PART 2:

CHAPTER 5:
ASPECT-ORIENTED PROGRAMMING

CHAPTER 6:
ASPECT-ORIENTED SECURITY

CHAPTER 5:

ASPECT-ORIENTED PROGRAMMING

5.1 Introduction

Aspect-oriented programming was first proposed by Gregor Kiczales and others at Xerox PARC in 1997. They developed it to offset redundancy in programs, thereby reducing complexity. Redundancies usually become apparent in areas such as security, memory management, resource sharing, and error and failure handling (Miller, 2001). Using aspect-oriented programming to address these redundancies is beneficial, as it has the potential to improve the reliability, maintainability, reusability (Viega and Voas, 2000) and robustness of an application. Although aspect-oriented programming is not ubiquitous in industry, it is receiving considerable attention from research and practitioner communities such as IBM, Northeastern University in the United States, the University of Twente in the Netherlands and Xerox (Miller, 2001). More recently, aspect-oriented software development has been successfully applied by Motorola, Siemens and Hewlett-Packard (Pohl et al., 2008). The field has matured to such an extent that it generated its own conference, and the first International Conference on Aspect-Oriented Programming took place in Twente in the Netherlands in 2002.

The object-orientated paradigm was found to be inadequate in terms of design and the implementation of cross-cutting concerns, as there is no elegant way of modularising such concerns. Aspect-oriented programming provides explicit language support for modularising design decisions that cross-cut a functionally decomposed program (Walker et al., 1999). That is, the developer is able to maintain the code (cross-cutting functionality) in a

modularised form. It is important to note that aspect-orientation maintains all the benefits of the object-oriented paradigm and should be viewed as an extension rather than a replacement of object-oriented technologies.

This chapter will commence with a discussion on the evolution of the different programming paradigms, after which the concept of aspect-oriented programming will be introduced. The next elaboration will cover the difficulties to be encountered with regard to the integration of aspect-oriented programming into software development.

5.2 Evolution to Aspect-Oriented Programming

Programming languages have evolved from assembly languages in the 1950s, to procedure-oriented languages in the 1960s, followed by structured programming and data abstraction in the 1970s. The next innovation involved object-oriented, distributed functional, and relational paradigms in the 1980s (Wegner, 1990). The sections that follow briefly describe the evolution of the aspect-oriented paradigm.

Structured programming enabled developers to provide modularity and reusability of constructs like procedures and functions(Constantinides and Hasson, 2002). However, as the size of software products increased, the structured programming paradigm was found to be inadequate and maintenance was becoming increasingly problematic. The basic tenets of the object-oriented programming paradigm, which embraced encapsulation and reuse, were viewed as a means for improving the quality and maintainability of software. The other advantage that the object-oriented paradigm had over the structured paradigm was the notion that it promoted thinking about software in a way that closely modelled the way humans perceive and interact with the real world. However, the object-oriented paradigm has not yet solved the problem of the duplication of code scattered throughout different objects. With object-oriented programming the attempt to reduce duplication was class inheritance. However, the problem is not related to the concept of inheritance. Instead, it is about unrelated objects that share some points of commonality (Padayachee and Eloff, 2006). The design and implementation of cross-cutting concerns still pose a problem to

object-oriented programming, as there is no elegant way of modularising these concerns. Aspect-oriented programming (AOP) addresses this problem and provides explicit language support for modularising design decisions that cross-cut a functionally decomposed program (Walker et al., 1999). In other words, the developer is able to maintain the code (cross-cutting functionality) in a modularised form.

Aspect-oriented programming is designed to exploit some of the advantages of object-oriented programming such as functionality, encapsulation, hierarchal classes and modularity. At the same time it manages to overcome an important disadvantage: objects cannot solve the problem of concerns that are not confined to a single class (Miller, 2001) . These concerns result in the 'tangling-of-aspects phenomenon' (Kiczales, 1996), which inevitably results in a system that is difficult to maintain (Raje et al., 2001). Aspect-oriented programming advocates abstracting these cross-cutting concerns into modular units called aspects. For instance, suppose a system had one cross-cutting concern, scattered throughout the system. Suppose also that this cross-cutting concern can be removed from the system and codified as a single separate aspect. Thereafter this aspect is woven into the system at specific points when required, thereby simplifying the code significantly and also promoting more effective coding of these aspects. Furthermore, an update can be done on the single entity or aspect, rather than searching across the whole system and modifying the cross-cutting concern several times (Padayachee and Eloff, 2006).

As with aspect-oriented programming, subject-oriented programming is also based on the separation-of-concerns principle. In the subject-oriented paradigm, applications are composed of subjects. The subject-oriented paradigm defines the state and behaviour pertinent to the application itself – usually as fragments of the state and behaviour of collections of relevant classes (Harrison and Ossher, 1993). Subject-oriented programming also addresses some of the object-oriented programming limitations such as non-invasive system extensions and system decomposition. The main difference between aspect-oriented programming and subject-oriented programming is that aspect-oriented programming achieves non-invasive system extension through intercepting base code at

join points, whereas subject-oriented programming achieves system extension through the development of composition rules needed to integrate existing components.

5.3 Aspect-Oriented Programming Terminology

Aspect-oriented languages have three critical elements: a join point model, a means of identifying join points and a means of affecting implementation at these join points (Kiczales et al., 2001). There are two categories of aspects: development and production aspects. Development aspects can be used during the development of an application to facilitate debugging, testing and performance tuning, and they are not part of the final build of the application. Production aspects are intended to be included in the build of an application and they provide additional functionality for the application.

Here follows brief definitions of terminology used in aspect-oriented programming extracted from Kiczales et al. (2001) and Kizcales et al. (1997):

Cross-cutting

Behaviour that cannot be encapsulated. Because of its impact across the whole system, it is called cross-cutting behaviour.

Join Points

Join points are certain well-defined points in the execution flow of a program.

Pointcut

A pointcut is a set of join points described by a pointcut expression. The pointcut is used to find a set of join points where aspect code would be inserted.

Advices

Advice declarations are used to define additional code that runs at join points. For example, AspectJ supports before and after advice, depending on the time the code is executed. 'Before' (after) advice on a method execution defines code to be run before (after) the

particular method is actually executed. 'Around' advice defines code which is executed when the join point is reached and has control over whether the computation at the join point (i.e. an application method) is allowed to execute.

Aspect

An aspect is a modular unit of a cross-cutting implementation that is provided in terms of pointcuts and advices, specifying what (advice) and when (pointcut) its code is going to be executed. In terms of codification, aspects are similar to objects.

Aspect Weaver

The final application is generated by incorporating cross-cutting concerns into a final executable form and invoking a special tool called an aspect weaver. Figure 5-1 illustrates the concept of weaving in AspectJ where aspects and normal Java code are woven together and compiled into Java byte code.

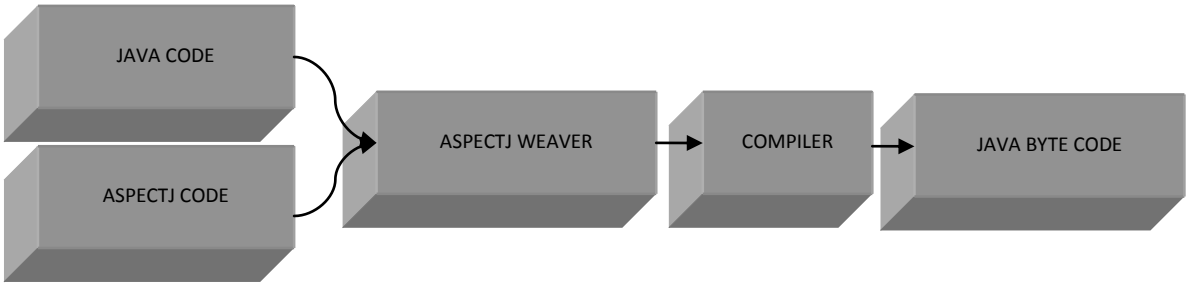

Figure 5-1: Illustration of the Weaving Concept

5.4 AOP Frameworks

The frameworks that are most widely known and used are AspectJ, AspectWerkz, JBoss AOP, and Spring AOP. The frameworks that are less popular are abc, aspect#, AspectC++, and JAC (Kersten, 2005; Wakaba, 2004)

AspectJ

AspectJ is an extension of the Java language syntax and semantics and provides its own set of keywords for developing aspects. In addition to containing fields and methods, AspectJ's aspect declaration contains pointcut and advice members

AspectWerkz

AspectWerkz, JBoss AOP and Spring AOP add aspect semantics without changing the Java language syntax. AspectWerkz provides two ways of making AOP declarations: Annotations and JavaDoc style declarations.

JBoss AOP

JBoss AOP has an XML-based aspect declaration style – aspect, pointcut and advice declarations are made in XML. Advice is implemented using plain Java methods that are invoked by the JBoss AOP framework.

Spring AOP

Spring AOP also uses an XML-based aspect declaration style, and similar to JBoss AOP, advices are implemented in a Java method with special parameters that are invoked by the Spring framework.

AspectJ was selected to develop the model concept presented in this thesis. It was assumed that as it is the most mature from all other aspect-oriented languages, it would be more thoroughly tested than other recent aspect-oriented compilers. Furthermore, since the AspectJ is more evolved than the other languages, it has a richer tool support. For example, the Eclipse platform which an integrated development environment, provides a mechanism to visualise an aspect-oriented system graphically.

5.5 Evaluating Aspect-Oriented Programming

Although aspect-oriented programming introduces an elegant implementation of separation of concerns, it does have its challenges. According to Murphy et al. (1999), the aspect-oriented approach makes it easier to reason about and develop 'certain kinds of application code'; this implies that aspect-oriented programming cannot be applied in every situation. According to Padayachee and Eloff (2006), aspect-oriented programming is not only inappropriate in the programming of small-scale systems, but several cross-cutting concerns will have to be identified across a system to warrant the use of aspect-oriented technology. As this is a new technology, problems are to be expected in terms of testing and debugging the code. To start with, Alexander and Bieman (2002) maintain that as a result of the weaving process, isolating faults will be difficult as they may reside in the source code, aspect or woven code. Secondly, aspects being applied to pointcuts can interfere with performance and predictability of the program behaviour (Läufer et al., 2003).

Another challenge, as noted by Chen (2004) is that of understandability. A many-to-many relationship may exist between aspects and the primary abstractions they integrate with, which would potentially require an understanding of many other aspects to understand only one. According to Chen (2004), a complication may arise when several different authors each writes a collection of aspects to be woven. Each programmer must be au fait with the set of primary abstractions that his/her aspects can be woven with. This implies that each programmer must know about the other aspects that are used, either by direct composition or indirectly as a result of weaving. All of these activities will inevitably place a greater cognitive burden on aspect authors.

The advantages gained from using aspect-oriented programming include increased modularity, reduced development time, increased maintainability, improved reusability, more flexibility and simpler class hierarchies (Elrad et al., 2001). In addition to the benefits of reduced complexity and improved maintainability of software, programmers may be better able to understand an aspect-oriented program when the effect of aspect code has a well-defined scope(Walker et al., 1999). The presence of aspect code may also alter the

strategies that programmers use to address tasks perceived to be associated with aspect code (Walker et al., 1999).

As with all technology, several disadvantages have been identified despite all the great advantages. As aspect-oriented programming is a new technology, some issues are still unresolved. For instance, Alexander and Bieman (2002) pose several pertinent questions regarding aspect-oriented technology: 'How do we measure the complexity that results from the weaving process?'; 'Can we control or minimise the cognitive distance induced by the weaving process?'; 'How do we maintain aspect-oriented programs?' and 'How do we effectively test and analyse aspect-oriented programs?' Furthermore, Baniassad and Clarke (2004) and Clarke (2002) assert that identifying cross-cutting behaviour is difficult as it is entangled with other behaviours.

Murphy et al. (2001) on the other hand, pose questions providing several opportunities for research, such as: 'Does aspect-oriented programming work for large, multi-developer projects?'; 'To what kinds of problems is it best suited?' and 'What kinds of constructs are most usable for specifying crosscuts?'. Based on their empirical study, Murphy and his fellow researchers argue that aspect-oriented programming shows clear 'promise', but there is still much to learn about it. This implies that there is a definite need to carry out more evaluative studies on aspect-oriented programming. This need will be addressed in Chapter 8, where the aspect-oriented implementation of the model will be assessed against the object-oriented implementation.

Providing aspect-oriented solutions to an access control problem may provide greater insights into the paradigm and allow the access control problem to be viewed from a different perspective. Incorporating an aspect-oriented approach may also involve a shift in philosophy, as researchers and developers may have to discover innovative ways of maximising the positives of the technology by rethinking the access control in an alternative way. Just as the move from structured programming to object-oriented programming involved readdressing access control implementations, so too will the shift from object-oriented programming to aspect-oriented programming.

5.6 Conclusion

This chapter provided a brief overview of the evolution towards the aspect-oriented paradigm. The paradigm's goals, strengths and weaknesses were identified and briefly contrasted to those of the object-oriented paradigm. The object-oriented paradigm is, however, here to stay and it will not be replaced by aspect-oriented programming. In fact, aspect-oriented programming is merely a next step in the evolution of object-oriented technology and a refinement of object-oriented technology. Despite all the difficulties acknowledged and outlined in this chapter, aspect-oriented programming has been touted as a way to resolve security issues. Aspect-orientation is viewed as a better context in which to implement security concerns efficiently. The relationship between security and aspect-oriented programming will be explored in more detail in the next chapter.

CHAPTER 6:

ASPECT-ORIENTED SECURITY

6.1 Introduction

Security is a constant and pervasive concern in software systems. A major cause of this fact is the structural difference between application logic and security logic. A significant amount of work has been done in aspect-oriented security to warrant making the process more systematic in terms of software design and development.

According to Bodkin (2004) aspect-oriented security design 'is relevant for all the major pillars of security: authentication, access control, integrity, non-repudiation as well as the supporting administration and monitoring disciplines required for effective security'. For example, Alexander and Bieman (2002) state that 'code that implements a particular security policy would have to be distributed across a set of classes and methods that are responsible for enforcing the policy. However, with aspect-oriented technology, the code implementing the security policy could be factored out from all classes into an aspect.' Accordingly, the code that affects the implementation of multiple classes and methods is localised in one cohesive place, namely an aspect.

In addition to the dimension of abstraction offered by aspect-oriented programming, it also facilitates the implementation of additional security features so as to constitute a fully operational software system.

This ease of extensibility has the following advantages:

- It allows for better separation of concern and therefore better division of labour between application developers and security engineers.
- It also allows security to be added in a more agile manner since it is not necessary to consider it during requirements and specification phases.
- Using aspect-orientation to enforce security policy at compile time is also advantageous as it is a lot more efficient than code reviews (Boström, 2004).
- Crosscutting concerns can be added or removed without making invasive modifications to original programs (Ubayashi et al., 2004).

6.2 Aspect-oriented programming and its application to security

The discussion that follows highlights the relevance of aspect-oriented technology in terms of implementing some of the major pillars of security (access control and authentication, accountability and audit, data protection and information flow controls) in software systems.

6.2.1 Access Control and Authentication

Access control is a fundamental part of computer security where every requested access must be governed by an access policy stating who is allowed access to what. The request must then be mediated by an access policy enforcement agent (Pfleeger and Pfleeger, 2003). The seminal work in this area was conducted by De Win et al. (2002), who actually generalised the aspects they developed for access control to promote the reusability of these aspects. In an earlier publication (2001) they delineated three types of aspects, namely **Identification**, **Authentication** and **Authorisation** for access control in the aspect-oriented paradigm.

Program Listing 6-1: Generalised Aspect Code for Access Control

```
abstract aspect Identification of eachobject(entities()){
abstract pointcut entities();
public Subject subject null;
}
abstract aspect Authentication of eachcflowroot(authenticationCall()){
      private Subject subect;
      abstract pointcut serviceRequest();
      .......
}
abstract aspect Authorization{
      abstract pointcut checkedMethods();
      .....
}
```

The **Identification** aspect is used to tag the entities that must be authenticated, and as a container for identity information of the subject. The subject (see Program Listing 6-1) included in the aspect is used to determine whether access should be allowed or not. The **Authentication** aspect passes authentication information to the access control mechanism, while the **Authorisation** aspect checks the access based on the identity information received through the Authentication aspect.

Slowikowski and Zieliński (2003) considered how aspect-oriented security could enhance container-managed security and also demonstrated how identification, authentication and access control may be applied to components without modifying the source code. They concluded that aspect-oriented security required no modification of the application's source code to introduce security and that the procedure was highly flexible and extensible. The significance of their research is that if access policies are unknown or vague, access control features may be implemented after the development of other requirements or when these policies have been defined more clearly. Furthermore, the abstraction of access control policies eases security management and development significantly, as security experts may be allocated specifically to the development of these features.

According to Padayachee and Wakaba (2007), the aspect-orientated paradigm's versatility in terms of access control measures has been further validated by studies conducted within differing approaches to access control. The paradigm has been leveraged to implement discretionary access control (see De Win et al. (2002)), role-based access control (see Pavlich-Mariscal et al. (2005)) and mandatory access control (see Ramachandran et al. (2006) and Padayachee and Eloff (2007)).

6.2.2 Accountability and Audit

Accountability and audit serve to collect and analyse the activity of an information system. They aim at detecting security violations and defining causes, which may also be easily implemented with aspects (Slowikowski and Zielinski, 2003). In Program Listing 6-2 below, Slowikowski and Zieliński (2003) go on to demonstrate that an aspect could keep a log of exceptions thrown from a specific component – without modifying the component.

Program Listing 6-2: Demonstrating Accountability and Auditing with Aspect-Orientation

```
aspect BankAspect{
      pointcut bankMethods():
execution (public *bank.*(..)) && this (SessionBean);
//Log information after throwing BankSecurityException from SessionBeans
// which belong to the bank package
      after() throwing (BankSecurityException e): bankMethods(){
            Log log = Log.getInstance();
            log.write(e);
      }
}
```

6.2.3 Cryptographic Controls

In an experiment conducted by Boström (2004), it was found that by using aspect-oriented programming, database encryption could be added after the initial system had been completed. This case study showed that using aspect-oriented programming resulted in better modularity, database independence and less code. However, in certain instances the logic developers could not be totally alienated from the process of encryption, because the development of the functionality sometimes depended on the encryption process.

6.2.4 Information Flow Controls

There is some duality between access control and information flow control as both mechanisms are concerned with the flow of information. However, information flow control is more than access control, as an illegal flow may occur even when only authorised requests are performed on an object. As such, most access control models are supplemented with some form of information flow control.

Masuhara and Kawauchi (2003) found that although sanitising was a cross-cutting concern, there was no possible way to define a pointcut that would be able to detect whether a string was from an unauthorised source or not, or whether it contained unwanted information. Hence, they proposed a new pointcut called *dflow* that addresses the dataflow between join points as an extension to the AspectJ Language. Recall that join points are well-defined points where calls to aspect code would be inserted. The authors did not address security classifications and their dataflow definition only dealt with direct information flow. As no studies have been performed exclusively on this area, and seeing that aspect-oriented programming is an evolution in object-oriented programming, it would be pragmatic to investigate information flow control from this context as well (Padayachee and Wakaba, 2007).

6.2.5 Protection from invasive software

Aspect-oriented programming has been used to implement software tampering detection mechanisms in applications running on untrusted hosts (Falcarin et al., 2004). This involved the use of aspect-oriented programming to realise self-checking, a process where a program checks itself to verify that it has not been modified. In terms of evolving verification techniques as security threats change, using the separation-of-concerns principles makes it easier to 'swap in and out and evaluate alternative treatment options' (Houmb et al., 2004)

6.2.6 Security kernels

A security kernel is responsible for enforcing the security mechanisms of the entire operating system (Pfleeger and Pfleeger, 2003). Engel and Freisleben (2005) developed a tool for deploying dynamic aspects in the kernel space of an operating system. They determined that dynamic aspect-oriented programming was suitable in this area owing to the changing requirements, internal conditions and cross-cutting functionality of security kernels, and also found that the performance impact of using aspect-orientation was negligible in most instances.

6.2.7 Verification

Kumar et al. (2001) developed a framework that uses the aspect-oriented programming paradigm to verify that commercial off-the-shelf components are developed as per security contracts. Validating a component's performance, resource usage, transaction support, security, data persistency and distribution, are concerns that cross-cut a system's components – hence the aspect-oriented paradigm is appropriate for these purposes. Furthermore, as verification procedures tend to be similar in most applications, using the aspect-oriented paradigm facilitates the reusability of these validation measures (Grundy and Ding, 2002) via abstraction.

As discussed above, the aspect-oriented paradigm is revolutionising security implementations. It allows for security implementations to be adapted to changing needs more rapidly, and this is critical, given that security needs are impermanent. Furthermore, security solutions can seldom cater for all possible violations and these are usually discovered retrospectively. The added value of treating security as a separable concern is that it promotes reusability. This is significant, as the solution developed for the model concept presented in this research may require only slight modification for reuse in other information security systems in view of the fact that security aspects tend to be generic

6.3 Conclusion

In this chapter, the relevance of aspect-oriented programming to information security was explored and it became evident that it was in fact relevant to all pillars of security. Security concerns must inform every phase of software development, from the requirements phase through to the design, implementation and deployment (Devanbu and Stubblebine, 2000). During systems development, security requirements may be vague or they may change, and they may well be considered only as an afterthought after the system has been deployed. The extensibility of the aspect-oriented paradigm allows for these concerns to be implemented after other requirements have been developed. Furthermore, if a security concern has to be maintained due to the discovery of new security threats in the environment, the separation of concerns would fundamentally simplify this task. Within other programming paradigms, it may result in several fixes across several components, thus resulting in inconsistencies and regression faults.

The prototype that was designed to test the proof-of-concept of the model will be presented in Chapter 7 while Chapter 8 deals with its implementation, using the aspect-oriented approach. As the model is intended to augment traditional access controls, it is posited that the aspect-oriented approach will be highly suitable to fulfil this need without compromising the integrity of the system as a whole.