Identification of Crosscutting in Software Design

Klaas van den Berg
Software Engineering Group
University of Twente
P.O. Box 217
7500 AE Enschede
the Netherlands
+31 53 4893783
k.g.vandenberg@ewi.utwente.nl

José María Conejero
Quercus SEG
University of Extremadura
Avda. Universidad s/n
C.P. 10071 Cáceres
Spain
+34 927 257268
chemacm@unex.es

Juan Hernández
Quercus SEG
University of Extremadura
Avda. Universidad s/n
C.P. 10071 Cáceres
Spain
+34 927 257204
juanher@unex.es

ABSTRACT
The identification of crosscutting is a prerequisite for applying aspect-oriented techniques in software development. We present an operationalization of the definition of crosscutting to support this identification. We use matrices to represent the relation between design elements at different levels of abstraction. We present some case studies about the identification of crosscutting concerns in order to illustrate the application of our approach. In particular, we apply the approach to the identification of crosscutting in some of the GoF's design patterns.

Keywords
Aspect-Oriented Software Development, Scattering, Tangling, Crosscutting, Crosscutting Concerns

1. INTRODUCTION
Several approaches to the modelling of crosscutting concerns in the different phases of the development life cycle have emerged. Some of these approaches propose mechanisms to model crosscutting focusing on particular phases such as design or requirements [18] [19] [22]. Other ones present mechanisms or techniques which allow the modeling throughout several stages or even the entire development process [5] [11]. In [2] a more exhaustive and detailed survey of these approaches is presented.

Usually, these approaches presuppose that crosscutting concerns have been previously identified somehow. The identification of such crosscutting concerns is based on either designer's experience or designs by others developers where crosscutting concerns have been identified early on. So these approaches allow us to model crosscutting concerns which are already well-know by the AOSD community. However, both in aspect-oriented designs and non aspect-oriented designs, new characteristics of the systems could emerge behaving as crosscutting concerns. These concerns could not be previously identified as crosscutting concerns. Obviously, we need some support to identify such crosscutting concerns in order to apply appropriate aspect-oriented techniques to handle them. Aspect mining is a research area which provides mechanisms to identify crosscutting. However most of aspect mining approaches (e.g. [10] [20]) are focused on code level. The lack of support for this identification in earlier phases is an impediment to apply aspect modeling approaches. In this paper we propose an approach to identify crosscutting concerns in software designs or models.

The approach is based on our own formal definition of crosscutting [6] and its representation by means of matrices. This approach helps to aspect-oriented modeling since it can be used to identify crosscutting concerns. It can be applied to design phases but also to other consecutive phases in software development.

The rest of paper is structured as follows. In section 2, we introduce the definition of crosscutting. We describe how to represent and visualize crosscutting in a crosscutting matrix and how to derive this matrix from the dependency matrix using a scattering and tangling matrix. In section 3, we show some case studies where we apply the concepts introduced in the paper. Finally in sections 4 and 5, we describe related work and present conclusions of the paper.

2. CROSSCUTTING DEFINITION
The definition presented in this paper is based on a framework of crosscutting proposed in [6][7]. The proposition is that crosscutting can only be defined in terms of 'one thing' with respect to 'another thing'. In other words, at least two domains (or two levels or two phases) are related with each other in some way.

- A *domain* could refer for example to a concern model with concerns or to a design with architectural elements.
- A *level* could refer for example to refinements in the Model Driven Architecture (e.g. CIM, PIM and PSM) [15].
- A *phase* could refer to any phase in the software development life cycle (e.g. requirements, design, and so on).

We use here the general terms *source* and *target* (as in [15]) to denote two consecutive domains, phases or levels. We assume that elements in the source are related to elements in the target: there is a mapping between source and target elements. The mapping can be established manually or be automated in transformation rules.

According to our definition in [6], crosscutting occurs when in a mapping between source and target, a source element is mapped to two or more target elements and at least one of these target elements has a mapping from one other source element. In Figure 1, we show an intuitive representation of an example mapping with target element t3 involved in crosscutting.

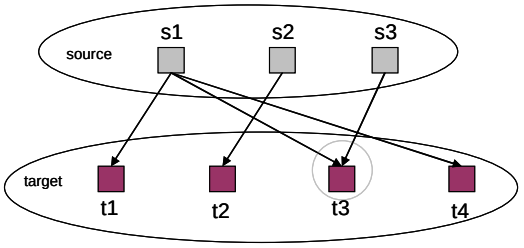

Figure 1. Mapping between elements at different levels of abstraction (s1, s2, s3 at source; t1, t2, t3 and t4 at target)

2.1 Matrix representation

In this section, we describe how crosscutting can be identified and represented in matrices. As starting point, the developer must establish a dependency matrix showing the mapping between source and target. From this matrix, we derive the crosscutting matrix, where we represent the crosscutting source elements. Then, we describe how the crosscutting matrix can be constructed from the dependency matrix with some auxiliary matrices. This is illustrated with some examples.

2.2 Definitions of matrices

The relation between source elements and target elements can be represented in a matrix that we called dependency matrix. As described before, the mapping can have different types, such as usage and abstraction dependencies (e.g. realization, refinement and tracing [24]). *A dependency matrix (source x target) represents the dependency relation between source elements and target elements (inter-level relationship).* In the rows, we have the source elements, and in the columns, we have the target elements. In this matrix, a cell with 1 denotes that the source element (in the row) is *mapped* to the target element (in the column). Reciprocally this means that the target element *addresses* the source element. Scattering and tangling can easily be visualized in this matrix (see the examples below).

We define an auxiliary concept *crosscutpoint* used in the context of dependency matrices, to denote *a matrix cell involved in both tangling and scattering*. If there is one or more crosscutpoints then we say we have crosscutting.

Crosscutting between source elements for a given mapping to target elements, as shown in a dependency matrix, can be represented in a crosscutting matrix. *A crosscutting matrix (source x source) represents the crosscutting relation between source elements, for a given source to target mapping (represented in a dependency matrix).* In the crosscutting matrix, a cell with 1 denotes that the source element in the row is crosscutting the source element in the column. In section 2.3 we explain how this crosscutting matrix can be derived from the dependency matrix.

A crosscutting matrix should not be confused with a coupling matrix. A *coupling matrix* shows coupling relations between elements at the same level (intra-level dependencies). In some sense, the coupling matrix is related to the design structure matrix [4]. On the other hand, a crosscutting matrix shows crosscutting relations between elements at one level with respect to a mapping onto elements at some other level (inter-level dependencies).

We now give an example and use the dependency matrix and crosscutting matrix to visualize the definitions (S denotes a scattered source element - a grey row; NS denotes a non-scattered source element; T denotes a tangled target element - a grey column; NT denotes a non-tangled target element). The example is shown in Table 1, representing the mapping from Figure 1.

Table 1. Example dependency and crosscutting matrix

dependency matrix

		target				
		t[1]	t[2]	t[3]	t[4]	
source	s[1]	1	0	1	1	S
	s[2]	0	1	0	0	NS
	s[3]	0	0	1	0	NS
		NT	NT	T	NT	

crosscutting matrix

		source		
		s[1]	s[2]	s[3]
source	s[1]	0	0	1
	s[2]	0	0	0
	s[3]	0	0	0

In this example, we have one scattered source element s[1] and one tangled target element t[3]. Moreover there is one crosscutpoint at matrix cell [1,3] (dark grey cell). Applying our definition, we arrive to the crosscutting matrix. Source element s[1] is crosscutting s[3] (because s[1] is scattered over [t[1], t[3], t[4]] and s[3] is in the tangled one of these elements, namely t[3]). The reverse is not true: the crosscutting relation is not symmetric.

2.3 Constructing crosscutting matrices

In this section, we describe how to derive the crosscutting matrix from the dependency matrix. We use a more extended example than the previous one. We now show an example with more than one crosscutpoint, in this example 8 points (see Table 2; the dark grey cells).

Table 2. Example dependency matrix with tangling, scattering and several crosscutpoints

dependency matrix

		target						
		t[1]	t[2]	t[3]	t[4]	t[5]	t[6]	
source	s[1]	1	0	0	1	0	0	S
	s[2]	1	0	1	0	1	1	S
	s[3]	1	0	0	0	0	0	NS
	s[4]	0	1	1	0	0	0	S
	s[5]	0	0	0	1	1	0	S
		T	NT	T	T	T	NT	

Based on the dependency matrix, we define some auxiliary matrices: the *scattering matrix* (source x target), and the *tangling matrix* (target x source). These two matrices are defined as follows:

- In the scattering matrix a row contains only dependency relations from source to target elements if the source element in this row is scattered (mapped onto multiple target elements); otherwise the row contains just zero's (no scattering).

- In the tangling matrix a row contains only dependency relations from target to source elements if the target element in this row is tangled (mapped onto multiple source elements); otherwise the row contains just zero's (no tangling).

For our example in Table 2, these matrices are shown in Table 3.

We now define the crosscutting product matrix, showing the frequency of crosscutting relations. *A crosscutting product matrix (source x source) represents the frequency of crosscutting relations between source elements, for a given source to target mapping.* The crosscutting product matrix is not necessarily symmetric. The *crosscutting product matrix* ccpm can be obtained through the matrix multiplication of the scattering matrix sm and the tangling matrix tm: $ccpm = sm \,.\, tm$ where $ccpm_{ik} = sm_{ij}\, tm_{jk}$

In this crosscutting product matrix, the cells denote the frequency of crosscutting. This can be used for quantification of crosscutting (crosscutting metrics). The frequency of crosscutting in this

matrix should be seen as an upper bound. In actual situations, the frequency can be less than the frequency from this matrix analysis, because in the matrix we abstract from scattering and tangling specifics. In the crosscutting matrix, a matrix cell denotes the occurrence of crosscutting; it abstracts from the frequency of crosscutting.

Table 3. Scattering and tangling matrices for dependency matrix in Table 2

scattering matrix

		target					
		t[1]	t[2]	t[3]	t[4]	t[5]	t[6]
source	s[1]	1	0	0	1	0	0
	s[2]	1	0	1	0	1	1
	s[3]	0	0	0	0	0	0
	s[4]	0	1	1	0	0	0
	s[5]	0	0	0	1	1	0

tangling matrix

		source				
		s[1]	s[2]	s[3]	s[4]	s[5]
Target	t[1]	1	1	1	0	0
	t[2]	0	0	0	0	0
	t[3]	0	1	0	1	0
	t[4]	1	0	0	0	1
	t[5]	0	1	0	0	1
	t[6]	0	0	0	0	0

The *crosscutting matrix* ccm can be finally derived from the crosscutting product matrix ccpm using a simple conversion: $ccm_{ik} = \text{if } (ccpm_{ik} > 0) \wedge (i \neq j) \text{ then 1 else 0}$.

The crosscutting product matrix and the crosscutting matrix for the example are given in Table 4.

Table 4. Crosscutting product matrix and crosscutting matrix for dependency matrix in Table 2

crosscutting product matrix

		Source				
		s[1]	s[2]	s[3]	s[4]	s[5]
source	s[1]	2	1	1	0	1
	s[2]	1	3	1	1	1
	s[3]	0	0	0	0	0
	s[4]	0	1	0	1	0
	s[5]	1	1	0	0	2

crosscutting matrix

		Source				
		s[1]	s[2]	s[3]	s[4]	s[5]
source	s[1]	0	1	1	0	1
	s[2]	1	0	1	1	1
	s[3]	0	0	0	0	0
	s[4]	0	1	0	0	0
	s[5]	1	1	0	0	0

In this example, there are no cells in the crosscutting product matrix larger than 1, except on the diagonal where it denotes a crosscutting relation with itself, which we disregard here. In the crosscutting matrix, we put the diagonal cells to 0. Obviously, this is because we interpret a source element can't crosscut itself.

As we can see in crosscutting matrix in Table 4, there are now 10 crosscutting relations between the source elements. The crosscutting matrix shows again that the crosscutting relation is not symmetric. For example, s[1] is crosscutting s[3], but s[3] is not crosscutting s[1] because s[3] is not scattered (scattering and tangling are necessary but not sufficient condition for crosscutting).

For convenience, these formulas can be calculated automatically by means of very simple mathematic tools. By filling in the cells of the dependency matrix, the other matrices are calculated automatically.

3. CASE STUDIES

In order to show the application of our approach in different domains, we demonstrate now how to identify crosscutting in some well-know examples of the literature: on the one hand a DVD store system and on the other hand some GoF's (Gang of Four) design patterns [8]. The latter is extracted from a more extended study where we applied the approach to most of these patterns. For space reasons, we do not show the complete study. We show the application of the framework to two particular design patterns: Mediator and Adapter.

3.1 The DVD System

In this section we describe an example to illustrate the use of our definitions to identify crosscutting in software design. The example is about a system to handle the selling of various DVD products (e.g. DVDs and boxsets). Each product has one or more suppliers. The example is based on Gradecki & Lesiecki ([9], Chapter 1). Here, we consider four concerns: the keeping of a price for each product (price keeping concern), the number of DVD's in a boxset (boxset size concern), the handling of titles of each DVD (DVD title concern), and the recording of any changes in the state of each product (change logging concern). These concerns can be extracted from requirements presented in [9] through concern modelling techniques. The object-oriented design of the system is shown in the UML class diagram in Figure 2. There is an abstract class Product with two subclasses DVD and Boxset. Each product has a price (in the attribute price) and one or more suppliers from the class Supplier. A DVD has a title (attribute title). A Boxset contains a number of DVDs (attribute number), representing the size of the boxset. Each product has a Logger object. This object is used for logging changes in the price (in the operation Product.setPrice), changes in the title (in the operation DVD.setTitle), and changes in the number of DVDs in a boxset (in the operation Boxset.setNumber).

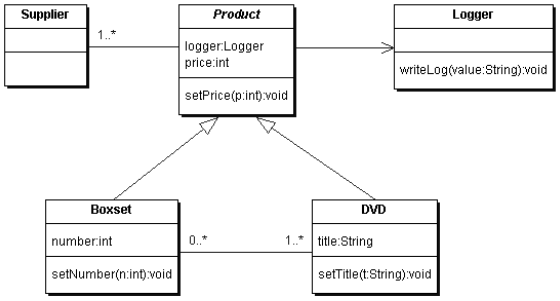

Figure 2 Class diagram of the example system for selling DVD products with logging (based on [9])

Based on an implicit and intuitive notion of crosscutting, Gradecki & Lesiecki [9] state that the logging concern is a crosscutting concern. However, in this paper we are interested in

the formal identification of crosscutting concerns such as this logging concern. For this purpose, we introduced the definitions in the previous section. Then we will apply it in this example to show the identification of crosscutting concerns.

As we mentioned as decomposition of the concerns, we have the price keeping concern, the change logging concern, the boxset size concern and the DVD title concern. These are the four source elements. As decomposition of the design, we have the 5 classes: Product, Supplier, Boxset, DVD, and Logger. These are the five target elements. We could establish the following dependency matrix (see Table 5). The price keeping concern is mapped onto the class Product, and only implicitly - through inheritance of attribute and operations - in the classes DVD and Boxset. The logging is performed in each class where a change of state could be performed. Therefore, the change logging concern is mapped onto the classes Product, Boxset and DVD because of the explicit call of *writeLog* in the set operations in these classes.

Table 5. A dependency matrix for the DVD products system

	Design Class					
Concern	Product	Supplier	Boxset	DVD	Logger	
price keeping	1	0	0	0	0	NS
change logging	1	0	1	1	1	S
DVD title	0	0	0	1	0	NS
boxset size	0	0	1	0	0	NS
	T	NT	T	T	NT	

Applying our definitions, we derive the crosscutting matrix for this case. This matrix is shown in Table 6. In this case, the logging concern is crosscutting as well the price keeping concern as the DVD title concern and the boxset size concern, but not the other way around (crosscutting is not symmetric).

Table 6. Crosscutting matrix for the dependency matrix in Table 5 for the DVD products system

	Concern			
Concern	price keeping	change logging	DVD title	boxset size
price keeping	0	0	0	0
change logging	1	0	1	1
DVD title	0	0	0	0
boxset size	0	0	0	0

This example shows that there should be well-defined mapping rules from source to target elements, with a rational about which mapping or dependency relations are included. In the example above, the methods calls of *writeLog* are represented as a dependency, but the inheritance of the price keeping is not represented in the dependency matrix. Other choices are feasible. Depending on the goal of the crosscutting analysis, one has to select the mapping rules. The impact of inheritance on crosscutting is illustrated in [25]. A tentative classification of crosscutting based on types of dependency relationships is given in [12].

Obviously, this logging crosscutting concern is well identified in the AOSD literature, and the obtained result is not surprising at all. However the same analysis may be done for systems where other crosscutting concerns may arise. We show some other case studies in subsequent sections.

3.2 Dialog system in GUI (Mediator Pattern)

A dialog box in a GUI commonly uses a window containing a wide collection of widgets such as text, list boxes, buttons, radio buttons and so on. The behaviour of the dialog box is distributed among the different widgets which usually interact with each other, enabling or disabling actions according with the widget behaviour. These interactions reduce the reusability of the objects participating in the GUI. The Mediator pattern allows widgets to be decoupled through the addition of a class which takes over the communication among widgets. The application of the pattern improves the reusability of widgets making them oblivious about the communication with other objects. The UML class diagram of the Dialog System based on the Mediator Pattern is shown in Figure 3. A more detailed explanation of this example and the Mediator pattern can be found in [8].

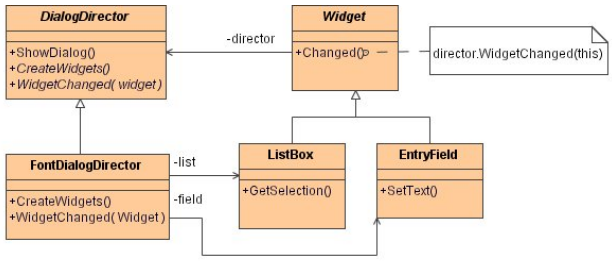

Figure 3. Mediator pattern applied to GUI design [8]

As it is stated in [8], there are three different participants in Mediator pattern: Mediator (the *DialogDirector*), ConcreteMediator (the *FontDialogDirector*) and Colleagues (the widgets). The goal of Mediator and ConcreteMediator participants is to provide the Colleagues with a mechanism to decouple them. When a change in a Colleague is produced it notifies the Mediator, which performs the corresponding actions (e.g. notify the change to the rest of Colleagues). So these participants perform the communication or notification protocol. On the other hand, the Colleague role is played by some classes which perform some functionality (e.g. the concrete widget behaviour).

Based on the analysis of participants, we determined the following concerns: Communication, because of the notification among colleagues and mediator; List and Text Field as a result of the different widgets behaviour; and finally Window, dealing with the behaviour of the window graphical component. A concern modelling technique such as [23] could also be used to discover the concerns.

Having these concerns (as source elements) and the classes of the UML class diagram shown above (as target elements), we obtain the dependency and crosscutting matrices (see Table 7).

As we can see in Figure 3, the *DialogDirector* class addresses both the Communication and the Window concerns, because it has methods for showing the dialog and for allowing widgets communication. In the dependency matrix, this is represented by mappings in cells [1,1] and [1,4]. The *FontDialogDirector* class only addresses the Communication concern because its behaviour (to notify changes produced in widgets). It must be observed that despite of *FontDialogDirector* inherits the *showDialog* method, this class doesn't redefine or even use this method. Consequently, there is no mapping to the Window concern (only a mapping in cell [1,2]). The *Widget* abstract class only provides the reference

of the *DialogDirector* to its subclasses. Accordingly, it only addresses the Communication concern. Finally, *ListBox* and *EntryField* simultaneously address their own behaviour and the communication concern (the inherited *Changed* method must be called once a change is produced).

Table 7. Matrices for the Dialog System

Concern	Dialog Director	Font Dialog Director	Widget	List Box	Entry Field	
Communication	1	1	1	1	1	S
List	0	0	0	1	0	NS
Field Text	0	0	0	0	1	NS
Window	1	0	0	0	0	
	T	NT	NT	T	T	

Design Class

Concern	Communication	List	Text Field	Window
Communication	0	1	1	1
List	0	0	0	0
Text Field	0	0	0	0
Window	0	0	0	0

Concern

In the crosscutting matrix, we can observe that the Communication concern crosscuts the List, Text Field and Window concerns. We conclude that - using our analysis based on dependency and crosscutting matrices - we identified crosscutting which emerged in a design based on the mediator pattern. The dependency matrix could be based on different decompositions. An AspectJ implementation of this design pattern can be found in [13]. This implementation removes the crosscutting from Mediator pattern.

3.3 Drawing editor (Adapter Pattern)

Sometimes, a toolkit class that is designed for reuse is not reusable only because its interface does not match the domain-specific interface required in an application [8].

The adaptation of a previously implemented interface to a new required one is known as the Adapter pattern or also a Wrapper. In [8] we can see an example where a *TextView* class which represents some text (that should be edited and drawn) must be adapted to fulfill a different interface. In the example, the authors add a new *TextShape* class where they implement this adaptation (the functionality regarding to Adapter pattern). The UML class diagram of this example is shown in Figure 4.

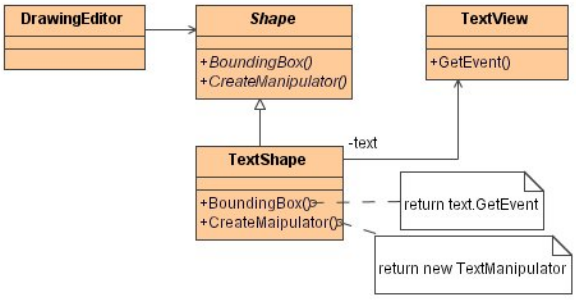

Figure 4. Adapter pattern applied to Drawing Editor [8]

The participants in this pattern are: Client, an object which must use an interface (the *DrawingEditor*); Target, the interface that Client wants to use (the *Shape*); Adaptee, the class whose interface must be adapted to the required one (the *TextView*); and Adapter, this is the class which adapts the Adaptee to the Target (the *TextShape*). As we did in last section, we analyze these participants in order to determine the concerns in this application. In this case we consider four concerns: one concern for each participant. On the other hand, the decomposition of the design is driven by the UML classes.

Taking these decompositions as input, the dependency and crosscutting matrices can be determined as shown in Table 8. The crosscutting matrix shows that there is no crosscutting in this case.

As we mentioned in section 3.2, the authors in [13] use AspectJ as implementation language to develop these patterns. They state that the advantages of implementing the Adapter pattern by means of AOP are almost inappreciable. From our analysis the reason becomes clear: this pattern does not require the utilization of AOP because there is no crosscutting.

Table 8. Dependency and crosscutting matrices for Adapter

Concern	Drawing Editor	Shape	Text Shape	Text View	
Client	1	0	0	0	NS
Target	0	1	0	0	NS
Adaptee	0	0	0	1	NS
Adapter	0	0	1	0	NS
	NT	NT	NT	NT	

Design Class

Concern	Client	Target	Adaptee	Adapter
Client	0	0	0	0
Target	0	0	0	0
Adaptee	0	0	0	0
Adapter	0	0	0	0

Concern

4. RELATED WORK

Several authors use matrices (design structure matrices, DSM) to analyze modularity in software design [4]. Lopes and Bajracharya [14] describe a method with clustering and partitioning of the design structure matrix for improving modularity of object-oriented designs. However, the design structure matrices represent intra-level dependencies (as coupling matrices mentioned in section 2.2) and not the inter-level dependencies as in the dependency matrices used for our analysis of crosscutting.

In [19], a relationship matrix (concern x requirement) is described very similar to our dependency matrix, and used to identify crosscutting concerns in requirements. However, there is no explicit operational definition of crosscutting.

The approach presented in [3] allows the requirements engineer to identify crosscutting concerns. However, the identification of crosscutting functional concerns is not yet clear and it lacks of explicit support (such as tool and guidelines) to identify non-functional crosscutting concerns. In [21] the authors have improved this approach by means of a mechanism based natural language processor to identify functional and non-functional crosscutting concerns from requirements documents. However this approach is focused only on requirements phases while our approach can be applied throughout the software life cycle.

In aspect mining area, there are several approaches to identify crosscutting, e.g. [10] [20]. However, most of these approaches are focused on the implementation level and they use different

pattern matching or dependencies graph for such identification. As we stated in section 1, we need support to apply such identification of crosscutting concerns in early phases. In that sense, the framework presented in this paper provides a mechanism which is perfectly suitable to be applied in any phase as we mentioned above.

A definition of crosscutting similar to ours can be found in [16] and [17]. Our definition is less restrictive showing that crosscutting is not symmetric property. The differences are explained in [6]. Moreover, our definition can be applied to consecutive levels of abstractions in software development, such as requirements, design and implementation. This can be achieved through the cascading of dependency matrices [6].

5. CONCLUSIONS

We proposed an operationalization of the definition of crosscutting, based on specific mappings between design elements at two levels, called source and target. We introduced the dependency matrix to represent and to visualize the mapping. We used this matrix to derive the crosscutting matrix and to identify crosscutting in software design.

We showed the application of this approach to well-know case studies to identify crosscutting concerns. The same analysis could be done in systems where new crosscutting concerns may emerge. The operationalization of crosscutting with matrices constitutes a helpful means to apply aspect-oriented modeling approaches in different scenarios or domains.

In another study (to be published), we applied a cascading of dependency matrices to model crosscutting relations across several levels, for example from concern modelling, to requirements, architectural design to detailed design and implementation. As such, the analysis of crosscutting can be based on traceability relationships as represented in the dependency matrices.

As a future work, we are establishing the requirements for the building of some automatic tools in order to derive the mappings between source and target elements. These tools avoid the manual filling of matrices making the approach more systematic and useful.

ACKNOWLEDGEMENT

This work has been carried out in conjunction with the AOSD-Europe Project IST-2-004349-NoE [1] and also partially supported by CICYT under contract TIN2005-09405-C02-02.

REFERENCES

[1] AOSD-Europe (2005). *AOSD Ontology 1.0 - Public Ontology of Aspect-Orientation.* Retrieved May, 2005, from http://www.aosd-europe.net/documents/d9Ont.pdf

[2] AOSD-Europe (2005*). Survey of Aspect-Oriented Analysis and Design Approaches.* Retrieved May, 2005, from http://www.aosd-europe.net/documents/analys.pdf

[3] J. Araujo, A. Moreira, I. Brito, & A. Rashid (2002), Aspect-Oriented Requirements with UML, In *Workshop on Aspect-Oriented Modelling with UML* at *International Conference on Unified Modelling Language*. Dresden, Germany.

[4] Baldwin, C.Y. & Clark, K.B. (2000). *Design Rules vol I, The Power of Modularity.* MIT Press.

[5] Baniassad, E. & Clarke, S. Theme (2004). An Approach for Aspect-Oriented Analysis and Design. In *26th International Conference on Software Engineering*, Edinburgh, Scotland.

[6] Berg, K.van den & Conejero, J. (2005). Disentangling crosscutting in AOSD: a conceptual framework, in *Second Edition of European Interactive Workshop on Aspects in Software*, Brussels, Belgium

[7] Berg, K. van den, & Conejero, J. (2005a), A Conceptual Formalization of Crosscutting in AOSD. In *Iberian Workshop on Aspect Oriented Software Development*, Technical Report TR-24/05 University of Extremadura (pp. 46-52). Granada, Spain.

[8] Gamma, E., Helm, R., Johnson, R., & Vlissides, J. (1995). *Design patterns. Elements of reusable object-oriented software.* Addison-Wesley.

[9] Gradecki, J.D. & Lesiecki, N. (2003). *Mastering AspectJ, Aspect-Oriented Programming in Java*. Wiley.

[10] Griswold, W. G., Kato, Y. & Yuan, J. J. (2000). Aspect browser: Tool support for managing dispersed aspects. In *Workshop on Multi-Dimensional Separation of Concerns* at *International Conference on Software Engineering*. Limerick, Ireland.

[11] Grundy, J. (2000). Multi-perspective specification, design and implementation of software components using aspects. In *International Journal of Software Engineering and Knowledge Engineering*, vol. 20.

[12] Hanenberg, S. & Unland, R. (2002). A Proposal for Classifying Tangled Code. *Workshop Aspekt-Orientierung der GI-Fachgruppe 2.1.9*. Bonn, Germany.

[13] Hanneman, J, & Kiczales, G. (2002). Design Pattern Implementation in Java and AspectJ. In *17th Annual ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications.* Seattle, USA.

[14] Lopes, C.V. & Bajracharya, S.K. (2005). An analysis of modularity in aspect oriented design. In 4th *International Conference on Aspect-Oriented Software Development*. Chicago, Illinois

[15] MDA (2003). MDA Guide Version 1.0.1, document number omg/2003-06-01

[16] Masuhara, H. & Kiczales, G. (2003). Modeling Crosscutting in Aspect-Oriented Mechanisms. In *17th European Conference on Object Oriented Programming*. Darmstadt

[17] Mezini, M. & Ostermann, K. (2003). Modules for Crosscutting Models. In *8th International Conference on Reliable Software Technologies*. LNCS 2655. Toulouse, France.

[18] Pawlak, R., Duchien, L., Florin, G., Legond-Aubry, F., Seinturier, L. & Martelli, L. (2002). A UML Notation for Aspect-Oriented Software Design. *In Workshop on Aspect-Oriented Modeling with UML* at *Aspect Oriented Software Development Conference*. Enschede, The Netherlands.

[19] Rashid, A., Moreira, A. & Araujo, J. (2003). Modularisation and Composition of Aspectual Requirements. In *Second Aspect Oriented Software Conference.* Boston, USA.

[20] Robillard, M. P. & Murphy, G. C. (2002). Concern graphs: finding and describing concerns using structural program dependencies. In *24th International Conference on Software Engineering* (pp. 406-416). Orlando, Florida.

[21] Sampaio, A., Loughran, L., Rashid, A. & Rayson, P. (2005). Mining Aspects in Requirements. In *Early Aspects 2005 Workshop* at *Aspect Oriented Software Development Conference*. Chicago, USA.

[22] Stein, D., Hanenberg, S. & Unland, R. (2002). A UML-based Aspect-Oriented Design Notation For AspectJ. In *First Aspect Oriented Software Development Conference*. Enschede, The Netherlands.

[23] Sutton, S. & Rouvellou, I. (2002). Modeling of Software Concerns in Cosmos. In *First Aspect Oriented Software Development Conference*. Enschede, The Netherlands

[24] UML (2004). *Unified Modeling Language 2.0 Superstructure Specification*. Retrieved October, 2004 from http://www.omg.org/cgi-bin/doc?ptc/2004-10-02

[25] Welch, I.S. & Stroud, R.J. (2003). Re-engineering Security as a Crosscutting Concern. The Computer Journal, 46(5), 578-589.