Technical Report No: 2003/13

Evolving Legacy System Security Concerns Using Aspects

Robin C. Laney

Janet van der Linden

Pete Thomas

11th November 2003

Department of Computing
Faculty of Mathematics and Computing
The Open University
Walton Hall,
Milton Keynes
MK7 6AA
United Kingdom

http://computing.open.ac.uk

Evolving Legacy System Security Concerns Using Aspects

Robin C. Laney
Dept. of Computing,
The Open University, UK
+44 (0)1908 654342
r.c.laney@open.ac.uk

Janet van der Linden
Dept. of Computing,
The Open University, UK
+44 (0)1908 652985
j.vanderlinden@open.ac.uk

Pete Thomas
Dept. of Computing,
The Open University, UK
+44 (0)1908 652695
p.g.thomas@open.ac.uk

ABSTRACT

This paper shows how aspects can be successfully employed in the support of system evolution. The context is a case study on migrating a legacy client-server application to overcome the security problems associated with 'message tampering' attacks. The focus is on authorization issues in which aspects are used to add a security mechanism based on digital signatures. The approach provides for future evolution of the system. In particular, it is shown how factoring of aspectual concerns allows the scope of the security boundary to be varied, illustrating reuse of the aspects.

Whilst the aspects are added non-intrusively, it is demonstrated how aspects can modify the control-flow behaviour of a server. An extension to AspectJ's exception mechanism that conforms to design by contract is proposed to facilitate this form of aspect.

Categories and Subject Descriptors

D.3.3 [**Programming Languages**]: Language Contructs and Features

General Terms

Design, Experimentation, Security, Standardization, Languages

Keywords

Aspects, security, legacy system

1 INTRODUCTION

In this paper we describe how a legacy software system was evolved using aspects. The aim of the investigation was to add certain security features to an existing system, written in Java, in a non-intrusive manner. That is, the additional features were to be added without changing the existing code (or its libraries) in any way: a prima facie case for the application of aspects. However, the investigation identified a number of issues that had to be overcome and led us to some insights into aspect oriented programming and revealed the need for additional expressiveness in the aspect language we were using (AspectJ).

A number of researchers [4, 14] have identified reasons why aspects are a good mechanism for improving security in software systems:

- Security is a concern that is located throughout the entire application and is therefore difficult to get right. Techniques that help to control and reduce this pervasiveness should be exploited.
- Security experts will be able to better focus their efforts on security aspects if these are concentrated and separated from other concerns.

In [4] it is discussed how security requirements can be viewed through a number of *how*, *what*, *where* and *when* questions. In the past, there have been successful attempts to factor out some of the *what* and *how* questions. That is, w*hat* and *how* can be addressed, for example, through the use of libraries, for cryptography, or through a service such as JAAS, the Java Authentication and Authorization Service. However, the question of *where*, for example, should each security concern be implemented at the beginning of each method, or only in some methods? and the question of *when*, for example, should a concern be addressed when some application condition is satisfied or not? are both still tangled with the application. This is because the calls to the security libraries or services have to be made from within the application. Currently, aspects are beginning to address some of these *where* and *when* issues. For example, [10] discusses how to make calls to JAAS using aspects.

In our work we have used aspects to implement authorization requirements based on digital signatures. We were able not only to separate the lower level mechanism of implementing a digital signature, but also to factor out the policy of where we want this security concern to be applied.

The use of aspects in software development has been divided into two categories [2, 9]: development and production aspects. Development aspects can be used during the development of applications to facilitate debugging, testing and performance tuning, and are not part of the final build of the application. In our work we made us of logging aspects to assist in debugging. Production aspects are intended to be included in the build of an application, and provide additional functionality for the application. There is evidence that production aspects are helpful in factoring the design of green-field developments [1, 8], as well as for existing systems [3, 5]. In this paper we explore their application in the evolution of a legacy system, under the strict constraint that no changes are to be made intrusively.

Our work is based on the use of a case study encompassing multiple servers in a prototype home banking application. The existing system is secure to the degree that users must log-on before accessing an account. However, messages between clients and servers are not signed, exposing the system to attacks caused by message tampering. Our initial goal in this work was to add digital signatures to the system to enhance the authorization mechanisms. We focused on authorization issues simply to clarify the separation of concern issues. Having achieved this and solved some interesting but unexpected problems concerning the degree of interaction between an aspect and the legacy code, we turned our attention to future system evolution. We give a specific illustration of evolving the system, in the light of revised security requirements involving a transfer of responsibility for signing and checking messages between servers. We discuss the benefits of well designed aspects that are reusable in system evolution.

The paper is organized as follows. In section 2, we discuss related work and present relevant details of the case study. In section 3, we specify our initial additional security requirements and discuss the design and implementation necessary to meet them. In particular, we describe the difficulties of developing aspects that must modify control flow behaviour and how modest enhancements to the exception mechanism used would be of assistance in solving this problem. Furthermore we show how our aspects were factored to enhance separation of concerns. In section 4, we introduce a requirement to move the security boundary and show how the factoring of our aspectual security code smoothes the evolution of the system. Section 5 draws conclusions and outlines future work.

2 BACKGROUND AND RELATED WORK

2.1 Related Work

Security is increasingly becoming an area that attracts the attention of the AOSD community. In many ways security represents an ideal candidate for the separation of concerns, because it permeates all areas of software systems. Security would thus benefit from techniques that help to structure and modularize it.

One example of using security aspects is to modularize access control. In [5] authentication is separated from the application, here an FTP server. Aspects were used to perform the task of checking username/password combinations, while other aspects represented the authenticated user thus holding important security information for the duration of an FTP session.

A different approach to modularizing access control is presented in [10]. Laddad shows how JAAS, the Java Authentication and Authorization Service can be added to applications using aspects, rather than making calls to this service from the application code.

Security issues can also relate to the notion of secure code: many programming languages have features that can easily be used in such a way as to leave security flaws. In [14] Viega, Bloch and Chandra discuss how aspects can help developers to be consistent in their implementation of security policies by, for example, checking that calls to the SecurityManager are included where needed, replacing all calls to non-secure functions by calls to secure functions, providing buffer overflow protection or logging data that may be relevant to security.

An alternative approach to implementing non-functional requirements is the use of reflection language systems/architectures. In particular Welch and Stroud [15] present a portable approach based on their Kava meta-object protocol. They present a case study in which a security system based on proxies and inheritance is re-implemented using Kava. This results in better enforcement of security but the use of reflection does raise issues of efficiency. Their approach is portable as it is not dependent on specialized language systems or architectures as is the case with most other work on reflection.

Other work has looked at the factorization of middleware software [16], based on a mining approach to discover scattered code in legacy systems that can be refactored as aspects. This is in contrast to the work presented in this paper where we are adding functionality.

2.2 The Case Study

The case study was developed as a tool for illustrating concepts in concurrent programming and for illustrating software engineering techniques. As such, the software has been continually amended as new requirements were specified.

The system was based on the following scenario. A bank wished to provide its existing customers with the ability to perform a number of different banking transactions using their home computers. The bank started this development from the point at which a customer already had one or more accounts at one or more of its branches and each branch had a computer-based banking system to maintain its accounts. In addition, the branch computers communicated with a central machine at the bank's headquarters. The architecture of the home banking system is shown in Figure 1.

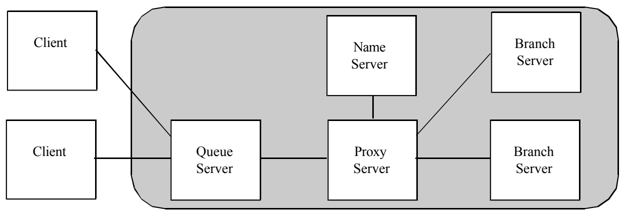

Figure 1 The home banking system

A Branch Server is an existing branch system: there are many branches. The Client represents a customer's machine. Between a Client and its Branch Server are three items of middleware. The Queue Server interfaces with all Clients and queues customer requests before passing them on to the Proxy Server. One of its purposes is to smooth out the flow of requests to the Proxy Server. The Proxy Server acts as a router, using the customer's account number to route the request to the appropriate branch. The Proxy Server uses the facilities of the Name Server to determine the IP address of the appropriate Branch Server. The home banking system was designed so that each of the servers could be hosted on separate machines. Indeed, there is provision for installing multiple Proxy Servers should the loading on a single server become too high. It was envisaged that the Queue, Proxy and Name Servers would be housed at the bank's headquarters. The shaded area illustrates that the Queue Server acts as an interface between the Clients and the Servers and deals with some security concerns.

Client requests are in the form of asynchronous messages packaged as strings constructed according to a proprietary protocol known to each server. Every message contains the customer's identifier and account number. Responses to client requests are also messages conforming to a specific protocol and are routed back from the Branch Servers via the Proxy and Queue Servers to the Client. The Queue Server is responsible for forwarding a response to the appropriate client.

In the initial prototype implementation, security is minimal: authentication is performed by the Branch Server using a single password. Client requests are rejected by the Queue Server unless the client has successfully logged-in. If the Queue Server receives a login request message it passes it directly to the appropriate Branch Server for authentication. If the login is successful, the Queue Server will accept all future requests from that Client until the client ends the session.

3 Adding Digital Signatures

By adding a mechanism for signing messages using digital signatures we are able to reduce greatly the susceptibility of our system to 'message tampering' attacks. Our focus in this research is on authorization: the signing and checking of messages, ignoring the separate issue of authentication. Aspects are relevant here as an implementation technique, since we need to interact with the message passing code that crosscuts the system. For reasons of traceability from requirements through design to implementation, it is important that we relate aspects structurally and functionally to the requirements. As we show below, these details emerge iteratively with design and implementation of the aspectual details. These issues, which we explore further below suggest that the use of early aspects i.e. at the requirements stage [6, 13], exploited within the context of a twin-peaks approach [12] aid the development of secure code.

3.1 Relating Requirements and Aspects

The requirement to improve authorization security through the use of digital signatures entails:

(1) Generating and checking signatures at appropriate places, that is, at a security boundary, and storing user keys at sites where the signing of messages and the checking of signatures takes place.

(2) Modifying message handling to allow incorporation of signatures in messages. In this system, the Queue Server, Proxy Server and Branch Server have legitimate reasons for viewing (some of) the contents of a message and therefore have to unmarshall and marshall the message.

The initial security boundary, that is the perimeter of an area encompassing those parts of the system that are to be secured, included all the servers and parts of the clients. The shaded area in Figure 2 illustrates this. Note that the aspects are not required to ensure security of the whole of the client. Prior to signing messages, security is necessarily the responsibility of some other combination of software/hardware.

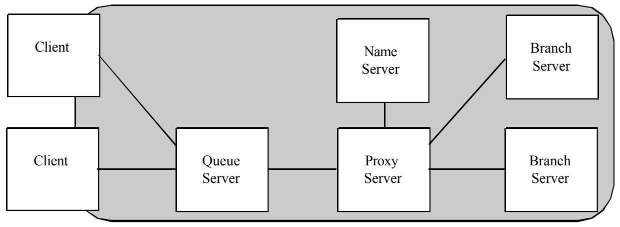

Figure 2: Banking application with security boundary

Our initial design involved two aspects crosscutting the two requirements (in addition to crosscutting within, and as it emerged, across servers). The aspects were a security aspect A_{SC} responsible for generating and marshalling signatures in the client and an aspect A_{SS} responsible for unmarshalling and checking signatures in the server.

In order to remove this crosscutting of the requirements, the aspects were factored as follows:

A_G	Aspect to generate a digital signature
A_C	Aspect to check a digital signature
A_M	Aspect to ensure signature is marshalled
A_U	Aspect to ensure a signature is unmarshalled

This allows us to factor the aspects across the system in the following manner. Aspects A_G and A_M are used at the 'signing' server, and aspects A_U and A_C at the 'checking' server. This factoring also allowed us to remove a degree of crosscutting with the original requirements. Our aspects now divide into two forms: those concerned with message manipulation (a low level mechanism) and those concerned with details specific to digital signatures (i.e. operating at more of a policy level). This distinction is significant when considering system evolution, as we discuss in section 4.

3.2 Aspect Details

Here we present the aspects that were developed to implement the digital signatures for our authentication service. In this discussion, we leave aside the issue of the distribution of the keys, and concentrate on some of the problems we encountered during this development and what we see as a possible shortcoming in the AspectJ tool.

On the *client* side, aspects A_M and A_G were implemented as shown in Figures 3 and 4 respectively. We defined a new class DigSig with a number of methods to handle messages and their digital signatures (this is not shown here). Aspect A_M defines a **pointcut** which intercepts the moment that messages are about to be sent to the server. The banking system communicates between its components using sockets, and the messages are sent as Strings using the method println(). The **around** advice ensures that marshalling takes place, using the pack() method from DigSig.

```
public aspect AM {

pointcut marshal (String anArg): call(* *.println(..))
                                          && (args(anArg));

void around (String oldArg): marshal(oldArg) {
               String newArg = DigSig.pack (oldArg, null);
               proceed (newArg);
          }
}
```

Figure 3 Aspect A_M

A separate aspect A_G has its **pointcut** defined on the call to pack() within the **around** advice of the aspect A_M. This ensures that an appropriate signature will be generated and passed on as a parameter to the pack() method of aspect A_M.

```
public aspect AG {

pointcut generateSig (String message, String signature):
                   within (AspectM)
                   && call(* DigSig.pack(..))
                   && (args(message, signature));

String around (String message,String signature):
   generateSig (message,signature){
   return (proceed (message,(DigSig.CreateDigSig(message))));
   }
}
```

Figure 4 Aspect A_G

On the *server* side it is the Queue Server where messages from the clients arrive and which must implement the checking activity. Within the Queue Server code is a class named QueueServerThread which extends Java's Thread class. In Java, the code that a Thread object executes must be implemented in an argument-less public void method named run(). The basic design of the run() method in QueueServerThread is shown in Figure 5. There is one thread for each client session.

The run() method is called from the start() method invoked by the constructor of QueueServerThread. The run() method contains a number of **try-catch** statements, as well as loop control statements, **break** and **continue**, which make it difficult to follow the flow of the program. This clearly illustrates why it would be best not to tamper with this application!

```
1 public void run() {
2   open client sockets for communication
3   while (true)  //serve for duration of client session
4     try {
5           read a client request message
6           if (message is not in expected format){
7              send Error message to client
8              continue
9           }//end if
10          if (request is a login command) {
11             try{
12                   forward client details to proxy server
13                   get response from proxy server
14                   if (response == succesfull login){
15                      log client account number for this session
16                   }
17                   forward response to client
18             } catch (Exception e1)
19          }// end if
20          else
21             add message to queue
22     } catch (Exception e2){break}
23  }// end while
24  close sockets
25 } //end run
```

Figure 5 Pseudo code for the run method in QueueServerThread

The point at which a client request enters the Queue Server is found on line 5. This represents a good join point to intercept the incoming message. Immediately the message arrives it is validated (lines 6 to 9) and, if found to be invalid, a message is sent to the client and the processing of the message stops – the thread returns to the beginning of the while loop to obtain the next client message.

Figure 6 shows the aspect A_U which takes care of the unmarshalling. It uses **around** advice, which first calls proceed() in order to obtain the result of the readline() statement, it then converts this using the unpack() method from DigSig, and returns the modified message to the program.

Finally, there is an aspect to check whether the digital signature is correct. Ideally, aspect A_C should be defined as follows*: 'if during the run() method it is discovered that there is an intruder, then stop execution of the run() method for that client session*'. The idea being that the discovery of an intruder is sufficiently serious that we would not want it to continue processing, and contacting other servers with potentially harmful data.

We resolved to define aspect A_C as presented in Figure 7.

```
public aspect AU{

pointcut unmarshal() : call (* *.readLine(..));

String around ()  : unmarshal() {
            String signedMessage = proceed ();
            String unsignedMessage =
                        DigSig.unpack(signedMessage);
            return (unsignedMessage);
            }
}
```

Figure 6 Aspect A_U

```
public aspect AC {

  pointcut checkSig (String uncheckedMessage):
                        within(AspectAU)
                        && call(* DigSig.unpack(..))
                        && (args (uncheckedMessage));

  before (String uncheckedMessage):
                        checkSig(uncheckedMessage) {
    if (!DigSig.IsCorrectSig(uncheckedMessage)) {
      printWriter.println("E*Aspect intruder");
      //any other code to raise alarm
      throw (new SecurityException());
    }//end if
  }
}
```

Figure 7[1] Aspect A_C

Notice that we defined a **pointcut** for A_C in the **around** advice of the aspect A_U as it does the unmarshalling, similar to the way aspect A_G was constructed on top of aspect A_M. Aspect A_C works as follows: it retrieves the unchecked message and ensures that it is checked through the DigSig class. If the result of this test is false, a message is sent to the client indicating that an intruder has been intercepted, and a SecurityException is thrown. It is this throwing of the exception which will ensure that the program will effectively abandon the run() method. This is because the exception thrown in the aspect will be caught on line 22 of the run() method which has the call to **break** to handle the exception.

An important feature of these aspects is their minimal coupling with the application code. In fact, the design of the aspects conforms to a more general situation where messages are transmitted asynchronously between nodes and are required to be validated on receipt.

[1] We have not shown the details of how to obtain access to the local variable, printWriter, representing the output stream to the client.

3.3. Altering the flow of program and extension to AspectJ

In aspect A_C we deliberately allow the aspect to throw an exception, in order to cause the program to stop executing the method it was in (here, run()). Throwing an exception in an aspect, thus causing the original program to change its course in such a dramatic way, appears drastic and against the spirit of AOSD. Before we resorted to this action a number of different strategies were explored:

- Rewrite the run() method entirely, and make sure that the new version is called instead of the original one. This is not possible, because the call to run is hidden within the start() method of the library Thread class. In addition, the complexity of the existing run() method is such that the risk of failing to re-implement it correctly is high.
- Alter the values of local variables in the run() method, for example setting the message to **null**, and then picking this up when the format of the message is checked (see line 6 in Figure 5). However, this strategy would tie our aspect a lot closer to the actual application code, making it less maintainable and less re-usable.

Given that the nature of the aspects is to improve the level of security in our application, and in particular to remove the danger of tampering with messages going into the bank system, there is some justification for the more serious action we chose.

However, this proposal violates the two aspect style rules identified by Kersten and Murphy [8]:

- do not allow exceptions introduced by a weave to percolate out from the weave, and
- before and after advice should not alter the pre- and post-conditions of a method.

They argue that these two rules together ensure that the application of an aspect to a class will not affect how the class fits into the existing class structure, nor will it modify the contracts between client and supplier methods in the existing class structure, which in turn makes understanding, debugging and testing of the application much easier.

In aspect A_C the exception is thrown deliberately to ‘percolate out from the weave’. Had the exception been caught in the aspect, the desired effect would have been lost.

A much preferred option would have been to use a dedicated aspectException, that is, an exception which can only be thrown by aspects. Assuming that such an exception would be a subclass of Exception it could still be caught within the run() method. Therefore, we suggest an extension to the current AspectJ tool, in the form of an exception as described above, in order to facilitate managed interactions between an aspect and the control-flow of legacy code.

With respect to the second style rule, it is an interesting question whether the pre- and post conditions of the run() method are affected by the exception thrown in the aspect. We argue that the pre-condition is not affected, because the design of the run() method is such that no assumptions are made about the validity of the message string it is designed to work with. The post-condition is in fact strengthened by the behavior of the aspect, because the original post-conditions of the method (i.e. those before introducing aspects) still hold, and an additional one has been introduced. This is permissible under subcontracting [11] and hence indicates that we still operate according to the principles of Design by Contract.

The above suggests that there should be some modification to the Kersten and Murphy aspect style rules, to cater for a structured 'percolation' that still adheres to the principles of Design by Contract.

4. System Evolution

In reality it may well be that some of the servers in the banking system, those towards the back-end, will be adequately protected by other mechanisms than our digital signatures, including physical isolation, private networks, and proprietary security protocols. Additionally some of the servers towards the client end might be to some degree outside the scope of the bank's own security domains, for example, networks of ATM's run by affiliated organizations. As businesses evolve in the context of internal restructuring, mergers, and loose federal arrangements with heterogeneous organizations (such as large supermarket chains), the security boundary we established in Section 3 may be required to shift across machine boundaries.

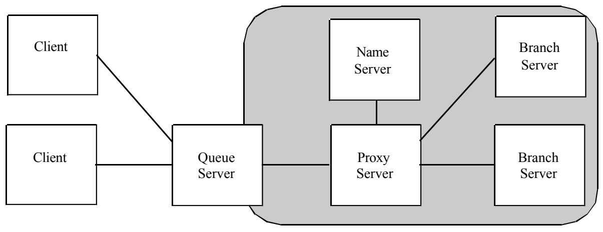

Figure 8: Banking application server structure with amended security boundary (now Proxy defines the boundary)

In Figure 8 we show one possible new machine boundary, where the Proxy Server now defines the point where clients interface to the banking system. The solution presented in section 3 is able to gracefully accommodate this change. The reason for this lies in the structuring of our aspects. The low level mechanism on which we have built security – aspects to marshal and unmarshal digital signatures – is independent of the aspects that generate and check digital signatures. This supports moving of our boundary: Aspects A_M and A_G both move to the Queue Server, and aspects A_U and A_C move to the Proxy Server.

During the development of this scenario an intermediate state may be required to test the system. This intermediate state could look as follows: the Bank Client with aspects to marshal and generate (A_M and A_G); the Proxy Server with aspects for unmarshalling and checking (A_U and A_C); and the Queue Server in between the two with aspects for marshalling and unmarshalling as well as generating signatures (A_U, A_M and A_G).

Furthermore, the structuring of the aspects means we can flexibly accommodate changes in security policy such as new algorithms, changes in key length and so on. Such changes in policy would be encapsulated in A_G and A_C with corresponding changes to A_M and A_U being easily deduced.

Finally, in Section 3 it was noted that the definition of the aspects displays minimal coupling with the application code. In this paper we have shown how they can be used to add digital signatures to messages under scenarios with differing security boundaries. In addition, given the generality of the aspects, particularly those defined for marshalling and unmarshalling, they will be able to provide support to scenarios entirely outside the realm of security. Applications with significant IO functionality can benefit from being able to easily validate or otherwise modify and enhance the input and output arguments by using the aspects.

5. Conclusions and Future Work

The contribution of this paper is in showing how aspects can be used to evolve legacy code. We have shown that taking one step of evolution (digital signatures), through an aspects approach, enhances future evolutions in a number of dimensions (moving the security boundary, changes to security policy and during the development phase). We have also shown how a server's control flow behaviour can be modified through the use of aspects, and that this does not necessarily break the two aspect style rules as defined in [8]. We propose an extension of the AspectJ exception mechanism that conforms to Design by Contract to allow for more flexible integration of aspects with legacy code.

Since our aspects relate to a single overall concern – authorization – their relationship is one of cooperation, rather than conflict. In a more realistic scenario, system evolution would be likely to involve simultaneous work across a number of cross-cutting concerns with a high likelihood of conflicting requirements. An approach to resolving such conflicts is given in [13]. Our approach to partitioning the aspects relating to a concern should allow conflict resolution to operate at a finer granularity: for example, replacing our signing algorithm with a faster one can be done independently of the marshalling aspects.

In future work we plan to investigate implementing further additions of functionality to our system. The aim being to explore a wider range of interaction patterns between aspects and legacy code, and also between the aspects themselves. From this we would hope to be able to identify some analysis and design patterns to encapsulate our approach in a general way. This will allow us to give further consideration to the question of whether there should be some modification to the Kersten and Murphy aspect style rules, to cater for a structured 'percolation' that still adheres to the principles of Design by Contract.

This work is a part of a larger programme of work on security, and in particular on security requirements. In contrast to the work presented here, which concerns itself with securing an implementation, we are also working on specifying security requirements independently of implementation [7].

6. ACKNOWLEDGMENTS

We gratefully acknowledge the input of our colleagues Charles Haley and Bashar Nuseibeh in the Department of Computing, The Open University, for their thought provoking discussions.

REFERENCES

[1] Aaltonen T., Helin J., Katara M., Kellomaki P., Mikkonen. T. Coordinating Aspects and Objects. Electronic Notes in Theoretical Computer Science, **68** No 3, Elsevier Science, 2003.

[2] AspectJ Team. The AspectJ Programming Guide. http://eclipse.org/aspectj

[3] Coady Y., Kiczales G. Back to the future: A Retroactive Study of Aspect Evolution in Operating System Code. In Proceedings of AOSD03, Boston, MA, 2003.

[4] De Win B., Piessens F., Joosen W., Verhanneman T. On the importance of the separation-of-concerns principle in secure software engineering. Workshop on the Application of Engineering Principles to System Security Design, WAEPSSD, Boston, MA, USA, November 6-8, 2002, Applied Computer Security Associates (ACSA).

[5] De Win B., Joosen W. and Piessens F. AOSD & Security: a practical assessment. Workshop on Software engineering Properties of Languages for Aspect Technologies (SPLAT03), 2003, 1-6.

[6] Grundy J. Aspect-Oriented Requirements Engineering for Component-based software systems. *In Fourth IEEE International Symposium on Requirements Engineering (RE'99).* Limerick,Ireland: IEEE computer Society Press, 7-11 Jun 1999

[7] Haley C.B., Laney R.C., Moffett J.D., Nuseibeh B.A, Using Trust Assumptions in Security Requirements Engineering. Second Internal iTrust Worshop on Trust Management in Dynamic Open Systems, Imperial College, London UK, l-17 Sep 2003.

[8] Kersten M.A., Murphy G.C. Atlas: A Case Study in Building a Web-based Learning Environment using Aspect-oriented Programming. Technical Report Number TR-99-04, Department of Computer Science, University of British Columbia, Canada, 1999.

[9] Kiczales G., Hilsdale E., Hugunin J., Kersten M., Palm J., Griswold W. G. Getting Started with AspectJ. Communications of the ACM. October 2001, Vol. 44, No.10, 59-65.

[10] Laddad R. AspectJ in Action. Manning, Greenwich, 2003.

[11] Meyer B. (1997) Object-Oriented Software Construction. Prentice Hall International [2nd edition, ISBN 0-13-629155-4]

[12] Nuseibeh B.A., “Weaving Together Requirements and Architecture”, IEEE Computer 34 (3):115-117, March 2001.

[13] Rashid A., Moreira A.M.D., Araujo J. Modularisation and Composition of Aspectual Requirements. In Proceedings of AOSD03 2003, Boston, MA, 2003.

[14] Viega J., Bloch J.T., Chandra P. Applying Aspect-Oriented Programming to Security. Cutter IT Journal, **14**, No.2, February 2001, 31-39.

[15] Welch I.S. and Stroud R.J. Re-engineering Security as a Crosscutting Concern. Computer J., **46** (5), 578-589.

[16] Zhang C., Jacobsen H. Quantifying Aspects in Middleware Platform. In Proceedings of AOSD03, Boston, MA, 2003.