Chapter 3

Aspect-Oriented Programming in Model Driven Development

In this chapter, Aspect-Oriented Programming (AOP) has been explored in context of Model Driven Development, where crosscutting concerns are inevitable in software development. Some situations are identified where there are some crosscutting concerns. These crosscutting concerns are encoded in aspect modules for implementation. This has been obtained by proposing a security module, which have some crosscutting concerns. Few state-chart diagrams and class diagrams are designed to depict the security system. Designing meta-models for the security aspects has been also advocated. These meta-models can be easily transformed to source code using suitable tools. The conversion of a meta-model to source code supports program transformation which is one of the major activities of MDD. This transformation is a case of refactoring which is required for upgradation of old models to new models. A case study has been made to demonstrate, that certain modules need AOP approach for enhancing the MDD techniques.

3.1 Overview of AOP and Crosscutting Concerns

The complexity of software development and maintenance has increased over past few decades. The reason behind the increase of complexity is not only development of large-scale software but also to cater to a demand for more sophisticated software. Hence, for sophisticated small or large-scale applications, modular approaches have been developed, where the system is broken up into internally coherent modules, each of which offers an external narrow interface [39].
The implementation of modules and interfaces are provided by the programming languages in different ways. The modular structures were first implemented in *procedural languages*. *Object-oriented languages* further modularized code into objects which encapsulate data and operations upon it. It also offers a narrow interface. The third approach is *component-oriented*. This deals with independent modular units with well defined interfaces. It can be composed and integrated into a generic architecture that encapsulates larger units which can be deployed and reused individually [40].

All these approaches are directed towards modularizing the software. However, the sophistication increases from procedure-oriented to component-oriented. These all are separating the concerns with more abstractions in component-oriented rather than from object-oriented and procedure-oriented. However, all concerns are not well modularized since they overlap with other modules. Hence such concerns which crosscut the concerns of other modules are known as *crosscutting concerns*.

Isolation of a number of concerns into modules often mean that other concerns cannot be placed in a module of its own, but have to be distributed over the existing modules. The code related to these concerns is then scattered, which means that it is distributed over several modules and not localized in a single one. The code is also tangled, which means that it is mixed with the code of other concerns. Such small code fragments which are scattered through several functional components are also known to be tangled codes. These concerns are identified as distinct concerns but they cannot be encapsulated as a generalized module in procedure-oriented or object-oriented programming [37].

### 3.1.1 Crosscutting Concerns and their Impact in Software Development

Crosscutting concerns are those concerns, which, due to interactions with other concerns cannot be encapsulated properly. Such concerns are scattered throughout the modules of a software system and contradict and violate the principles of separation of concerns and modularity. So, there is a need to identify the nature of crosscutting
concerns. Many kinds of concerns do not align with the modularizations and hence they end up as scattered, tangled and replicated code.

- **Code Scattering** [9, 37, 41] refers to a concern implementation that is scattered across many other concerns within a module. This is a threat to modularity and information hiding.

- **Code Tangling** [97] is the interlacing of the codes within the same unit of the system design. This is due to the interactions among the different concerns, where the elements of code for two concerns are in the same unit and cannot be dissociated.

- **Code Replication** [34, 98] occurs when a concern interacts with multiple concerns and when all interactions are implemented identically.

Hence it can be said that scattered, tangled and replicated codes are three types of crosscutting concerns. Scattered and tangled codes come into existence due to improper decomposition of software modules. This motivates to modularize the software with higher level of abstraction than in Object-Oriented Programming. Scattered and tangled codes decrease reusability, maintainability and customizability since the concerns become coupled and also violate the principle of separation of concerns [9, 37]. It has been observed that replicated codes [42-43] occur due to copy and paste of code fragments which reduces the software maintainability.

The above classification is made on the basis of the code representations on software development and maintenance. On the contrary, it has been demonstrated that crosscutting concerns are classified in two different dimensions on the basis of crosscut structure or program structure [10, 44] as follows.

(i) A crosscut structure can be homogeneous or heterogeneous.

(ii) Concerns can crosscut the static structure or the dynamic structure of a program.

A homogeneous crosscut extends a program at multiple joinpoints by adding one extension which is a modular piece of code [45]. A heterogeneous crosscut extends multiple joinpoints by adding multiple extensions where each individual extension is implemented by a distinct piece of code, which affects exactly one joinpoint [45]. So
the difference is that all the joinpoints affected by heterogeneous crosscuts are having different codes which are not for homogeneous crosscuts.

Crosscutting concerns are again classified from the perspective of compile time or runtime and are known as static or dynamic crosscutting concern respectively. This classification is based on the implementation of crosscutting concerns. A static crosscutting concern adds new classes and interfaces etc. and a dynamic crosscut affects the runtime control flow of a program which can be realized in an event-based model [46, 47].

3.2 Role of AOP in Modularizing Crosscutting Concerns

All the classifications are done in different perspectives. But the main objective is to segregate the identified concerns into separate units and to execute according to the situation. AOP addresses the problems caused by crosscutting concerns in the following ways.

(a) Decomposition of the software into modules which contains the concerns that do not crosscut through a host programming language.

(b) Identification of the concerns that crosscut the implementation of other concerns and encapsulating and implementing these concerns into modules known as aspects.

3.2.1 AOP Implementations

In software development, modules are obtained by designing the crosscutting and non-crosscutting concerns. So, code-modules (non-crosscutting concerns) and aspects are designed separately. Both the code and aspects are combined into a final executable form using the aspect weaver. As a result, a single aspect can contribute to the implementation of a number of methods, modules or objects, increasing both reusability and maintainability of the code. Figure 3.1 expresses the weaving process.
AOP is a concept and hence is not bound to a specific programming language. AOP concepts have been implemented in different languages like C++, Java, Smalltalk etc. AspectC++ and AspectJ are the extension of C++ and Java respectively where both provide the compiler and the aspect weaver. Though these softwares are available but they are in the development process to enhance C++ or Java with AOP concepts. AOP allows weaving the crosscutting concerns into the applications, rather than having explicit calls in the code.

### 3.2.2. Identification of Crosscutting Concern Situations

AOP helps to understand the relationships among the different concerns and identifies the crosscutting concerns. It supports software development at analysis and design levels. At the analysis level, it is carried out by first identifying a set of actions in the requirements list. In the design level the set of actions as crosscutting or non-crosscutting behaviors are segregated.

In general, it has been observed that there are certain areas in which crosscutting concerns are inevitable whether it is in a large-scale or in a small-scale application. Authentication, authorization, caching, communication, configuration management, exception management, logging and instrumentation, state management, validation etc. are few situations where crosscutting concerns occur. Logging is a homogeneous

---

**Figure: 3.1** Aspect Weaver
crosscut as it is the same in different points whereas authentication, authorization, exception management are heterogeneous crosscutting concerns since the codes are different for different joinpoints.

### 3.2.3 Significance of AOP in MDD

Modular software development needs to identify the crosscutting and non-crosscutting concerns. This can be obtained in the following ways.

a) In requirement analysis phase, one can identify the activities required

b) In design phase, one can differentiate the activities as crosscutting and non-crosscutting concerns

In order to demonstrate the above concepts, it is required to develop a module, using MDD. This MDD technique uses AOP as a part of the development. It is aspired to develop such modules that cater to handling of a linear data structure like array and any product line application. AOP can play a significant role by restricting the users of software or product line. As triggers are designed for database security, similarly, a security module has been designed for a software product line. This module can be triggered before, after or during the handling of the software. The following are the two different concerns for the present module.

a) The software development for array related activities

b) Design of the security module, to restrict and monitor the software handling

The former is the non-crosscutting concerns whereas the later is the crosscutting concerns. The crosscutting concerns need to be activated in the following situations.

a) Whenever any access to the software is required

b) Whenever messages are to be generated according to the users

The advantage of this designing is that the security module can be integrated to any type of software accessibility. Though the security module has been designed as a part of the core software development but this security module can be used for any other existing software accessibility. The portability of the security module is an added advantage to the existing MDD techniques. Hence, the designing of the crosscutting and non-crosscutting concerns are presented.
The designing of the non-crosscutting concerns will be demonstrated in later chapters. The next section provides the designing of the crosscutting concern activity.

### 3.3 Proposed Security Module

Security is a prime concern for any software handling. Software developers need to consider security concerns while making architectural, logical and physical designing. Very few concepts and approaches have emerged that help and guide developers to integrate security into software. This motivates researchers to design security modules in different ways like security design patterns, secure coding, security code injection using aspect-oriented programming etc. [108]. On analyzing some existing security modules the following have been observed.

(a) A framework has been proposed in to integrate security into applications by providing plans, patterns and aspects for different security issues and problems [108].

(b) Aspect-oriented approach has been used in [75] called Role Models, which is a technique for modelling and integrating security concerns into designs.

All the existing techniques for designing security modules are capable to design security modules only for large-scale software. Moreover, no such security module has been developed which can be used for small-scale software product line or for any product line. There is thus a motivation in the present work to design a security module using aspects as it has a crosscutting behavior. This module can be integrated to any product line.

Login module has been chosen in this thesis as a crosscutting concern for several reasons. It is called as a security module because it will include few authorization components which will be responsible for detecting the right user. A part of communication component will take care of throwing suitable messages to the user. It is aspired to design the security concern in such a way that it customizes software in a very interactive mode. This interactive customized software can be obtained neatly by encapsulating the authorization and communicating messages into aspects.

This reveals that aspects are essential from refraining crosscutting concerns. So, the traditional communication concern done in OOP has to be modified into AOP. This
security module is a general aspect module and can be integrated into large or small-scale applications. It needs AOP because certain message communications are imposed whenever software is used for accessibility. This is done through before, after and/or around advice. OOP lacks this facility.

In the proposed scheme, a security system has been designed, with appropriate message generation for accessing any software. In order to access the software, two types of users are categorized, *end-users* and *developers*. The end-users are entitled to handle the software as a tool and the developers have the accessibility of the source code of the software. The two basic functions of the software are the query method and the update method.

The security module will perform as follows.

(i) The end-users visit the system through query() method and uses the software.
(ii) Developers log in the system through update() method and perform software modifications.
(iii) Before visiting the system, it verifies user’s authenticity. For a valid user, appropriate message is generated and the user is permitted for the action. Otherwise, after giving a suitable message, it returns to the initiated state.
(iv) Before and after the query()/update() operation, a log function is triggered.

### 3.3.1 Designing of Concerns in Proposed Security Module

In the proposed security module, it is now needed to identify the non-crosscutting concerns and crosscutting concerns which are termed as the core (functional) concerns and the non-functional concerns respectively. The core concerns are a part of the software but the non-functional concerns are the part of the security activities. Since the security activities will be required frequently as and when the software is used, they will be placed in several parts of the program. This demands the codes to be encapsulated in a unit called aspects. The advantage of this aspectual unit is that it can be triggered before or after or at a certain point when an event is taking place. In the present work, security module is to be triggered whenever there is an attempt to access the software. Hence, the security activities are crosscutting concerns and need to be used in aspects.

The identified core and non-functional concerns are the following.
(i) Functional concerns: query and update
(ii) Non-functional concerns: access and log

The non-functional concerns are specified as aspect classes in which communication modules are integrated. The access-verifying mode is required by most functional components and hence in a traditional approach it will lead to a tangled situation. After the introduction of AOP, all components do not need to examine the authority of the user. However, an operation will be automatically triggered when required. This enhances the security of the system and reduces the tangled structure of the system. Login is important for the security of the software as it logs the user according to their status.

3.3.2 Composing Concerns through Class and State-Chart Diagrams

Since access and log are used for both functional (core) concerns, query and update, so the two aspect classes, access and log crosscut the method of query() and update(). Again, message communication is required for the access aspect. So, the three activities i.e. access, log, and message communication are combined into a single unit known as the security aspect (Figure 3.2). When any user visits the software by query() or update(), security aspect is triggered which then checks the authority. For a valid user, an appropriate message is generated which performs the log operation. Otherwise, the process is terminated by displaying a suitable message. The class diagram of the proposed scheme is depicted in Figure 3.2.

![Class Diagram of Security Aspect](image_url)
In the class diagram, crosscutting attributes are separated and modeled separately. However, the class diagram does not express the executing process. So, the dynamic model of the security system is given in the state-chart diagram (Figure 3.3).

![State-chart diagram of Security Aspect](image)

**Figure 3.3:** State-chart Diagram of Security Aspect

The various states of the state-chart diagram of the security aspect are.

- **S1:** Idle state,
- **S2:** Query state,
- **S3:** Checking validity for query,
- **S4:** Update state, and
- **S5:** Checking validity for update.

The state S1 is the initial state, where the both types of users (end-users and developers) start their accessing process of the software. The S2 state is for end-users and S3 state is for developers. So, when an end-user accesses the software, there is a transition from S1 state (idle state) to S2 state (Query state). Similarly, the developers accessing the software make a transition from S1 state to S4 state (Update state).

The transition from S2 to S3 (Validity checking for query) is to check the identity of the end-user. In case of a valid user, there is a transition back from S3 to S2, whereas for an invalid user the transition is from S3 to S1, i.e. back to the
idle state. Similarly, for developers the transition from S4 to S5 (Validity checking for update) is to check the authenticity of the developers. The transition from S5 to S4 implies a genuine entry whereas the transition from S5 to S1, back to the idle state, is for unidentified user.

The state-chart diagram of query class is now provided in Figure 3.4, where Q1 denotes the starting state for an end-user and Q2 denotes the query state.

![State-chart diagram of query class]

**Figure 3.4:** State-chart diagram of query class

The state-diagram of Figure 3.4 is for the detail transition between S2 to S3. The user’s request to access the software is the state Q1 and for evaluation of user’s identity it changes to S3 state. Q2 state performs the query only for valid user and returns back to Q1. On the other hand, for an invalid user S3 state changes back to S1.

The following state-chart diagram is for update, as shown in Figure 3.5, where U1 denotes the starting state for the developers and U2 denotes update state.
Figure 3.5: State-chart diagram of update class

Figure 3.5 states that U1 state is for developers who request to make an entry for accessing the software which changes the state to S5. On account of a valid request the state S5 changes to U2. State U2 allows software development activity and finally it returns back to U1 state. S5 state reverts back to S1 for any unidentified person. Figure 3.4 and Figure 3.5 depict the validity checking processes for end-users and developers respectively.

The complete state-chart diagram of the system is now presented in Figure 3.6. This is done to simulate the system requirement which may be validated if required.
Figure 3.6: The complete state-chart diagram of login system

Figure 3.3, Figure 3.4 and Figure 3.5 are now placed in a single state-chart diagram to show the entire functioning of all the states in the security system. As already explained, S1 is the idle state which is changed to either S2 for end-user accessing or to S4 for developers accessing. The rest of the states and state transitions are similar to the previous discussions.

The class and the state diagrams cleanly abstract the crosscutting concerns and the non-crosscutting concerns. This is the first step of model representation in the proposed module.

3.4 Aspect-Oriented Modeling of the Proposed Security Module
Model driven development can benefit from AOP. Crosscutting concerns are implemented through aspects containing source code. As has been demonstrated in the previous section, isolating the concerns (crosscutting and non-crosscutting) are difficult, so it is desirable to first design the state-chart and class diagrams. This is a partial
process of designing the concerns, as it shows very limited information in context of crosscutting concerns. In order to design a complete model of crosscutting concerns and obtain the necessary source code, one needs aspect-oriented modeling. The aspect-oriented modeling not only assists in designing the crosscutting models but also helps in generating suitable source code using AOP model interpreter. The modeling of the proposed security system has been described in the following section.

### 3.4.1 Development of Aspect-Oriented Login Model

In order to develop an Aspect-Oriented model, any domain specific modeling tool can be used from meta-level specifications. Generic Modeling Environment (GME) [82-83] is used for meta-modeling, modeling and for generating application code. The following steps are required to be implemented for the whole process of designing crosscutting concerns from model to code.

(i) Meta-modelling – In order to specify the objects, attributes and relationships, meta-models are used in meta-modelling language. Meta interpreter maps the meta-models into graphical Domain Specific Modelling Language (DSML).

(ii) Modelling – The models are constructed for the security scheme using DSML specified in meta-level process. Since it is not a complex model, simple models are generated.

(iii) Mapping – The domain models generated in the above step are mapped into frame code using model interpreters.

(iv) Building – The source code is built from the frame code.

The following Figure 3.7 depicts the entire process.

---

**Figure 3.7:** Model to code transformations of the login system
Figure 3.7 shows how meta-models are transformed to source code. The main objective to design meta-models is to convert the models to code. Though different tools are available, but, to demonstrate the process of code generation from meta-models, the manual process has been used in the present work. Figure 3.7 depicts the following.

(i) As the login system caters two types of situations (a valid/invalid user), so in the meta-model two different messages are present. The meta-models of the messages contain attributes etc. and also give the relationship among the other meta-models.

(ii) The meta-interpreter converts the meta-models into domain models which are graphical DSL.

(iii) Mapping of the domain model into frame code, which is a structural code, is done through model interpreters.

(iv) From the structural code the source code is developed.

The above discussions show that using suitable tools a meta-model can be easily transformed to source code. Hence, the code generation becomes very handy in such a situation. In this login module, the designed models are converted to source code manually and not by using any tool. This is because designing of models are done initially (in design phase) and later converted into code. All the models are developed in the design phase.

3.4.2 Implementation of the Proposed Security System

The complete state-chart diagram is presented in Figure 3.6. In the proposed complete state-chart model, both aspect and non-aspect constructs are presented in a single model. This is done in order to have a better understanding of both the aspect and non-aspect weaving behavior. Generally, in the context of any software development, after the design phase, primarily the focus is shifted to coding phase, as the output of the design phase confronts the input to the coding phase.
In order to implement the model, the language AspectJ has been chosen. This is an aspect-oriented extension to the Java programming language. It supports general-purpose aspect-oriented programming. Typically, an “AspectJ” program is divided into two parts: the non-aspect code which includes classes, methods, constructors etc. and the aspect code that includes aspects, pointcuts, advices, introductions etc. Similarly, an AspectJ program can be viewed as two important parts. These are the language specification part, which defines the grammar and semantics of the language and the language implementation part, which includes weavers that take various forms such as a ‘compiler’ and a ‘linker’. A weaver produces byte-code that conforms to the Java byte-code specification, allowing any compliant Java Virtual Machine (JVM) to execute those class files. Thus, AspectJ has become a widely-used de facto standard for AOP by laying emphasis on simplicity and on usability for end-users. It uses Java like syntax, and includes IDE integrations for displaying crosscutting structure since its initial public release in 2001. All valid Java programs are also valid AspectJ programs but AspectJ lets programmers define special constructs called aspects. Aspects can contain several entities that are unavailable to standard classes. These are inter-type declarations or introductions, advices and pointcuts. Aspects also support inheritance like classes.

The software used for implementation are Java – jdk 1.7.0, IDE – Eclipse SDK – 4.2 (windows 32-bit), AspectJ Development Tool Kit for Eclipse IDE (ajdt 4.2) and ajc 1.7.2 compiler for compiling AspectJ Programs.

A regular Java class called MessageAcceptor has been created in the present work, which contains two methods named details() and deliver(). Later, aspects are introduced to inject additional behaviour without modifying the class. The details() method accepts the entered details for a particular user (End-user/Developers) to the system and the deliver() method delivers a simple message to an authenticated user. Next, a simple class namely ‘Main’ has been written to exercise the functionality of the MessageAcceptor class. The Main Class contains the main method along with an object ‘obj’ which is used to call the methods of MessageAcceptor Class.

Generally AOP methodology focuses on finding and separating core and crosscutting concerns of a system and on implementing them independently. In the present context, authentication needs to consider as the cross-cutting functionality. The main role of this functionality is to ensure the authentication mechanism. This
A mechanism has been implemented independently with the help of an aspect named as ‘SecurityAspect’. This functionality is injected in the core concern (Java Class). The functionalities implemented through an aspect can be treated in a very advantageous way during the debugging and maintenance phases because it provides an access to the system only to legitimate users. The effort from the developers/programmers end is reduced, because everything regarding the authentication is specified in an aspect. Whenever a method needs to ensure authentication, it can simply call the aspect.

If this has to be implemented in Java or any other Object-Oriented Programming languages, then it is required to add a call in each method of the authenticator class, which needs authentication. Thus implementing through AspectJ has a significant advantage over OOPs as there is no repetitions of code and code scattering is very less. Further, it ensures a better traceability and code reusability as all the modules are developed independently and integrated later on. This integration of codes between the aspect and the non-aspect code does not change the source object code, which means that the business logic of one code can be comfortably used with another code.

The aspect named ‘SecurityAspect’ uses a pointcut secureAccess() which executes the public method named ‘details ()’ of the MessageAcceptor Class. Pointcut secureAccess() captures the join point from the Java code. In this, the pointcut used is having a matching criteria execution(* MessageAcceptor.details(..)), which is used by the ajc compiler to find join points of a project from the source Java classes to weave any relevant advice written for those pointcuts. The ‘*’ indicates that the pointcut matches any return type, and the ‘..’ inside parentheses after the method details() specifies that it matches regardless of the number or types of arguments. An advice has been defined to execute before reaching the join points selected by the secureAccess() pointcut. The before() part indicates that the advice should run prior to execution of the matched join point with the help of pointcut prior to executing any MessageAcceptor.method(). In the advice, the current user is authenticated. With the aspect now present in the system, each time MessageAcceptor.method() is executed, the advice code performs the authentication logic before the execution of the method from the matching criteria of the pointcut secureAccess(). This is achieved by using an Authenticator Class. An object ‘authenticator’ is created.
private Authenticator authenticator = new Authenticator();

authenticator.authenticate();

In this context the ‘Authenticator’ class asks for credentials (username and password) when the authenticate() method is called for the first time in a thread. Upon successful authentication, it stores the user in a thread local and so it doesn’t ask for credentials in the same thread again. Upon failure, it will simply throw a runtime exception. The detailed snapshot for the unauthenticated user is shown in Figure 3.8.

Figure 3.8: A snapshot of an unauthenticated user to the system

After successful authentication the Users (end-users/developers) will have an access to enter their queries. The detailed output snapshot of the implemented model for an authenticated end-user to the system is shown in Figure 3.9.
The main advantages of the above designed module are as follows.

a) It will increase the reusability, as the security module can be integrated to filter the users for accessibility of any software after carrying out minor changes.

b) Aspects help to trigger codes before, after or during any event of the software handling. This facility is not available in OOP.

c) Login can be an ideal example for security module, as accessing any software needs some login mechanism.

d) As AOP is an emerging technique, we can incorporate this in our designed software to improve the efficiency of MDD techniques.

3.5 Case Study: Smart Telephone Network Product Line

The existing standards for developing secure systems generally address system security in broad sense and require extensive resources and expertise to adapt and use during design of a specific system. Also, they do not address low-level details and methodology for designing secure systems [106]. This is the motivation to design a security system for any product line that needs less expertise and which provides the methodology for developing such security systems. The designed security system can
be a component to any product line which checks the authenticity of the user. Such a
security system can be easily integrated to any product line.

This case study is to illustrate this approach in a telephone network system
called Smart Telephone Network Product Line (STNPL) using aspects. This is an
automated telephone system used in any organization. The objective of this case study
is to demonstrate the relevance of AOP and other MDD techniques. The present Object-
oriented techniques lack some automatic method calls which are required before, after
or during an event. This is supported by AOP. In this study, a complex login process to
access the telephone network has been introduced. This login process acts as a security
measure in accessing the telephone network. The functioning of the telephone network
has been described subsequently.

3.5.1 Introduction to STNPL

The telephone network is called smart because it allows the user to access the
network through a vigorous security scanning with a record of the user’s duration of
call. The usage details of the user can also be monitored and stored. For the sake of
simplicity, the recording of user call details has not been considered. The functioning of
the system is as follows.

a) Initially the user’s identity is checked by password
- Unauthorized person is debarred
- Authorized person is allowed to dial a security number which is
  obtained from a message in the registered mobile

b) The user is allowed to access the Telephone Network after dialling the
security number

c) The STNPL provides an interactive mode to the user for obtaining the
destination connection

d) The call operation is terminated with a message of call duration

In order to demonstrate the role of aspects in STNPL, an interface using aspects has
been designed which caters to the first two functions (i.e. (a) and (b)). In the subsequent
chapters, the next two functions have been demonstrated.
3.5.2 Modularization of Security unit of STNPL

According to A.Ray et al. [105], during aspect-oriented software design phase the relationship of the pointcut and aspect needs to be found. This can be well represented in a single sequence diagram. Hence an aspect-oriented based UML sequence diagram of Telephone Network System for a login security process use case has been designed.

The major units of the security system are the following:

1) User

2) Aspect called Phone
   - A before Advice to check the password
   - An after Advice to send a message for dialling the security number that is obtained as SMS in the registered mobile, to the authorized user
   - An around Advice to start the timer immediately after the security number dialled

3) A Public Service Telephone Network (PSTN)

All mentioned units are presented in Figure 3.10.
Figure 3.10: An AO based sequence diagram of STNPL for login security process use case

The major three units (User, Phone and PSTN) in the sequence diagram have been presented. The aspect Phone has advices for checking the authentic person, providing message to dial the security number and to setup the timer. Though the advices are part of the aspect Phone but all these advices are shown as separate units to clarify the functioning. The security unit functions in the following way.
a) A person picks up the phone and provides the password  
b) The aspect Phone verifies the validity through a legitimate() and assigns a boolean value to auth  
c) If auth is false, the system terminates with an exit  
d) Otherwise, a security code required to be dialled which has been provided by the system  
e) Message to dial the security number is triggered by the Phone aspect  
f) On dialling the security number the timer starts functioning through the Phone aspect  
g) The user is connected to PSTN  

3.5.3 Role of aspects in the Security unit of STNPL

The modularization obtained through the aspects in STNPL is a refinement of the login module as described in section 3.4. The login module is a simple password verification that is also used in STNPL. The following refinements make a complex login system for STNPL.

a) A high security number is generated only when the password verified. This needs a database that has to be linked with STNPL which tracks the user’s password with the mobile number.  
b) A time recording system is given for the user’s information.

Implementation of these two mechanisms in the simple login module will make a sophisticated security system that can be integrated to any other application. On implementing the above two mechanisms, aspect becomes mandatory for the following advantages.

a) OOP lacks code execution before, after and during a certain event which aspect through advice does not lack. Password verification, message generation, fetching information from a database and time recording are obtained through suitable advices. This makes AOP mandatory for such security system.  
b) Reusability of existing module (simple login module) has been carried out with some refinements. Efficient MDD techniques demand more and more reusability
of existing modules. Existing aspect modules can easily be refined as per the requirement.

It can be considered that, efficiency of MDD techniques can be enhanced by introducing aspects as a simple or complex module. The security system of STNPL can be further improved by introducing a biometric identification that has not been considered in this case study.

3.6 Summary

The designing of the login module is achieved through class diagrams, meta-models and through AOP codes. This login module is different from a traditional one as it encapsulates messages and security concerns in a separate entity called aspects. Message generation is different in different situations which can be better achieved by the AOP source code rather than by OOP. Though the source code has been generated manually, better approach has been suggested in the present work, in which meta-models are converted into source code using suitable tools. The designed source code for security aspect is a general login module which can be integrated to any software handling for security concerns. This supports reusability of existing components. Hence, though designed manually, the methods for achieving it using Generic Modeling Environment has been given. This approach of generation of aspect-oriented source code through models is a very powerful mechanism in MDD. On the other hand, Aspect-oriented modeling approach is closer to problem space (i.e. analysis, design and modeling) rather than to solution space (i.e. implementation and coding). GME bridges this gap by providing implementation and coding. The development process of the proposed scheme also provides an efficient way to isolate the crosscutting concerns through state-chart diagrams and through class diagrams. The case study of STNPL refines the login module designed initially by incorporating a high security number obtained from a database. This refinement not only makes a simple login module to a complex one but also advocates reusability. The login module and the designing and refining of the login module justifies that AOP is one of the major tools for increasing the efficiency of MDD. The security concerns can be modularized in an interactive way through aspects rather than through objects. Thus, aspects can be an inevitable part of
sophisticated modular software development. The designed login modules can be integrated to any existing product line through aspect weaver. Hence, analysis of AOP in design and code level supports reusability of modules like login. Moreover, the interactive login modules enhance the quality of model driven software, in context of security. This extensive analysis on the role of AOP supports for a better approach of MDD.