The Early Identification of Functional and Non-Functional Crosscutting Concerns

Narender Singh
Department of Computer Science & Applications,
Maharshi Dayanand University, Rohtak, India.

Nasib Singh Gill
Department of Computer Science & Applications,
Maharshi Dayanand University, Rohtak, India.

ABSTRACT
Over the last few years, several research efforts have been devoted for handling crosscutting concerns at the early phases of software development especially at requirements level. These efforts are meaningless unless all the crosscutting concerns are properly identified. Many approaches only consider non-functional concerns as crosscutting concerns. However, crosscutting concerns may also be functional. In this paper, we are proposing an integrated approach that supports complete separation of concerns i.e. handles both functional and non-functional concerns as crosscutting. Our work will surely contribute some positive in this direction.

Keywords
Separation of concerns, crosscutting concerns, aspect-oriented programming, aspect-oriented requirements engineering.

1. INTRODUCTION
The term *separation of concerns [1]* was first introduced by E. Dijkstra, where, a *concern* [2] is any matter of interest in a software system. This described the process of dividing the large complex problem into smaller ones for reducing the complexity of software systems. A lot of significant work exists on separation of concerns in the literature such as viewpoints [3], use cases [4], and goals [5]. Some success in the direction to modularize the complex software system has been achieved. But, still it is difficult to achieve complete separation of concerns through today's most popular programming paradigm such as *Object-Oriented Programming (OOP)* because some concerns are too tightly coupled with others hence spanning over multiple classes and are so called as *crosscutting concerns*. They are responsible for *scattering* and *tangling*. Several empirical studies provide evidence that crosscutting concerns degrade code quality because they negatively impact internal quality metrics such as program size, coupling, and separation of concerns [6]. However, these approaches do not explicitly focus on crosscutting concerns. The work on *advanced separation of concerns* [7], therefore, complements these approaches by providing systematic means for handling such crosscutting concerns.

Aspect-Oriented Programming (AOP) introduced by Kiczales et al. [8] is an alternative programming paradigm to Object-Oriented Programming (OOP). It is also based on the concept of separation of concerns [9]. It is a step towards achieving improved modularity during software development and provides a solution to some difficulties encountered with object-oriented programming, sometimes scattering and tangling. It focuses on crosscutting concerns by providing means for their systematic identification, separation, representation and composition [10]. It encapsulates crosscutting concerns in separate modules, known as *aspects*. It later uses composition mechanism to weave them with other core modules at loading time, compilation time, or run-time [11].

Aspect-orientation is firstly implemented at code level and many aspect-oriented programming languages have been proposed such as AspectJ [12], AspectC [13], AspectC++ [14], JBoss AOP [15], JAsCo [16], HyperJ [17] etc. A lot of significant work also has been carried out at the design level mainly through extensions to the UML meta-model e.g. [18] [19]. Research on the use of aspects at the requirements engineering stage is still young. Aspect-Oriented Requirements Engineering (AORE) [20] [21] improves the modular representation by focusing on identifying, analyzing, specifying, verifying, and managing the crosscutting concerns at the early stages of software development. This early understanding of aspectual trade-offs plays a significant role in shaping the system architecture [22].

Over the last few years, several research efforts have been devoted for handling crosscutting concerns at the early phases of software development especially at requirements level. These efforts are meaningless unless all the crosscutting concerns are properly identified. Many approaches only consider non-functional concerns as crosscutting concerns. However, crosscutting concerns may also be functional, such as auditing, or validation [23] [24]. In this paper, we have proposed a systematic approach to identify both the functional and non-functional crosscutting concerns during requirements engineering along with its application on a case study. Our approach supports the identification of both the functional and non-functional concerns as crosscutting concerns.

The rest of paper is organized as follows: Section 2 presents the related work; Section 3 outlines the proposed systematic approach to identify crosscutting concerns at requirements level. Section 4 illustrates proposed approach by means of a case study. Section 5 draws some conclusions and points to directions of future work.

2. RELATED WORK
An approach called Aspect-Oriented Component Requirements Engineering (AOCRE) was proposed by *John Grundy* [25]. Its main focus was on the identification and specification of both the functional and non-functional requirements each component provides or requires. Early-aspects: a model for aspect-oriented requirements engineering was proposed by *Rashid et al.* [26]. *Baniassad and Clarke* [27] proposed an approach called Theme that is based on Theme/UML which is augmented by the Theme/Doc process. Aspect-oriented software development with use cases by *Jacobson et al.* [28] is an extension to the traditional Use Case approach proposed by the same authors. *Z. Jingjun et al.* [29] proposed aspect-oriented requirements modeling based on UML (Unified Modeling Language) aiming to apply AOP paradigm at requirements level of software development. Here, core concerns and crosscutting concerns are

identified using OOP and AOP respectively and then represented as core class and aspect class in UML. An approach to identify and model candidate aspects from functional and non-functional requirements of the system was proposed by *Hamza and Darwish* [30]. It uses Formal Concept Analysis and Enduring Business Themes to understand the interaction among NFRs and FRs, and to identify candidate aspects in early stages of the development. A use case and non-functional scenario template-based approach was proposed to identify aspects by *Liu et al.* [31]. Use cases and scenario templates are described first here and later they are mapped to specific functional use case features.

3. PROPOSAL OUTLINE

A lot of approaches exist in literature for handling crosscutting concerns are discussed in section 2 of this paper. But, most of them lack in handling both the functional and non-functional crosscutting concerns. Here, we are proposing an integrated approach that supports complete separation of concerns. The approach consists of following systematic activities:

3.1 Identify Concerns

The first activity in our approach is to identify concerns. A *concern* [2] is any matter of interest in a software system and can be defined as a set of coherent requirements. Each set defines a property that the future system must provide. This activity is further divided in many sub-activities like identify actors and use cases, identify relationship among actors and use-cases, elicit functional concerns (FCs), elicit non-functional concerns (NFCs), and finally integrate the NFCs with use cases. The description of each sub-activity is as follows:

3.1.1 Identify Actors and Use Cases

The first sub-activity during identifying concerns is to identify all stakeholders that may interact with the future system. We call these stakeholders as actors and they may be human being or some other system. An actor is a role abstraction that can play various roles in different times. A use case describes the behaviour of the system from an external point of view and used to represents the functionality of the system as a complete flow of events.

3.1.2 Identify Relationships among Actors and Use cases

After identifying the actors and use cases, we need to identify relationship among them. Many types of relationships are established for reducing the complexity of use case diagrams and increasing the understandability of the models. The *communication relationship* is established among actors and use cases when information is exchanged between them. It is represented by a solid line between the actor and the use case along with <<initiate>> or <<participate>> stereotype. The <<initiate>> stereotype is used for the actor who initiate the use case whereas the <<participate>> stereotype is used by all those actors who did not initiate the use case but having communication with the use case. Hence, by establishing such relationships, we can specify access control for the system i.e. which actor can access that functionality and which cannot. The *include relationship* is established to identify the commonalities among the use cases. If two or more use cases share the common behaviour, then factor out that behaviour into a separate use case. The main advantages of it are to reduce the complexity of the model and fewer redundancies. The *extend relationship* is established by including the behaviour of one use case with another use case for exceptional cases like errors, help, and other unexpected conditions. In Unified Modelling Language (UML), we represent include relationships as dashed open arrow starting from including use case and labelled with <<include>> stereotype whereas extend relationship is labelled with <<extend>> stereotype.

3.1.3 Elicit Functional Concerns (FCs)

Functional requirements describe the interaction between the system and its environment independent of its implementation. The environment includes the user and any other system that interacts with the system. A concern may be addressed by a single or multiple requirements. Hence by analyzing the functional requirements, we can identify functional concerns.

3.1.4 Elicit Non Functional Concerns (NFCs)

NFCs are usually system-wide quality concerns that are not directly related to the functional behaviour of the system. They are described as declarative statement including a broad variety of requirements such as usability, reliability, robustness, performance, response time, security etc. They must be defined during concern identification because they can not be modelled directly using use cases and also have much impact on the development cost of the system. To identify NFCs efficiently, both the client and the developers need to collaborate. In practice, an analyst uses taxonomy such as unified process for generating a check list of questions for understanding the non-functional behaviours of the system. This check list of questions can be organized by roles of actors, as they are already identified in the first sub-activity.

3.1.5 Integrate NFCs with Use Cases

The final sub-activity of identifying concerns is to integrate NFCs identified earlier with the base use cases. This is accomplished by using a special stereotyped <<constrain>> relation that extends and links the base use cases to non-functional use cases stereotyped as <<NFC>>.

3.2 Specify Concerns

To specify a concern, we use the modified template of [32] and [33] as shown in table 1. The *name field* contains the name of the concern. The *description field* contains the short description in terms of textual explanation of the concern. The *primary actor* field names the principal actor. The decomposition field shows the *decomposition* of concerns into simpler concerns if possible. The c*lassification* field describes the concern according to its type, e. g. functional, non-functional. The *precondition field* and *postcondition field* contain the conditions to be satisfied before and after the execution of the concerns respectively. The list of *Responsibilities field* lists all the operations that the concern must provide. The list of c*ontributions* field lists positive or negative interactions which the concern has with other concerns. This field helps detecting conflicts whenever concerns contribute negatively to each other. These conflicts may be resolved through the *Stakeholder priorities* field, which assigns priorities to concerns from the stakeholders' perspective. Finally, the *Required concerns* field acts as a dependency reference to other concerns in the system. This field will be used to identify which concerns are crosscutting.

Table 1: Template to specify a concern

Name	The name of the concern.
Description	Short description of the intended behaviour of the concern.
Primary actor	Name of the principal actor.
Stakeholders	Users that need the concern in order to accomplish their job.
Decomposition	Concerns can be decomposed into simpler ones
Classification	Helps the selection of the most appropriate approach to specify the concern. For example: functional, non-functional, goals.
Preconditions	Condition to be satisfied before the execution of the concern.
Post conditions	Condition to be satisfied after the execution of the concern.
List of Responsibilities	
Ri	List of what the concern must perform; knowledge or proprieties the concern must offer.
List of Contributions	
Ci	List of concerns that contribute or affect this concern. This contribution can be positive (+) or negative (-)
List of Priorities	
Stakeholder	Expresses the importance of the concern for a given stakeholder. It can take the values: *Very Important*, *Important*, *Medium*, *Low* and *Very Low*.
List of Required Concerns	
RCi	List of concerns needed or requested by the concern being described.

3.3 Identify Crosscutting Concerns

It is significant to identify crosscutting concerns i.e. candidate aspect at early stages because they may create differing situations and result as undesirable affect on later stages of software development. This is achieved by considering the list of required concerns field in concern specification template and building a matrix shown as table 2 to relate concerns to each other and identifying their crosscutting nature.

In matrix depicted as table 2, both rows and columns represent the concerns identified earlier in section 3.1. Each row represents that the concern requires other concerns and contains the value 1 if the column concern is required by row concern, 0 if not required, and X for diagonal values. Crosscutting concerns refer to those concerns that are required by more than one concern. After analyzing the table, we are able to identify both the functional and non-functional crosscutting concerns.

4. CASE STUDY

For illustrating the approach, we apply it to a case study presented in [34]. The case study is about the First Responder Interactive Emergency Navigational Database (FRIEND), an accident management system. The system is being developed to help and manage the enormous amounts of information involved with accident management [35]. It supports several classes of users including first responders (workers in the field), field supervisors, Dispatchers, and resource allocators. These users collaborate with the help of this system to manage the information associated with an accident(s), including resource information, activities and actions taken in response to an accident, geographical information, Emergency Operations Plan (EOP) information, and hazardous materials information. The requirements are stated as follows: "In FRIEND system, a field officer, such as a police officer or fire fighter, has access to a wireless computer that enables them to interact with a Dispatcher. The Dispatcher in turns can visualize the current status of all its resources, such as a police van or a fire unit or a paramedic unit, on a computer screen and dispatch a resource by issuing commands from a workstation. The system administrator is responsible for managing all users and terminals and also for assigning permissions to different users. The system administrator should be able to store the different users and their permissions, restricting their access."

Table 2: Matrix representing relation among concerns

	C1	**C2**	**C3**	**C4**	**C5**	**.**	**.**	**.**	**Cn**
C1	X	1	1	1	0	0	0	0	1
C2	0	X	0	1	0	1	1	0	1
C3	1	0	X	0	0	0	1	0	0
C4	1	0	0	X	0	1	1	0	0
C5	1	0	0	0	X	1	0	0	1
.	0	0	0	0	0	X	0	0	1
.	0	0	0	0	0	0	X	0	1
.	1	0	0	0	0	1	1	X	0
Cn	1	0	0	1	0	1	1	0	X

$$Ci, j = \begin{cases} \text{X, if } i = j \\ \text{1, if } Cj \text{ is required by } Ci \\ \text{0, otherwise} \end{cases}$$

4.1 Identifying Concerns

The first activity is to identify concerns which is further divided in many sub-activities like identify actors and use cases, identify relationship among actors and use-cases, identify non-functional concerns (NFCs), and finally integrate the NFCs with use cases.

4.1.1 Identify actors and use cases

For example in FRIEND system, many actors are identified such as *FieldOfficer* who is a police and fire officer and responsible for responding to an incident, *Dispatcher* who is a police officer and responsible for answering 1073 calls and allocating resources to an incident, *SystemAdministrator* who is responsible for managing all users and end terminals, and *Librarian* who is responsible for archiving an incident and generating reports. The other actors are *investigator*, *governor*, *mayor*, and *other databases*.

The identified use cases in FRIEND system are *ReportEmergency* to notify a Dispatcher about a new emergency, *OpenIncident* to create an incident report and initiate the incident handling, *AllocateResource* to assign the additional resources to an incident, *ArchieveIncident* to archive an incident, *SearchArchive* to search an incident and generate reports from archived incidents, and use cases for system administration e.g. *ManageUser* for managing users and *ManageTerminal* for managing end terminals.

After the identification of actors and use cases, we can easily define the boundaries of the system. The actors are outside the boundary of the system, whereas the use cases are inside the boundary of the system. The use case diagram for FRIEND system is depicted in figure 1. To simplify the complexity of the case study, we here consider only three use cases ReportEmergency, OpenIncident, and AllocateResources and two actors FieldOfficer and Dispatcher.

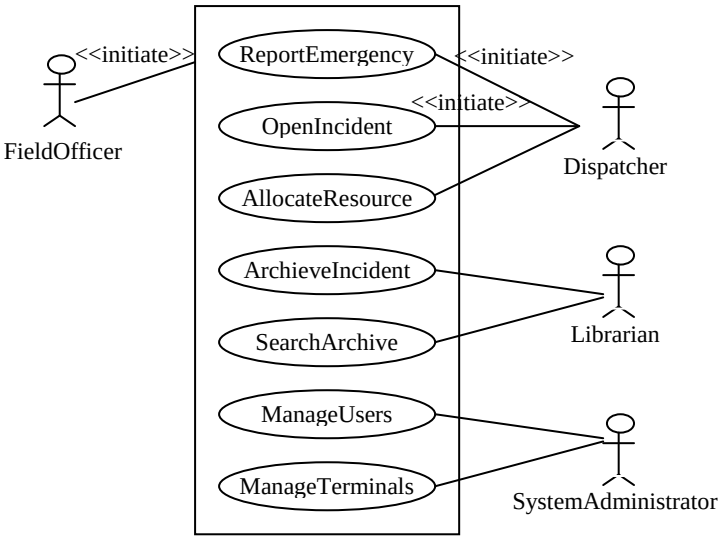

Figure 1: Use case diagram of FRIEND

4.1.2 Identify relationships among actors and use cases

In our case study, there are communication relationships among the actor FieldOfficer and use case ReportEmergency; among the actor Dispatcher and use cases ReportEmergency, OpenIncident and AllocateResources; among the actor librarian and use cases ArchiveIncident and SearchArchive; and among the actor SystemAdministrator and use cases ManageUsers and ManageTerminals. The Dispatcher views the city map to find out the exact position of incident happening and also for allocating the resources to the incident nearby. Here, both the use cases OpenIncident and AllocateResource share the common behaviour of viewing the city map. Hence, a new use case *ViewMap* can be described using include relation for sharing this common behavior as shown in figure 2 [36]. Also, suppose that there may be network failure at any time during communication between the FieldOfficer and Dispatcher. Hence, we need a new use case that describes the flow of events needed to recovery due to network failure. *ConnectionDown* is new use case that extends the use case ReportEmergency, OpenIncident, and AllocateResource as shown in the figure 3 [36].

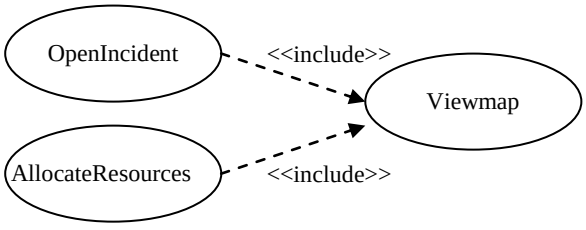

Figure 2: Include relationship among use cases

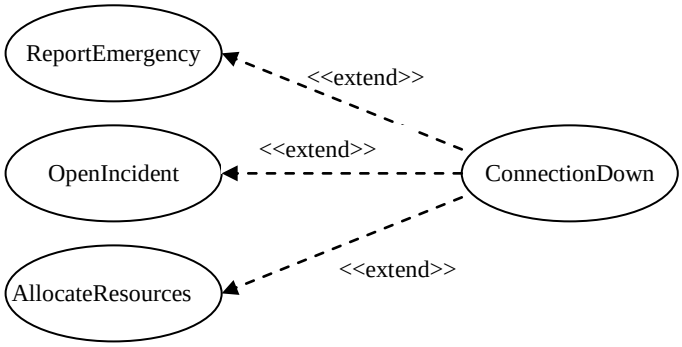

Figure 3: Extend relationship among use cases

4.1.3 Elicit the Functional Concerns (FCs)

The functional requirements for FRIEND system are:

- The FieldOfficer must be logged into FRIEND system.
- The FieldOfficer reports an emergency into FRIEND system.
- FRIEND system responds by presenting a form to the FieldOfficer.
- The FieldOfficer fills out the form by selecting emergency level, type, location, and brief description of the situation. Once the form is completed, he/she submitted the form.
- FRIEND system acknowledges the successful submission to the FieldOfficer.
- FRIEND system receives the form and notifies the Dispatcher.
- The Dispatcher reviews the submitted information and creates an incident in the database.
- The dispatcher selects a response and acknowledges the report to the FieldOfficer.
- The Dispatcher in turns can visualize the current status of all its resources, such as a police van or a fire unit or a paramedic unit, on a computer screen and dispatch a resource by issuing commands from a workstation.

A concern may be addressed by a single or multiple requirements. Hence by analyzing the functional requirements, we can identify functional concerns. The Functional Concerns identified here are login system, report emergency, open incident, allocate resources, view map, and connection down. These FCs are shown in Table 3.

4.1.4 Elicit the Non-Functional Concerns (NFCs)

To identify NFCs efficiently, both the client and the developers need to collaborate. In practice, an analyst uses taxonomy such as Unified Process for generating a check list of questions for understanding the non-functional behaviours of the system. For example, NFCs for our case study are derived from these NFRs:

Concurrency:

- The primary purpose of FRIEND is to provide users concurrent access to a set of shared information. They may access the data simultaneously or serially.
- Concurrent users must see changes to the data as quickly as possible.
-

Response Time:

- Responds within time (<t) by presenting a form to FieldOfficer when the FieldOfficer activates the "Report Emergency" function from his/her terminal.
- Responds within time (<t) by acknowledging the successful submission of form to the FieldOfficer.
- Responds within time (<t) by notifying the Dispatcher about new emergency after receiving the form submitted by FieldOfficer.
- Responds within time (<t) by acknowledging FieldOfficer the selected response submitted by Dispatcher.

Logging:

- When the connection downs, the situation is logged by the system and recovered when the connection is re-established.
- The system administrator is responsible for monitoring different activities that occur in the system e.g. check-in operations.

Accuracy:

- During an accident, decisions about resource allocations must be made quickly and correctly.

Mobility:

- The system must handle mobility as the accident management personnel will need to access the system on the move during his/her tour. The mobility required by accident management personnel requires FRIEND system to employ state-of-the art wireless communication technology.

Compatibility:

- The system must be compatible with the external services it has to interact with; in particular, hotel and theatre ticket reservations.

Availability:

- The system must always be available to react to be accessed by the FieldOfficer and Dispatcher.

Security:

- The system administrator is responsible for managing all users and terminals and also for assigning permissions to different users.
- The system administrator should be able to store the different users and their permissions, restricting their access.

Hence, in our case study, we have identified NFCs: concurrency, response time, logging, availability, mobility, accuracy, compatibility and security. These NFCs are shown as in table 3.

4.1.5 Integrate NFCs with Use Cases

To simplify the complexity of the case study, we here consider only *Response Time* NFC to integrate with use cases. The refined use case diagram of FRIEND system after integrating NFCs with use cases is depicted in following figure 4. Also, all the identified concerns in FRIEND are represented in table 3.

4.2 Specify Concerns

To specify a concern, we use the template shown as table 1. *Report Emergency* concern and *Response Time* concerns are specified using the template as shown in table 4 and table 5 respectively.

Table 3: Concerns identified in FRIEND system

Concern	Description	
FC1	Login system	Functional Concerns (FCs) identified
FC2	Report emergency	
FC3	Open incident	
FC4	Allocate resources	
FC5	View map	
FC6	Connection down	
NFC1	Concurrency	Non-functional Concerns (NFCs) identified
NFC2	Response time	
NFC3	Logging	
NFC4	Accuracy	
NFC5	Mobility	
NFC6	Compatibility	
NFC7	Availability	
NFC8	Security	

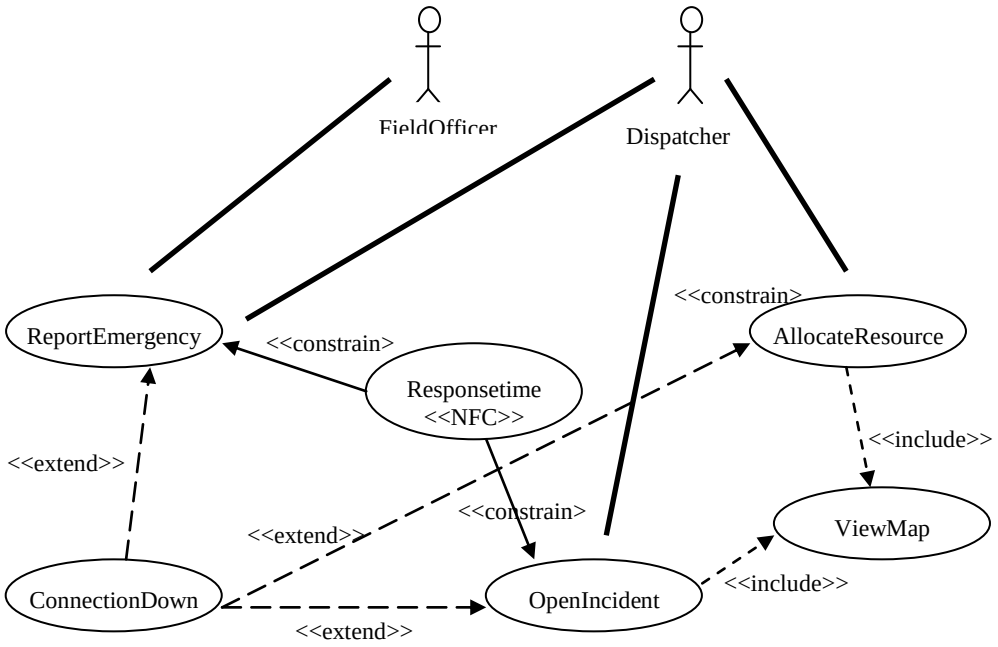

Figure 4: Refined Use Case Diagram of FRIEND system after Integrating NFCs with Use Cases

Table 4: Specifying Report Emergency concern

Name	ReportEmergency
Description	The FieldOfficer reports an emergency into FRIEND system.
Primary actor	Fieldofficer
Stakeholders	FieldOfficer, Dispatcher
Decomposition	<None>
Classification	Functional
Preconditions	The FieldOfficer must be logged into FRIEND system.
Post conditions	The FieldOfficer must be received an acknowledgment and selected response from the Dispatcher, OR The FieldOfficer must be received an explanation indicating why the transaction could not be processed.
List of Responsibilities	
1. Responds by presenting a form to the FieldOfficer. 2. Receives the form filled by the FieldOfficer. 3. Acknowledges the successful submission to FieldOfficer. 4. Notifies the Dispatcher.	
List of Contributions	
<None>	
List of Priorities	
1. FieldOfficer: Very Important 2. Dispatcher: Very Important 3. Developer: Important	
List of Required Concerns	
1. Login system	

4.3 Identify Crosscutting Concerns

To identify crosscutting concerns, we use a matrix shown as Table 2. Hence it results as a matrix shown as table 6 relating all identified concerns (LS: Login System, RE: Report Emergency, OI: Open Incident, AR: Allocate Resources, VM: View Map, CD: Connection Down, CN: Concurrency, RT: Response Time, LG: Logging, AC: Accuracy, MB: Mobility, CP: Compatibility, AV: Availability, SC: Security).

After analyzing the table, we identify LS (login system), VM (View Map), CD (Connection Down), CN (Concurrency), RT (Response Time), LG (Logging), AC (Accuracy), MB (Mobility), AV (Availability), and SC (Security) as crosscutting concerns. Crosscutting concerns may be functional or non-functional concern. For example, LS (Login System) and VM (View Map) are functional crosscutting concerns. The non-functional crosscutting concerns are Connection Down, Concurrency, response Time, Logging, Accuracy, Mobility, and Security.

5. CONCLUSION AND FUTURE WORK

It is beneficial to handle crosscutting concerns at early stages of software development rather than handling them at later stages because it not only makes the design simpler, but also helps to reduce the cost and defects that occur in the later stages of development. Over the last few years, several research efforts have been contributed for handling crosscutting concerns at the early phases of software development. These efforts are meaningless unless all the crosscutting concerns are properly identified. AORE has emerged as a new way to modularize and reason about crosscutting concerns during requirements engineering. It improves the modular representation by focusing on identifying, analyzing, specifying, verifying, and managing the crosscutting concerns at the early stages of software development. Many existing AORE approaches consider only non-functional concerns as crosscutting but, crosscutting concerns may be functional as well as non-functional.

In this paper, we have proposed a systematic AORE approach to identify these crosscutting concerns at early phases of software development. Further, we have implemented the proposed approach on a case study and achieved some success to identify both the functional as well as non-functional concerns as crosscutting concerns. But, still we need more efforts on the proposed approach to realize it as a complete AORE approach. These efforts include exploring the activity of identifying crosscutting concerns, managing concerns, composing concerns, and validating it with more case studies using aspect-oriented metrics. Our future work will focus on improving the proposed approach by incorporating all the aspects which are left here.

Table 5: Specifying Response Time concern

Name	Response Time
Description	The FieldOfficer's is acknowledged within 30 seconds after the submission of form. Also, the selected response arrives no later than 30 seconds after it is sent by the Dispatcher.
Primary actor	<None>
Stakeholders	FieldOfficer, Dispatcher, System Administrator, Developer
Decomposition	<None>
Classification	Non-functional
Preconditions	<None>
Post conditions	<None>
List of Responsibilities	
Responds within time (<t) by presenting a form to FieldOfficer when the FieldOfficer activates the "Report Emergency" function from his/her terminal. Responds within time (<t) by acknowledging the successful submission of form to the FieldOfficer. Responds within time (<t) by notifying the Dispatcher about new emergency after receiving the form submitted by FieldOfficer. Responds within time (<t) by acknowledging FieldOfficer the selected response submitted by Dispatcher.	
List of Contributions	
1. Availability (+) 2. Accuracy (-) 3. Concurrency (-)	
List of Priorities	
1. FieldOfficer: Very Important 2. Dispatcher: Very Important 3. System administrator: Very Important 4. Developer: Important	
List of Required Concerns	
<None>	

Table 6: Matrix representing relation among concerns of FRIEND system

	LS	**RE**	**OI**	**AR**	**VM**	**CD**	**CN**	**RT**	**LG**	**AC**	**MB**	**CP**	**AV**	**SC**
LS	X	0	0	0	0	1	0	1		1	1	1	1	1
RE	1	X	0	0	0	1	1	1	1	1	1	0	1	1
OI	1	0	X	0	1	1	1	1	1	1	1	0	1	1
AR	1	0	0	X	1	1	1	1	1	1	1	0	1	1
VM	1	0	0	0	X	1	1	1	0	1	1	0	1	0
CD	0	0	0	0	0	X	0	0	1	0	0	0	0	0
CN	0	0	0	0	0	0	X	0	0	1	0	0	1	0
RT	0	0	0	0	0	0	1	X	0	1	0	0	1	0
LG	0	0	0	0	0	0	1	0	X	1	0	0	1	1
AC	0	0	0	0	0	0	1	0	0	X	0	0	1	1
MB	0	0	0	0	0	0	1	0	0	0	X	0	1	0
CP	0	0	0	0	0	0	0	0	0	1	1	X	1	0
AV	0	0	0	0	0	0	0	0	0	0	1	0	X	0
SC	1	0	0	0	0	0	0	0	0	0	0	0	0	X

6. REFERENCES

[1] Dijkstra, E.W., 1976, A Discipline of Programming, Prentice Hall, Englewood Cliffs, New Jersey, USA.

[2] S. M. Sutton Jr and I. Rouvellou., 2002, Modeling of Software Concerns in Cosmos. In Proceedings of the 1st International Conference on Aspect-Oriented Software Development, ACM, pages 127–133.

[3] Finkelstein, A., Sommerville, I. 1996, The Viewpoints FAQ, Software Engineering Journal: Special Issue on Viewpoints for Software Engineering, IEE/BCS. **11**(1): 2-4.

[4] Jacobson, I et al., 1992, Object-Oriented Software Engineering - a Use Case Driven Approach, 978-0201544350, Addison-Wesley.

[5] Chung, L.et al., 2000, Non-Functional Requirements in Software Engineering, 0-7923-8666-3, Kluwer Academic Publishers.

[6] Mark.E. et al., 2008, Do Crosscutting Concerns Cause Defects?", IEEE Transactions On Software Engineering, Vol. 34, No. 4, July/August.

[7] I. Brito, A. Moreira, 2003, "Advanced Separation of Concerns for Requirements Engineering", VIII Jornadas de Ingenieria del Software y Bases de Datos (JISBD'03), pp. 47-56, Alicante, Spain. November 12-14.

[8] G. Kiczales et al., 1997, Aspect-oriented programming, in: Proceedings of the 11th European Conferenceon Object-Oriented Programming (ECOOP'97), Springer, pp. 220–242.

[9] Laddad Ramnivas, 2002, I want my AOP! Part 1: Separate software concerns with aspect-oriented programming. Java World Fueling Innovations. URL: http://www.javaworld.com/javaworld/jw-01-2002/jw-0118-aspect.html.

[10] Rashid, A., Moreira, A., Araújo, J., 2003, Modularization and Composition of Aspectual Requirements, In 2nd Aspect-Oriented Software Development Conference (AOSD'03), Boston, USA, ACM Press. 11-20.

[11] Baniassad et al., 2006, Discovering Early Aspects, IEEE Software Special Issue on Aspect-Oriented Programming. 23(1): pp. 61-70.

[12] AspectJ Project (2007). http://www.eclipse.org/aspectj/.

[13] AspectC Project (2007). http://www.cs.ubc.ca/labs/spl/projects/aspectc.html.

[14] Spinczyk, O., Lohmann, D., Urban, M. (2005). "AspectC++: an AOP Extension for C++." Software Developer's Journal **1**: 68-76.

[15] JBoss Project (2007). http://labs.jboss.com/jbossaop/.

[16] JAsCo Project (2007). http://ssel.vub.ac.be/jasco/.

[17] HyperJ (2007). http://www.alphaworks.ibm.com/tech/hyperj.

[18] Clarke S. and Walker R. J., 2001, Composition Patterns: An Approach to Designing Reusable Aspects, ICSE.

[19] Suzuki J. and Yamamoto Y., 1999, Extending UML with Aspects: Aspect Support in the Design Phase, ECOOP Workshop on AOP.

[20] Y. Yu et al. 2004, From Goals to Aspects: Discovering Aspects from Requirements Goal Models, Proc. RE 2004, IEEE CS, pp. 38-47.

[21] E. Baniassad, S. Clarke, 2004, Theme: An Approach for Aspect-Oriented Analysis and Design, ICSE 2004, IEEE CS, pp.158-167.

[22] A. Moreira et al., 2005, Multi-Dimensional Separation of Concerns in Requirements Engineering" RE 2005, pp. 285-296.

[23] Moreira, A., Araújo, J., Brito, I, 2002, Crosscutting Quality Attributes for Requirements Engineering. In 14th Software Engineering and Knowledge Engineering Conference (SEKE'02), Ischia, Italy, ACM Press. 167 - 174.

[24] Rashid, A., Moreira A., 2006, Domain Models are NOT Aspect Free. In 9th Model Driven Engineering Languages and Systems Conference (MoDELS/UML'06). Genova,

Italy, Lecture Notes in Computer Science 4199. Springer. 155-169.

[25] J. Grundy, 1999, Aspect-Oriented Requirements Engineering for Component-based Software Systems, IEEE International Symposium on Requirements Engineering, IEEE CS, pp. 84-91.

[26] Rashid, A. et al., 2002, Early Aspects: a Model for Aspect-Oriented Requirements Engineering, Proc. of Int. Conference on Requirements Engineering (RE'02).

[27] E. Baniassad, S. Clarke, 2004, Theme: An Approach for Aspect-Oriented Analysis and Design, In Proceedings of the 26th Int. Conf. on Software Engineering (ICSE04).

[28] Jacobson, I., 2004, Aspect-Oriented Software Development with Use Cases, 978-0-321-26888-4, Addison-Wesley.

[29] Zhang Jingjun et al., 2007, Aspect-Oriented Requirements Modeling, Proceeding of the 31st IEEE Software Engineering Workshop SEW-31 (SEW'07), Baltimore, MD, USA.

[30] S. Hamza and D. Darwish, 2009, On the Discovery of Candidate Aspects in Software Requirements, Proc. Of Sixth International Conference on Information Technology: New Generations.

[31] Xiaojuan et al., 2010, Use case And Non-functional Scenario Template-Based Approach to Identify Aspects, Second International Conference on Computer Engineering and Applications.

[32] I. Brito, 2004, Aspect-Oriented Requirements Engineering, UML'04 DocSym, Doctoral Symposium 7th International Conference on the Unified Modeling Language, Lisbon, Portugal.

[33] A. Amirat et al., 2006, An Aspect-Oriented Approach in Early Requirements Engineering, IEEE, pp. 1055-1058.

[34] Bernd Bruegge et al., 1994, Design Considerations for an Accident Management System, In Proceedings of the Second International Conference on Cooperative Information Systems, Toronto Press, May 1994.

[35] M. Bookser and B. Bruegge. 1993, Information, Technology and Police Management: The FRIEND system," Proceedings of the 1993 Society of Police Futurists International Symposium, Baltimore, Maryland, May 1993.

[36] B. Bruegge and A.Dutoit, 2007, Object-Oriented Software Engineering Using UML, Patterns, and Java™, 2nd Edition, Pearson Education.