Software Security using Aspect-Oriented Software Development

Dr. Buthainah F. AL-Dulaimi Buthynna@yahoo.com
Dr. Afaf Badea Al-Kaddo Afafbad@yahoo.com
Amer Abdulmejeed Abdulrehman amer6567@yahoo.com
College of Education for Women - Computer Dept.

Abstract

Aspect-Oriented Software Development (AOSD) is a technology that helps achieving better Separation of Concern (SOC) by providing mechanisms to identify all relevant points in a program at which aspectual adaptations need to take place. This paper introduces a banking application using of AOSD with security concern in information hiding.

Keywords: Aspect-oriented, AOSD, Security Concern

أمن البرمجيات باستخدام تعديل الهيئة لتطوير البرمجيات

د. بثينة فهران عبد د. عفاف بديع قدو عامر عبدالمجيد عبدالرحمن
كلية التربية للبنات - قسم الحاسبات

المستلخص

تساعد تكنولوجيا تعديل الهيئة على تحقيق افضل فصل في Soc, عن طريق توفير اليات لتحديد جميع النقاط المترابطة المطلوب اجراء تعديلات على هياتها او اتجاهها. اعتمدت هذه الورقة البحثية تطبيق التكنولوجيا المقترحة على نظام مصرفي باستخدام AOSD باعتماد الامنية في اخفاء المعلومات.

1.Introduction

Aspect-Oriented Software Development (AOSD) is a new approach to software design that addresses modularity problems that are not handled well by other approaches, including Structured Programming and Object-Oriented Programming (OOP). AOSD complements, but doesn't replace those approaches. An aspect is a module that encapsulates a concern. In some approaches, an aspect may also contain classes and methods. The focus of Aspect-Oriented Software Development (AOSD) is in the investigation and implementation of new structures for software modularity that provide support for explicit abstractions to modularize concerns. These modularized concerns are called aspects, and aspect-oriented approaches provide methods to compose them. Some approaches denote a root concern as the base. Various approaches provide different flexibility with respect to composition of aspects. AOSD allows multiple concerns to be expressed separately and automatically unified into working systems [1,2,3,4].

Typical enterprise and internet applications today have to address "concerns" like security and the subsystems that provide these services can be implemented in a modular way. However, to use these services, you must insert the same code fragments into various places in the rest of the application to invoke these services. This compromises the overall system modularity, because the same invocation code is scattered throughout the application. For example, if we want to control access to certain services in the application, we could insert authorization-checking at the beginning of every method that needs this control. It would now be difficult and error prone to modify or replace this security approach later. Also, your application code is now tangled with a code for the security (and probably other) concerns, which both compromises clarity and makes it hard to reuse your code in another context. Since you would have to drag along the same security approach, which may not be appropriate. Because concerns like security typically cut across a number of application module boundaries, called cross-cutting concerns. [1, 2, 3, 4]

2. The Purpose for using AOSD

It stem from the problems caused by code scattering and tangling. The purpose of Aspect-Oriented Software Development is to provide systematic means to modularize crosscutting concerns. The implementation of a concern is scattered if its code is spread out over multiple modules. The concern affects the implementation of multiple modules. Its implementation is not modular. The implementation of a concern is tangled if its code is intermixed with code that implements other concerns. The module in which tangling occurs is not cohesive. Scattering and tangling often go together, even though they are different concepts [1,2,3,4].

Aspect-oriented software development considers that code scattering and tangling are the symptoms of crosscutting concerns. Crosscutting concerns cannot be modularized using the decomposition mechanisms of the language (object or procedures) because they inherently follow different decomposition rules. The implementation and integration of these concerns with the primary functional decomposition of the system causes code tangling and scattering [1,2,3,4].

Scattering and tangling of behavior are the symptoms that the implementation of a concern is not well modularized. A concern that is not modularized does not exhibit a well-defined interface. The interactions between the implementation of the concern and the modules of the system are not explicitly declared. They are encoded implicitly through the dependencies and interactions between fragments of code that implement the concern and the implementation of other modules. The lack of interfaces between the implementation of crosscutting concerns and the implementation of the modules of the system impedes the development, the evolution and the maintenance of the system. [1, 2, 3, 4]

3.System Development

A module is primarily a unit of independent development. It can be implemented to a large extent independently of other modules. Modularity is achieved through the definition of well-defined interfaces between segments of the system. The lack of explicit interfaces between crosscutting concerns and the modules obtained through the functional decomposition of the system imply that the implementation of these concerns, as well as the responsibility with respect to the correct implementation of these concerns, cannot be assigned to independent development teams. This responsibility has to be shared among different developers that work on the implementation of different modules of the system and have to integrate the crosscutting concern with the module behavior. Furthermore, modules whose implementation is tangled with crosscutting concerns are hard to reuse in different contexts. Crosscutting impedes reuse of components. The lack of interfaces between crosscutting concerns and other modules makes it hard to represent and reason about the overall architecture of a system. As the concern is not modularized, the interactions between the concern and the top-level components of the system are hard to represent explicitly. Hence, these concerns become hard to reason about because the dependencies between crosscutting concerns and components are not specified[1,2].

Finally, concerns that are not modularized are hard to test in isolation. The dependencies of the concern with respect to behavior of other modules are not declared explicitly. Hence, the implementation of unit test for such concerns requires knowledge about the implementation of many modules in the system. An aspect-oriented approach supports the implementation of concerns and how to compose those independently implemented concerns[3,4].

4.Aspect-Oriented Requirement Engineering

Aspect-oriented requirement Engineering focuses on the identification, specification and representation of crosscutting properties (, mobility, availability and real-time constraints) at the requirement level. Crosscutting properties are requirements, use cases or features that have a broadly scoped effect on other requirements or architecture components. Aspect-oriented requirements engineering approaches are techniques that explicitly recognize the importance of clearly addressing both functional and non-functional crosscutting concerns in addition to non-crosscutting ones. Therefore, these approaches focus on systematically and modularly treating, reasoning about, composing and subsequently tracing crosscutting functional and non-functional concerns via suitable abstraction, representation and composition mechanisms tailored to the requirements engineering domain[3].

5.Aspect-oriented system architecture

Aspect-oriented system architecture focuses on the localization and specification of crosscutting concerns in architectural designs. Crosscutting concerns that appear at the architectural level cannot be modularized by redefining the software architecture using conventional architectural abstractions. Aspect-oriented architecture starts from the observation that we need to identify, specify and evaluate aspects explicitly at the architecture design level to redesign a given architecture in which the architectural aspects are made explicit.

6.Aspect-Oriented Modeling and Design

Aspect-oriented design has the same objectives as any software design activity, i.e. characterizing and specifying the behavior and structure of the software system. Its unique contribution to software design lies in the fact that concerns that are necessarily scattered and tangled in more traditional approaches can be modularized. The process takes as input requirements and produces a design model. The produced design model represents separate concerns and their relationships[4].

7.System Maintenance and Evolution

The lack of support for the modular implementation of crosscutting concerns is especially problematic when the implementation of this concern needs to be modified. The comprehension of the implementation of a crosscutting concern requires the inspection of the implementation of all the modules with which it interacts. Hence, modifications of the system that affect the implementation of crosscutting concern require a manual inspection of all the locations in the code that are relevant to the crosscutting concern. The system maintainer must find and correctly update a variety of poorly identified situations[3].

8.Discussion

Traditional software development focuses on decomposing systems into units of primary functionality, while recognizing that there are other issues of concern that do not fit well into the primary decomposition. The traditional development process leaves it to the programmers to code modules corresponding to the primary functionality and to make sure that all other issues of concern are addressed in the code wherever appropriate. Programmers need to keep in mind all the things that need to be done, how to deal with each issue, the problems associated with the possible interactions, and the execution of the right behavior at the right time. These concerns span multiple primary functional units within the application, and often result in serious problems faced during application development and maintenance. The distribution of the code for realizing a concern becomes especially critical as the requirements for that concern evolve — a system maintainer must find and correctly update a variety of situations. Aspect-Oriented Software Development focuses on the identification, specification and representation of cross-cutting concerns and their modularization into separate functional units as well as their automated composition into a working system.

9.**Bank Account Application Security using AOSD**

Aspect-oriented programming entails breaking down program logic into distinct parts (so-called *concerns*, cohesive areas of functionality). Nearly all programming paradigms support some level of grouping and encapsulation of concerns into separate, independent entities by providing abstractions (e.g., procedures, modules, classes, methods) that can be used for implementing, abstracting and composing these concerns. But some concerns defy these forms of implementation and are called *crosscutting concerns* because they "cut across" multiple abstractions in a program.An aspect can also make binary-compatible structural changes to other classes, like adding members or parents. For example, consider a banking application with a conceptually very simple method for transferring an amount from one account to another.

```
void transfer(Account fromAcc, Account toAcc, int amount) throws Exception {
  if (fromAcc.getBalance() < amount) {
    throw new InsufficientFundsException();  }
   fromAcc.withdraw(amount);
  toAcc.deposit(amount);}
```

However, this transfer method overlooks certain considerations that a deployed application would require: it lacks security checks to verify that the current user has the authorization to perform this operation; a database transaction should encapsulate the operation in order to prevent accidental data loss; for diagnostics, the operation should be logged to the system log, etc.
A version with all those new concerns, could look somewhat like this:

```
void transfer(Account fromAcc, Account toAcc, int amount, User user, logger logger)
  throws Exception {
  logger.info("Transferring money...");
  if (! checkUserPermission(user)){
    logger.info("User has no permission.");
    throw new UnauthorizedUserException();  }
  if (fromAcc.getBalance() < amount) {
    logger.info("Insufficient funds.");
    throw new InsufficientFundsException();  }
   fromAcc.withdraw(amount);
  toAcc.deposit(amount);
   //get database connection
   //save transactions
   logger.info("Successful transaction.");}
```

In this example other interests have become *tangled* with the basic functionality (sometimes called the *business logic concern*). Transactions, security, and logging all exemplify *cross-cutting concerns*. Now consider what happens if we suddenly need to change (for example) the security considerations for the application. In the program's current version, security-related operations appear *scattered* across numerous methods, and such a change would require a major effort. AOP attempts to solve this problem by allowing the

programmer to express cross-cutting concerns in stand-alone modules called *aspects*. Aspects can contain *advice* (code joined to specified points in the program) and *inter-type declarations* (structural members added to other classes). For example, a security module can include advice that performs a security check before accessing a bank account. The pointcut defines the times (join points) when one can access a bank account, and the code in the advice body defines how the security check is implemented. That way, both the check and the places can be maintained in one place. Further, a good pointcut can anticipate later program changes, so if another developer creates a new method to access the bank account, the advice will apply to the new method when it executes.
So for the above example implementing logging in an aspect:

```
aspect Logger {
      void Bank.transfer(Account fromAcc, Account toAcc, int amount, User user, Logger
logger)  {
            logger.info("Transferring money...");       }
       void Bank.getMoneyBack(User user, int transactionId, Logger logger)  {
            logger.info("User requested money back.");       }
      // other crosscutting code...}
```

All accounts have some basic things in common: there will be a current balance, an interest rate, and ways of depositing and withdrawing money. A normal savings account might only have these features, but other types of account are likely to need to have further information. For example, a current (cheque) account usually has an overdraft limit, and an associated interest rate for when the account is overdrawn; a fixed deposit will have a termination date when the money will be paid out; etc. A generic bank account class might look something as in figure(1):

Account
balance, accountNumber interestRate
deposit, withdraw

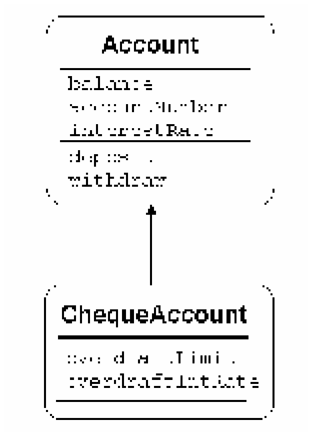
Figure(1): Generic Bank Account

The withdraw method, which should really check to make sure that there is enough money in the account. If we now consider a cheque account this will need exactly the same information as the generic Account class, plus additional data members for the overdraft limit and overdraft interest rate figure (2):

ChequeAccount
balance, accountNumber interestRate, overdraftLimit overdraftIntRate
deposit, withdraw

Figure(2): Cheque Account with Additional Information

The diagrams are always used to show the relationships between classes that are related by inheritance. For our simple example, the inheritance hierarchy would be as shown in the figure(3).



Figure(3): The Inheritance Hierarchy

10. The Class Hierarchy for Account and ChequeAccount

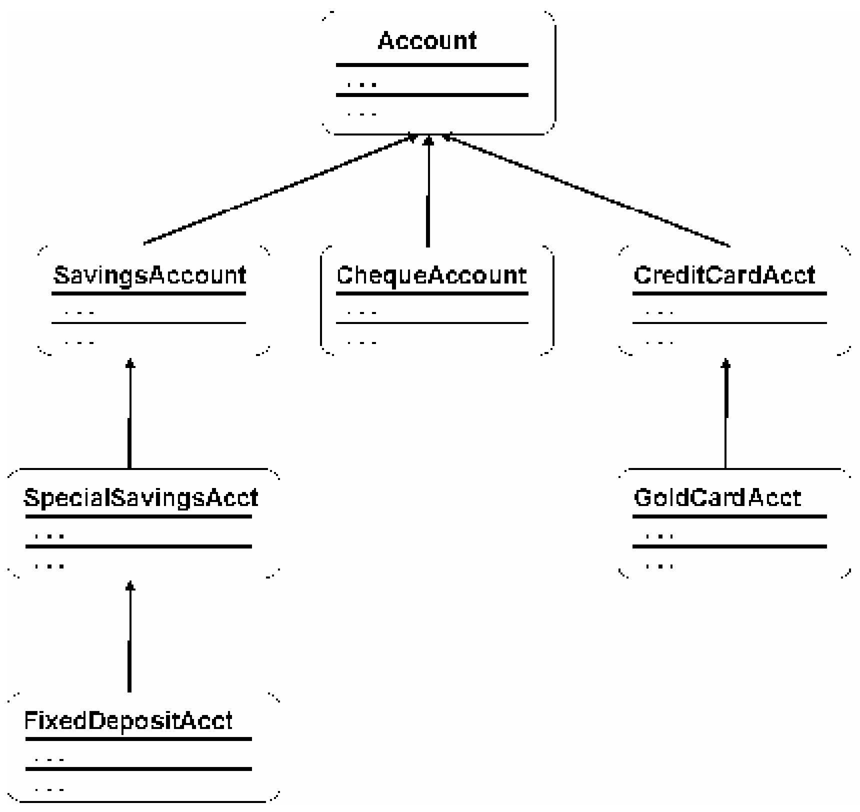


Figure (4): Class Hierarchy for Account and Cheque Account

11.<u>Conclusions</u>

This paper introduced the notion of AOSD. It argues that AOSD further enhance the modularization of models by enabling encapsulation of crosscutting concerns. However, full integration of AOSD in the system development is effective through supporting the life cycle of the system. Particularly, there is a need for extending composition rules, providing tool support, and automatically analyzing requirements and their aspect-oriented implementations for the solution space.

References
[1] S. Clarke and E. Baniassad. Aspect-Oriented Analysis and Design: The Theme Approach. Addison Wesley, 2005.
[2] J. Kienzle, W. AlAbed, and J. Klein. Aspect-oriented multi-view modeling. In ACM, editor, AOSD'09, Charlotteville, Virginia, USA, March 2009.
[3] J. Klein and J. Kienzle. Reusable Aspect Models. In 11th Aspect-Oriented Modeling Workshop, September 2007.
[4] N. Loughran and A. Rashid. Framed Aspects: Supporting Variability and Configurability for AOP. In International Conference on Software Reuse (ICSR-8), Springer, 2004.