and-utilities.html>.
- [2] Apple mail. <http://www.apple.com/macosx/what-is-macosx/apps-and-utilities.html>.
- [3] Aspectj development tools (ajdt). <http://www.eclipse.org/ajdt>.
- [4] Mozilla firefox. <http://www.mozilla.com/firefox>.
- [5] New directions in cryptography. <http://www.cs.jhu.edu/~rubin/courses/sp03/papers/diffie.hellman.pdf>.
- [6] Rsa encryption algorithm. <http://www.rsa.com/>.
- [7] Safari. <http://www.apple.com/safari>.
- [8] What are inter-type declarations. <http://www.eclipse.org/aspectj/doc/released/faq.php>.
- [9] R. Anderson. *Security Engineering: A Guide to Building Dependable Distributed Systems*. Wiley, 2001.
- [10] M. Bellare, R. Canetti, and H. Krawczyk. Keying hash functions for message authentication. pages 1–15. Springer-Verlag, 1996.

- [11] J. Daemen and V. Rijmen. *The Design of Rijndael: AES - The Advanced Encryption Standard*. Springer-Verlag, 2002.
- [12] E. Gamma, R. Helm, R. Johnson, and J. M. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional, 1994.
- [13] W. L. Hursch and C. V. Lopes. Separation of concerns. Technical report, 1995.
- [14] C. Kaufman, R. Perlman, and M. Speciner. *Network Security: Private Communication in a PUBLIC World*. Prentice Hall PTR, 2002.
- [15] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold. An overview of aspectj. pages 327–353. Springer-Verlag, 2001.
- [16] H. Lipmaa, P. Rogaway, and D. Wagner. Comments to nist concerning aes modes of operations: Ctr-mode encryption, 2000.

Aspect Oriented Development of a Client-Server Based Game Application

Selçuk Onur Sümer, Alper Karaçelik

Department of Computer Engineering, Bilkent University
Ankara, Turkey 06800

ssumer@bilkent.edu.tr, alperk@bilkent.edu.tr

Abstract – Multiplayer gaming is usually based on networking architecture such as the client-server architecture. For developing a multiplayer gaming application on the client-server architecture, different concerns need to be implemented such as request synchronizing and rendering. Some of these concerns cannot be easily modularized in a single object or module. So they crosscut multiple modules in the system and this case reduces the maintainability. In this paper, we propose an aspect-oriented programming approach for separating these crosscutting concerns in multiplayer gaming applications.

Index Terms – Multiplayer game, aspect oriented programming, client-server, crosscutting concern

1. Introduction

One of the efficient ways to develop and implement a multiplayer game is the client-server approach. Client-server networking is a distributed application architecture that partitions tasks or workloads between service providers (servers) and service requesters (clients). [1] Mostly, servers and clients are placed on different machines. Generally server side shares its sources, and provides some functions to the clients. Clients do not have to share their sources or states. They only use the services (functions) that server provides.

Multiplayer game concept provides all players (users) to share the same gaming platform. Since, AI-controlled opponents often lack the flexibility and the ingenuity of regular human thinking [2], playing against human players offers different experiences. We can classify multiplayer games in several ways. Here, we prefer to classify them as turn-based multiplayer games and real-time multiplayer games.

In this paper, we explain how to design and implement a client-server based multiplayer game application using the object-oriented and the aspect-oriented programming. The remainder of the paper is organized as follows. In section 2, we introduce the object oriented design structure of our application. The use-case diagram and the class diagram of the project are given in the section too. Finally, the implemented design patterns are explained. In section 3, problems and crosscutting concerns of the application are discussed. Furthermore, applied aspect oriented programming concepts are introduced in section 4. In section 5, we briefly make a conclusion, and mention about future works.

2. Object Oriented Design

In this section, we examine the actors of our system and behaviors of them with a use case diagram. Additionally, class structures of some important system components are introduced.

2.1. Model of Application

Most fundamental behavior of the server is waiting for and accepting connections. A client actor assumes that a server application is running and waiting for connections. So, the first behavior of the client is connecting to the server. After connecting, clients have to login to the system in order to use the service. Clients can choose a game to play but, they have to join an existing room or create a new room for selected game. After entering a game room, they can play the game. Fig. 1 shows the use-case diagram of our system.

The diagram illustrates the use cases and relationships for a game system. It features a central system boundary containing several use cases, with external actors (Client and Server) interacting with them. The use cases are represented by yellow ovals with black text. The relationships are shown with solid lines for associations and dashed lines with labels for include relationships.

Actors:

- Client**: Represented by a stick figure on the left.
- Server**: Represented by a stick figure on the right.
- System Element**: Represented by a stick figure at the top, with dashed arrows labeled <<Extends>> pointing to the Client and Server actors.

Use Cases:

- Produce Request**: Associated with the System Element actor.
- Consume Request**: Associated with the System Element actor.
- Play Game**: Associated with the Client actor.
- Wait Connections**: Associated with the Server actor.
- Accept Connections**: Associated with the Server actor.
- Enter Room**: Associated with the Client actor.
- Create Room**: Associated with the Client actor.
- Choose Game**: Associated with the Client actor.
- Login to System**: Associated with the Client actor.
- Connect to Server**: Associated with the Server actor.

Relationships:

- Include Relationships** (dashed arrows with <<Include>> label):
 - Play Game includes Enter Room and Create Room.
 - Enter Room includes Choose Game.
 - Create Room includes Choose Game.
 - Choose Game includes Login to System.
 - Login to System includes Connect to Server.
- Association Relationships** (solid lines):
 - Client to Play Game
 - Client to Enter Room
 - Client to Create Room
 - Client to Choose Game
 - Client to Login to System
 - Server to Wait Connections
 - Server to Accept Connections
 - Server to Connect to Server

that are joined to itself. Moreover, our server model consists of users. A user can be a normal user or a premium one. (In our project, premium user is not added to implementation) Readers can see the class diagram of the server model in the Fig. 2.

2.2. Design Patterns

22

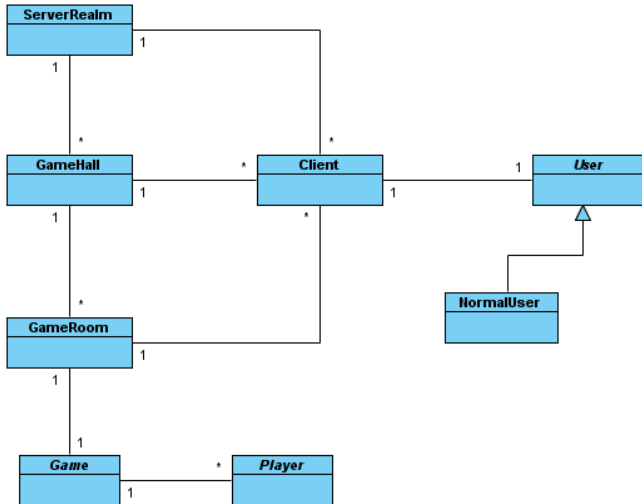


Fig. 2 Class diagram of the server model

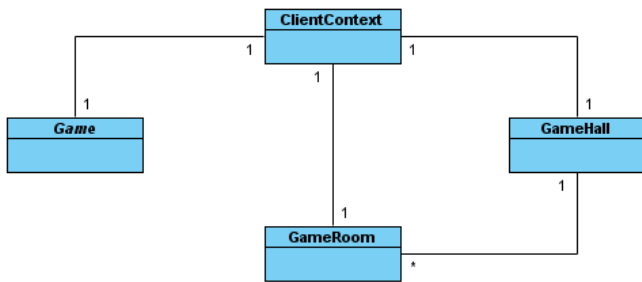


Fig. 3 Class diagram of the client model

In our project, both the client and the server has a model (the data that application operates), a view (user interface) and a controller (which process input and makes model and view communicate with each other) module. So, we applied model-view-controller (MVC) pattern in our project. Furthermore, we can think a game package as the family of the game. This package contains rules of the game, the game platform and the user interface. These three components construct a game family, and if the family of a product is the subject, abstract factory pattern is a proper solution.

2.2.1. Model-View-Controller

MVC pattern isolates business logic from input and presentation. It is often used by applications that need the ability to maintain multiple views of the same data. MVC pattern consists of three components. Model component represents the data which the application operates. View component is used for displaying all or a portion of the data. Controller component is used for handling events that affect the model or the view (or views). Controller is acted like a mid-layer between model and views. [3]

In our project, model represents the data structure. Model of the client side knows the game which it plays at the current time, the room which it has joined and the game hall which it was connected to. Model of the server side knows all available games, game halls and game rooms. In addition, it knows all connected clients. Every game hall belongs to a game, and holds available rooms. Every room is dedicated to a single game and holds the list of the players joined to that room.

For every client, different views have to be displayed. Viewing concern is handled by the controller component (mostly with rendering aspect (See Section 3.3.3.)). So, changes in the model component can affect every view in a different way. Controller component is also responsible for interpreting the commands produced from a network message or a view component and forwarding these commands to the proper action handlers. After applying the actions, both the model and the view component may have to be changed. In Fig. 4, readers can find package diagram of model-view-controller pattern applied in the client application.

2.2.2. Abstract Factory

“Abstract factory pattern provide an interface for creating families of related or dependent objects without specifying their concrete classes.” [4]

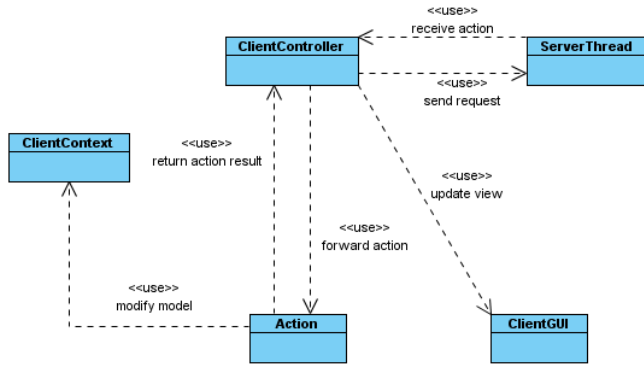


Fig. 4 Model-view-controller pattern applied in the client application

For our application every game can be thought as a family. Family members of a game are: the game class (*Chess*), the player class (*ChessPlayer*), and the view class (*ChessPanel*). Our concrete game class contains structural components (e.g. chess board) and the logic (e.g. validation of piece movements) of the game. Our concrete player class contains only individual attributes (e.g. color of player). A view class extends *JPanel* class. Game panels contain visual components of the game family it belongs to (e.g. visual representation of current state of chess board). Our view class is strongly related with rendering aspect (See Section 3.3.3.). You can find illustration of abstract factory patterns in the Fig. 5 [4]

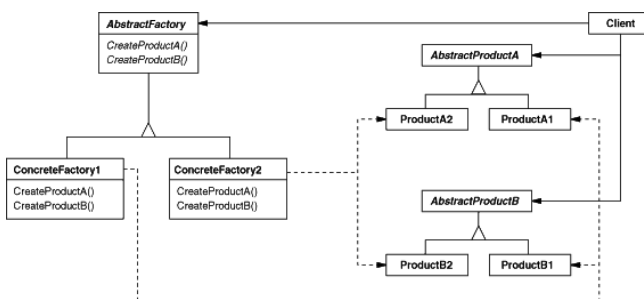


Fig. 5 Abstract factory

We can think our abstract *Game* class is replaced with *AbstractProductA* and abstract *Player* class is replaced with *AbstractProductB*. According to this approach, *ProductA1* is replaced with *Chess* and *ProductB1* is replaced with *ChessPlayer*. If we want to add a new game

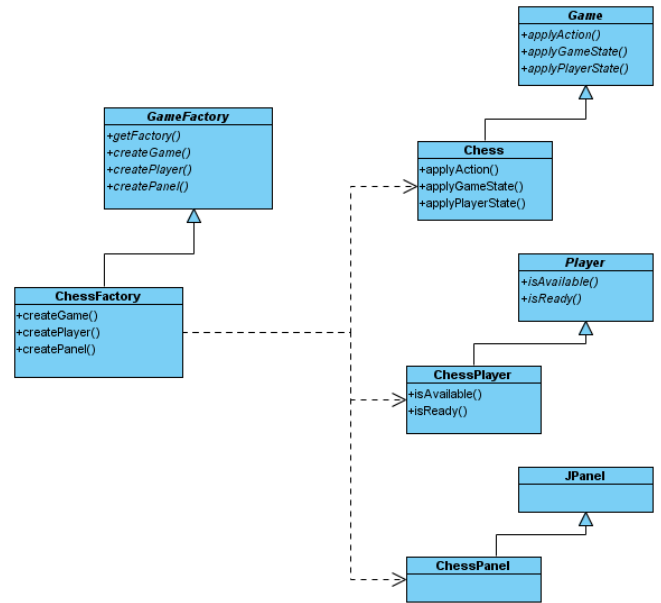


Fig. 6: Chess Family for the client side

to our application, *ProductA2* and *ProductB2* will be replaced with proper classes. Readers can find the illustration for our chess family in the Fig. 6.

3. Identifying Crosscutting Concerns

There are some design decisions that cannot be implemented clearly with Object-Oriented Programming (OOP) or procedural programming because of their crosscutting behavior. These crosscutting elements are called aspects. Aspect-Oriented Programming (AOP) provides techniques to modularize these aspects. [5]

In this section, first we describe the problems of our case study, and then specify concerns with crosscutting behavior.

3.1. Problems

A client-server based application simply consists of two main sides: The server and the client. Each side has its own specific problems in addition to the common problems for both server and client.

On the server side, synchronization of multiple client requests, distribution of the data to the clients, authorization of the client actions,

storage of the user related information are the main problems needed to be resolved.

On the client side, rendering of the state changes in the model is one of the most important problems.

Request buffering and synchronization is a common problem for both the server and the client. The server deals with synchronization of multiple client requests whereas the client deals with request coming from the server and the client GUI. Secure transfer of the data between the server and a client is also a common problem for both sides.

3.2. Crosscutting Concerns

Some of the problems defined in the section 3.1 may lead to tangling and scattering when they are solved with traditional approaches.

Distribution of the data to the clients, rendering of the data on the client side, and synchronization of requests are the important concerns with crosscutting behavior.

The crosscutting behavior of the distribution concern is shown in the Fig. 7. As you can see in the Fig. 7, the distribution concern is scattered among the different modules of server.

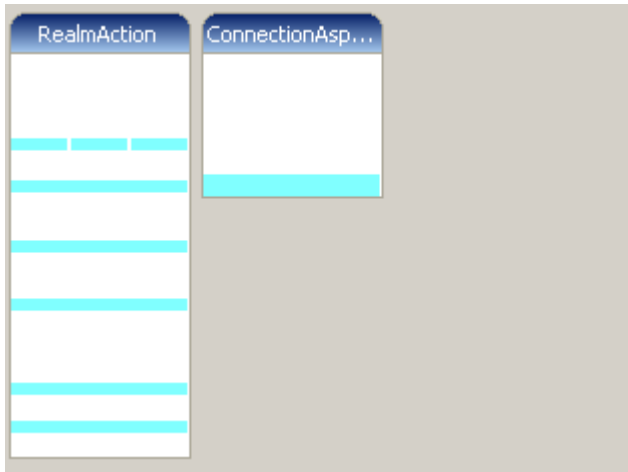


Fig. 7: Distribution concern

In the Fig. 8, crosscutting behavior of the rendering concern is illustrated. Similar to the distribution concern of the server side, rendering is scattered among the modules of the client.

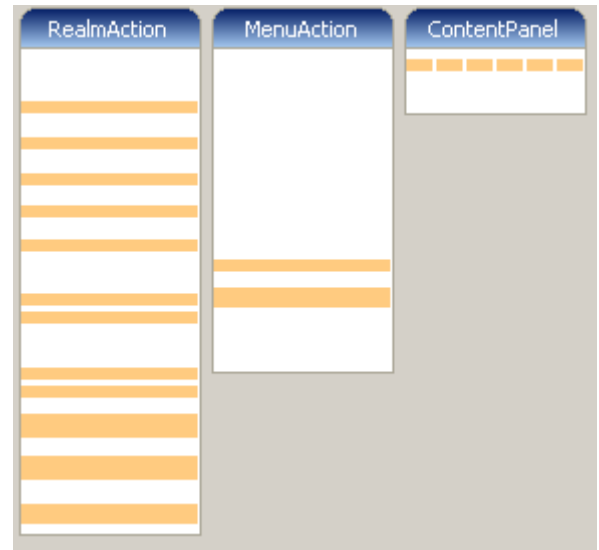


Fig. 8: Rendering concern

As you can see in the Fig. 9, request synchronization concern is scattered among the modules of both the client and the server.



Fig. 9: Request synchronization concern

In order to achieve the separation of crosscutting concerns, an aspect oriented approach should be applied to obtain modularization. In the following section, detailed solutions to these crosscutting concerns are explained in an aspect oriented manner.

4. Aspect-Oriented Programming

This section focuses on the details and implementation of the selected aspects which are proposed as a solution to the crosscutting concerns defined in the section 3.

4.1. Producer-Consumer

The producer-consumer problem is encountered when there are multiple data producers and a single consumer for these produced data. The main concern of this problem is synchronization of the shared buffer in which multiple producer threads put the produced data. The data in this buffer is then consumed by the consumer until the buffer becomes empty. [6]

In our case study, the server simultaneously accepts incoming requests from the clients. These simultaneous requests are handled with a producer-consumer approach in order to develop a consistent system. Although requests are accepted in parallel, they are processed sequentially in order to prevent inconsistency among the shared resources.

Consider the situation when two users are trying to enter the same game room which has only one more available slot. In this case processing both requests in parallel may lead to a game room having more users than its maximum capacity.

On the client side, although it seems to be easier to handle the requests, it is still required to synchronize incoming requests, since the client application accepts requests from both the server and the client GUI.

As it can be observed that the source of a request is not single and there may be more than one source which can produce a request or an action. Therefore, producing and handling of an action is a crosscutting concern and an aspect oriented approach is required to overcome this problem.

In our case study, methods that can produce a request are annotated with the *Producer* tag and a pointcut is defined for the methods having the *Producer* annotation. This pointcut is advised in order to add a new request to the synchronized list of commands. The code fragment (Fig. 10) shows the details of the pointcut and the advice.

```
pointcut production() :  
    call(@Producer * *(..));  
  
after () returning (Command command) :  
    production()  
{  
    synchronized (commandList)  
    {  
        commandList.add(command);  
        commandList.notifyAll();  
    }  
}
```

Fig. 10: The pointcut and the advice for the methods that produce requests.

Requests produced by this approach are consumed by a consumer thread one by one until the list becomes empty. After the list becomes empty, the thread waits until a new request added to the list.

4.2. Distribution

One of the main concerns of a distributed system is the distribution of the data among the clients. Clients should be notified by the server after the change of the server state.

In our case study, there are many actions which can change the state of an object on the server. Notifying every client about every change would be a simple but time and resource consuming approach. Instead, a more specific approach should be considered. For example, a change in a particular game hall should only be distributed to the clients in that particular game hall, not to all clients that are connected to the server. Similarly a change in the state of a game should only be forwarded to the clients in the game room where that game is being played.

In a game hall, entering of a user to the hall, exiting of a user from the hall, creation of a new game room, joining of a user to a game room, leaving a game room are the examples of the actions which are needed to be distributed to the clients. As it can be observed, distribution of the state changes in the game hall is a crosscutting concern which leads to scattering of the code related to distribution among the action methods related to game hall.

Similarly in a game room, the state of the game, number of users in the game room and content of the chat panel change rapidly and continuously. Distribution of these changes is also crosscutting and leads to scattering among the game related action methods.

In order to resolve the crosscutting behavior of the distribution concern, an aspect oriented approach is required. In our case study, several pointcuts are defined for the actions that are needed to be distributed and these pointcuts are advised according to the action type.

In the Fig. 11 a pointcut, which is defined for the entering of a new user to a game hall, is illustrated with its before and after advice blocks. These advices are executed in order to send required information to the client who has just entered the hall, and to the clients that are already in that game hall. Note that, the list of the clients are sent to the new entering client, before the execution of the add method in order to prevent duplicate appearance of his own name in the user list.

```
pointcut enteringHall(GameHall hall,
    Client client) :
    call(@Distributed * add*(..)) &&
    target(hall) && args(client);

before (GameHall hall, Client client) :
    enteringHall(hall, client)
{
    // send list of all clients
    controller.getDistributor().
        sendClientList(client,
            hall.getClientList());
}

after (GameHall hall, Client client) :
    enteringHall(hall, client)
{
    // send list of all rooms
    controller.getDistributor().
        sendRoomList(client,
            hall.getRoomList());
    // send new user to other clients
    controller.getDistributor().
        sendClient(hall.getClientList(),
            client);
}
```

Fig. 11: The pointcut and the advices for the entering of a new user to a game hall.

4.3. Rendering

Rendering in general can be defined as the image creation process from a model. [7] Although rendering is widely used for 3D models, we only consider 2D rendering for our case study.

Rendering is an important concern for the client side, since the clients should always see the most up-to-date graphical representation of the state of the client context.

When a client is notified by the server about the changes in the state of a game hall, a game room, or a game, this change is applied to the client model. However after each update of the model of the client, it is also required to update the view of the client. Therefore, behavior of the rendering concern is crosscutting and should be separated from the model concern.

In order to separate rendering concern, an aspect oriented approach is required. In our implementation, pointcuts are defined for the actions that update the model objects which require rendering. These pointcuts are advised in order to update the corresponding view object.

In the Fig. 12 a pointcut, which is defined for the chess piece movement in the chess game, is illustrated with its advice block. The advice updates the chess board view after the update of the model of the chess board.

```
pointcut movementMade(Chess chess,
    int fromX, int fromY, int toX, int toY):
    call(* Chess.makeMovement(..)) &&
    target(chess) &&
    args(fromX, fromY, toX, toY);

after(Chess chess,
    int fromX, int fromY,
    int toX, int toY) returning():
    movementMade(chess,
        fromX, fromY, toX, toY)
{
    ChessPanel panel = (ChessPanel)
        ClientController.getInstance().
            getGui().getGameRoomPanel().
            getGamePanel();
    panel.makeMovement(fromX, fromY,
        toX, toY);
}
```

Fig. 12: The pointcut and the advice for the chess piece movement rendering

5. Conclusion and Future Work

In this study our focus was on developing a reliable and maintainable client-server based game application with aspect oriented software development techniques.

We have presented the object oriented design of our client-server application first. Then, we identified the problems and some of the crosscutting concerns of the design. Finally, we proposed aspect oriented solutions to those crosscutting concerns.

On the other hand, there are many concerns left to be identified and implemented by using AOP. Authorization, security, persistency, and latency are some of these concerns needed to be implemented for a fully functional system.

The communication protocol between server and clients is based on unencrypted and fixed-sized ASCII messages. One of our future goals is to implement a communication protocol between the server and the clients which supports encrypted and variable-sized messages.

Currently, on the game server there is only one turn based game. However, our design is also suitable for real-time games. We would like to implement a real-time game and test its playability in the future.

6. Acknowledgments

We would like to thank to Gözde Kapısız for her chess icon drawing for the game selection panel, and to Cumhuri Kılıç for his contribution to the software testing process. We would also like to thank to Assist. Prof. Bedir Tekinerdoğan for his useful suggestions and feedback.

7. References

- [1] <http://java.sun.com/developer/Books/jdbc/ch07.pdf>
- [2] http://en.wikipedia.org/wiki/Multiplayer_video_game
- [3] <http://www.enode.com/x/markup/tutorial/mvc.html>
- [4] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Professional Computing Series. Addison Wesley, Reading, MA, USA, 1995.
- [5] Kiczales, G., et al.: *Aspect-Oriented Programming*. In: Proceedings of the European Conference on Object-Oriented Programming (ECOOP). Springer-Verlag, Finland (1997)
- [6] http://en.wikipedia.org/wiki/Producer-consumer_problem
- [7] http://en.wikipedia.org/wiki/Rendering_%28computer_graphics%29

Developing Aspect Oriented Software Components for C2 Systems

İskender Yakın, Bahar Pamuk

Department of Computer Engineering, Bilkent University,
Department of Computer Engineering Middle East Technical University
yakin@cs.bilkent.edu.tr, bahar@ceng.metu.edu.tr

Abstract

As operational expectations from Command & Control (C2) systems increase, new functionalities or abilities are expected to be integrated to the existing systems. These functionalities introduce new requirements most of which address new software components or modifications to the existing ones. In both cases integration or modification effort increases as the number of components addressed by a requirement increases and this fact implies weak separation of concerns. Such a problem affects the whole software development cycle from reclassification of requirements to testing of the whole system. In this paper we separate some common C2 concerns such as communication, verification, and simulation through Aspect Oriented Programming (AOP). To overcome the crosscutting among concerns we consider each concern given above as different aspect oriented software components. The approach is illustrated with an example case, TALOS Mission Planning Software developed in ASELSAN.

Key Words: Aspect-Oriented Programming, AspectJ, Mission Planning

1. Introduction

Command & Control (C2) is best defined by functions - including command, control, communications - that are integrated systems of procedures, organizational structures, personnel, equipment, facilities and technologies. Command and control is about decision making, the exercise of direction in the accomplishment of a mission, and is supported by information technology [7].

As operational expectations from Command & Control systems increase new functionalities or abilities are expected to be integrated to the existing systems. These functionalities introduce new requirements most of which address new software components or modifications to the existing ones. It is mostly impossible to get or predict all

the requirements of a software project in early stages of its life cycle. This requires the software to be designed with maximum cohesion and minimum coupling from the beginning by separating different features or functionalities.

As a general problem in software engineering, integration or modification effort increases as the number of components addressed by a requirement increases and this fact implies weak separation of concerns. Such a problem affects the whole software development cycle from reclassification of requirements to testing of correlated components and even the whole system. For instance communication infrastructure plays an important role in the design of C2 systems. A new requirement or a change in an existing requirement can change the structure of the communication infrastructure even it can require the whole communication infrastructure to be changed with another communication infrastructure. In this case all the components dependent on the communication infrastructure are affected from such a change and all of them are required to be recoded even if communicating entities in the C2 systems stays the same and still communicating with messages including the same information.

To overcome the problem mentioned above we employ Aspect Oriented approach in order to separate concerns and to eliminate the crosscutting among these concerns. For instance, as a solution to the given problem on communication concern the following Aspect Oriented approach can be applied. It is common for all C2 systems that an event or a command created by an operator triggers a message to be sent to another entity using the same system. Once all the events and commands which require communication among different entities are detected, the communication concern can be separated from the rest of the system in a separate software component since what this component does is just to detect event and commands and realize the communication requirements within itself. In a case that the communication infrastructure changes, for example if STANAG [10] is determined to be used instead of JAUS

[5] what is required to be done is to change the communication component without affecting any other components dependent on communication component.

The remainder of this paper is organized as follows. In section 2 background on C2 systems and concerns are given. In section 3 we present our example case, TALOS Mission Planning Software. In section 4 we describe object oriented design of this software. Problem statement including the identification of aspects is given in Section 5.

2. Background

Command and control, or C2, can be defined as the exercise of authority and direction by a properly designated commanding officer over assigned and attached forces in the accomplishment of the mission [11].

Command and control functions are performed through an arrangement of personnel, equipment, communications, facilities, and procedures employed by a commander in planning, directing, coordinating, and controlling forces and operations in the accomplishment of the mission. Commanding officers are assisted in executing these tasks by specialized staff officers and enlisted personnel. Military staff is a group of officers and enlisted personnel that provides a bi-directional flow of information between a commanding officer and subordinate military units. The purpose of a military staff is mainly that of providing accurate, timely information which by category represents information on which command decisions are based. The key application is that of decisions that effectively manage unit resources. While information flow toward the commander is a priority, information that is useful or contingent in nature is communicated to lower staffs and units [12].

Crosscutting concerns emerging from coupled software components is a well known problem in software engineering. This problem can be faced commonly in C2 Software development because of high communicational and relational requirements between software components. In order to overcome this problem different software development approaches have been tried by C2 Software developers.

Pure Object-Object oriented approaches for developing C2 Software produces highly coupled components since C2 concerns such as communication, verification or simulation are required to be implemented within each component separately. Even if design patterns reduce the crosscutting among components still they do not provide mechanisms to accumulate concerns in a single component.

Aspect Oriented approach provides an efficient solution to crosscutting concerns through accumulating

the code segments addressing the same concern but implemented in different components within a single component. The mechanism to accumulate these segments into a single component is managed through point cuts. A point cut is a point of execution in a given software component. Identifying all the point cuts corresponding to the same concern provides that all the code related with the same concern to be accumulated in a single component. The responsibility of this component is to identify the point cuts and inject the code related with the concern to location represented by point cuts.

In this paper some of the crosscutting concerns present in C2 Systems will be examined by means of example C2 software called TALOS Mission Planning Software.

3. Mission Planning

Mission can be defined as an objective together with the purpose of the intended action [11]. A mission consists of multiple tasks, and each task should be planned according to rules that allow using sources in the best way to maximize yield while minimizing source usage. Enhanced Mission Planning capability in C4I will choose best rules/ways in a variety of rules/ways autonomously to help operator plan the mission in the most optimized way.

Digital integration of Enhanced Mission Planning will help operator not only in planning a mission by choosing best rules to success a mission objective but also will supply pre-defined mission templates which will make it easy and fast for mistake-free mission planning.

As an example case related with C2 systems we use TALOS Mission Planning Software developed in ASELSAN as the infrastructure on which we develop aspect oriented software components. TALOS project, which is an EU Seventh Framework Programme project, is aiming to develop a mobile, scalable and autonomous system for protecting unregulated land borders. The complete system applies both aerial and ground unmanned vehicles, supervised by a command and control centre.

TALOS Mission Planning Software was developed through MDSD with Eclipse EMF/GMF [6] and aims to plan missions at commander and operator level as an activity diagram. Each task in a mission plan corresponds to an activity in an activity diagram. Mission Executer Component iterates a mission plan and assigns each task in the plan to the corresponding operators or commander with respect to the execution order of tasks. Each task is sent to the corresponding operator as a JAUS message over network.

Since the code of Mission Planning component is 100% auto-generated, the only configuration control is applied to the models which are used to generate the code. If the models are updated the code is generated again.

This procedure necessitates that no manually written code can be inserted to the auto-generated code since manually written code is deleted once the models are changed and the code is auto-generated again. This implies that a software component developed through MDSD cannot be modified within the component. This addresses the use of AOP for feature injections through manually written code.

Some of the concerns related with the example case which are also common to most of C2 systems are communication, verification, mission timeline and simulation concerns. These separate concerns can be considered as separate software components. The definition of these components in the context of the mission planning case is as follows:

- **Communication Component:** This component encapsulates the communication part of the existing software.
- **Verification Component:** This component verifies that mission plan is suitable to be executed.
- **Mission Timeline Component:** This component is responsible for displaying the details of mission in term of duration of each task along with start and finish times. TALOS Mission Planning Software is developed through MDSD, so we do not want to edit the auto-generated code manually. Since the meta-model of each task does not include start and finish times as data fields it will be required to maintain this information externally with this aspect oriented component.
- **Simulation Component:** This component simulates C2 consoles of operators to whom tasks are assigned during mission execution.

4. Object-Oriented Design

An instance of the auto-generated Ecore data model with field names in the classes of the object-oriented design is given in **Error! Reference source not found..**

As can be seen from the data model all mission related elements are of type MissionEntity. Mission element in the model includes all types of mission elements. The first element illustrated in the figure, TaskTrigger, is used to connect tasks in terms of execution order. While the task at the source edge of a TaskTrigger is called sourceTask, TaskTrigger points to targetTask meaning that after completion of sourceTask, targetTask will be executed. TaskGroup in the model can be considered as submission of mission and it may include tasks or other task groups. Task represents a job assigned to one of the actors in the mission. While MissionStartPoint and MissionEndPoint

correspond to the start and finish points of a mission, TaskStartPoint and TaskEndPoint correspond to the start and finish points of task similarly.

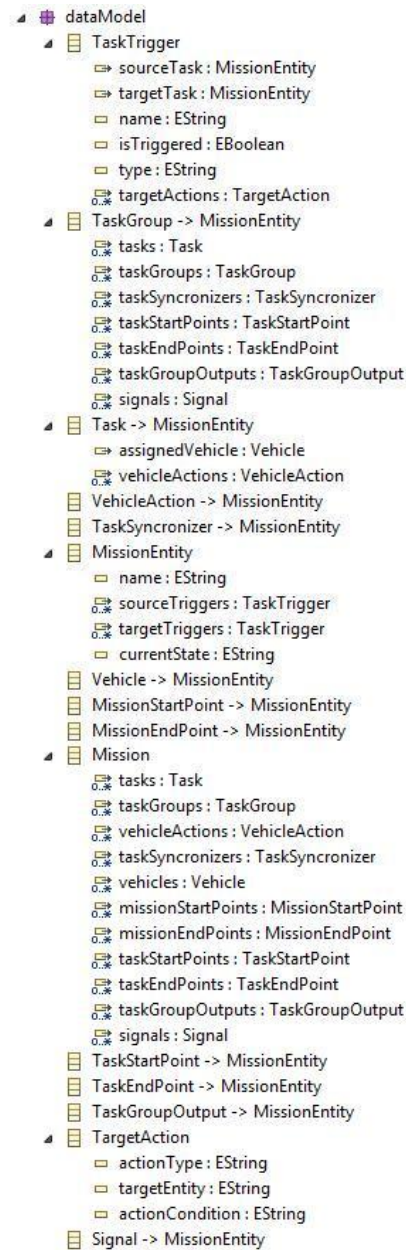


Figure 1. Ecore Data Model

Composite and Observer Patterns are used in the implementation of the MissionExecuter. The definitions along with their usages in the software are presented in the next sections.

4.1 Composite Pattern

“Composite pattern allows a group of objects to be treated in the same way as a single instance of an object. The intent of composite pattern is to compose objects in to tree structures to represent part-whole hierarchies. Composite pattern lets clients treat individual objects and compositions uniformly.” [1].

In this project, a TaskGroup object includes tasks, taskSynchronizers, taskStartPoints, taskEndPoints, taskGroupOutPuts, signals as well as other taskGroups, as represented in Figure 2, where Composite Pattern provides the required object relation.

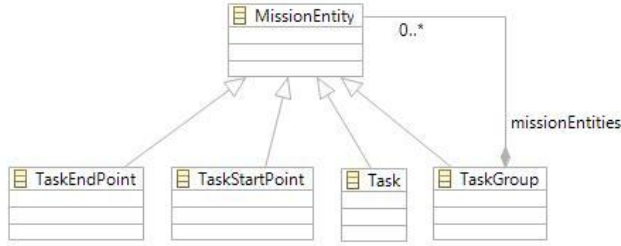


Figure 2. Illustration of Composite Pattern in TALOS Mission Planning Software

4.2 Observer Pattern

Observer Pattern is used to notify objects in a system without knowing the objects explicitly when a change is applied on the notifier object in the system.

We employed Observer Pattern in our project to provide the implicit invocation among the Properties Window and the Editor in the user interface. The modification applied on the text field of the mission in the Properties window is notified to the Ecore structure of the system which notifies the other classes to make the appropriate change on the instances. Therefore, the modification on a selected mission entity on the Properties window is reflected on the Editor window. In the same way, any change made on the Editor window is seen on the Properties window by sending notification to the Ecore and other objects. Observer Pattern in TALOS software is illustrated in Figure 3.

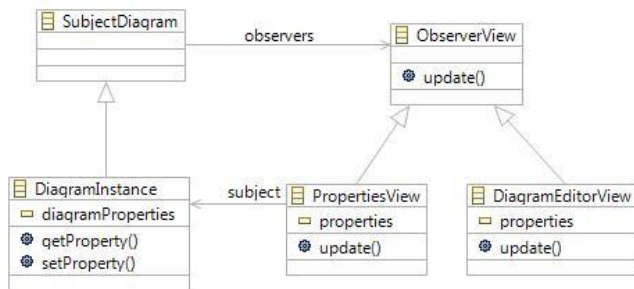


Figure 3. Illustration of Observer Pattern in TALOS Mission Planning Software

An example mission plan is illustrated in Figure 4. The mission plan includes tasks connected with triggers. A trigger is produced upon completion of tasks and starts the execution of the next task in the mission plan. Parallel tasks are started by task synchronizers corresponding to the black bars in Figure 4.

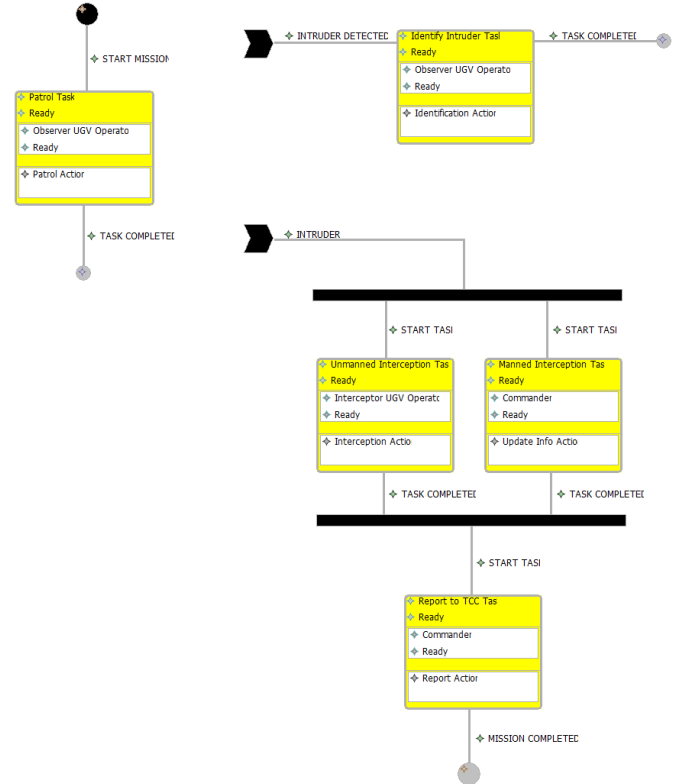


Figure 4. An example mission plan as an activity diagram.

5. Identifying Aspects

As a general problem C2 Software includes many components corresponding to different concerns. In this work we try to show the crosscutting between mission execution concern and several other concerns. As a result of high interaction among these components through interfaces, considerable amount of crosscutting can be identified among these components. Crosscutting among the components introduces difficulties in integration, maintenance, and testing.

The components identified in Section III are considered as the main concerns other than execution concern. Those concerns cause the implementation to be scattered among multiple modules. The available OO paradigm of the system lacks the reusability of the functionalities provided by those components and it lacks

the maintainability of the scattered code. Hence we applied aspect oriented approach to separate the concerns and decrease the complexity by implementing the aspects by AspectJ. Below are the descriptions of each aspect:

5.1. Communication Aspect

During the execution of mission plan, the Mission Executer is responsible for executing the plan and send messages to the entities who are assigned the current task for each iteration of task. The procedure of message sending is done via a JAUS with a protocol such as UDP or STANAG. The message sending functionality is required more than once inside the mission execution code which causes tangling of code and lowers the maintainability. Any modification desired to be made on the communication protocol causes to make changes on lines where the protocol is used which prevent robustness of code. Therefore, the communication concerns can be implemented by means of aspects.

5.2. Verification Aspect

During mission planning, the connections among mission components may not be appropriate for simulation. For instance a connection may be absent between two components which prevents the mission to be completed as intended. A loop may exist in the plan where the user is noticed to get his/her approval. When the mission plan is ready and the Fire button is pressed, it should be verified before the execution to catch those situations. On the other hand, the necessary inputs for Task entity such as operator, task name, state, executer and action may be missing. Whenever a Task is inserted in the plan, the checks for its inputs should be made and the missing components should be presented to the user as warnings which lead verification to be made during the mission planning.

In communication component, there are entities which require verification. The messages prepared to be sent between mission executer and actors (commander or operators) are required to be checked. In order for a message to be a valid message ID and destination address should be among the predefined message IDs and the destination addresses respectively. Hence the communication concern crosscuts with the verification concern.

The verification process arises in various cases as stated above before executing the mission plan. The control of each case for verification results in tangling code. Therefore the verification concerns are accumulated to be used in every verification procedure.

5.3. Simulation Aspect

In mission plan, there may be branches in the mission plan which causes different paths to be generated. In simulation, the aim is to calculate the expected duration of execution for every iteration of task in the mission plan. Hence, the expected duration for every execution path in the mission plan with the path itself is presented to the user. The user has the opportunity to learn the duration of every alternative of execution whenever he/she wants by pressing the simulation button. The user can select the most optimum path by predicting the duration of every path before executing the plan. These operational requirements indicate the crosscutting between mission execution concern and simulation concern.

In order to simulate the operator command consoles, the messages sent to the operators by the mission executer should be absorbed and signals or task completed messages should be sent to the mission executer. This simulation requirement necessitates the simulation and the communication components to interact which imply that the communication and simulation concerns crosscut.

5.4. Mission Timeline Aspect

The aim of mission timeline is to show the start and end times of each task during execution and inform the user at each iteration of task. In mission timeline the user tracks the durations of tasks at the time of execution.

6. Aspect-Oriented Programming

6.1. Production Aspects

Production aspects are essential for software systems to facilitate the components which have crosscutting features in that the application would not work without them when excluded. In this study we developed communication and verification components as production aspects.

6.1.1 Communication Aspect. In order to send messages among operators and commander, each of these actors should have a unique JAUS address, so this field is required to be added to the existing Actor class as illustrated in Code Fragment 1.

```
// JAUS address of an operator or commander
private JAUSAddress Actor.address;

public JAUSAddress Actor.getAddress() {
    return address;
}

public void Actor.setAddress(JAUSAddress address) {
    this.address = address;
}
```

Code Fragment 1. JAUS address is added to the existing Actor class.

Each actor is assigned a unique JAUS address after these actors are extracted from the current mission plan as illustrated in the first advice in Code Fragment 2. The second advice in the same code fragment registers the aspect to the JAUS communicator node corresponding to the commander.

```
pointcut addJAUSAddresses(MissionExecutionView mev):
    target(mev)
    && call(void MissionExecutionView.getActorList(..));

after(MissionExecutionView mev): addJAUSAddresses(mev) {
    /* Assign a JAUS address for each actor */
}

// Register this aspect to the communicator node for detecting
// received messages from other nodes
after(MissionExecutionView mev):
    target(mev) && call(void MissionExecutionView.init(..)) {
    CommanderNode.getInstance().getSubSystemCommander()
        .addMessageListener(this);
    this.mev = mev;
}
```

Code Fragment 2. JAUS address assignment and communicator node registration

When mission executer iterates the mission and starts new tasks in the mission flow, these tasks should be assigned to the corresponding operators through JAUS messages. Advice given in Code Fragment 3 illustrates that for a task started by the mission executer, corresponding task assignment message is sent to the operator.

```
// If the mission starts a new task send a message to the
// corresponding actor
after(MissionExecutionView mev, int messageId, String actorName):
    target(mev)
    && args(messageID, actorName)
    && call(void MissionExecutionView.startTask(int, String)) {
    JAUSAddress address = null;

    /*
     Find the address of the actor specified by actorName parameter
     */

    CommanderNode.getInstance().getSubSystemCommander()
        .sendTaskStartMessage(messageID, address);
}
```

Code Fragment 3. Sending task assignment message

6.1.2. Verification Aspect. Before the commander starts the mission, the validity of the mission should be verified. Code Fragment 4 illustrates two different verification operations. The first advice verifies that each task is assigned to an actor and the type of operator is one of Observer UGV Operator, Interceptor UGV Operator, and Commander. The second advice verifies that each task is started by a trigger and each task starts a trigger after its completion.

```
// Pointcut corresponding to the method call for loading a mission
pointcut verifyTaskExecuter(MissionExecutionView mev):
    target(mev) && call(void MissionExecutionView.init(..));

//Task-executer verification
void around(MissionExecutionView mev): verifyTaskExecuter(mev) {
    /* Verify 1) Task is assigned to an operator or commander
     2) Type of the operator is one of
     - Observer UGV Operator
     - Interceptor UGV Operator
     - Commander */
}

//Mission-flow verification
void around(MissionExecutionView mev): verifyTaskExecuter(mev) {
    /* Verify 1) Each task is started by a trigger
     2) Each task starts a trigger */
}
```

Code Fragment 4. Task Executer verification

Each task message includes a message ID and JAUS address fields. These fields are checked to ensure that their values are among predefined values as illustrated in Code Fragment 5.

```
// Message verification
void around(CommanderSubSystemCommander csc,
    int messageId, JAUSAddress destinationAddress):
    target(csc)
    && args(messageID, destinationAddress)
    && execution(void CommanderSubSystemCommander
        .sendTaskStartMessage(int, JAUSAddress)) {
    /* Verify 1) Message ID is among possible message IDs
     2) Destination is among possible destination
     addresses */
}
```

Code Fragment 5. Message verification

6.2 Development Aspects

Development aspects are used in order to comprehend the flow of software components or can be used to develop new software components. In this section simulation and mission timeline aspects are given as development aspects which are present in C2 software systems.

6.2.1. Simulation Aspect. Whenever the mission executer sends commands to the operators, the operators send the corresponding messages to the other operators by the help of the advice depicted in Code Fragment 6. So the behavior of the operators is simulated instead of controlling them via the consoles.

The simulation aspect is a type of Development Aspect since the simulation of operator behaviors helps to check the messages sent by the executer.


```

after(MissionExecutionView mev, int messageID, String actorName):
    target(mev)
    && args(messageID, actorName)
    && call(void MissionExecutionView.startTask(int, String)) {

        try {
            Thread.sleep(2000);
        } catch (Exception e) {
            e.printStackTrace();
        }

        mev.missionTriggered(messageID, MissionExecutionView.TASK_COMPLETED);
    }

```

Code Fragment 6. Simulation of operator C2 consoles

6.2.2. Mission Timeline Aspect. As the mission plan iterates, the start and end times of the mission and the start and end times of each task are important to see the timeline of the execution flow. So the timeline aspect is a production aspect type.

During mission execution several types of events can be required to be time stamped in order to verify that whether these events happened at expected time. The following code adds the timestamps to events whose times are to be observed as illustrated in Code Fragment 7.

```

void around(MissionExecutionView mev, String timelineEvent):
    target(mev) && args(timelineEvent)
    && call(void MissionExecutionView.addTimeLineEvent(String)) {
        Date date = new Date(System.currentTimeMillis());
        SimpleDateFormat df = new SimpleDateFormat("dd.MM.yyyy hh:mm:ss");
        timelineEvent = df.format(date) + " " + timelineEvent;
        proceed(mev, timelineEvent);
    }

```

Code Fragment 7. Adding time stamp to time related events

7. Related Work

The study conducted for distributed real-time and embedded (DRE) systems running on a command-control platform with multiple middleware QoS management technologies [2] makes use of aspect-oriented approach in one of their middleware. In the case study by Bold Stroke avionics system's application context, the experiments were conducted on Weapons Systems Open Architecture which consists of two airborne servers (one command-control aircraft and one F-15 fighter aircraft). In the implemented and flight-tested multi-layered QoS architecture based on Real-Time CORBA standard, the highest middleware layer in the system which is Quality Objects (QuO) framework, aspect-oriented approach is used in the development of QoS behavior. In QuO Denial of Service (DOS) protection is inserted multiple places in the message processing flow which cross-cuts the decomposition based on processing messages [3]. An aspect component inserts itself into the data flow and maintains its own state and services.

Another case study addresses a command-control simulator for Army fire support which makes use of aspects in the context of Product-line Architectures (PLA)

and Domain-specific languages (DSL) [4]. It is discovered that the components of distributed simulations are not conventional DCOM and CORBA components but rather "aspects" whose addition or removal impacts the source code. In the implementation, a component was found to be decomposed into ten largely independent layers which deal with distinct components which are:

- Reading from simulations scripts
- Communication with local and remote processes
- Proxies
- Different weapons
- GUI displays for graphical depiction of continuing simulations

Treating these capabilities as distinct components simplifies the specifications and debugging. Then, by the use of these components, the extensibility and understandability goals are achieved through PLA and DSL technologies.

8. Discussion

In this paper we showed that several C2 concerns can be separated into different software components and that separation of concerns reduces effort in different phases of software life cycle.

Application of Aspect Oriented approach for communication, verification, execution timeline, and simulation concerns over TALOS Mission Planning Software indicates that these concerns can be separated into different software components.

Communication concern of C2 requires that different entities of the C2 system to exchange messages among themselves and so message sending procedures should be accessible by each of these entities. A software component which sends different messages for different operations or commands requires the communication procedures to be called from different parts of code. A change in the communication infrastructure or even change of the whole communication infrastructure requires all code segments which include calls to communication procedures to be changed one by one. This also requires testing all the components which use communication procedures. By separating the communication concern into a single software component through Aspect-Oriented approach provides advantages such that even when the whole communication infrastructure is desired to be changed, the only part of the C2 software that should be modified is just the communication component.

Similarly verification of different operations or commands requires that all components should implement

verification procedures separately where operations or commands are defined. Accumulating all verification procedures into a single software component eliminates crosscutting of verification concern.

Timeline concern in our example case provides a good example of using aspects over auto-generated code. For instance the task structure does not include variables related with start, end or total execution time of a task. Aspect-Oriented approach also provides mechanisms to inject variables related with aspects to existing code. By injecting such variables we are able to compute time related computations for each task and even the corresponding mission as a whole.

Simulation concern is a common concern present in almost all C2 systems. While simulation can be used to test C2 software components, it can also be used to evaluate operational scenarios before they are executed on the operational area. Since simulation requires the interaction of all software components as if they are operating with real data, simulation is regarded as a crosscutting concern. If the procedure of running a simulation changes, this requires all the software components which include a code segment related with simulation to be modified. Accumulating simulation concern into a single software component eliminates such problems.

9. Conclusion

C2 software includes many concerns whose implementation is distributed in more than one software component resulting in crosscutting concerns among different components. This property of software components causes difficulties in all phases of software life cycle including integration, maintenance and testing.

Applying Aspect-Oriented approach helps to separate C2 concerns into separate software components and as a result eliminates crosscutting concerns. Separation of concerns into different software components provides easy integration of new concerns to the existing software systems. Modification or extension of a concern through changing or new requirements can easily be reflected to the existing software components and testing effort are reduced such that only modified components are required to be tested against the other ones.

Aspect Oriented approach was tried on an existing command control system called TALOS Mission Planning Software. Using this software as an infrastructure to probe separation of concerns showed that concerns such as communication, verification, execution timeline, and simulation aspects can be accumulated into different software component by means of AspectJ through Aspect Oriented approach.

10. Acknowledgement

This paper was prepared in collaboration with Mr. Bedir Tekinerdoğan, Assistant Professor at the Department of Computer Engineering of Bilkent University.

11. References

- [1] Composite Pattern
http://en.wikipedia.org/wiki/Composite_pattern
- [2] C. Gill, J. Gosset, J. Loyall, R. Schantz, M. Atighetchi, and D. Schmidt, "Integrated Adaptive QoS Management in Middleware: A Case Study", *Real-Time Systems*, 29, 101-130, 2005
- [3] J. Zinky, R. Shapiro, Aspect-Oriented Interceptors Pattern Dynamic Cross-Cutting Using Conventional Languages, BBN Technologies, ACP4IS 2003
- [4] D. Batory, C. Johnson, B. Macdonald, and D. von Heeder, Achieving Extensibility Through Product-Lines and Domain-Specific Languages: A Case Study, *ACM Transactions on Software Engineering and Methodology*, Vol. 11, No. 2, April 2002, Pages 191–214
- [5] M JAUS, Domain Model, Volume 1. Version 3.2., 10 March 2005
- [6] Eclipse EMF/GMF, <http://www.eclipse.org>
- [7] C4ISR Handbook for Integrated Planning (CHIP), DoD Integrated C4I Architectures Division, April 1998
- [8] DoD Architecture Framework Version 1.5
- [9] TALOS - Transportable Autonomous patrol for Land bOrder Surveillance system, <http://talos-border.eu>
- [10] <http://www.nato.int/cps/en/natolive/stanag.htm>
- [11] Builder, Carl H., Bankes, Steven C., Nordin, Richard, "Command Concepts - A Theory Derived from the Practice of Command and Control", RAND, ISBN 0-8330-2450-7, 1999
- [12] http://en.wikipedia.org/wiki/Command_and_control

An Aspect-Oriented Development of A Banking Management System

Buğra Mehmet Yıldız, Çağrı Toraman, Uğur Bilen
Department of Computer Science, Bilkent University
Ankara, Turkey 06800
{bugra, ctoraman, bilen} @ cs.bilkent.edu.tr

Abstract— *Banking applications can be considered as one of the most complex systems in software development area. Such systems are also hard to manage when the crosscutting concerns come into consideration that affect the most of the system's components. The Object-Oriented approach provides some solutions for such concerns although this leads to tangling of the code and scattering of the concerns through the system. In this paper, we will focus on an existing client-server based banking management application as an example case and will show how to deal with these problems through some extension examples using Aspect Oriented Development approach and comparing it with the Object Oriented approach. In the end, we will provide a discussion on the experience we had on Aspect-Oriented programming issue.*

Keywords— *Aspect-Oriented Software Development, Banking Application, Banking Software Systems, Crosscutting concerns.*

1 . Introduction

Banking applications can be considered as one of the most complex systems in software development area. The difficulty comes from several reasons. The data entities and their relations should be derived from the business domain in a precise and complete manner in the requirements stage. Also the business rules should be well defined and the data entities should be modeled in coordination with these rules. As a separated concern from data model, the system should also provide persistence for the data relation model in order to store the information. So, a database support is needed for such systems. With the concept of database, synchronization arises as another concern for controlling the accesses to objects and keeping the data consistent.

Other than data-related concerns, monitoring of the system's server is another point that must be taken into consideration. There can be the need of seeing what is going on the server for error checking, performance or forecasting reasons.

For an implemented banking application, a multi-language support may be needed depending on the requirement changes. But after finalizing or at the final steps of the project, it can be very painful to modify the system for supporting the languages other than the implementation language. Most of the interface components and some application components must be revised for providing such functionality.

These problems force the change of the design of numerous components if the solutions are based on Object-Oriented Approach. Also these changes are tend to violate separation of

concerns principle because they lead to tangling code and scattering of the concerns over many system components.

The main aim of this work is to show how to solve the problems that we stated in the previous paragraphs without facing any side effects of crosscutting concerns by applying Aspect-Oriented Development Approach using AspectJ. As an example case, we took a simple banking management application that does not provide the mentioned functionalities and extended it with aspect-oriented programming using Aspect Mining and Early Aspect Identification techniques. The extended system does not have a professional level of business support but we tried to provide an idea of how to aspect technique to a server-client based business application.

In Section 2, the analysis of the existing system is explained. This analysis includes requirement analysis and architectural design with the explanation of applied patterns. In Section 3, we mentioned about the aspects that we applied including their requirement specifications, object-oriented design effects when Object-Oriented Approach is applied instead of aspects and coding details of aspects in AspectJ. Section 4 is the discussion part where we shared our experience with Aspect Oriented approach in detail. Following it, Section 5 gives a brief idea of other works that are done in related areas. Finally, we provide a conclusion in Section 6.

2 . Existing banking management system

2.1 Requirement Analysis

A typical bank management system involves clerk and admin actions. Clerk works as somehow client that requests various demands from a main server. There might be several clerks distributed over various locations. The main function of a clerk would be to use the bank management system for the purpose of satisfying customer demands. For example, a clerk uses the system to withdraw some money from a customer's account, and the system acts as a client requesting withdraw from a main server. On the other hand, there should be an administrator using the system for general managing.

As the diagram shows, an administrator of the system can add/remove/edit a branch, add/edit/remove a clerk and manage an account in terms of system based settings.

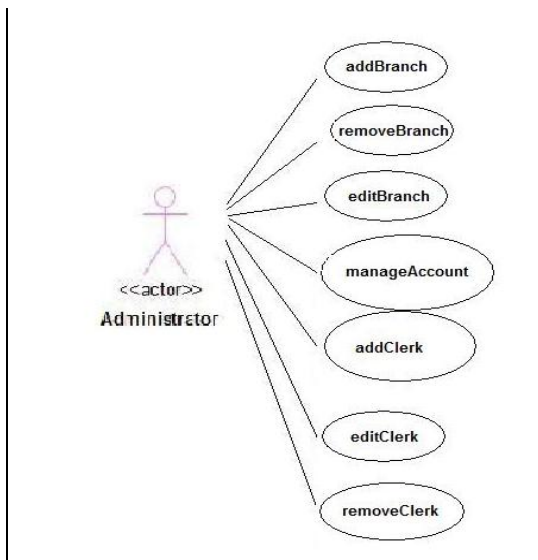


Fig 1. Use case diagram of administrator

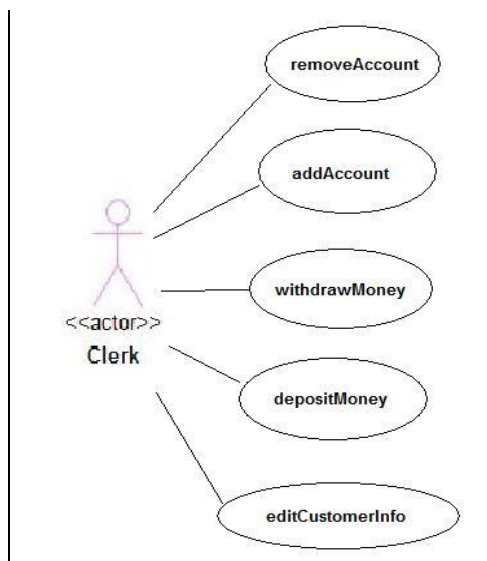


Fig 2. Use case diagram of clerk

As the second user of the system, clerk, can remove an account, add an account, deposit money to an account, withdraw money and edit customer information as it can be seen in Fig 2.

2.2 Architectural Design

The bank management system consists of five layers: Application, data, communication, interface and protocol layer. Fig 3 shows the general structure of the system including these five layers. The function of the application layer is to construct the basic line of the system. All of the complex system requirements are conducted in there. It actually manages the data layer entities that construct the basic objects/components of the business domain. The components may for instance be used for keeping the basic information of a checking/time deposit account or a customer. In order to provide a formal communication between a clerk and the main server, we introduce the communication and protocol layer.

The communication layer's function is mainly to provide a client-server interaction mechanism, and the protocol layer cooperates with the communication layer for defining a formal interaction by providing standardization in request and reply messages. As a metaphor, the communication layer can be seen as a language and the protocol layer is the grammar and words we use for that language. In other words, the protocol layer acts somehow a bridge between communication layers of server and client side. Therefore, communication and protocol layers are meaningless unless they use together. Lastly, we introduce an interface layer that provides the user an easily understandable system interface.

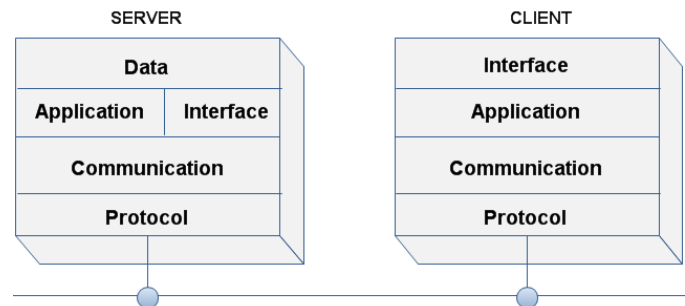


Fig 3. The layered diagram of the application

Fig 4 represents the class diagram of the application layer of the server side. The application layer consists of only "BankManager", which is a singleton class, on the server side. All customers, branches, accounts and other important components that are banking entities of the bank management system are kept and managed in there. It exerts all the core functions that the application needs, such as adding a new customer, removing an existing customer, adding or removing an account or withdrawing/depositing some money from/to an account etc. All of these operations are actually used when a request comes from a client-side application.

The class diagram of the data layer is given in Fig 5. As mentioned before, the data layer represents the business domain entities for the application, such as customer and account information. There are three types of person in the application: Customer, Clerk and Admin. Two of them, clerks and admin, are allowed to use the application directly. Each account is beloved to a specific customer and branch. Also each clerk using the application is located in a specific branch. There are also two types of accounts: Checking and Time Deposit. Any kind of person/account and also any branch in the system are considered as a business entity.

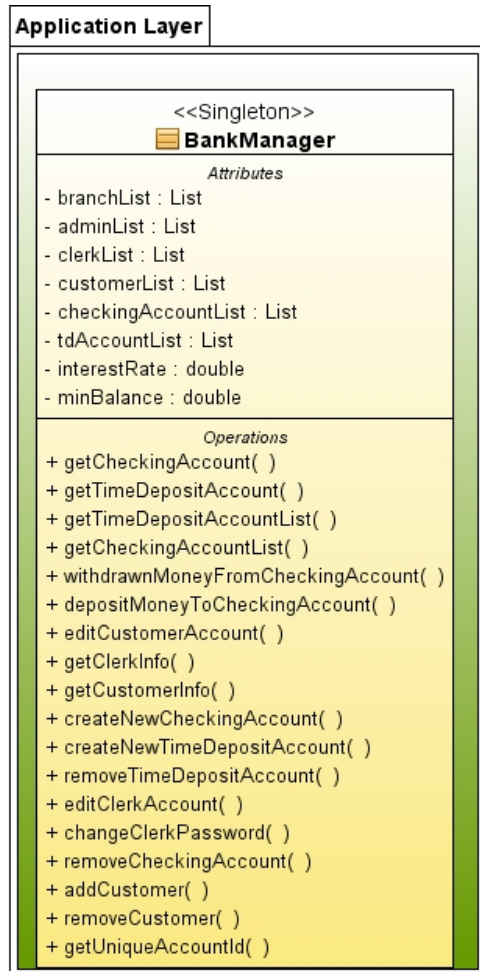


Fig 4. The class diagram of the application layer

2.3 Applied Patterns

2.3.1 Layers Pattern (Architectural Pattern)

The layers architectural pattern divides the system into decomposed groups of sub-tasks in which each group of sub-tasks is at a particular level of abstraction [1]. We separated our system to five layers: application, communication, data, interface, and protocol Layers. Fig 3 demonstrates our layered design. We tried to separate our classes into different levels according to their similarity in tasks accomplished.

2.3.2 Singleton Pattern

Singleton pattern is used to ensure a class has only one instance, and a global point of access to it. In our application there is a class called **BankManager** which handles all the requests coming from clerks and administrators. Figure 4 shows the class diagram of bank manager class and in order to keep the information synchronized, we have to ensure that there is only one instance of it. After we create one instance, all other classes interact only that unique instance of the **BankManager**.

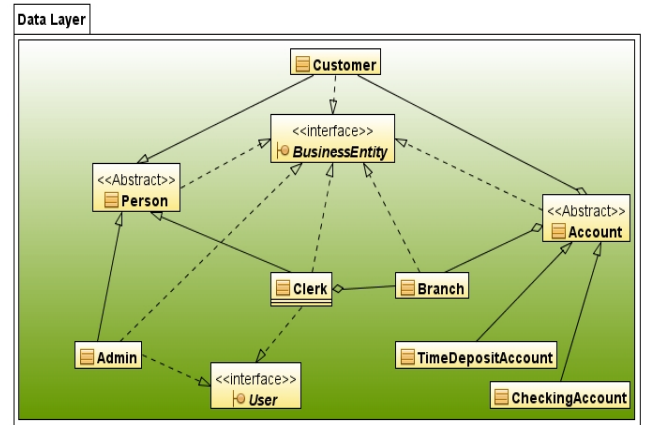


Fig 5. The class diagram of the data layer

2.3.3 Proxy Pattern

In systems that need multiple copies of complex objects, it is better to use proxy pattern [2] since proxy pattern provides an interface for that complex object. In the banking management system, the complex object is the bank itself. Fig 6 shows the applied proxy pattern in the banking management system. As mentioned before, this object is represented with “**BankManager**” class. In this case, the proxy of the bank object is “**ClientApplication**” which is executed on the client-side machine. The communication between the real object and its proxy is provided by the server-client mechanism that is described in section 2.2. Both real and proxy objects implements an interface (**BankInterface**), which keeps the core operations of the bank. Finally, the clerk executes the application and the interface **ClerkApplicationFrame** reaches to the real object by means of the proxy one.

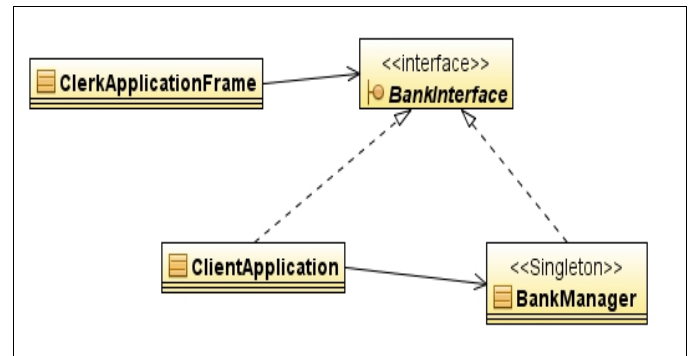


Fig 6. The proxy pattern applied in the system

Applying the proxy pattern in the banking management system, we provide an efficient way to represent the bank object on the client side. The system creates a basic instance of the bank on the server side, and there would be multiple instances of this bank object on the client side by giving references to the real object. By this way, there is no need to create the complex object again and again, and therefore the applied pattern provides an efficient way for the client-server mechanism.

3 . Introducing Aspects

In this section, the details of applied aspects can be found. For each aspect, we defined the requirement specification of the aspect, explained how to implement the requirement without aspects and gave coding details of the aspects in AspectJ.

3.1 Multi-Language Aspect

The aspect-free banking management application we had in the beginning is implemented in English language. We want the application to support other languages also. The assumption we made here is the list of displayed language-text messages are known so that any translation can be done from English to desired language. For our example case, we used Turkish support.

For managing the language translation and the text reading purpose, there is need of a separate class that we called LanguageStringList. This class should provide language service to the interface layer. In interface layer, there are 6 classes (counting both client and server interface classes) and there is need of change in 175 lines of code for Object Oriented solution that seems to be an unpleasant situation for a programmer.

As it can be easily understood from the object-oriented solution, the multi-language concern is scattered through the whole user interface components. The object-oriented solution gives the jobs of language management and user interface to the interface layer classes that leads to tangling of the code.

So we solved the problem with LanguageAspect class. We first determined the pointcuts that must be captured. These points cuts include setText(...) and setTitle(...) operations of the JButtons, JFrames and JLabels. The coding of these capture operations is as follows:

```
public pointcut adminLanguagePointcut( String setString):
    (call( void javax.swing.JFrame.setTitle(String)) ||
     call( void javax.swing.JLabel.setText(String)) ||
     call( void javax.swing.JButton.setText(String)) ) && args( setString);
```

Fig 7. Pointcut code of multi-language aspect

So this pointcut captures all setText and setTitle operations of JFrame, JLabel and JButton objects with a String parameter. Then we implemented an around advice to bypass the method call with the given String parameter and put our language specific String as the new parameter to the method.

```
void around(String setString):adminLanguagePointcut( setString){
    proceed( LanguageStringList.getInstance().getValue(setString));
}
```

Fig 8. Advice code of multi-language aspect

LanguageStringList is implemented as a singleton class so that there won't be any time spending on creating a new instance for each call of setText(...) or setTitle(...) operation. The class simple keeps a HashMap to keep the mapping between English and other language correspondence. The corresponding language text is returned with the getValue(...) method as used in advice.

3.2 Server Monitoring Aspect

In Client-Server based applications, the monitoring of the server operations can become important for performance, error checking, forecasting and other concerns related to knowing what is going on in the system. In this project, the requirement we implemented is monitoring of the request and replies that the server receives and sends so that the activities of the clients can be observer. The request details of each connected client are supposed to be displayed dynamically in an interface that is separate from the initial administrator interface during the whole connection time of the client.

There are about 25 monitoring points in the server side. For each monitoring point, some method call should be coded for displaying the incoming request or outgoing reply. In the case of increase in the number of monitor points, the number of method call must be also increased manually. Also, when the requirement concerns about the monitoring change or extended (such as the monitoring of working threads, details of specific accounts operations), the new monitoring points should be determined and manual code injection should be done for these points.

The management of monitoring points and corresponding method calls is a difficult job if it is done manually. In addition to this, with each monitor method call, the monitoring concern is scattered through the monitored classes which causes the low-cohesion since a class starts to do some job that it does not supposed to do.

We implemented an Aspect class called MonitoringAspect that captures the desired monitoring points, and for each monitoring point, it makes the needed method call for display purpose. The pointcut code that captures the monitoring points is as follows:

```
public pointcut incomingRequest():
    (call ( Object ObjectInputStream.readObject()) && within( WelcomeThread));
public pointcut outgoingReply(Object curReply):
    (call ( void ObjectOutputStream.writeObject( Object)) && within( WorkerThread)
     && args( curReply);
```

Fig 9. Pointcut code of monitoring aspect.

As it can be understood, the incomingRequest pointcut captures the incoming request at the time the request object is returned from the incoming stream. This pointcut is limited to Server class's code. For capturing the Reply object that is supposed to be sent to the client is done by outgoingReply pointcut at the time of converting the target object to the stream.

For displaying the captured Request and Reply objects, the following advises are implemented:

```
after() returning ( Request o):incomingRequest(){
    ServerMonitorFrame newFrame= ServerMonitorFrame.getInstance();
    newFrame.handleRequest(o);
}
before( Object r): outgoingReply( r){
    ServerMonitorFrame newFrame= ServerMonitorFrame.getInstance();
    newFrame.handleReply((Reply)r);
}
```

Fig 10. Advice code of monitoring aspect.

Here, the Request object returned from the readObject() method of ObjectInputStream class is captured and given as the parameter to ServerMonitorFrame singleton class's handleRequest() method. Similarly, the Reply object that is the parameter of the writeObject() method of the ObjectOutputStream Class is captured before the method is executed and given to ServerMonitorFrame class as the parameter of handleReply() method.

3.3 Synchronization Aspect

Since this is a client-server based multi-thread application, there can be multiple clients that want to reach to the same business entity. So some synchronization mechanism is needed for providing safe access to these entities in order to keep the data consistent. For instance, a client should not be allowed to access a customer's account while another client is editing the same account.

In order to provide synchronization, a new class (let it be 'Synchroner') that manages the synchronization of a business entity should be implemented. This class prevents a second access to a business entity while some other client uses that entity. All other threads that want to access this entity are also forced to wait. When a thread finishes its operation on that business entity, then it should notify one of waiting threads. These wait and notify operations can be implemented by synchronized 'wait()' and 'notify()' methods of Java. Besides, 'Synchroner' class has a variable called 'beingUsed' to keep the current status of the business entity. If it is true then it will mean that someone currently uses the object, and no one uses vice versa.

Although synchronization seems to be implemented easily with the help of wait and notify methods, there are some issues that should be considered. First issue is the implementation of the code blocks that provide synchronization. Since we have more than one business entity that needs to be synchronized, the code blocks are scattered over the java classes of these business entities. Other problem is that an individual module or class will get a second concern - synchronization unless we imply the aspect-oriented way. Thus, an individual module gets a tangled code. The synchronization should be organized such that each module is related with its own concern, not synchronization.

In order to provide a consistent aspect-oriented approach, we assign a separate 'Synchroner' for each customer and account. The top lines of Figure 11 are the assigned variables and get methods for each. Since all customer and account operations are conducted with respect to the request coming from a clerk, we assign a pointcut on the run() method of the 'WorkerThread' class. This pointcut is also given in the below of the same figure.

```
private Synchroner Customer.synch = new Synchroner();
public Synchroner getSynchroner(Customer c) { return c.synch; }
private Synchroner CheckingAccount.synch2 = new Synchroner();
public Synchroner getSynchroner(CheckingAccount a) { return a.synch2; }

pointcut synchCut(WorkerThread w): this(w) && execution(void run());
```

Fig 11. Assigned variables and their get method with the pointcut defined for the synchronization.

Before the run operation of any thread with an incoming request, the aspect should force the thread to wait if it is being used by someone else. In order to do so, it needs to know whether the worker thread wants to do an operation that requires synchronization such as editing a customer info or depositing/withdrawing a checking account etc. Figure 12 displays the before advice for the above purposes. In this figure, pause method is used to illustrate the thread gets busy with the operation while another clerk wants to access the same object meanwhile.

```
before(WorkerThread w): synchCut(w) {
    Request r = w.getRequest();
    Object obj=null;
    Synchroner syn=null;
    if( r.hasCodeOf( RequestCodes.EDIT_CUSTOMER) || r.hasCodeOf( RequestCodes.DELETE_CUSTOMER) ||
        r.hasCodeOf( RequestCodes.CUST_GET_INFO){
        obj = BankManager.getInstance().getCustomerById(r.getId());
        if( obj!= null){
            syn = getSynchroner((Customer)obj);
            if( syn.getBeingUsed()){
                syn.makeWait(obj);
            }
            else{
                syn.setBeingUsed(true);
                syn.pause(15);
            }
        }
    }
    else if( r.hasCodeOf( RequestCodes.DEPOSIT_MONEY) || r.hasCodeOf( RequestCodes.WITHDRAW_ACC) ||
        r.hasCodeOf( RequestCodes.DELETE_CHK_ACC) || r.hasCodeOf( RequestCodes.GET_CHK_ACC){
        obj = BankManager.getInstance().getCheckingAccountById(r.getId());
        syn = getSynchroner((CheckingAccount)obj);
        if( obj!= null){
            syn = getSynchroner((CheckingAccount)obj);
            if( syn.getBeingUsed()){
                syn.makeWait(obj);
            }
            else{
                syn.setBeingUsed(true);
                syn.pause(15);
            }
        }
    }
}
```

Fig 12. Advice for before running a worker thread.

Similarly after executing a request of a worker thread, the aspect should notify one of the threads waiting on the object's monitor. Figure 13 shows the after advice for this purpose.

```
after(WorkerThread w): synchCut(w) {
    Request r = w.getRequest();
    Object obj=null;
    Synchroner syn=null;
    if( r.hasCodeOf( RequestCodes.EDIT_CUSTOMER) || r.hasCodeOf( RequestCodes.DELETE_CUSTOMER) ||
        r.hasCodeOf( RequestCodes.CUST_GET_INFO){
        obj = BankManager.getInstance().getCustomerById(r.getId());
        if( obj!= null){
            syn = getSynchroner((Customer)obj);
            syn.makeNotify(obj);
            syn.setBeingUsed(false);
        }
    }
    else if( r.hasCodeOf( RequestCodes.DEPOSIT_MONEY) || r.hasCodeOf( RequestCodes.WITHDRAW_ACC) ||
        r.hasCodeOf( RequestCodes.DELETE_CHK_ACC) || r.hasCodeOf( RequestCodes.GET_CHK_ACC){
        obj = BankManager.getInstance().getCheckingAccountById(r.getId());
        if( obj!= null){
            syn = getSynchroner((CheckingAccount)obj);
            syn.makeNotify(obj);
            syn.setBeingUsed(false);
        }
    }
}
```

Fig 13. Advice for after running a worker thread.

3.4 Persistence Aspect

For the initial banking application case, the assumption was the server is always on so there is no need of storing data. Such an assumption seems to be not rational for such a domain but as we stated in introduction part, we try to give an idea of how to implement the crosscutting concerns in an Aspect-Oriented manner.

Since, a data-oriented domain such as our banking case needs its data to be stored, the requirement is to upgrade the initial application to have a database for persistence purpose.

In order to provide persistence application with the database, a new class called 'DatabaseHandler' should be implemented. This class actually does the all database work to achieve a well-organized interaction between the database and the application. The operations such as connecting to the database, executing a mysql query etc. are all handled by this class.

Database handler should be a separate concern that is related with almost all business entities. However, it is scattered over those entities unless we use aspect-oriented way. Also each business entity like customer relates with the database handling concern whereas aspect orientation avoids this problem.

To implement the database concern as an aspect, we get help of 'DatabaseHandler' class. Figure 14 displays a list of pointcuts. These pointcuts are related to specific methods such as creating/removing a clerk, branch or a customer. However, this figure shows some sample pointcuts for persistency aspect. There are also several pointcuts we regarding the database-oriented operations of business entities.

```
pointcut clerkCut(Clerk c): (target(BankManager) && call(int addClerk(Clerk))) && args(c);
pointcut clerkCutRemove(Clerk c): (target(BankManager) && call(int removeClerk(Clerk))) && args(c);

pointcut branchCut(Branch b): (target(BankManager) && call(int addBranch(Branch))) && args(b);
pointcut branchCutRemove(Branch b): (target(BankManager) && call(int removeBranch(Branch))) && args(b);

pointcut cusCut(Customer c,String s): (target(BankManager)
&& call(Reply addCustomerSafe(Customer,String))) && args(c,s);
```

Fig 14. Pointcut code of persistency aspect.

The application should save a new data created on the application to the database immediately. Figure 15 displays a sample of operations whose pointcuts are given in the previous figure. Note that these operations are controlled by an object called 'dh' which is an instance of 'DatabaseHandler' class. There are several advices handling other operations of persistency aspect as well, but we just give a sample of them for the sake of report simplicity.

```
after(Clerk c) returning(int i): clerkCut(c) {
    if( i == 0 || i == 1)
        dh.addClerk(c,i);
}

after(Clerk c) returning(int i): clerkCutRemove(c) {
    if( i == 0)
        dh.removeClerk(c);
}

after(Branch b) returning(int i): branchCut(b) {
    if( i == 0)
        dh.addBranch(b);
}

after(Branch b) returning(int i): branchCutRemove(b) {
    if( i == 0){
        dh.removeBranch(b);
    }
}

after (Customer c,String s) returning(Reply r): cusCut(c,s){
    if( r.hasCodeOf(ReplyCodes.OPERATION_SUCCESS))
        dh.addCustomer(c);
}
```

Fig 15. Advice code of persistency aspect.

4. Discussion

In this work, we have chosen to implement the target system from scratch. During the implementation of the aspects, we both assumed that there is (this) existing system and also the requirement that must be satisfied will be developed without considering the existing system. This approach we followed allowed us to experience two different ways of identifying aspects. For multi-language aspect, we applied Aspect Mining technique. In the existing code, we captured the points where the translation method will be called (These calls are made especially in setText(..) operations and scattered over user interface code) and implemented the translation method call as aspect advice. In other implementations, we identified the aspects from the requirements by the Early Identification technique. When we compare these two techniques, we concluded that the mining of the aspects from the existing code is relatively easier compared to getting the aspects from the requirements. The reason of this especially the following: the capturing points in Early Identification technique is a bit more difficult compared to Aspect Mining since mining of the aspects can be done in a more formal, standardized way (Even there is tool support for mining although we didn't use because of the size of our project).

During the project process, we also realized that there is a trade-off between using the object-oriented solution and Aspect-Oriented solution. Although Aspect-Oriented approach offered us some excellent solution in some cases like multi-language and monitoring, we decided not to implement some aspects that we thought during brain-storming period such as error and constraint checking. Because, these concerns were not very crosscutting for our case since they were just placed in some particular components. For example, some domain-constraint checking done on parameters of the data objects can be done specifically on those objects, where the implementation of aspects requires much more work than just implementing error or constraint checking code integrated in data components. So the selection of aspects or object-oriented coding basically depended on how much the target concern is scattered for our project.

Another point that we learnt in this work is that Aspect-Oriented approach is not an alternative of Object-Orientation, it can be rather qualified as a complement or helper to Object-Oriented approach. For each aspect we implemented, we needed to code an application class that will provide the functionality of the aspect's concern. As examples, `LanguageStringList` and `ServerMonitorFrame` are two classes that do the real computational work of the multi-language and monitoring aspects. The jobs of these aspects were to remove the language and monitoring concerns out of the classes whose job is different.

The main difficulty we faced during the project was not finding the pointcuts, but the managing of the coding syntax of some of the pointcuts. This difficulty we experienced has several reasons. First one was the legacy code's coding style was not standardized in some points of the system. As an example, it was easy to capture all the `setText(..)` operations because the Java library uses the same conventions in all of the classes. But, during capturing of some points in our custom code, we needed to increase the size of the pointcut since we couldn't shorten it to handle all the method calls. For this reason, we thought that it could be an excellent idea to add the auto-capturing property to the Eclipse environment we used. For example, a capture selection can be implemented in Aspect plug-in that captures the desired method calls by right clicking on them. After this, the programmer can modify the resulted customizable code in some way.

5. Related Work

In their paper [3], Rashid and Chitchyan assert that persistence - the storage and retrieval of application data from secondary storage media - is often used as a classical example of a crosscutting concern. It is widely assumed that an application can be developed without taking persistence requirements into consideration and a persistence aspect plugged in at a later stage. However, there are no real world examples showing whether persistence can in fact be aspectised and, if so, can this be done in a manner that promotes reuse and is oblivious to the application. In this paper, they provide an insight into these issues drawing upon their experience with a classical database application: a bibliography system. They argue that it is possible to aspectise persistence in a highly reusable fashion, which can be developed into a general aspect-based persistence framework. Nevertheless, application developers can only be partially oblivious to the persistent nature of the data. This is because persistence has to be accounted for as an architectural decision during the design of data-consumer components. Furthermore, designers of such components also need to consider the declarative nature of retrieval mechanisms supported by most database systems. Similarly, deletion requires explicit attention during application design as mostly applications trigger such an operation.

In another work [4], Coelho et.al. states that modern applications are typically complex, multithreaded, distributed, and often should provide real-time responses and small-footprint. Due to such characteristics, most often, it is hard to understand the behavior of such systems and consequently

detect the root causes of performance or reliability problems. In order to collect information about system's runtime behavior - operations' performance, internal threads status - the system developer is required to instrument the target application (and sometimes also its execution platform). Such monitoring code which allows the developer to reason about the code execution is not localized in a single application module; it must be included in many modules. As a consequence, the monitoring concern tends to be scattered across multiple application/platform modules and tangled with other application concerns. The Application Monitor pattern supports the separate definition of monitoring-related functionalities concerns through the use of aspect-oriented programming. It decouples such concerns from the implementation of application-specific concerns, which in turn improves the system reusability and maintainability.

6. Conclusion

In this work, we implemented some crosscutting concerns that can be encountered in most of the data-related complex systems by taking a bank management system as an example case. It can be seen that these general concerns are related to the several components in the system, classes for our case, and the management of these concerns can be very difficult if they are approached from Object-Oriented perspective. The difficulty comes from the fact that the crosscutting concerns lead to scattered code that causes to low-cohesion and high coupling in components that is the most unpleasant situation for the software engineer. We showed that Aspect Oriented approach provides a good way of managing these crosscutting concerns without living the handicaps of Object Oriented approach. To make clear the benefits of aspects, we also provided brief descriptions of the object solutions.

While we were identifying aspects, we followed two aspects identification methods that are Aspect Mining and aspect derivation from requirements. Aspect Mining is relatively easier compared to derivation from requirements since the first approach has a more formal methodology. On the other hand, it must be stated that it is not possible to do Aspect Mining if there is not a legacy code or a software product line, so the early identification of aspects from the requirements is a critical issue that must be expertise by the designer.

Finally, we realized that Aspect Oriented programming is not an alternative to the Object Oriented approach. Rather, it offers a good way of solving the crosscutting problems that Object Orientation suffers, so that aspects complete objects in such concerns. There is a trade-off between using these two approaches and the decision of the approach that will be applied depends on the designer's evaluation of this trade-off.

7. Acknowledgment

We would like to specially thank to Asst. Prof. Bedir Tekinerdoğan for giving us the opportunity to show ourselves in this workshop of the one of the leading software techniques that is Aspect-Oriented Software Development.

8 . References

- [1] Buschmann, F., R. Meunier, H. Rohnert, P. Sommerlad, and M. Stal. *Pattern-Oriented Software Architecture: A System Of Patterns*. West Sussex, England: John Wiley & Sons Ltd., 1996
- [2] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable ObjectOriented Software*. Addison-Wesley, Reading, Massachusetts, 1995.
- [3] Rashid, A., Chitchyan, R. *Persistence as an Aspect*. In: proc.of 2nd International C. on Aspect Oriented Software Development Boston-USA, March 2003
- [4] Coelho, R., Dantas, A., Kulesza U., Cime W., von Staa, A., and Lucena, C. *The Application Monitor Aspect Pattern*.

Aspect-Oriented Refactoring of a J2EE Framework for Security and Validation Concerns

Başak Çakar, Elif Demirli and Şadiye Kaptanoğlu

Department Of Computer Engineering, Bilkent University
06800 Bilkent, Ankara, Turkey
{cakar, demirli, sadiye} @ cs.bilkent.edu.tr

Abstract— Object-oriented frameworks are widely used for exploiting reusability in application development. The degree of reuse is determined by how much of the application's required functionality is provided by framework's modules. The concerns whose enforcement policies differ among applications cannot be implemented in the framework core, so those concern's implementations cannot be reused. In this paper, we provide an aspect-oriented extension of a J2EE Object-Oriented framework BORFWCORE for security and validation concerns. The extension contributes to the framework in two ways. First, it makes the security and validation concerns' implementations reusable. Secondly, it prevents code scattering and tangling in application development related to those concerns.

Keywords---Aspect-Oriented Programming, Spring AOP, Security, Validation

1. Introduction

A software system is designed and implemented as asset of modules. A module is a program unit that has an explicit syntactic definition and an explicit method of activation, is semantically independent, performs a definite subtask in a software system, and has some definite set of "contact points" with the rest of the program, in both data flow and control flow [1]. In software systems there can be a lot of concerns which cannot be implemented just by the use of a module hierarchy. Thus concerns generally have to crosscut the existing module hierarchy in order to add new features or modify existing ones in the software system. Lots of crosscutting concerns in the system decreases extensibility and maintainability by increasing complexity. These crosscutting concerns cause scattering and tangling code. Scattering is the situation of having code for a specific concern spread out throughout the implementation. The reason of this situation is that single concern affects multiple modules. In addition, tangling is the condition of having multiple concerns interleaved in a single module [2]. In order to solve crosscutting concern problem Aspect-Oriented Programming (AOP) can be used.

AOP is a new programming paradigm developed to support software component reuse, modification, and enhancement, based on defining and applying (weaving) aspects that implement cross-cutting concerns [1]. An aspect is a new unit

of modularization and the main goal of AOP is designing and developing programs by separation of concerns in terms of aspects.

In our experiment we use a framework which is developed to implement mobile applications. Frameworks are often used to develop components which are used to simplify the development of infrastructure and they affect the flow of control between classes. Since frameworks aim at reusability, implemented components in the frameworks are used in different applications which are developed by using the existing framework. The main problem of the framework we work on is that the modules of the framework do not handle security and validation concerns. Because of that developers have to handle these each time when they are required. This causes crosscutting concerns in applications. In order to solve this problem we use aspect oriented approach and write aspects for each crosscutting concern.

One important crosscutting concern is security. Security in its general sense is a condition in which an entity does not suffer harm from threatening events [3]. In our case, application security encompasses measures taken throughout the application's life-cycle to prevent exceptions in the security policy. Systems are often developed without security in mind, and when the time arrives to deploy these systems, it quickly becomes apparent that adding security is much harder than just adding a password protected login screen [4]. Security aspects are important to improve application strength. There are many different security aspects and we selected three of them to implement because they are more important for our sample application. First one is the authentication aspect which is used to identify the user in the system in order to allow him/her to use the functionalities of the system. Second one is authorization which decides which functionalities of the system can be used by the current authenticated user. The last security aspect is data access control which ensures that each user can access only his/her own emails and messages.

Our other aspect type is input validation. Input validation refers to those functions in software that attempt to validate the syntax of user provided commands [5]. This aspect

protects database from invalid inputs and prevent users from entering wrong information.

Extending framework with AOP provides developers to perform checking operation before implementing all related methods in the developing application so they do not have to deal with these crosscutting concerns. By this way they can focus on only their developing applications without thinking security and validation concerns in each module. This also decreases the development time of new applications.

The remainder of this paper is organized as follows: In section 2 BORFWCORE; a mobile application development framework is explained in detail. Section 3 describes the problem statement. Section 4 explains the aspect-oriented extension of the framework. The extension is implemented with Spring AOP. Section 5 provides an alternative implementation: JBoss. Related work to our case is explained in Section 6 with some comparisons. In section 7 we discuss our work and results emphasizing the limitations. In the next section we conclude. Section 9 and 10 include Acknowledgement and References respectively.

2. BORFWCORE: A Mobile Application Development Framework

2.1 Overview of the framework

Reusability is an important issue in today's software development practice and research. Since it increases the productivity, it is highly considered by the software companies whose product portfolio consists of similar products. There are various approaches for designing reusable software and object-oriented framework is a popular one. An object-oriented (OO) framework is a kind of reusable software architecture comprising both design and code.

OO frameworks are used for developing various product families such as mobile applications. For mobile software, compatibility of the system across multiple mobile devices is an important concern. Since development with OO frameworks ensures compatibility, they are widely used for mobile application development [6].

BORFWCORE is a mobile application development framework that we took from Bor Yazilim for aspect oriented extension. Various mobile applications had been developed by Bor Yazilim using this framework.

BORFWCORE is a J2EE (Java 2 Enterprise Edition) based mobile application development framework; J2EE is a platform for server programming in Java programming language. In BORFWCORE, Model-View-Controller (MVC) pattern is applied and some third-party open source libraries are used. The open source libraries used in the work described in this paper are Struts, Spring, JSF and Hibernate.

2.2 Architecture of the framework

BORFWCORE is a J2EE application that is implemented based on Model-View-Controller pattern structure. To explain the basic structure of the framework we provide the Model-View-Controller architecture of the system. The architecture model based on MVC is shown in Figure 1.

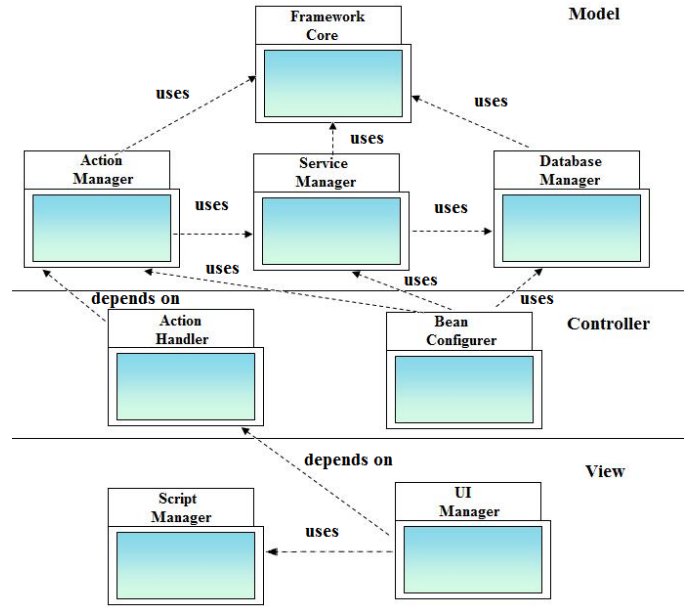


Figure 1. MVC Architecture of BORFWCORE

As modeled in Figure 1, the system consists of three fundamental parts.

View consists of UI Manager and Script Manager. UI Manager uses scripts of Script Manager. View gets the events caused by actions of users and directs those events to Action Handler which is an element of Controller.

Controller consists of Action handler and Bean Configurer. Action Handler triggers the corresponding action in Action Manager as a result of the event directed from View. In other words, Action Handler maps user interface events to actions. Bean Configurer configures beans for required objects in model for instantiating them at runtime.

Model consists of four fundamental modules. Action Manager gets commands from Action Handler and uses services provided by Service Manager in order to complete the action. Service Manager reads/writes database via Database Manager.

2.3 Open source libraries in the framework

In the implementation of the framework, some popular third-party libraries are used. Those libraries are Struts, Spring, JSF and Hibernate.

Struts is an open-source framework for creating Java web applications. It extends the Java Servlet API to encourage developers to adopt a MVC architecture [7].

The Spring framework is another open source application framework for the Java platform. The core features of the Spring Framework can be used by any Java application, but there are extensions for building web applications on top of the Java Enterprise platform [8].

JavaServer Faces (JSF) is a Java-based Web application framework intended to simplify development integration of user interfaces and the operation on the values they hold [9].

Hibernate is an object-relational mapping (ORM) library for the Java language, providing a framework for mapping an object-oriented domain model to a traditional relational database [10].

2.4 Object-Oriented Design of the Framework

The partial class diagram of the system is shown in Figure 2. It includes classes which are commonly used in our sample application. There are four types of classes; service implementation classes (MessageServiceImpl, SmsMessageServiceImpl, EmailMessageServiceImpl, UserServiceImpl), action classes (EmailAction, SmsAction, UserAction), entity classes (EmailMessage, SmsMessage, User) and hibernate classes (SmsMessageDaoHibernate, EmailMessageDaoHibernate, UserDaoHibernate).

and hibernate classes (SmsMessageDaoHibernate, EmailMessageDaoHibernate, UserDaoHibernate).

User invokes the operations of action classes from User Interface of the application. Action classes use service implementation classes to execute the request which activates from User Interface. After service implementation classes complete the execution result is recorded to database by using methods in hibernate classes.

2.5 Sample Application working on the framework

Our application focuses on three basic use cases; ManageEmail, ManageSms and ManagerUser. These CRUD use cases includes Edit, Delete, Save and View of related object. System controls emails and sms regularly so as to send unsent messages. After any user save a new email or sms, system fetches this unsent message and sends it to destination address or number automatically. In our sample application there are three types of users. Admin can perform all use cases. Some operations in the system are not used by all users. Admin give users a right to use edit and delete operations. Authorized user can use all operations in ManageEmail and ManageSms use cases. However; regular user can only save and view their own email and sms messages. The use case diagram of the sample application is provided in Figure 3. Some screenshots of the sample application can be seen in Appendix A.

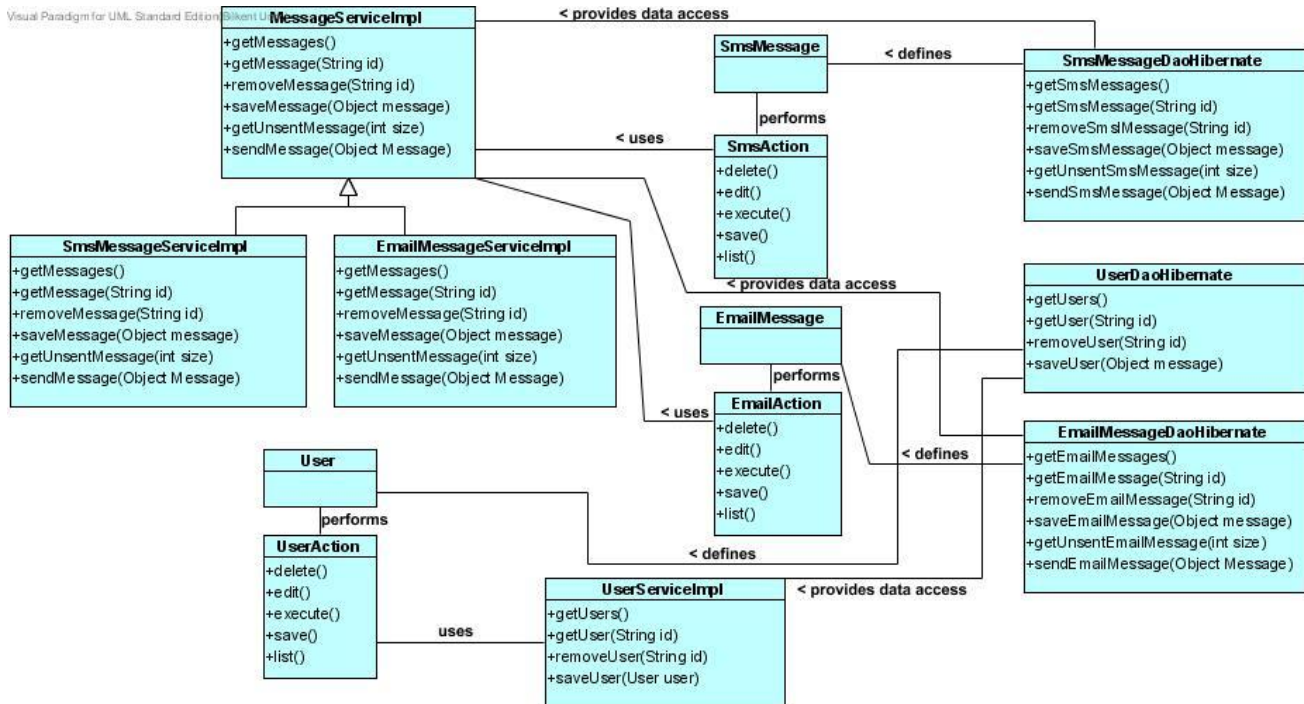


Figure 2. Partial Class Diagram of BORFWCORE

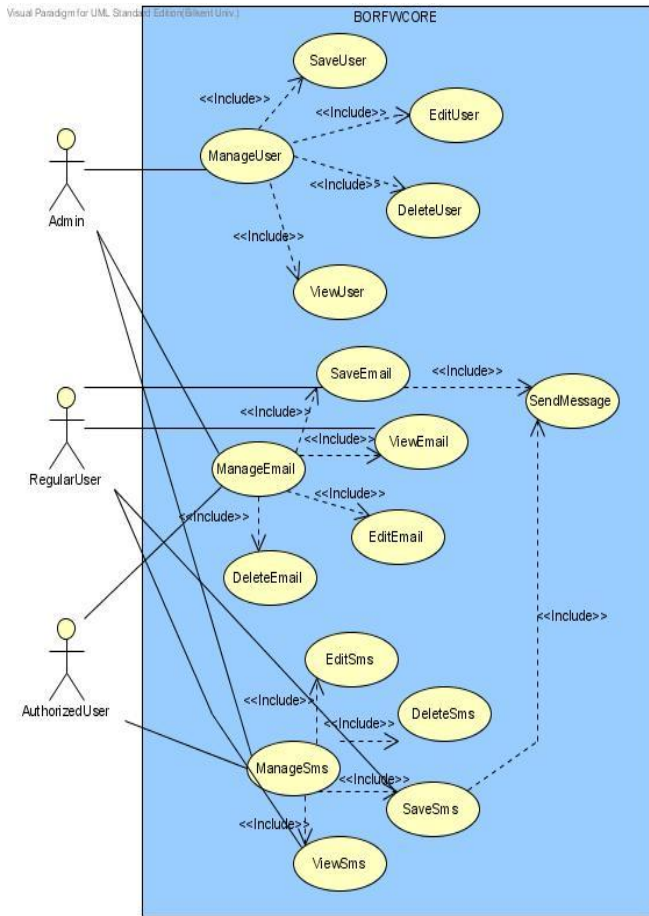


Figure 3. Use Case Diagram for Sample Application

2.6 Application of Adapter Pattern

Adapter pattern is used to convert the interface of a class into another interface clients expect. It provides to create a reusable class that cooperates with classes that do not necessarily have compatible interfaces [11]. In the original implementation of the framework, there was a structure that we need apply Adapter pattern on. The original design of that structure is shown in Figure 4.

In the design, they implement UserServiceImpl class separately and they have to write operations in this class although all the operations work in the same way with the operations in MessageServiceImpl class. UserServiceImpl is possibly separated from MessageServiceImpl because of the following reason:

MessageServiceImpl contains the operations related to message object such as getMessages(), removeMessage() etc. Since both EmailMessage and SmsMessage are some type of

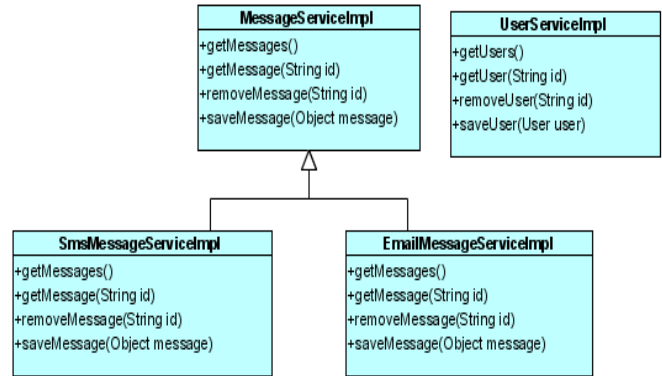


Figure 4. The Structure of ServiceImpl Package before Adapter Pattern

message both EmailMessageServiceImpl and SmsMessageServiceImpl extends MessageServiceImpl. However, since user is not a type of message, and its operation names don't match the super class MessageServiceImpl (getUsers() instead of getMessages()), it is designed separately.

We apply object adapter pattern to this design in order to reuse operations in MessageServiceImpl class by creating an additional adapter class UserServiceAdapterImpl. The resulting structure of the ServiceImpl package is shown in Figure 5.

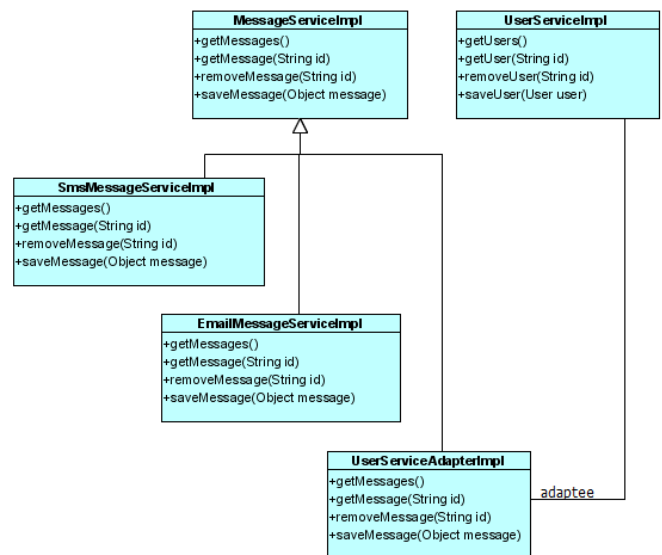


Figure 5. Structure of ServiceImpl package after Adapter Pattern

3. PROBLEM STATEMENT

Developers and users of BORFWCORE states that the main challenges of developing applications on this framework is implementing security and validation. Security consists of authorization, authentication and access control [4]. Validation checks the format of user inputs. These concerns have impact on many modules of the overall system. To implement these concerns within each module they are related to cause code scattering and tangling in the application.

Since developers have to handle security and validation concerns in each single module code tangling occurs. An example to possible code tangling in BORFWCORE is provided in Figure 4.

In our application clients need to authenticate themselves before accessing the resources. In the original implementation of the framework, it needs to be checked whether the user is authenticated or not at the beginning of each operation that only authenticated users has the right to operate it. Figure 4 shows the delete() method of SmsAction class. It is seen that authentication status is checked before deleting the message. Authentication concern is implemented in the same way at the beginning of many other similar methods such as list(), save() of SmsMessage class and the similar methods of EmailMessage class. This situation results in scattering of authentication across many modules and makes development and maintenance of the concern hard.

Authorization is another concern that causes similar problems. In our application, doing some operations require authorization. For example, regular users aren't authorized to delete sms or email messages. Thus, as it is seen in Figure 4 at

the beginning of delete() method the authorization of the user is checked, too. This situation causes the same scattering problem as in authentication case, and also introduces tangling problem. It is seen that both authentication and authorization concern is implemented in the same module that lowers the cohesion of the module.

Access control checks whether the user has the right to access some certain data or not. In our case, regular and authorized users have the right to manipulate only their own sms and email messages. Admin has the access to all messages. At the beginning of list methods, it is determined whether the whole messages will be listed or only the user's own messages will be listed depending on the access control rights of the current user. Thus, this situation causes the similar problems as in the authentication and authorization.

Validation is to check whether the user input is in the correct format or not. At the beginning of each method that is invoked when the user wants to input data to the system, the correctness of the entered data is checked. When the format of a new input field needs to be checked, the developer needs to change the implementations of the related functions based on the new policy. This situation also causes code scattering and tangling.

As it is seen, the implementation of security and validation concerns results in scattering and tangling which make maintainability harder. The complexity of programming cross-cutting issues can be decreased by applying aspect oriented programming. AOP encapsulates the crosscutting concerns in explicit modular unit this increases the modularity of system for reuse and also decreasing the scattering and tangling codes.

```
public String delete() {
    User user = (User)ActionContext.getContext().getSession().get("user");

    if( user != null){

        int userType = Integer.parseInt(ActionContext.getContext().getSession().get("userType").toString());

        //If the user is authorized user or admin
        if( userType == 1 || userType == 2){
            service.removeSmsMessage(smsMessage.getId().toString());

            List args = new ArrayList<Long>();
            args.add(smsMessage.getId());
            ActionContext.getContext().getSession().put("message", getText("meMessage.deleted", args));

            return SUCCESS;
        }
        else{
            return INPUT;
        }
    }else{
        return INPUT;
    }
}
```

Figure 6. The original implementation of delete method of SmsAction class

4. Aspect-Oriented Programming

AOP is a new paradigm which provides separation of crosscutting concerns with a new type of modularization [12]. This unit is called aspect and it itself crosscuts other modules. By this way we implement the crosscutting concerns in aspects instead of distributing them in core modules. As a result AOP modularizes the crosscutting concerns in a clear manner and results in a system architecture that is easier to design, implement, and maintain.

In this work, we adopted aspect-oriented methodology to an existing framework with the use of Spring AOP. Spring AOP is implemented in pure Java and it does not need to control the class loader hierarchy, so it is suitable for use in a J2EE web container or application server. However, there is one important limitation of Spring AOP that it currently supports only method execution join points (advising the execution of methods on Spring beans). Field interception is not implemented so we have only method execution join points in our work. This limitation is not important for us because the aim of Spring AOP is not to provide the most complete AOP implementation but rather to provide a close integration between AOP implementation and Spring IoC. By this way aspects are configured using normal bean definition syntax which eases our work because the framework that we work on uses Spring library. We used the schema support of Spring AOP and with this configuration an aspect is simply a regular Java object defined as a bean in the Spring application context. So, the pointcut and advice information is captured in the XML.

6.1 Validation Aspect

Web applications usually need to collect data from their users. This can be achieved by text fields, radio buttons, etc. The problem of this concern is that there can be many inappropriate inputs since what the user enters do not always make sense. For example, a phone number may be lacking a digit or may have letters inside. The reason of this situation can be because the input field is not clear or because the user is not paying attention. In any case, this results in getting nothing from the program which is disgusting for users. As a result, one needs to validate user input by some means.

In our case, we take input from users via free form text fields. An example input form is shown in Figure 7. For this situation we need to confirm that “To Number” and “From Number” fields are not empty. Furthermore, these two fields should have exactly eleven digits which represent the phone number. A similar validation should also be carried out for the email form.



Figure 7. Sms Form User Interface

Normally validation is implemented in every class and method which takes user input. For this reason scattering occurs in the code. So, we want to have validation rules be stored separately from markup or Java code so that they can be modified independent of any other source code. That is why we decided to implement an aspect for this concern.

Defining an aspect includes the following: defining a bean, a pointcut, a specific type of advice, and the advice method implementation. The bean names the aspect and gives the class path of advice method implementation. We have two pointcuts and two corresponding methods for the advices of these since we could not combine them for sms message and email message. The bean definition and the pointcut for our validation aspect are presented in Figure 8. The advice can be seen in Figure 9.

```
<bean id="validationAOP" class="org.borsoft.aspect.ValidationAspect" > </bean>

<aop:aspect ref="validationAOP">
  <aop:pointcut id="smsSavePointcut"
    expression="execution(* org.borsoft.web.SmsAction.save(..))" />
  <aop:pointcut id="emailSavePointcut"
    expression="execution(* org.borsoft.web.EmailAction.save(..))" />
  <aop:around pointcut-ref="smsSavePointcut"
    method="smsValidation" />
  <aop:around pointcut-ref="emailSavePointcut"
    method="emailValidation" />
</aop:aspect>
```

Figure 8. Spring AOP Declaration of Validation Aspect


```

public Object smsValidation(ProceedingJoinPoint pjp) throws Throwable
{
    SmsAction o = (SmsAction) pjp.getThis();
    SmsMessage sms = o.getSmsMessage();
    if ( sms.getFromNumber().length() == 0 || sms.getToNumber().length() == 0 )
    {
        ActionContext.getContext().getSession().put("message", getText("errors.required"));
        return INPUT;
    }
    Pattern p = Pattern.compile("\\d{11}");
    Matcher m = p.matcher(sms.getToNumber());
    boolean matchFound1 = m.matches();
    m = p.matcher(sms.getFromNumber());
    boolean matchFound2 = m.matches();
    if ( !matchFound1 || !matchFound2 )
    {
        List args = new ArrayList();
        args.add("smsMessage.toNumber");
        ActionContext.getContext().getSession().put("message", getText("errors.format"));
        return INPUT;
    }
    return pjp.proceed();
}

```

Figure 9. Advice Code for Validation Aspect

The pointcuts for sms and email message classes include the method executions of save methods. That is because input is taken from user only within the save operations. We have a separate pointcut for the two concepts because the input to be validated for each is different, so we take the objects in the advice and check the necessary fields according to these. For sms case, we check whether the phone number fields are empty and also whether they conform to the specified pattern which is basically having eleven digits and nothing else. For email case, we again check whether the two address fields are empty, and also whether they conform to the specified email pattern. We defined an around advice since if the user entered invalid input, the save operation should not be performed; instead the user should be warned and asked new input. So, if one of these is not satisfied then we return a value "INPUT" which means that the system needs input to continue this operation. As a result, the user sees the form page again.

This concern could also be implemented using the Validator property of Spring by the use of xml files. However, this has some disadvantages which are solved by the aspect implementation. First, in order to maintain validation rules one needs to synchronize the page markup, the ActionForm, and the Validator and Struts configuration files. Whereas, when we use an aspect, we only need to modify the ValidationAspect class that implements the advice method instead of Validator and Struts xml files. Secondly, the use of Validator results in lack of data conversions and transformations, but with an aspect we can achieve these functions.

6.2 Authentication Aspect

Authentication is any process which is used to verify that someone is who they claim they are. In our case we try to

identify the users by the use of username and password information. We constrain that a user cannot perform any of the system activities if he/she is not logged in to the system. For this reason, one needs to check whether the user have logged in before any type of operation.

We defined an authorization aspect for this purpose. The pointcut of this aspect includes the method executions of view operations, which are the list functions of our case. The pointcut definition can be seen in Figure 10. The reason of just including the list methods for the listing of sms and email messages is that, a user cannot do other operations before seeing these lists. So, it is sufficient to have these functions in our pointcut definition.

```

<aop:pointcut id="authenticationPointcut"
expression="execution(* org.borsoft.web.*Action.list(..))"/>
<aop:around pointcut-ref="authenticationPointcut"
method="authenticate" />

```

Figure 10. Declaring Authentication Pointcut

In the advice, we check whether the user's id is present in the current session. We just perform this check because we add the user's id to the session when he/she tries to login from the login page, if the username and password exist in the database of course. We again used the around advice type. The reason is that if the user is not authenticated, then the desired operation should not be performed, but the user should be asked to login. As a result, if the user tries to view information before logging in, the system forces authentication by redirecting the user to authentication page.

6.3 Authorization Aspect

Authorization in general sense is finding out if a user, once identified, is permitted to do something. In our case, authentication is the precondition of authorization, meaning that if a user is not authenticated then there is no check for authorization. If a user is logged in, then he/she is able to view information in the system. After this point we need to check whether this user is authorized to perform specific operations. As stated before, there are three types of users in our system: admin, authorized user and regular user. Regular user cannot perform edit and delete operations on the system, so when a user wants to perform these operations we should check the user type parameter. This parameter is obtained from the session since once a user is logged in; the user type is saved in the session. The pointcut of this aspect includes all edit and delete function executions as seen in Figure 11.

```
<aop:aspect ref="authorizationAOP">
  <aop:pointcut id="authorizationPointcut"
    expression="execution(* org.borsoft.web.*Action.delete(..)
    || execution(* org.borsoft.web.*Action.edit(..))" />
  <aop:around pointcut-ref="authorizationPointcut"
    method="authorize" />
</aop:aspect>
```

Figure 11. Declaring Authorization Pointcut

We applied around advice type because if the user is not authorized to perform these operations the system should redirect user to another page instead of continuing with the method execution. In the advice part, we check the user type parameter from the session and either perform the asked

operation or redirect the user to another page according to this information.

6.4. Access Control Aspect

Access control is an important aspect of any system which is the act of ensuring that an authenticated user accesses only what they should and nothing more. Access control has many dimensions so we selected a way that is appropriate to the framework we are working on.

We constrain that only admin can see all saved messages and users in the system. So, every other authenticated user should see their own sms messages and email messages. For this reason, our pointcut includes method executions of getting sms and email messages as seen in Figure 12.

In the advice part, we had two options to implement one of which was retrieving all messages from the database and filtering them with an after advice. The second option was to write an around advice and retrieve only necessary information from the database. We selected the second option to implement since the size of all messages in the system can be very large and performing the retrieval of this data for each user would be a cumbersome activity. However our approach also has an improper part which is extending a class related to database connection for an aspect related class. In this class we check whether the current user is an admin, the execution continues if he/she is an admin. If the user is another type, then the database query to retrieve messages is generated according to the user id and so, only the messages belonging to this user are retrieved. The advice implementation is shown in Figure 13.

```
<aop:aspect ref="accessControlAOP">
  <aop:pointcut id="accessControlPointcut"
    expression="execution(* org.borsoft.dao.hibernate.*MessageDaoHibernate.get*Messages(..))"/>
  <aop:around pointcut-ref="accessControlPointcut"
    method="controlAccess" />
</aop:aspect>
```

Figure 12. Declaring Access Control Pointcut

```

public class AccessControlAspect extends HibernateDaoSupport{
    public Object controlAccess(ProceedingJoinPoint pjp) throws Throwable
    {
        if(ActionContext.getContext().getSession().get("admin_flag").equals(new Integer(1)))
        {
            // The user is administrator, show all
            return pjp.proceed();
        }
        else
        {
            if( pjp.getThis().toString().startsWith("org.borsoft.dao.hibernate.Email"))
                return getHibernateTemplate().find("from EmailMessage where user_id=?",
                    ActionContext.getContext().getSession().get("userId").toString() );
            else
                return getHibernateTemplate().find("from SmsMessage where user_id=?",
                    ActionContext.getContext().getSession().get("userId").toString() );
        }
    }
}

```

Figure 13. Advice Code for Access Control Aspect

5. JBoss: Alternative Approach

As a different implementation technique for applying aspect-oriented programming to our case, we consider JBoss AOP. JBoss AOP is a 100% pure java aspect oriented framework where aspects and advice code blocks are written in regular Java. However, pointcut definitions are put into an xml file. In addition to these, the weaving is dynamic and performed at run time. With all the above specifications, JBoss AOP is really similar to Spring AOP. However, when we consider the functionality deeply, we see that JBoss AOP more capable than Spring AOP in some concepts. For example, with Spring AOP we were unable to define field pointcuts, thus we used method execution pointcut definitions in all aspects. But with JBoss AOP one can define method execution and constructor execution, field (get, set), method call, and all types of pointcuts. All captures the other types together.

If we write our validation pointcut and advice using JBoss AOP, we would get the code in Figure 14 for pointcut and aspect definition. The pointcut definition is really similar but there is one concern. It is that, an interceptor class which defines the advice specifications must implement the “org.jboss.aop.advice.Interceptor” interface and this interface defines two methods which are “getName” and “invoke”. By this way we do not give explicit method names in the aspect definition, so can have only one advice in a class. This changes our Spring AOP implementation since we should have two advice classes for the sms and email message classes’ save method pointcuts.

```

<aop>
  <bind pointcut="execution (public * org.borsoft.web.SmsAction->save (...))">
    <interceptor class="org.borsoft.aspect.ValidationAspect"/>
  </bind>
  <bind pointcut="execution (public * org.borsoft.web.EmailAction->save (...))">
    <interceptor class="org.borsoft.aspect.ValidationAspect2"/>
  </bind>
</aop>

```

Figure 14. JBoss implementation example

6. Related Work

Aspect-oriented programming is a new paradigm that aims to ensure separation of concerns. In the context of security, this means to isolate security implementation from the main program’s business logic. For this reason, security is regularly claimed to be a typical example of a crosscutting concern that can be handled with AOP/AOSD techniques and research on this issue have been carried out. At this point, we will talk about two different related work concepts, where one is solving security problems with an object-oriented approach and the other is using AOSD.

Research on object-oriented solutions for security problems has been carried out approximately ten years before now and they are mostly related to security of database systems. Two such research outcomes are presented in [13], [14]. Although object-oriented paradigm is good at separating concerns by mapping them to objects, it is not that strong in representing abstract notions [15]. Security is one such concept and it is really difficult to model security with object-oriented paradigm. Since the security concern is spread throughout the

code, if a critical check is forgotten in the code it would be a huge problem [15].

For these reasons researchers have started to use AOP to solve this problem. One related work is presented in [16]. In this work, authors applied AOSD techniques to a specific security-related real-life case study. Their case consisted of modularizing the access control and auditing concerns of an existing FTP server implementation [16]. They achieved modularity with a complete separation of the security features they have selected. Moreover they state that modularization of crosscutting concerns improves the comprehensibility and analyzability of security features [16]. This work is really similar to our case since they have also applied AOP concepts to an existing real-world application. Since the existing application was not developed considering this, applying AOP was not easy for both. Moreover, the application of AOP is restricted to security concern. The related work has considered only two security conditions, whereas we defined and implemented three aspects related to security, and also closely related to each other. In addition, we have implemented the aspects in Spring AOP but they have used AspectJ.

7. Discussion

An aspect-oriented approach helped us to resolve the crosscutting concerns in the framework. We solve Authentication, Authorization, Access Control and Validation concerns in separate units independent of the core modules of the framework. This will make development of applications on this framework easier. For example, if we want to add a new user type to the sample application, we only will need to modify `AccessControlAspect` and `AuthorizationAspect` correspondingly. Or if we need to validate another input field we will only need to add a new policy to `ValidationAspect` class. This will reduce the application development effort on the framework dramatically.

However, there are still some problems that we cannot solve with aspect-oriented structure without code scattering. When we consider authorization scenario in the sample application, some users are authorized to edit and delete and some users are not. For the users that are not authorized to delete any message, we want to make delete button invisible. For this purpose, we need to check the type of user in user interface creation stage. We could not find an aspect oriented solution to this problem. The first solution we considered was to write if-else statements into .jsp files in order to determine create or not to create a UI element. However, this solution is not even close to be aspect-oriented because it results in both scattering and tangling. Another solution might be catching the user interface generating code by a pointcut and decide whether to display the delete button or not there. However, we could not find a way to implement such as solution in Spring AOP. So, we are not able to implement our concerns for user interface creation stage in an aspect-oriented way.

Another point we need to discuss here is some limitations of Spring AOP. It is quite easy to implement aspects with Spring AOP on a Spring framework. The aspects, pointcuts and advice references are declared in bean configuration files and advices are written in Java classes. However, when compared to AspectJ, expressiveness of Spring AOP is low. Spring AOP only supports method execution join points for Spring beans.

There are some things that cannot be done easily or efficiently with Spring AOP, such as advising very fine-grained objects (e.g. domain objects). AspectJ is a better choice in such cases [17]. However, our experience is that Spring AOP provides easy solution to most problems in J2EE applications.

Spring AOP uses a proxying mechanism to create the proxy for a given target object [18]. When a pointcut is defined for a method execution, both the method in the original object and the method in the proxy object are caught and the advice is executed twice. This situation did not cause a problem for none of our aspects. However, this point should be kept in mind because it may cause important problems for especially transactional advices.

8. Conclusion

Throughout the paper, we described our work for extending a J2EE framework BORFWCORE with Aspect-Oriented programming for security and validation concerns. BORFWCORE is a Spring framework for developing mobile applications on it. Despite the fact that the framework itself helps to reduce the application development effort, there are some problems that it cannot solve effectively. The developers in Bor Yazilim stated that it requires much effort to implement especially security and validation concerns without an aspect-oriented approach. So we provide an aspect-oriented design and implementation of those concerns in the paper.

The aspects we defined are Authentication, Authorization, Access Control and Validation. We do the aspect-oriented implementation with Spring-AOP. We observed that with the aspect-oriented implementation, the application development effort related to our aspect-oriented concerns has decreased. Before the aspect-oriented implementation, there was no implementation related to these concerns in the framework core. When access control is considered, for each application different access control policies would be required. Even the user types would differ from application to application. That is why; those concerns had to be implemented while developing each application. For these concerns, we could not exploit the reusable nature of the framework. After the aspect-oriented implementation both defining new user types and modifying the policies is quite easy because we just need to modify the advice in `AccessControlAspect` class.

Our experiences show that aspect-oriented implementation of these concerns in BORFWCORE makes valuable


contribution to the framework both eliminating code scattering and tangling and increasing the degree of reusability of the framework.

9. Acknowledgements

We would like to thank to Ozer Aydemir in Bor Yazilim for providing BORFWCORE to us and explaining the problems they face without an aspect-oriented implementation. We also thank to Asst. Prof. Dr. Bedir Tekinerdogan for consulting us during our project and organizing the 4th Turkish Aspect Oriented Software Development Workshop.

10. References

- [1] V. O. Safanov, *Using Aspect-Oriented Programming for Trustworthy Software Development*, 2008.
- [2] T. Cottenier, *Model-Driven Software Development in a Large Industrial Context*, 2007.
- [3] Security Concern
<http://www.rogerclarke.com/EC/IntroSecy.html>
- [4] J. Yoder and J. Barcalow, *Architectural Patterns for Enabling Application Security*. In *Pattern Languages of Program Design - 4*. Addison Wesley, 1999.
- [5] J. H. Hayes and A. J. Offutt, *Increased software reliability through input validation analysis and testing*. In *Proceedings of The Tenth IEEE International Symposium on Software Reliability Engineering*, pp. 199-209, 1999.
- [6] Verkata
Available: <http://www.verkata.com>
- [7] Apache Struts
Available: <http://struts.apache.org>
- [8] Spring Framework
Available: <http://www.springsource.org>
- [9] JavaServer Faces
Available: <http://java.sun.com/javaee/jaserverfaces/>
- [10] Hibernate
Available: <http://www.hibernate.org/>
- [11] E. Gamma, R. Helm, R. Johnson and J. M. Vlissides *Design Patterns: Elements of Reusable Object-Oriented Software*, First Edition: November 1994.
- [12] R. Laddad, *AspectJ In Action: Practical Aspect Oriented Programming*, 2003.
- [13] J. Millen and T. Lunt, *Security for Object-Oriented Database Systems*. In *Proceedings of the IEEE Symposium on Security and Privacy*. Oakland (Ca), USA, 260-272, 1992.
- [14] B. Thuraisingham, *Mandatory Security in Object-Oriented Database Systems*, *Proc. Int'l Conf. Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, 1989.
- [15] J. Viega, J. T. Bloch, and P. Chandra, *Applying aspect-oriented programming to security*. *Cutter IT Journal*, 14, 2, 31-39, 2001.
- [16] B. De Win, W. Joosen, and F. Piessens, *AOSD & Security: a practical assessment*, *Workshop on Software engineering Properties of Languages for Aspect Technologies (SPLAT03)*, pp. 1-6, 2003.
- [17] Aspect Oriented Programming with Spring
Available:
<http://static.springsource.org/spring/docs/2.0.x/reference/aop.html>
- [18] Spring AOP Proxying Mechanisms
Available:
<http://static.springsource.org/spring/docs/3.0.x/spring-framework-reference/html/ch07s06.html>



Home Sms Email Mobile

Verkata Core Spring Framework

User Name:

Password :

Login

Integrated Packages

- Spring 2.0
- Hibernate 3.0
- Sitemesh
- Velocity Templaing Engine
- DWR (Direct Web Remoting)
- AJAX (Prototype/Scriptaculous)

[About](#) [Terms and Policy](#) [Privacy](#) [Help](#) Copyright 2007 © Verkata

Figure 1. Home Page

In the Home page user should login to system in order to use system functionality.



Home Sms Email Mobile

Add Sms

To Name:

To Number*:

From Name:

From Number*:

Subject:


Message:

Save Cancel

[About](#) [Terms and Policy](#) [Privacy](#) [Help](#) Copyright 2007 © Verkata

Figure 2. Sms Form

User should fill required fields in order to save SMS messages which are sent by the system automatically.



Home Sms Email Mobile

Add Sms

One item found.

Sms Message Id	To Name	To Number	From Name	From Number	Subject	Message
1	Basak	15054444444	Elif	15053333333	Hello	See you next time

[About](#) [Terms and Policy](#) [Privacy](#) [Help](#) Copyright 2007 © Verkata

Figure 3. Advice Code for Access Control Aspect

This page shows SMS messages, which are sent or waiting to be sent, of authenticated user.

Aspect-Oriented Extension of Vector Image Drawing Tool: Joodle

Kemal Eroğlu, Aytuğ Murat Aydın, Mehmet Ali Abbasoğlu

*Department of Computer Engineering, Bilkent University
06580, Bilkent, Ankara*

{k_eroglu, a_aydin, abbasoglu}@ug.bilkent.edu.tr

Abstract

Different kinds of drawing tools have been developed over the years by taking different design issues and development ways into consideration. Although lots of concerns are handled by Object Oriented Software Programming, crosscutting concerns still decrease the cohesion and increase the coupling between software components. In this paper, we will discuss the extension process of a Vector Image Drawing tool “Joodle” in a modular way by using Aspect Oriented Software Development. In the end we will remark the benefits of Aspect Oriented Software Development and the lessons learned.

Keywords

AOOSD, AspectJ, SVG, Vector Image, OO Patterns

1. Introduction

Vector graphics are aimed for the use of geometrical primitives such as points, lines, curves, and shapes or polygons, and those kinds of geometrical shapes are all based on mathematical equations so as to represent images in the computer graphics. [1] The main purpose of vector graphics is excluding unnecessary details. Because of that they are useful in areas of text, multimedia, 3D

rendering and many other information and art areas.

Vector images decrease the minimal amount of information stored in a much smaller file size compared to large bitmap images. Correspondingly, with indefinitely zoom, vector image can remain smooth. In a different manner, parameters of objects are stored and can be modified later, as consequence moving, scaling, rotating, filling does not reduce the quality of the drawing. From 3D perspective of vector images, rendering shadows is also much more realistic, as shadows can be abstracted into the rays of the light form which they are formed.

Joodle as a drawing tool is implemented by using Object Oriented Programming. It has a modular design. However Joodle might need some extensions such as adding new shape types as add-ons, undo/redo operations, warning handling and modularizing the existing Observer Pattern. We focus in particular on those concerns. However, it appears that these concerns cannot be easily modularized and tend to crosscut over multiple classes. This reduces maintainability, reuse, etc. To cope with these problems we provide an Aspect Oriented approach in which we implement the crosscutting concerns as aspects.

The paper is organized as follows: Section 2 describes the background work to

understand the paper and Joodle, the application for application of AOP in detail. Section 3 describes Object Oriented Design of the system. Section 4 shows problem statement and concerns. Section 5 explains aspect-oriented solution with AspectJ as a language for AOP and Section 6 briefly indicates the related work in the literature. Finally, Section 7 terminates by discussing the effects of AOP on the case study.

2 Joodle

The motivation behind vector images is that vector images are necessarily descriptions of shapes to be drawn and not the pixel information per se. As a result of this kind of description style, vector images are impressive alternatives to other common image formats. SVG format which is advocated by the World Wide Web Consortium (W3C) is defined in XML plain-text files. SVG stands for scalable vector graphics. [2]

In Joodle application, user can draw vector images with using defined tools, and save with the SVG format.

In this part, we will introduce Joodle which is a drawing tool which uses vector graphics. This tool is implemented by using Java and the functionalities of Object Oriented Design. As an illustration, figure 1 shows a running screenshot of Joodle.

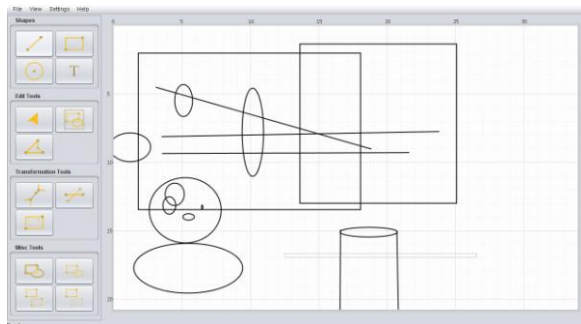


Figure 1- A snapshot from Joodle

Joodle is a drawing tool that aims to create and edit vector graphics. Users can do various operations by using Joodle such as creating vector graphics images and manipulating, coloring and merging existing vector graphics images. Joodle uses SVG (XML) format instead of other graphical image formats.

Users can draw many shapes by using this program. The drawable shapes include rectangular, squares, ellipses, circles and many others by using its Add-on option.

Users can group existing shapes by using grouping function of Joodle. In addition to that, another functionality of Joodle is editing drawn shapes by rotating, moving with respect to an angle. Besides, user can change the attributes of the shapes by making amendments on line type, color preferences. This gives the users opportunity of making alterations on the shapes.

Users are allowed to save their work on Joodle. Drawings are saved as SVG files which preserves the data description in a XML file. Besides, user can also load their saved works and make new operations on those drawings.

3. Object Oriented Design

This implementation of the system is a good example of Object Oriented Software Development. The design issues of Agile Software development are strictly obeyed and Design Patterns are applied to this design.

There are four packages in the proposed OO design: domain, si, ui and tools as indicated in Figure 2.

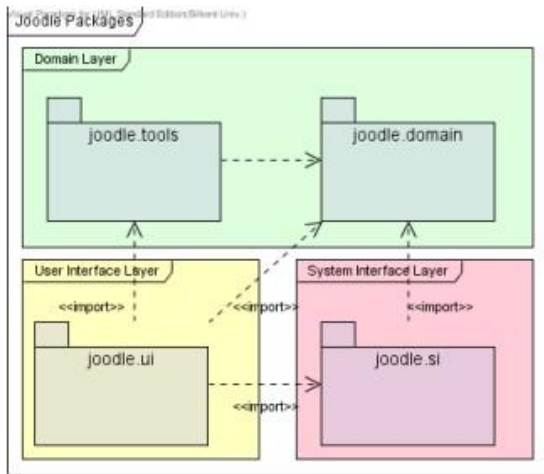


Figure 2-Packages of Joodle Implementation

Each package consists of classes which have different types of functionalities. Domain package includes the classes that are responsible for shapes. Basic shape classes are held by this package.

Si package is responsible for handling loading and saving functionalities as well as add-on operations. In addition to that, this package also deals with resource managing. In brief, classes of Si package deals with system operations.

Tool package has the classes that are used for creating new shapes, editing, rotating the drawings. Tool package actually works as a backend for UI classes.

Classes of the UI package are mainly focused on User Interface of the Joodle. UI package has also the functionality of User Interface operations. Frontend operations are done via those classes in UI package.

The main operations of the Joodle are held by Domain Package. Hence in the following, Domain package will be investigated in a deep manner.

Design patterns are one of the most crucial solutions that are used in OO designs. Since they have such importance, in Joodle, they are used in several places.

In Joodle, Singleton Pattern, Observer Pattern and Composite Pattern are used to enhance the OO design of the program.

3.1. Observer Pattern

3.1.1 Motivation

Observer Pattern is intended to solve the problem of maintaining consistency among several classes that are part of a model data. However Observer Pattern of OO design arises some problems. The main problem lies behind it is that you should modify the classes that participate in this pattern.

3.1.2 Description

Observer Pattern is a software design solution where an object enlists its dependents which are called observers and it notifies them in any state changes by calling one of their methods. [3]

In Joodle, in the domain package we have Canvas class and Shape and SelectiveObserver interfaces as seen in Figure 3. In case selection state of a shape changes, Canvas notifies the observers.

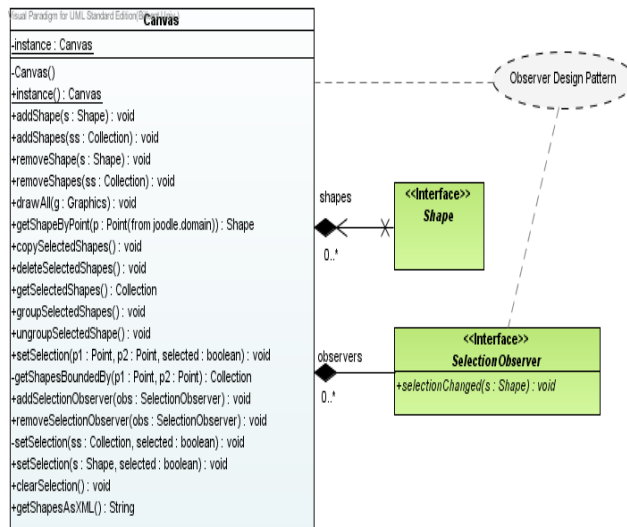


Figure 3- Showing Observer Pattern in Joodle

Canvas Class uses three functions for defining observers and notifying them.

- AddSelectionObserver: It is for defining new observers.
- RemoveSelectionObserver: It is for extracting an observer from the set of observers.
- FireSelectionChanged: Observers are notified when this function is called since it makes alterations on selection.

3.2. Composite Pattern

3.2.1 Motivation:

Like in any other graphics application, in Joodle, users can build complex diagrams using simple component types. User can also use the group of components to create larger components. However problems may occur during making distinction between container and primitive shapes. Composite Pattern helps the user do not make this distinction. In other words, user does not bother to make distinction of CompositeShape or other shape types.

3.2.2 Description

Composite Pattern is a structural software solution which works by composing objects into tree structures to represent part-whole hierarchies. Composite pattern allows clients to treat individual objects and compositions of objects uniformly.[3]

CompositeShape, Shape and AbstractShape classes are the parts of the Composite Pattern. As seen in the Figure 4, interface Shape is implemented by the classes AbstractShape and CompositeShape Classes.

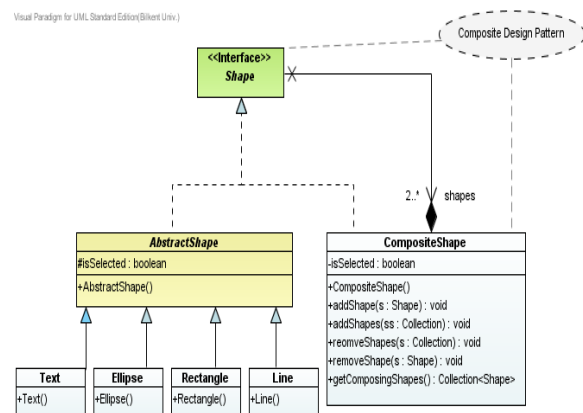


Figure 4- Composite Pattern in Joodle

3.3. Abstract Factory Pattern

3.3.1 Motivation

In the application, there are a lot of shapes which contains information. According to which tool uses this shape, it can be shown on the application differently, while it still have to contain the same information. Abstract Factory Pattern helps us create a composite class which must contain the same members but created differently depending on the application.

3.3.2 Description

Abstract Factory Pattern is a creational design pattern that provides an interface to create families of related or dependent objects without specifying their concrete classes. [3]

The structure of the pattern is shown in the figure 5. Shape and its subclasses with ShapeFactory and its subclasses reify Abstract Factory Design Pattern.

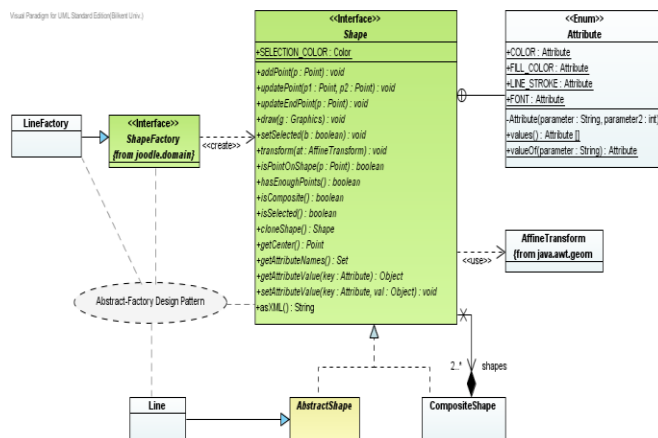


Figure 5- Abstract Factory Pattern

4. Problem Statement

In existing design, identification of existing concerns has not been done and we needed to find out crosscutting concerns to provide a high cohesion among components. In addition to that, having low coupling and increasing modularity are other aims that we take into consideration. Besides, existing patterns are affecting the system modularity and therefore they crosscut the basic structure of some classes. They add behaviors and modify roles of the classes.

Although this can be helpful for the system, it can make the system hard to implement and it may add more complexity to the code. Therefore, Aspect Oriented Software Development would be beneficial for handling separating the concerns.

5.1 Identifying Crosscutting Concerns

We have identified the following crosscutting concerns:

- Observer Concern
- Undo/Redo Concern
- Plug-in Concern
- Warning Concern

4.1.1 Observer Concern

In the OO implementation of Joodle, Observer Pattern is handled in a non-modularized way. The main problem is that the classes of observer pattern include the methods related to the pattern and this may obstacle future modifications. When the user needs to make alterations to the classes of the Observer Pattern, the structure of the pattern is needed to taken into consideration.

Another problem arises when the user wants to apply the same pattern to other classes. User needs to implement all the related work from the beginning to achieve this aim.

4.1.2 Undo/Redo Concern

Undo & Redo Operations are not implemented by Joodle. However they have significance in drawing tools, since user may need to make several changes and may want to return one of the previous drawings.

This problem is supposed to be supported by all of the drawing operations in the application. When this problem is not handled as an aspect for AOP, each class that is affected by any of the amendments in the system would be altered to add required operations.

Another problems occurs, when new plugins are added to the application. When the user wants to make an undo/redo operation, required alterations should also be made.

However, handling this problem as a subject to AOP, an outside Aspect class will handle this operation. The code required to add undo/redo operations will be separated from application classes. Thus implementing those operations to the new classes will be easy.

4.1.3 Add-on Concern

In the OO implementation of Joodle, allowing the user adding new shape types is not implemented. However, it is essentially a necessary function, because existing shape types are not sufficient for complex drawings. Adding new shape types as add-on to the system causes implementing a new class and editing existing classes of Joodle, if OO is used. This eventually causes scattering modules and tangling between classes which is not wanted for a neat implementation.

4.1.4 Warning Concern

Some warning concerns cause some significant deficiencies in the code. There are some error handling and sequence issues in the Joodle which resulted in scattered and

tangling code. There are some classes which are dependent to each other because of handling these functionalities. Besides when user does unintended operations such as closing the program accidentally, the application does not give any control message in order to handle the errors. In the same manner, when user does a sequence of operations, application does not prompt saving previous operations. For example, when user wants to load an image, the application does not prompt saving previous work.

5. Aspect Oriented Programming

In the previous section, problem statement and concerns are introduced. As explained above, since each concern scatters multiple classes, this will enhance dependency and it will result in low cohesion. Since this situation is an unwanted occurrence, some improvements are needed. At this point, Aspect Oriented Programming is a good way to deal with such problems. As AOP language, we used AspectJ for several reasons. The main motivation behind this choice is its popularity and simplicity.

In this section, we explain how we solved the problems defined in the previous section by using AspectJ and we provide AspectJ code samples as an illustration.

5.1 Observer Aspect

The fundamental goal of Observer Aspect is to modularize the Observer Pattern so as to avoid the problems may occur because of the scattering classes. [4]

In the Joodle, Canvas class is the subject of the Observer Pattern. The observers are notified when a change occurs in Canvas class. Canvas class can add new observers or delete any of the observers from the observers list.

In our redesign of Joodle, we created an abstract Aspect named SelectionObserverProtocol and an Aspect named SelectionObserverProtocolImp.

```
package Aspect;
import java.util.ArrayList;

abstract aspect SelectionObserverProtocol
{
    abstract pointcut selectionChanged( Shape s);

    after(Shape s) : selectionChanged(s) {

    }

    public Collection<SelectionObserver> Canvas.observers
        = new ArrayList<SelectionObserver>();

    * Adds an observer to the set of observers
    public void Canvas.addSelectionObserver()

    * Deletes an observer from the set of observers
    public void Canvas.removeSelectionObserver()

    * Returns observer list.
    public Collection<SelectionObserver> Canvas.getObservers() {}

    * If the given shape object has changed
    public void Canvas.fireSelectionChanged(Shape s) {}
}
```

Figure 6-AspectJ Code for Observer Concern-1

SelectionObserverProtocol aspect has several functionalities. The core function of this aspect is notifying, adding and deleting observers. After advise calls the fireSelectionChange method of the Canvas, if pointcut selectionChanged is called.

```
package Aspect;
import joodle.domain.Shape;

public aspect SelectionObserverProtocolImp
    extends SelectionObserverProtocol
{
    pointcut selectionChanged( Shape s):
        withincode(* Canvas.setSelection(..)) &&
        call(* setSelected(..)) &&
        target(s);
}
```

Figure 7-AspectJ Code for Observer Concern-2

As indicated in Figure 7, SelectionProtocolimp extends abstract SelectionObserverProtocol. This aspect actually defines our pointcuts.

5.2 Undo/Redo Aspect

As explained in the problem definition section, Undo/Redo concerns are one of the aspects that are not held in the original application. Since it has crucial functions in an application, in Joodle they needed to be held.

Each needs to implement some functions to handle undo/redo concerns. Each class that can get influenced by Undo/redo operations should be modified to handle these operations.

However, as explained in the problem definition section, such a solution may cause some deficiencies in the system. Therefore AOSD would be very beneficial to avoid such problems. In our design, we implemented an aspect named UndoAspect. Aspect UndoAspect is the aspect class that handles those Undo/Redo Concerns. UndoAspect aspect adds a pointcut in various methods of classes.

```

18 public aspect UndoAspect
19 {
20
21     public ArrayList<String> undoList = new ArrayList<String>();
22     int index = 0;
23     boolean undo = false;
24
25
26     // SAVE PROCESS AFTER AN OPERATION
27     @pointcut canvasChanged():
28         (withincode(* TransformationTool.handleRelease(..)) && call(* Shape.transform(..))) ||
29         (call(* Canvas.copySelectedShapes())) ||
30         (call(* Canvas.deleteSelectedShapes())) ||
31         (withincode(* Canvas.groupSelectedShapes(..)) && call(* Canvas.addShape(..))) ||
32         (call(* Canvas.ungroupSelectedShape())) ||
33         (withincode(* CreatorTool.handlePress(..)) && call(* Canvas.addShape(..))) ||
34         (withincode(* PointEditTool.handleRelease(..)) && call(* foo(..))) ||
35         (withincode(* PointEditTool.handleKeyPress(..)) && call(* foo(..)));

```

Figure 8-Code Fragment of Undo/Redo Aspect

In the advice part of the aspect, all the actions are added to an `ArrayList` as shown in Figure 9. We track this list according to the undo/redo operations. In the `ArrayList` we store all the actions done by the user. However if the user starts a work in a new file, all the data stored in `ArrayList` is reset.

```

37 @after() : canvasChanged()
38 {
39     if(index==0 )
40     {
41         String empty="<?xml version='1.0' standalone='yes'>" +
42             "<svg width='100%' height='100%' version='1.1' xmlns='http://www.w3"
43             +"></svg>";
44         undoList.add(0,empty);
45         index++;
46     }
47
48     undoList.add(index, Canvas.instance().asXML());
49     index++;
50
51     if(undo)
52     {
53         for (int j = index ; j < undoList.size(); j++)
54         {
55             undoList.remove(j);
56         }
57     }

```

Figure 9-Code Fragment of Undo/Redo Aspect

5.3 Add-on Aspect

Joodle has a set of pre-defined vector-graphics. However existing primitive shapes sometimes can be insufficient for the users. At this point, we wanted to add this functionality to the system by using AOP, because by that we will not deal with scattered classes and any deficiencies caused by crosscutting concerns can be avoided. In the end, we intend to achieve the goal of high cohesion and low coupling.

In Plugin Aspect, we have `AddonAspect`, `ShapeToolBar` and `ToolsPanel` classes as participants. In order to allow user to add new shapes to the application as plugin, we implemented `AddonAspect` class, as indicated in Figure 10.

```

package Aspect;
import java.io.File;

public aspect AddonAspect
{
    @pointcut addonFound():

    after() returning(ArrayList<ShapeFactory> sf):addonFound()

    * By defaults add-ons are in the current directory
    public static String addonsDir = ".";

    * Sets the add-ons' directory to the given new directory
    public void setAddonsDir(String dirName) {}

    * Tries to instantiate a Shape from the class
    private static Shape makeInstance(URL u, String cn) {}

    public static ArrayList<ShapeFactory> findAddons()
}

```

Figure 10-AspectJ Code for AddonAspect

5.4 Warning Aspect

As explained in the problem definition section, warning concerns results deficiencies in the code. Some of the warning concerns are not held, and the implemented ones cause scattering classes because of dependence on each other.

In order to solve these problems, by using AOP, we implemented an aspect named Warning Aspect. This aspect retrieves the required joinpoints by using pointcuts and handles those warnings via advises.

```
public aspect WarningAspect
{
    // EXIT IMAGE WARNING
    pointcut exitImage():[]

    before(): exitImage()[]

    // NEW IMAGE WARNING
    pointcut newImage():[]

    before(): newImage()[]

    // LOAD IMAGE WARNING
    pointcut loadImage():[]

    before(): loadImage()[]

    // LOAD ERROR HANDLING
    pointcut loadError():[]

    before(): loadError()[]

    // SAVE ERROR HANDLING
    pointcut saveError():[]

    before(): saveError()[]
}
```

Figure 11- WarningAspect

Using this aspect, we prevent exiting the application without prompting verification for saving the work. Therefore lots of unwanted problems will be avoided.

In addition to that, the error can occur during the operations of “Load and Save file” is handled by using this aspect.

6. Related Work

There are many different commercial and non-commercial graph visualization tools. Some of them generally the ones with better capabilities and user interfaces are commercial. On the other hand, there are some non-commercial and good ones. Drawplus SE[6], InkScape[7], Turtle are some of the leading ones.[8] As a commercial sample Adobe Illustrator[9] and MS Expression[10] are the most famous ones.

However none of these tools used AOP techniques as far as we know. Joodle is different than these tools since AOP approach is applied in the development cycle of Joodle.

7. Discussion

When we are doing Aspect Oriented refactoring of Joodle, we gained experience of benefits of Aspect Oriented Software Engineering. However, we had difficulty in defining aspects on vector image drawing. For example, there is no example of undo/redo aspect or addon aspect in literature. It was actually a new approach in the AOSD and therefore there is a lack of study on this matter.

However, during aspectizing observer pattern in the original implementation, we could easily find documents and related work about it. Modularizing Observer Pattern is a popular subject on AOSD. There are motivating and promising studies in this area.

We used AspectJ in our implementation for several reasons. For example, JBoss lets the user to add/remove advices dynamically in the execution time. However, in our concerns we do not such an attribute. Besides AspectJ and Eclipse integration supplied an ease of use of the development environment.

8. Conclusion

In this paper, we discussed the benefits of the AOP with respect to using an example project. The main aim was to enhance modularity and as such existing OOP design is not sufficient for modularizing some abstraction mechanism as well as dealing with crosscutting concerns.

Firstly we identified the crosscutting concerns of the application Joodle which are Observer concern, Undo/Redo concern, Add-on concern and Warning concern. Add-on concern and Undo/Redo concern are mostly not handled in many of the drawing tools. Warning concern and Observer concern are actually two significant concerns that are very frequently encountered in drawing tool applications. And these concerns result in tangling and scattering which is not wanted in good implementation.

Further, we have indicated that all these concerns can be handled by using aspects to reduce problems from crosscutting concerns. In the Observer Aspect, we modularized the Observer pattern to let the developer apply this pattern to other classes without requiring altering their classes. Besides future development on the classes of the pattern require some amendments for sustaining the

Observer Pattern. However with the help of AOP, we made the whole process modularized by handling scattered concerns.

In the Undo/Redo Aspect, we added a new functionality to the application by allowing the user making Undo/Redo operations. Instead of scattering the Undo/Redo operation to the classes, we developed an aspect which handles the operations in a more cohesive way.

In the add-on aspect, a new functionality added to the system which enables the user adds new shape types to the system. This process is also handled by AOP to handle scattering classes. In the same manner, our Warning aspect modularizes the several error handling operation warning processes.

In conclusion it can be deduced that AOP is a good approach when there are lots of scattering classes that cause tangling and dependency on the classes.

9. Acknowledgements

We want to thank Asistant Prof. Bedir Tekinerdogan for his sincere support and advices as well as his contributions to this workshop.

10. References

- [1] Vector Graphics, Retrieval Date: 14.12.2009, http://en.wikipedia.org/wiki/Vector_graphics
- [2] Scalable Vector Graphics, Retrieval Date: 14.12.2009 <http://www.w3.org/Graphics/SVG/>

[3] E. Gamma, R. Helm, R. Johnson and J. Vlissides. Design Patterns Elements of Reusable Object Oriented Software. Addison-Wesley, 2008

[4] Jacob Borella, The Observer Pattern Using Aspect Oriented Programming, IT University of Copenhagen, 2003

[5] Ivan Kiselev, Aspect Oriented Programming with AspectJ, Sams Publishing, 2003

[6] The AspectJ(TM) Programming Guide.
Retrieval Date: 14.12.2009
<http://www.eclipse.org/aspectj/doc/released/progguide/>

Comparative Analysis Of Strategies For Applying AOP On Legacy Code

Duygu Sarıkaya, Can Ufuk Hantaş, Dilek Demirbaş^{#1}

[#]Bilkent University, Computer Engineering Msc.

dsarikay@bilkent.edu.tr, hantas@bilkent.edu.tr, ddemirba@bilkent.edu.tr

Abstract

One of the basic issues in Aspect Oriented Programming (AOP) is cross cutting concern and likewise several strategies have been proposed in this context. However, it is not possible to find a silver bullet for solving cross cutting concerns because of the ill designed code. Therefore, we applied three different strategies to find the optimum procedure which are object oriented re-factorization, aspect oriented programming and combination of these two strategies.

While applying these strategies to a legacy code, we handle both existing concerns and new concerns that we applied to the case study. We use “EbookManager” application as a case study to test our approach.

1. Introduction

Object-Oriented Programming(OOP) has become the mainstream programming paradigm since it is able to define the real world problems in small units called objects. It is highly preferred since it encourages modularity; thus software re-use. Especially by using suitable patterns, the modularity is highly improved. However in practice we see that OOP techniques are not always sufficient to clearly capture all the important design decisions the program must implement. This forces the implementation of those design decisions to be scattered throughout the code, resulting in “tangled” and “scattered” code that is excessively difficult to develop

and maintain[1]. At this point Aspect Oriented Programming (AOP) addresses these decisions as aspects to solve these problems that could not be solved by using OOP. Aspect Oriented approach proves to be successful in this manner, and is efficiently used complementary

where the OOP techniques fall behind while dealing with crosscutting concerns.

The real world programmers have to deal with large amounts of legacy code everyday. In many existing applications today, we still observe that the OOP techniques or the patterns are not effectively used. In these cases, although it is possible to fix some of the problems we face with by refactoring the code considering OOP techniques, the time and effort spent is usually even more than starting the same application from scratch. Comprehending the legacy code with poor design, implementation and documentation could be an obstacle to fix even simple problems in the limited time. Providing good quality of the code and design in software applications is by all means crucial in order to be successful, however especially in the case of working on the legacy code, we face the real life constraints “scope”, “cost” and “time” so we need to make a trade-off and decide when to refactor the legacy code or when to use AOP, or to use both complementary to each other.

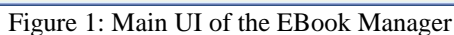
In our paper, we suggest that it is possible to define strategies for applying AOP on legacy code and using these strategies to work more efficiently by reducing the time, effort and cost spent. We discuss several strategies for applying AOP on legacy code. In particular, we discuss the strategies:

- a) Dealing crosscutting concerns by Object-Oriented refactoring without AOP
- b) Direct application of aspects to legacy code without refactoring
- c) Applying AOP after Object-Oriented refactoring

Our paper is organized as follows: In part 2 we give an overview of our study case, in part 3 we define our strategies considering both existing and new concerns for each, in part 4 we mention the related previous studies, in part 5 we discuss the lessons learnt from our study, in part 6 we conclude our study and we discuss the future works in part 7. The references are given in part 8.

2.1 E-Book Manager Application

Application basically requests a folder from user where ebooks are located and then lists them on the graphical user interface. From this interface users can view or update several properties of books such as title, author etc. A view from the application can be seen on figure **Error! Reference source not found..**



When user selects an ebook, application queries book information by using Amazon API 3.0 to bring book cover image and a short summary of the book to give the user a quick hint about the selected item.

2.2 Object Oriented Design

```

classDiagram
    class CheckOut {
        + CheckOut()
        + getNoOfSelectFiles()
        + setNoOfSelectFiles()
        + getOutFolder()
        + setOutFolder()
        - initComponents()
        - cmdBrowseActionPerformed()
        - cmdOKActionPerformed()
        - cmdCancelActionPerformed()
        + getSaved()
        + main()
    }
    class Utility {
        + main()
        + getRemotePageData()
        + getTaggedData()
        + getTaggedDataFromBegin()
        + replace()
        + getOccurrences()
        + removeAllTags()
        + getLocalPageData()
        + getXMLTagValue()
        + copyfile()
    }
    class Preferences {
        + Preferences()
        - initComponents()
        - cmdBrowseActionPerformed()
        - cmdCancelActionPerformed()
        - cmdSaveActionPerformed()
        + main()
    }
    class Main {
        + Main()
        + main()
    }
    class MainFrame {
    }
    class BookTableDataModel {
        + BookTableDataModel()
        + loadFilesIntoModel()
        - loadFiles()
        - breakFileName()
        + getValueAt()
        + setValueAt()
        + getColumnCount()
        + getRowCount()
    }
    class Ebook {
        - title
        - authors
        - extn
        - readingList
        - tags
    }
    class BookDetails {
    }

    MainFrame --|> Main
    MainFrame --> CheckOut : «Call, Import»
    MainFrame --> Utility : «Call, Import»
    MainFrame --> Preferences : «Call, Import»
    MainFrame --> BookTableDataModel : «Call, Import»
    MainFrame --> BookDetails : «Call, Import»
    BookTableDataModel --> Ebook : «Call, Instantiate, Import»
    Ebook --|> BookTableDataModel
    
```

Application stands on the Ebook class which is used to create ebook objects and contains fields to store details of the book. Utility class contains the methods to manipulate the XML data coming from web service.

MainFrame is the main user interface class of the application. It also scans the given folder including

subfolders for ebook files and lists them on a BookTableDataModel instance which extend JTable class.

BookDetails class is the details editing window GUI class and it also has functionality to search books with Amazon API to bring book details automatically.

CheckOut class is the GUI used to ask user for a destination folder where the selected files will be copied.

Preferences class is the user interface class which is used to set the applications properties.

2.3 Ill Defined Design

E-Book Manager has a really poor design. Functionality of the application is placed all around user interface classes. Several features are tangled in these classes. This makes understanding and maintaining code a really complex issue.

Main functionality of the application was placed in the MainFrame class which is actually built for the initial user interface of the application. It is so hard to trace the code to distinguish several different aspect from each other.

BookTableDataModel is used to show the book details in a spread sheet like way and stores some attributes of e-books. Even though there is an EBook class it does not contain all the related information. For example file type, name, path and extension information is not stored in EBook objects. This causes read and write operations on BookTableDataModel to be hard coded not by using methods of EBook objects.

Due to this ill design it is unfeasible to maintain the code. The Amazon API used in the application is deprecated since 2008 and it is no more working. To be able to update it developers should look around several classes to find web service related functions and make changes on all of them.

Also the detail storage and checkout functions are poorly thought and designed. They don't have separate classes developers can extend and add new behaviour and functionality. Details of the books are stored within the file names and it is not a possible to store data longer than 260 characters which can not be also changed easily.

Obviously the design is not modular, however it could easily be extended by applying Aspect Oriented Approach. In Section 3: "Strategies" we provide several strategies that could be used to enhance the legacy code with respect to cross-cutting concerns.

3. Strategies

3.1. Direct Application Of Aspects To Legacy Code Without Refactoring

3.1.1. Existing Concerns

3.1.1.1. Detail Storage

For detail storage function we wrote the Detail Storage aspect. This aspect changes the behavior of the operation completely. With this aspect book details are no more stored at the file name instead of that EBook class is declared to implement serializable interface and details are stored as separate files.

To be able to change the functionality we had to catch three operations with point cuts. These are done with saveDetails, loadDetails and copyFile advices.

```
public aspect DetailStorage {

    pointcut saveBook(Ebook book, int i, MainFrame mf)
    :call(* MainFrame.SaveBook(Ebook , int)) &&
    args(book,i) && target(mf);

    pointcut loadBook(String fileName,
    BookTableDataModel btdm) : call(*
    BookTableDataModel.breakFileName(String)) &&
    args(fileName) && target(btdm);

    pointcut copyFile(String src, String dst):call(*
    Utility.copyfile(..) && args(src, dst) &&
    this(FileCopyCheckOut);

    declare parents: Ebook implements Serializable;//use
    serializable interface to store book information

    //Advice for overriding method calls to breakFileName
    method to get boot information from file name and create
    book object
    Ebook around(String s, BookTableDataModel btdm) :
    loadBook(s, btdm){

        ...

    }

    //Advice for overriding method calls to saveBook function
    to store book information on separate file instead of
    changing filename.
    String around(Ebook book, int i,MainFrame mf) :
```

```

saveBook(book, i, mf) {
    ...
}

//Checkout function uses copyFile function in Utility
//Class. Override it to copy both book file and data file to
//destination
boolean around(String src, String dst):copyFile(src, dst){
    ...
}
}

```

Detail storage operation was originally done by changing the names of the e-book files. We wanted to store these details in separate files to be able to store more information in a way that is simple to implement and efficient to process. For this purpose we have decided on serialized objects stored on files.

To achieve this first we have used a pure aspect orientation. Our solution was straight forward to override detail storage related operations. For this purpose we have built an aspect that has 45 lines of code excluding imports and helper methods. Which has 3 point cuts and 3 advices.

3.1.1.2. Checkout

Checkout function in E-book Manager application is used to copy selected books to a destination folder. But the function did not care about the operating system on which the application runs and it was causing some exceptions to occur.

Apart from the existing checkout function we also wanted to add some extension to it which lets users to send selected books to a given email address as attachments. This new functionality is quiet similar to the existing one since they both have same parameters to execute. They both get a list of e-book objects and a destination (folder or email). Just the actual operation is required to be changed which is easily done with AspectJ.

```

public aspect CheckOutManager {

pointcut checkOut(MainFrame mf) : call(*
MainFrame.CheckoutFiles()) && target(mf);
pointcut checkOutFileMenuFunction(MainFrame mf):
call(* MainFrame.mnuCheckoutActionPerformed(*)) &&
target(mf);

pointcut menuCreation(MainFrame mf): call(*
MainFrame.initComponents()) && this(mf);

```

```

//override checkOut function so that it will work
//according to selected check out manager
void around(MainFrame mf) : checkOut(mf) {
    ...
}

//Create send-by-email menu
after(final MainFrame mf): menuCreation(mf){
    ...
}

//Set manager to fileCopyManager before menu action
//performed
before(MainFrame mf):checkOutFileMenuFunction(mf){
    ...
}
}

```

Our solution with pure aspect orientation was done by only 1 aspect which contains 52 lines of code excluding imports. With this aspect we caught the menu creation part of the application and added a new element which asks user to enter an email address and sends books to that address.

3.1.1.3. Web Service

To override the deprecated Amazon API 3.0 and override it with the new version we wrote an aspect.

We have caught the web service related parts of the code and returned the expected results by using the new Amazon API. To be able to make requests from the new web service we also needed to create a class which is used to make signed request. In prior versions of Amazon API every user registered to Amazon web services could make infinite number of requests but with the new one users should sign their requests with a cryptographic function.

3.1.2. New Concerns

3.1.2.1. Internationalization

Internationalization is the process of designing an application so that it can be adapted to various languages and regions without engineering changes. Sometimes the term internationalization is abbreviated as i18n, because there are 18 letters between the first "i" and the last "n." An internationalized program has the following characteristics:

- With the addition of localized data, the same executable can run worldwide.
- Textual elements, such as status messages and the GUI component labels, are not hardcoded in the program. Instead they are stored outside the source code and retrieved dynamically.
- Support for new languages does not require recompilation.
- Culturally-dependent data, such as dates and currencies, appear in formats that conform to the end user's region and language.
- It can be localized quickly.

In order to apply internationalization in java, you need .properties files each language that you want which contains the key and the value.

For example *Messages_en_US.properties* file Contains;

```
EditM=Edit...
ReadDefaultM=Read in Default Application
CheckoutM=Checkout...
...
```

these key and value information. Then you can increase the files according to abbreviation of you want like *Messages_tr_TR.properties*, *Messages_fr_FR.properties* and so on.

In our application, we utilized internationalization process for textual elements. In object oriented software development process, it can be achieved also but at least one line of code should be inserted after each component initialization. In our program, there are 135 lines of codes which are responsible to assign textual components. By the help of pointcuts in aspect, we caught these 135 lines of codes in by just 1 line of code. Internationalization is a concern by itself and we thought that it should be applied separately. By the help of aspect implementation, it is done at the and global which means it can be applied in anywhere. We do not need to change code it is enough to change just one line of code for another language support.

```
public aspect internationalization
{
```

```
pointcut setText(String s,JComponent
comp):call(* *.setText(String)) && args(s)
&& target(comp) &&
!this(internationalization);
void around(String s, JComponent
comp): setText(s, comp) {
//some variable declarations
...
locale = new Locale("tr","TR");
rb =ResourceBundle.getBundle("Messages"
, locale);
//getting locale value according to keys
...
if(comp instanceof JMenuItem){
JMenuItem jm=(JMenuItem)comp;
jm.setText(s);
}
// check for other types
...
}
```

3.1.2.2. Look&Feel

The Java look and feel is the default interface for applications built with the JFC. The Java look and feel is designed for cross-platform use and can provide:

- Consistency in the apperance and behavior of common design elements.
- Compatibility with industry-standard components and interaction styles
- Aesthetic appeal that does not distract from application content

```
public aspect lookandfeel {
pointcut catchButton(Jbutton j):
set(Jbutton *.*) && args(j);
after(Jbutton j):catchButton(j){
int osType =DetailStorage.osType();
switch (osType){
case 0://Unknown OS
break;
case 1://Windows
j.setUI(new
javax.swing.plaf.basic.BasicButtonUI());
break;
case 2://Mac
...
break;
case 3://Unix, Linux
j.setUI(new
javax.swing.plaf.metal.MetalButtonUI());
break;
}
```

It is possible to change whole appearance with the help of one line of code. However, whenever you want to change appearance of some certain components, you

should change code line by line. In our eBookManager application, we wanted to change appearance of just buttons according to operating system. In order to achieve this target, we firstly caught the pointcuts where the button fields are first initiated. Then we embedded an aspect after this initiation which changes the appearance of the button

3.1.2.3. Agent Based Communication

We add the new concern of providing distributed agent based book communication which is not supported by the existing system.

A distributed system consists of multiple autonomous computers that communicate through a computer network. The computers interact with each other via agents in order to achieve a common goal. Agents typically possess several characteristics like being Autonomous, mobile, goal oriented, communicative and collaborative, persistent, flexible, active or proactive. They also have the capability to adapt and learn.

An agent based distributed system has a main container which provides the registration of all the other containers as soon as they start, the main container also manages the agents by providing name service and it makes it possible for the agents to find and communicate with each other. Each container in the platform can release as many as agents they want with different roles. And these agents communicate with each other regardless of the location and they interact in order to achieve their goal.

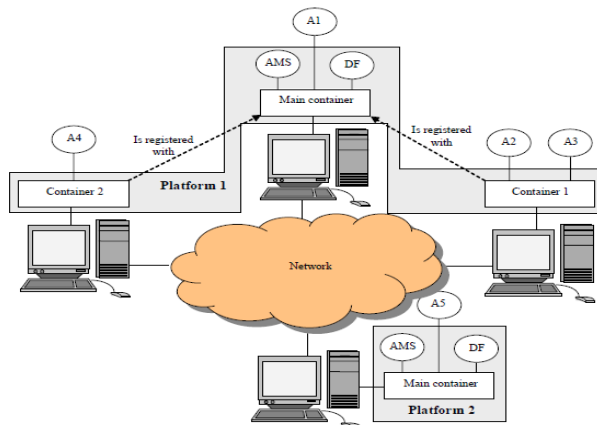


Figure 1 Containers and Platforms

Figure 3: Main UI of the EBook Manager

For our test case we add the mentioned concern and adapt a functionality to the existing system that uses

the distributed agent communication. According to this, there are two types of agents, buyer agents and seller agents. Each buyer agent is given the target book's title and it periodically requests all known seller agents to provide related price information about the book. Once it is provided the price information the buyer agent gets this information and evaluates the prices it got from several seller agents. After accepting the lowest price it terminates. Each seller has a local catalogue of book titles and associated price information. Seller agents continuously wait for requests from buyer agents. When asked to provide the price information for a book they check if the requested book is in their catalogue and in this case reply with the information and service them. Otherwise they refuse.

```
public aspect DistributedSellerAgent {

    pointcut loadFilesIntoModel(BookTableDataModel
    btdm) : call(*
    BookTableDataModel.loadFilesIntoModel(..) &&
    target(btdm);

    //Advice to create a seller agent after execution of the
    pointcut
    after(BookTableDataModel btdm) :
    loadFilesIntoModel(btdm){

        ...

    }

}
```

For the implementation of this part we used JADE (Java Agent Development Framework): a software Framework fully implemented in Java language to develop agent based distributed systems.

To add this functionality by directly applying AOP on legacy code we needed to create agent classes first. Then we used AOP to connect these agents to several points of the application to do their job.

3.2. Dealing crosscutting concerns by Object-Oriented re-factoring without AOP

3.2.1. Existing Concerns

3.2.1.1. Detail Storage

We have created abstract DetailStorage class which contains abstract methods for storing and reading book

details. Then we have implemented two different detail storage classes, `FileNameDetailStorage` and `SerializedDetailStorage`. Prior one has the functionality that the original eBook manager supplies. It stores the book information by changing file name. The latter one stores the information on separate files which contains serialized eBook objects. This way we can store more information compared to file names and reading back the details is a simple cast to eBook object.

When we wanted to do the same change on detail storage function that we did by directly applying AOP on legacy code by refactorization we needed to write an abstract class and two implementations to it. Those totally make 187 lines of code. The diagram of Detail Storage related classes can be seen on fig. 4.

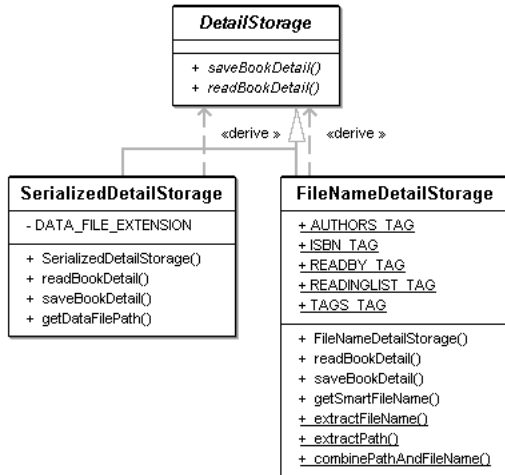


Figure 4: Detail Storage Class Diagram

3.2.1.2. Checkout

During re-factorization of checkout function we have created four new classes and we have used factory method pattern. `CheckOutFactory` class is used for creation of different `CheckOutManagers`.

We have implemented two different checkout managers. One for the original functionality which copies files to given destination folder and the other sends the selected e-book files to given email address.

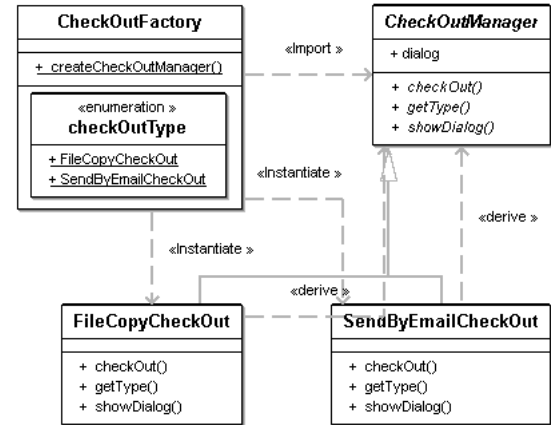


Figure 5: Checkout Class Diagram

To make the decided changes on checkout function by refactorization we have build 4 classes and applied factory method pattern. We have built a factory class to create checkout manager objects and an abstract class to create body of managers. Then we have created two concrete implementations of the abstract checkout manager class.

3.2.1.3. Web Service

We have made changes on web service related files directly. Since the old web service is deprecated it is meaningless to keep codes related to it. So we have made direct changes on it.

3.2.2. New Concerns

3.2.2.1. Internationalization

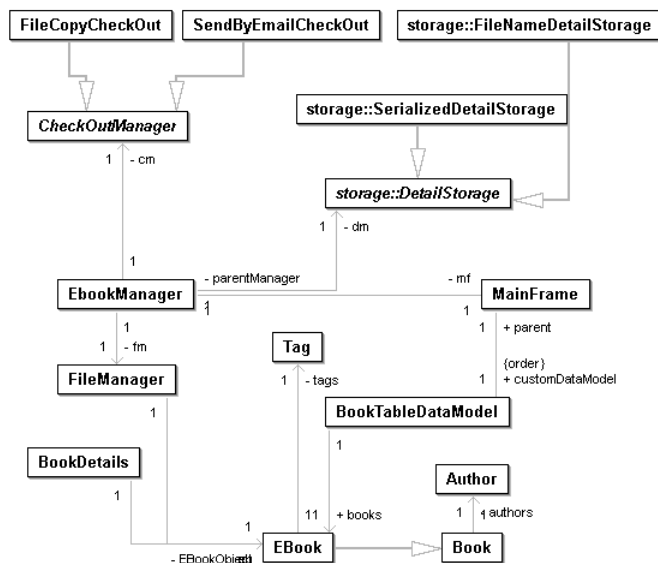
Internationalization is a crosscutting concern by its nature. To change the text property of all swing objects with an object oriented manner we need to type it explicitly after each object initialization. This is not an efficient solution. Some frameworks might be used for internationalization but it is not easy to port a legacy code to a new framework. It might require many changes to be done on the existing code.

3.2.2.2. Look & Feel

Look & Feel concern has a similar nature to internationalization. There is again no easy way to achieve it with object orientation.

3.2.2.3. Agent Based Communication

To be able to use the buyer and seller agent classes we have created we needed to manually connect these classes to existing ones. Since the currently intended functionality of the agents are known we have added some code to search function and table data model class to invoke these agents.



For our third strategy we have refactored the whole project. The class diagram of the refactored Ebook Manager can be seen on fig. 6.

classes. Previously they were stored as String objects and it was hard to manipulate.

Even after the complete re-factorization of the application internationalization still remains as a crosscutting concern. We have used our AOP solution in this case which is extremely easy and effective.

Look & Feel concern also remains as a crosscutting concern since we have several user interface classes and button objects are scattered all around them. To catch those objects easily and make the change on look & feel we used AOP approach explained in previous section.

3.2.2.3 Agent Based Communication

Agents are created in a combined manner of OOP and AOP. Since agent classes are new to the application we have built them outside the aspects and connected them to the application by using AOP to let them work exactly the points we want.

4. Related Work

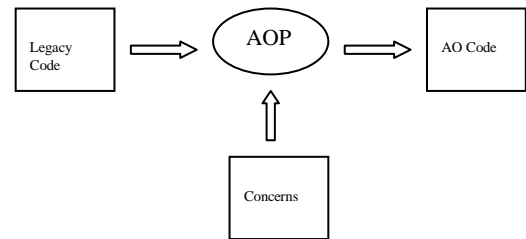
There are some previous works that compares the aspect-oriented software development and object oriented software development in web applications. One of these studies, is Reina, A. et al [2]. To show the advantages of the aspect-oriented development in web applications, they developed an application using AspectJ and Java Server Pages (JSP) and Cocoon. Tsang et al. [3] present an initial study illustrating the tradeoffs involved in increasing modularity. They empirically compare AO vs OO solutions in the context of Real Time Java, illustrating that modularity is improved with AOP but that in terms of other factors such as maintainability, reusability and testability, the original OOP solutions are often favorable. Baniassad et al. [4] show evidence for the presence of crosscutting concerns in practice and how they are implemented without AOP. Three main types of such crosscutting concerns are identified as well as implementation strategies. The study represents a cornerstone for the assessment of AOP. A very recent study [5] compares pure OOP and AOP with respect to software evolvability. As expected, changes to crosscutting concerns are better handled in AOP software, while changes of a more fundamental and architectural nature can often be better assimilated by OOP solutions. The authors state that this is due to the observation that AOP narrows the boundaries of concern dependencies, but tightens their interaction. We note that explicit interfaces as studied in this work can become a focal point to guide and control this interaction. Another study [6] reports their experience using AspectJ, a general purpose aspect-oriented extension to Java, to implement distribution and persistence aspects in a web-based information system. This system was originally implemented in Java and restructured with AspectJ. They directly use aspect oriented programming on to the pure java.

In our study we focus on applying AOP on legacy code and we define several strategies for that.

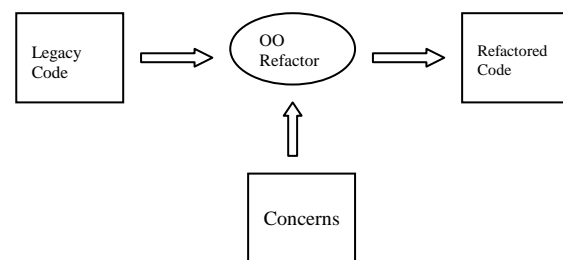
5. Discussion

In our paper, we suggest that it is possible to define strategies for applying AOP on legacy code and using these strategies to work more efficiently by reducing the time, effort and cost spent. We discuss several strategies for applying AOP on legacy code. In particular, we discuss the strategies:

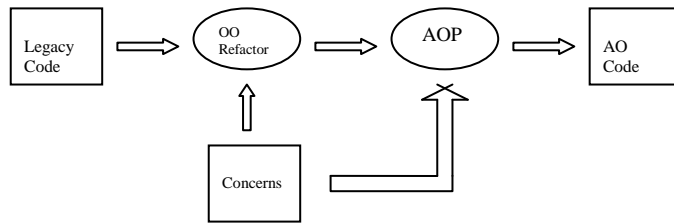
a) Direct application of aspects to legacy code without refactoring



b) Dealing crosscutting concerns by Object-Oriented refactoring without AOP



c) Applying AOP after Object-Oriented refactoring



With the provided strategies that could be used to apply aspect oriented programming to create solutions for the legacy code, we suggest that it is possible to achieve solutions in a less time consuming and less costly manner without sinking in the complexity of the existing code. We visualized and grounded our approach by implementing our strategies on a test case. To emphasize our point we provided the comparisons with the common existing approaches.

Our observations and conclusions could be listed as follows:

- By directly applying the aspect oriented approach on existin concerns, it is possible to create solutions with relatively a small amount of coding when compared to refactoring the legacy code with respect to the OOP techniques. Not being limited to and not spending too much time on the comprehension of the legacy code, by directly applying AOP we could deal with existing concerns more efficiently.
- For new concerns directly applying AOP does not provide a reasonable efficiency in terms of time and linecone, however we see that applying aspect oriented approach enables an improved extensibility to the system; which means by AOP we are creating more generic solutions. Thus, for future aspectual concerns when we add related methods it will prevent potential crosscutting concerns, moreover these new concerns could be applied to the Project easily in a less time consuming manner.
- We could create solutions fort he existing concerns by refactoring the legacy code; however we see that when compared to AOP, this process

is more time consuming and also more difficult because needs a good comprehension of the legacy code. On the other hand, it increases the quality and the reuseability of the legacy code. When we use this approach for new concerns we see that OOP techniques are limited to deal with crosscutting concerns; which means the scattered and tangling code could not always be solved by re-factorization.

- For the cases of existing concerns or future aspectual concerns that are still tend to be scattered and tangled over the system even after refactorization, to deal with the shortages of the OOP techniques we apply AOP as complementary to eliminate these crosscutting concerns.

It is possible to summarize the advantages and disadvantages of the strategies on both existing and new concerns in table-1. Using this table the advantages and disadvantages of the different approaches could be evaluated and it may be possible to take action considering where the case fits in the table.

	Existing Concerns	New Concerns
Directly Applying AOP	<ul style="list-style-type: none"> • Easy to implement • Does not require understanding all parts of code. • No need to make change on existing code. 	<ul style="list-style-type: none"> • Easy to implement • Do not require wide understanding on existing code. • Could be applied to the project easily • Improves extendibility for future aspectual concerns.
Re-factorization	<ul style="list-style-type: none"> • Decreases crosscutting concerns • In some cases may not be possible. • Hard to understand existing scattered code • Time consuming • Increases code reuse-ability 	<ul style="list-style-type: none"> • For some concerns not possible to solve by re-factorization • Requires hard coding within existing classes
Re-factorization and AOP	<ul style="list-style-type: none"> • Could deal with crosscutting concerns which 	<ul style="list-style-type: none"> • Prevents crosscutting concerns • Improves

	OOP re-factorization could not solve itself <ul style="list-style-type: none"> Improves modularity 	modularity
--	---	------------

Using these strategies in the proper cases we could use AOP to create solutions in a good and fast way in applications with legacy code.

6. Conclusion

In today's world creating software solutions within the minimum time possible is highly appreciated and the strategies we provided in our study could be used as a benchmark in the area and can be extended. These strategies could be used while determining how to handle both the existing and new concerns on legacy code by applying AOP. These strategies when used in proper cases will reduce the time spent in terms of preventing the

Acknowledgment

We would like to thank Asst. Prof. Dr. Bedir Tekinerdogan, for his efforts and contributions and also for organizing the Turkish Aspect-Oriented Software Development Workshop 2009.

8. References

- [1] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier and John Irwin, Aspect-Oriented Programming, ECOOP'97 Object-Oriented Programming, 1997
- [2] Reina, A.M., Torres, Toro, M. Aspect-Oriented Web Development vs Non Aspect-Oriented Web Development
- [3] S.L. Tsang, S. Clarke, and E. L. A. Baniassad. An evaluation of aspect oriented programming for java-based real-time systems development. In ISORC'04, 2004.
- [4] E. L. A. Baniassad, G. C. Murphy, C. Schwanninger, and M. Kircher. Managing crosscutting concerns during software evolution tasks: an inquisitive study. In AOSD'02, pages 120-126, 2002.

programmer deal with the drawbacks of the legacy code such as the complexity and the limitations of the code. We believe that our approach will be appreciated since it is a common problem in real world practice where the programmers have to deal with such drawbacks of the legacy code.

7. Future Works

We believe that our study provides a benchmark for future researches in the area. Our study can be extended by using large-scale applications with legacy code. Since we picked a fairly small application, it would add to our study to discuss our strategies on such application. Moreover, new strategies could be suggested.

- [5] P. Greenwood, T. Bartolomei, E. Figueiredo, M. Dosea, A. Garcia, N. Cacho, C. Sant'Anna, S. Soares, P. Borba, U. Kulesza, and A. Rashid. On the impact of aspectual decompositions on design stability: An empirical study. In ECOOP'07, 2007.
- [6] S. Soares, E. Laureano and P. Borba. Implementing Distribution and Persistence Aspects with AspectJ. In OOPSLA '02, 2002.

An Aspect-Oriented Simulator for Distributed MST Construction on Wireless Networks

Abdullah Bulbul, Omer Faruk Uzar, Utku Ozan Yilmaz

Department of Computer Engineering, Bilkent University
{bulbul,uzar,uyilmaz}@cs.bilkent.edu.tr

Abstract

Distributed algorithms are essential in wireless network environments where a centralized control does not exist. In those environments, ensuring the correctness and stability of the processes running on separate components of the network and their combined effect become significant concerns. Aspect Oriented Software Development (AOSD) makes it possible to control, validate and monitor the general progress in a distributed algorithm. In particular, this paper proposes an AOSD based solution to the Minimum Spanning Tree (MST) construction problem in wireless networks and compares it to the Object Oriented approach. Using AOSD, it is possible to validate the network components, synchronize them, check the correctness of the solution, and demonstrate the running algorithm imperceptibly to the running separate processes.

Keywords—AOSD, OOP, separation of concerns, Minimum Spanning Tree, Distributed Algorithms, Wireless Networks

1. Introduction

Today, wireless communication covers our daily life inevitably such that mobile phones and wireless internet services became the main necessities for most of us. From the software development perspective, wireless communication brings the requirement of using distributed algorithms which emerges extra concerns for mobile software developers such as synchronization, stability and adaptability to mobile variability.

Wireless and wired networks are excessively used in everyday life. However, due to its time consuming and costly nature, deploying a network which is not suitable

for the needs of the situation is out of question. Thus, simulating the behavior of a network in a software environment prior to actually building it is a key concern. Moreover, network simulators bring diversity to test cases while only a few cases can be tested in hardware. Furthermore, simulators make it possible to test marginal situations, which are hard to reproduce and test in a deployed system. Therefore, a simulator which imitates a wireless environment brings down the modification and maintenance costs by decreasing the probability of deploying unsuitable networks and enhances the variability of the use cases to simulate.

In a wireless environment, usage of distributed algorithms becomes inevitable. Upon usage of distributed algorithms, processors of different members execute concurrently to accomplish the same goal. Consequently, the software developers need to consider new concerns such as stability of the running algorithm on different members and synchronization. Thus, separation of concerns becomes a more difficult principle to accomplish in a distributed algorithm, revealing the advantages of AOSD which mainly aims to separate different concerns while developing and maintaining software.

In this study, we develop a simulator for the distributed version of a well known problem, Minimum Spanning Tree, using two software development approaches: the Object Oriented Software Development approach and the Aspect Oriented Software Development approach.

Our first approach for simulating the distributed MST construction is the OOSD approach. In this approach we make use of several of the Design Patterns of OOP. After implementing the simulator in an Object Oriented fashion, we solve the remaining problems using AOSD. Using AOSD, we are able to synchronize the separate processes, validate the correctness of each

process and the network structure, and visualize the overall packet send-receive events.

The organization of the remaining parts of the paper is as follows: Firstly, the background section describes the ancestors of this study. Then we explain our Object-Oriented Design and the used Patterns. Section 4 describes the necessary aspects and their implementation details are given in Section 5. The Related Work section describes the similar studies to our study which are performed previously and points out the differences. Section 7 contains the evaluation of our study and concludes the paper.

2. Background

In a wireless network, a node has access to only a subset of the network that is determined by factors like its transmission power and the node positions. The other nodes inside this subset form the neighborhood of this node. As opposed to the wired communication, we cannot explicitly control the exact topology of the network by determining the neighborhood of a node using wires and switches. None of the network members is aware of the overall structure of the network and that results in lack of all-knowing nodes. Therefore, implementing centralized algorithms in wireless networks is not efficient, which makes distributed algorithms the main requirement when wireless communication means are used instead of wired ones.

The MST is a fundamental problem in the fields of Graph Theory and Computer Networks. A spanning tree of a network is any tree that includes all nodes of a network, while MST is the spanning tree on which the total weights (costs) of the tree edges are the minimum. The MST related studies can be found in the survey of Graham et al. [1] and the most well-known solutions to this problem are Kruskal's [2] and Prim's [3] algorithms which are centralized algorithms. MST is especially useful for efficiently broadcasting in a network. Furthermore, having an MST is essential for efficiently solving many other problems. A distributed algorithm for MST construction is proposed by Gallager, Humblet, and Spira [4] and it is known as the GHS algorithm. The MST calculation in our system also makes use of this algorithm. An example MST is shown in Figure 1.

In a previous study [6], a distributed program that builds a minimum spanning tree (MST) for a network is implemented. It is a synchronous implementation of

the GHS algorithm in pure Java. The steps of the program are as follows:

- 1) *Read the partial network topology from a configuration file:* Information about total number of nodes in the network, nodes local to the running host and their neighbors are read from a configuration file.
- 2) *Build the MST:* MST is built in $O(\log n)$ steps and is modified to root it at a particular node.
- 3) *Broadcast messages:* After the MST is built; any node can broadcast the given messages to the subtree rooted at it.

Inspired by this distributed program, we designed and implemented an aspect-oriented simulator for distributed MST construction on wireless networks in a centralized manner.

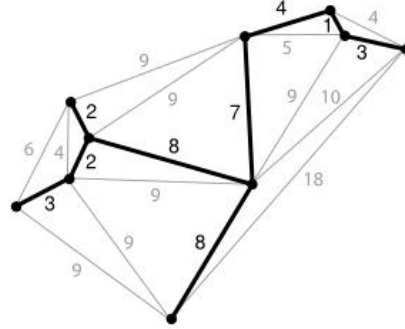


Figure 1: Minimum Spanning Tree Example [5].

3. Object-Oriented Design

In this section, we explain how to construct the minimum spanning tree (MST) in a distributed manner. As a result of the distributed approach, a centralized, all-knowing entity does not exist and the nodes' information about the network topology does not go beyond their neighbors.

3.1. Description of Application

Our application uses the aforementioned GHS algorithm to build a minimum spanning tree of wireless nodes by simulating the distributed environment. Although the full network topology is available to the application, each node is oblivious to the nodes beyond its immediate neighbors to fulfill the aim of simulating the actual wireless environment. After the wireless backbone is formed by connecting each node to its neighbors, the GHS algorithm starts running. In the

beginning, each node is a subtree itself. Then, at each round, number of subtrees is halved in the worst case by merging them. Thus, the algorithm terminates in $O(\log n)$ rounds. When only one subtree remains, the MST is built and the nodes are ready to broadcast messages given to them.

Our application has a GUI where users can place nodes and specify connections between them. The GUI also lets the users to suspend the execution to see the intermediate steps in detail, and then continue the execution. Furthermore, the final MST and the transmitted messages are visualized by our interface. Figures 5-6-7 shows parts of the GUI.

3.2. Object-Oriented Design

The program has 17 classes, one of which is used for visualization. Their functionalities are as follows:

Node: The main building block of the network. Moreover, it provides the main functionality of the system by running the GHS algorithm. It has a number of Neighbors.

Neighbor: Represent a neighbor of a node and the link in between. Has a Sender and a Receiver.

Sender and Receiver: They respectively send messages to and receive messages from the neighbor they belong to. They represent the output and the input ports of the link between a node and its neighbor.

Packet class hierarchy: Each subclass of Packet represents a different message. Sender and Receiver

provide communication between nodes by using them.

State class hierarchy: Each subclass of State represents a state of the GHS algorithm. They realize the state pattern in our application.

MessagePanel: Handles the visualization of our application.

The UML diagrams are shown in Figures 2-3-4.

3.3. Implementation of Design Patterns

State Pattern: There are 5 states for the construction of MST, namely *Search*, *Examine*, *Reply*, *Add* and *New Leader* states. On each state a *Node* sends some specific packets for this state and similarly waits for a certain number of packets of the same kind. Thus, it is appropriate to use the State Pattern of Object Oriented Design. Each state has a *handle* method which takes a *Node* instance as a parameter. In that method, the packets to send and receive are in different states. A node moves to the next state when it finishes its execution in its current state by calling *nextState()* method. Figure 3 shows the application of State Pattern to our design.

Observer Pattern: We have many Nodes working as Threads at the same time. They are computing the MST in a distributed manner. We need to visualize the computation process. For doing that we need observer pattern since we have many nodes and the graphical user interface changes if only a Node changes its state or attributes. So we have an observer (MessagePanel) and

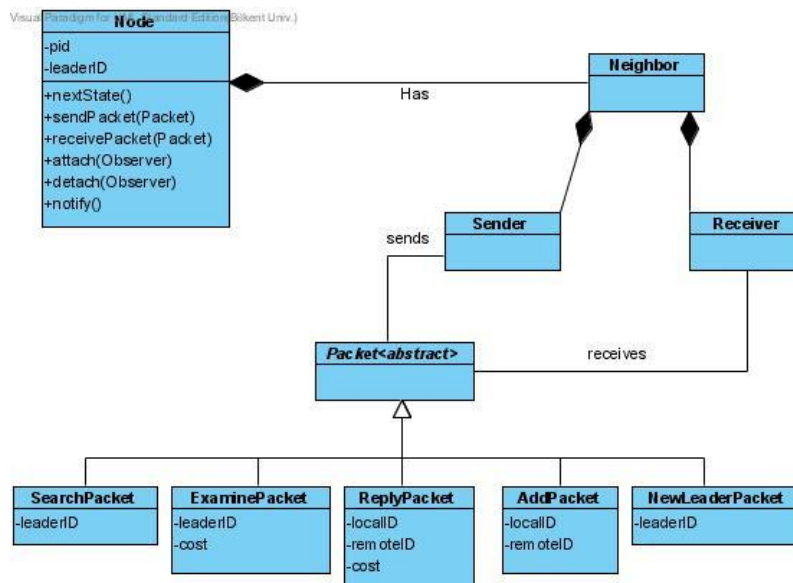


Figure 2: Class diagram of the system (Object-oriented approach)

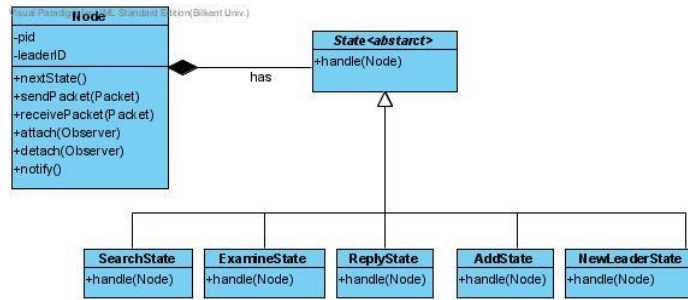


Figure 3: Application of the State Pattern to our design

many subjects (Node). When a node decides that it changed something important, it calls the notify method. Then observer updates its information according to the changes in the node. Figure 4 shows the application of Observer Pattern to our design.

4. Identifying Aspects

The previous section explains the object oriented backbone of our program. However, it has some crosscutting concerns that result in scattering and tangling. Therefore, in order to identify our aspects, we started with discovering the crosscutting concerns in our program.

Since our program runs a distributed algorithm, it has to handle synchronization, which is crucial for the algorithm's correctness. As indicated before, we accomplished it by defining states for parts of the algorithm. The states of the threads need to be checked at multiple points and if necessary, the executions of some threads need to be suspended until others catch up. However, the synchronization code is scattered all over the algorithm. It makes the code hard to understand and makes it hard to make changes to the synchronization code. Therefore, we defined synchronization as our first crosscutting concern.

In our program, every message sent/received is shown in the GUI. Moreover, we have different views

for nodes and edges; the current views of some elements change during the course of the algorithm as a response to multiple events. Tracking all these events to update the GUI results in tangled code. Thus, we defined display as our next crosscutting concern.

In addition to the crosscutting concerns in our object oriented implementation, there are extra crosscutting concerns which we didn't try to implement in an object oriented manner. In one such case, we decided to make it possible for the user to control the execution of the program by pausing it at certain points. The necessary code would be scattered if we implement it without using AOP, as at every control point we would need to check if the user sent a signal to pause or not. Hence, we defined control as another crosscutting concern.

Validation of the inputs and outputs is necessary to avoid running the distributed MST algorithm in erroneous networks and it increases the robustness of the program. As the input validation code would tangle our code and we want to use output validation only in development phase, we defined validation as our last crosscutting concern.

In this section, we identified four production aspects corresponding to the four crosscutting concerns defined above, to handle them with the help of AOP. Synchronization, Display, Control and Validation Aspects are described below in detail.

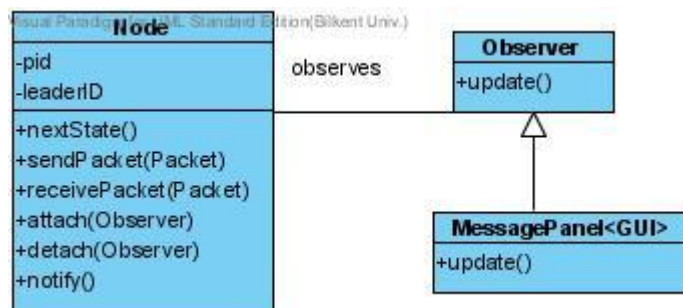


Figure 4: Application of the Observer Pattern to our design

4.1. Synchronization Aspect

In plain object oriented implementation there are 5 states: search, examine, reply, add and new leader respectively. Since there are as many threads as nodes, they need to be synchronized. In other words every node needs to move to the next state at the same time. If it is not the case the program will not work correctly. In pure OO implementation, it is hard to accomplish, since the synchronization code is scattered all over the program and makes it hard to understand and it has a large overhead. This aspect solves this restraint by ensuring that all the nodes move to the next state at the same time by catching the execution of the nodes that are ready and making them wait until every node is ready to move to the next state.

4.2. Display Aspect

We need to display the behavior of nodes for user interaction. In plain object oriented implementation the display code is scattered, hence difficult to modify and maintain. This scattering is a result of the displayable elements, which can be categorized as follows:

- 1) Nodes can be leaders.
- 2) Nodes can send message to other nodes.
These messages have 5 types: search, examine, reply, add and new leader.
- 3) An edge between two nodes can be part of the MST.

For each category the visualization elements differ. For category 1 we have different views for leader and non-leader nodes. For category 2, for each message sent there is an arrow between the specified node and color of arrow will change according to the type of the message. For category 3, an edge between two nodes has a different visualization if it is part of the MST.

4.3. Control Aspect

In plain object oriented implementation, once the program starts to execute, it is not possible to interact with it until the MST is built; there is no intermediate step that can be seen for simulation. However, the user may want to see the intermediate steps in detail to identify nodes that cause bottlenecks, to grasp the working principals of the algorithm etc. Therefore, we should be able to intercept the program in various control points for the user to pause the execution until he wants to continue. There will be 3 such control points:

- 1) Waits for user in each message send.
- 2) Waits for user for each state.
- 3) Waits for user for each round.

We know that each round has 5 states and in each state a lot of messages are sent, so the user can continue the execution of the program for short (until next sent message), medium (until next state) and long (until next round) periods by using this aspect.

4.4. Validation Aspect

In plain object oriented implementation there is no validation of inputs or outputs. In other words, the program assumes that the input is correct, and this assumption leads to another one: the correctness of the output. However, making assumptions when developing software is a grave mistake; we cannot assume that user always enters correct input and also we cannot assume the correctness of output in each case. So we need to validate the input and the output. When the simulator starts, the validation of the input is verified. If the verification fails, then the user is warned and will not run the program. Similarly, when the GHS algorithm finishes, program will check if the output is correct or not and warn user if it is necessary.

5. Aspect-Oriented Programming

In this section we will explain the implementation of the aspects defined in the previous section.

5.1. Synchronization Aspect

This aspect synchronizes the state transitions. It has one pointcut named *nextState*, which captures the call to the *nextState* method of *Node* class. The advice, which is of type “after”, suspends the execution of the calling thread until all nodes are ready to move to the next state. If all nodes are ready for the state transition, then all suspended threads are awakened and they move to the next state simultaneously. Code Fragment 1 shows the pointcut and advice of this aspect.

5.2. Display Aspect

Display aspect handles three distinct cases. These cases are explained below in detail and their corresponding pointcuts and advices are shown in Code Fragments 2-3-4.

Case1: Any node can be the leader of a subtree at a given moment, so leader nodes must have a different


```

pointcut nextState( ) : call( * Node.nextState(..));

after (Node n) : target(n) && nextState() {
    NetworkVisualizer.getInstance().enableButtons(true);
    if( count < maxNodeCount ){
        count++;
        try {
            lock.lock();
            stateLock.await();
            lock.unlock();
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        if(count == maxNodeCount){
            count = 0;
            if( nextRound ){
                countOfState++;
                if( countOfState <= 5 ){
                    lock.lock();
                    stateLock.signalAll();
                    lock.unlock();
                }
                else{
                    nextRound = false;
                    waitingMessage = true;
                    countOfState = 0;
                    NetworkVisualizer.getInstance().enableButtons(true);
                    NetworkVisualizer.getInstance().getVisualizer().repaint();
                    NetworkVisualizer.getInstance().deleteMessages();
                }
            }
            else{
                waitUserInput();
                NetworkVisualizer.getInstance().deleteMessages();
            }
        }
    }
}

```

Code Fragment 1: Synchronization Aspect

view than the non-leader nodes for ease of recognition by the user. We have two different shapes and colors for leader and non-leader nodes to do that, as shown in Figure 5. In order to accomplish that, we need a pointcut that selects the joinpoint that captures the write action on the *leaderID* field of class *Node*. If that field is written, it means that the node can be of either type (i.e. leader or non-leader) at the moment. Therefore, we need to update its status according to the value of *leaderID* and *pid* (the own id of the node). The advice is an after advice. After the *leaderID* field is updated, the node's status is also updated accordingly.

Case2: A node can send 5 different types of messages to another node. We need a pointcut that catches the call operation on *send* method of class *Node*. By using target and args methods of AspectJ, we are able to get the necessary properties for the visualization of a sent message. The advice is an after

advice which simply calls the *sendMessage* method of *NetworkVisualizer* class with the properties that have been gotten by pointcut.



Figure 5: Different visual elements for different types of nodes.

Case3: An edge can be a part of a subtree at any given moment and once it becomes a part of a subtree it will be a part of the final MST. Adding an edge to a subtree is accomplished by the add message, which is represented by an instance of the *AddPacket* class. So,

```

pointcut newLeader( ) : set( int Node.leaderID );

after( Node n ) : target(n) && newLeader() {
    NetworkVisualizer.getInstance().setLeaderMessage((int) (n.getPid()), n.getLeaderID() );
}

```

Code Fragment 2: Display Aspect – Case 1

```

after(Neighbor n, Packet p) returning() : target(n) && args(p) && (call( * Neighbor.sendMessage( Packet ) ) ) {

    //next message'a tiklandiyisa waitingMessage true oluyor
    if(waitingMessage){
        NetworkVisualizer.getInstance().enableButtons(true);
        try {
            NetworkVisualizer.getInstance().enableButtons(true);
            lock.lock();
            waitingMessageCount++;
            System.out.println( "Waiting message " + waitingMessageCount );
            messageLock.await();
            lock.unlock();

        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }

    NetworkVisualizer.getInstance().sendMessage(n.getNodeId(), (int) n.getPid(), p);
    NetworkVisualizer.getInstance().getVisualizer().repaint();

}

```

Code Fragment 3: Display Aspect – Case 2

```

after(int node1, int node2,int senderId ) : args(node1,node2,senderId) && call( ADD_MWOE.new(..) ) {

    NetworkVisualizer.getInstance().addMSTedge( node1, node2 );

}

```

Code Fragment 4: Display Aspect – Case 3

we need a pointcut that captures the call to the *new* method of *AddPacket* class and gets the information of the constructor of this class by args method of AspectJ. Then, we know that an add message is created to add the specified edge to a subtree (hence to the final MST); therefore the color and shape of the specified edge is changed on the visualization part. Different visual representations of edges can be seen in Figure 6.

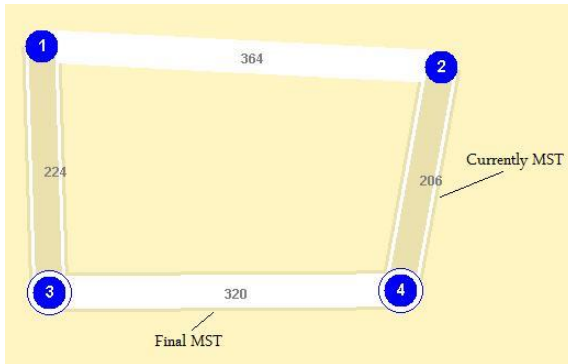


Figure 6: Different visual elements for different types of edges

5.3. Control Aspect

Control aspect handles three distinct cases. These cases are explained below in detail and their corresponding pointcuts and advices are shown in Code Fragment 5.

Case1: The program suspends execution and waits

for user input to continue each time a message is sent. We need two pointcuts to accomplish that: one for selecting the *sendMessage* method to suspend the execution and the other one for selecting the action performed by clicking next packet button to continue execution. Similarly, there are two advices: The first one is for suspending the execution, which is an around advice that performs wait operation on the *Node* that sent the message. The other one is for waking up the thread that is waiting for user input. User clicks the next packet button so that the next message can be sent by waking up the thread.

Case2: The program suspends execution and waits for user input to continue each time the state of the nodes is changed. In order to do that, we need pointcuts that select *nextState* method of class *Node* to suspend the execution and the action performed by clicking next state button to continue execution. Likewise, there are two advices: The first one is an after advice that suspends the threads after next state method called. The other one is for waking up all the threads that are suspended by the above around advice by performing a *signalAll* operation.

Case3: The program suspends execution and waits for user input to continue at the beginning of each round. Each round consists of 5 states. So we can use a modified version of Case 2. While Case 2 waits user input for each state transition, Case 3 waits user input for every fifth state transition.

```

after(ActionEvent e) : args(e) && execution(* *.actionPerformed(ActionEvent)){
    if (e.getSource() instanceof JButton) {
        if(((JButton) (e.getSource())).getText().equals("Next Message")){
            NetworkVisualizer.getInstance().getVisualizer().repaint();

            waitingMessage = true;
            lock.lock();
            if( waitingMessageCount > 0 )
            {
                waitingMessageCount--;
                messageLock.signal();
            }
            else{
                NetworkVisualizer.getInstance().enableButtons(false);
                stateLock.signalAll();
            }
            lock.unlock();
        }
        if(((JButton) (e.getSource())).getText().equals("Next Step")){
            NetworkVisualizer.getInstance().enableButtons(false);
            waitingMessage = true;
            lock.lock();
            messageLock.signalAll();
            stateLock.signalAll();
            lock.unlock();
        }
        if(((JButton) (e.getSource())).getText().equals("Next Round")){
            NetworkVisualizer.getInstance().enableButtons(false);
            NetworkVisualizer.getInstance().getVisualizer().repaint();
            nextRound = true;
            waitingMessage = false;
            lock.lock();
            messageLock.signalAll();
            stateLock.signalAll();
            lock.unlock();
        }
    }
}

```

Code Fragment 5: Control Aspect

5.4. Validation Aspect

As the validation is done in two points throughout the program (input and output validation at the beginning and at the end of the program, respectively), the validation aspect has two duties to handle. The pointcuts and advices of these two cases are shown in Code Fragment 6.

In the case of input validation, we need to check if the input is correct or not before running the necessary algorithm. To do that, we need a pointcut that selects the joinpoint where the algorithm starts execution. The algorithm starts running when the start button is clicked. We use an around advice. The reason for it is the possible prevention of further execution of the program; if the input is correct the algorithm starts running, but there will be a warning message and the algorithm will not start otherwise (Figure 7).

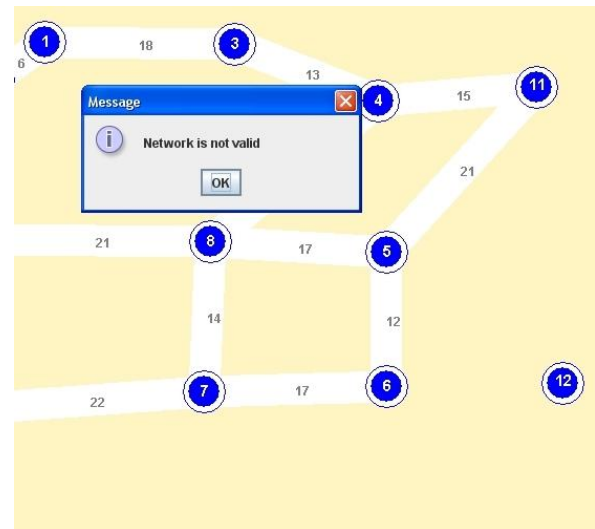


Figure 7: Demonstration of the effect of the validation aspect for input correctness. Since there is a non-connected node in the graph (Node 12), the program does not start and warns the user.

```

Object around(ActionEvent e) : args(e) && execution(* *.actionPerformed(ActionEvent)) {
    if (e.getSource() instanceof JButton) {
        if (((JButton) (e.getSource())).getText().equals("Start Algorithm")) {
            if (NetworkVisualizer.getInstance().validateInput())
                return proceed(e);
            else {
                JOptionPane.showMessageDialog(null, "Network is not valid");
                return null;
            }
        }
        else {
            return proceed(e);
        }
    }
    else {
        return proceed(e);
    }
}

after() : call( * GUIMessagePanel.updateFinished(..) ) {
    JOptionPane.showMessageDialog(null, "MST Constructed");
    NetworkVisualizer.getInstance().restartState();
}

```

Code Fragment 6: Validation Aspect

On the other hand, in the case of output validation, we cannot assume that the output is valid. Hence we need to check if the output (final MST) is correct or not. If the output is not correct, the coder is warned and he will need to check the code and debug it according to the mistake.

6. Related Work

There are some previous software in the area of wireless (in some cases together with wired) network simulators including, but not limited to, popular network simulators like J-Sim [8], OMNeT++ [9], ns-2 [10] and OPNET Modeler [11]. Although their aim is simulating the network behavior as a whole; while our aim is simulating the process of building minimum spanning tree of a wireless network prior to deployment, they are comparable to our work to an extent. A comprehensive overview of these simulators is given in [7]. In addition to their general advantages (e.g. simulating network traffic) /disadvantages (e.g. lack of step-by-step simulation, which we accomplished using AOP) over our work, they all have their strengths and weaknesses, which are described below:

J-Sim: J-Sim has no built-in wireless network support; it handles them through an extension. Furthermore, it lacks built-in graphical user interface. The visual network simulation can be accomplished by ns-2's Network Animator (Nam). On the other hand, the simulator can be extended by creating new components.

OMNeT++: Like J-Sim, OMNeT++ has no built-in wireless network support and handles them through an extension. However, it is a generic simulation tool and can handle a variety of different systems.

Ns-2: Ns-2 has no built-in wireless network support and handles them through an extension, like the above simulators. Although it is a fairly complete simulator, it is complex and has a significant learning overhead. On the upside, it can be connected to a real network and send/receive packets. Moreover, it has a lot of built-in protocols.

OPNET Modeler: OPNET Modeler also has no built-in wireless network support and handles them through an extension. On the other hand, it has a lot of protocol models implemented in it.

Aside from the differences addressed above, which are mostly functional, these simulators do not benefit from AOP paradigm [8] [9] [10] [11] unlike our simulator. As our code's maintainability and understandability are enhanced by AOSD, it is possible that their code has a lower level of maintainability and understandability.

7. Discussion & Conclusion

We have implemented a Distributed MST construction program using OO design principles. In the OO design, *State* pattern is used to control the behavior of the nodes in the network in various states of the algorithm. Execution of each node is determined according to its current state. Additionally, *Observer* pattern is used to notify the GUI components upon an action on the Nodes such as sending and receiving

messages.

After completing the Object Oriented programming, AOSD is used to separate crosscutting concerns and to add new functionality, making the code more robust and functional.

We have identified and then implemented four production aspects: Synchronization, Display, Validation and Control. Synchronization and Display concerns resulted in tangled modules and scattered code in pure Object Oriented implementation, while Validation and Control concerns were entirely missing.

We used AOSD to enhance the understandability of our code by implementing crosscutting concerns as aspects. Moreover, implementing crosscutting concerns as aspects made our code easier to modify and maintain. We also added some functionality which would complicate the code too much without the use of AOSD.

While we enhanced our program with the help of AOP, it can be developed further. We can model device failures stochastically, to increase the reality of the simulation. Concerns like energy consumption can be implemented in the future. We can implement other propagation models. We can simulate the actual network traffic once the MST is built. This additional functionality could help our program to be a generic network simulator.

References

[1] R.L. Graham and R. Hell, "On the History of the Minimum Spanning Tree Problem," *Annals of the History of Computing, IEEE*, vol. 7, no. 1, pp. 43-57, 1985.

[2] J. B. Kruskal, "On the shortest spanning subtree of a graph and the traveling salesman problem," in *Proceedings of the American Mathematical Society* 7, 1956, pp. 48-50.

[3] R.C. Prim, "Shortest connection networks and some generalizations," *Bell System Technical Journal*, vol. 36, p. 1389-1401, 1957.

[4] R. G. Gallager, P. A. Humblet, and P. M. Spira, "A Distributed Algorithm for Minimum-Weight Spanning Trees," *ACM Trans. Program. Lang. Syst.*, vol. 5, no. 1, pp. 66-77, 1983.

[5] S. Pemmaraju, "22C:21: Computer Science II: Data Structures,"
<http://www.cs.uiowa.edu/~sriram/21/fall07/project2.html>

[6] A. Bulbul, O. F. Uzar, and U. O. Yilmaz, "Distributed Minimum Spanning Tree Construction in Wireless Networks," 2009.

[7] M. Koksal, "A Survey of Network Simulators Supporting Wireless Networks," 2008.

[8] J-sim Official.
<http://sites.google.com/site/jsimofficial/>

[9] OMNeT++ Community Site.
<http://www.omnetpp.org/>

[10] The Network Simulator - ns-2.
<http://www.isi.edu/nsnam/ns/>

[11] OPNET Technologies, Inc.
<http://www.opnet.com/>