Analysis of crosscutting features in Software Product Lines

José M. Conejero
Universidad de Extremadura
Avda. de la Universidad s/n
10071, Caceres, Spain
+34 927257190
chemacm@unex.es

Juan Hernández
Universidad de Extremadura
Avda. de la Universidad s/n
10071, Caceres, Spain
+34 927257257
juanher@unex.es

ABSTRACT

Software Product Lines has emerged as a new technology to develop software product families related to a particular domain. The software products developed by this methodology are based on the combination of a set of common and variable assets. However, in order to combine these assets to build different products, coupling between common and variable parts must be highly reduced. In that sense, crosscutting features make evolution and adaptability of software difficult. In this paper we propose a framework to identify crosscutting features at early stages in order to use aspect-oriented techniques to modularize them and reduce their dependencies. This framework is based on a crosscutting pattern and uses traceability matrices to perform the analysis of crosscutting. Finally, applicability of the framework is shown by identifying crosscutting features in the Arcade Game Maker product line.

Categories and Subject Descriptors

D.2.1 [**Software Engineering**]: Requirements Engineering – *Elicitation methods, Methodologies, Tools.*

D.2.13 [**Software Engineering**]: Reusable Software – *Domain engineering, Reuse models.*

General Terms

Theory, Management.

Keywords

Software product lines, crosscutting features.

1. INTRODUCTION

Software Product Line engineering has emerged as a new technology to build complex software systems. This new methodology is focused on developing software products families related to a particular business area or domain. As it is stated in [20]: *Software product line engineering has proven to be the methodology for developing a diversity of software products and software-intensive systems at lower costs, in shorter time, and with higher quality.* One of the key techniques used in Product Line Engineering (PLE) is the Feature-Oriented Analysis (FOA). By this analysis, variability between products of a same family is identified. In this way, software development becomes now a process based on the combination of a set of common and variable assets. However, as it was stated in [11] and [6] dependencies between these assets make their combination difficult, reducing flexibility and adaptability of the products developed. The more independent the assets are, the easier the products family may be built. Even, the problem is more important when dependence relations exist between common and variable assets or vice verse. In that sense, some features may crosscut each other, thus reducing the modularity of the PL assets.

In [11], the authors introduce different approaches to improve modularity in PL such as the utilization of design patterns or Aspect-Oriented Software approaches. Although the former has been widely used in the industry, it still has several problems that can be solved by the latter [12] [10]. In this paper, we focus on the utilization of aspect-oriented techniques to improve modularity in SPL. Obviously, the utilization of aspect-oriented techniques needs a previous step where the crosscutting features must be identified. In that sense, we present a framework to allow such an identification. By means of aspect-oriented techniques, those features that crosscut other ones may be isolated and modularized so that dependencies are highly reduced. The framework presented in the paper is based on our previous work in [5] where we provided a new formal definition of crosscutting. The framework is based on a crosscutting pattern and traceability relations between source and target domains. These trace relations may be represented by a special kind of traceability matrix that we called dependency matrix. By using simple matrix operations, we obtain a different matrix called crosscutting matrix. This matrix allows identifying the elements of the source domain that crosscut to other elements of the same domain. Although the framework may be applied to different domains or life-cycle stages (e.g. [5] [4]), in this paper we focus on the early phases of a product line development so that source and target are related to features and requirements artifacts respectively. Therefore, this paper adds a new contribution to our previous work in [5], in particular, we adapt the framework for the product line domain, adding some tasks related to the feature-oriented analysis. Moreover, we improve the traditional feature-oriented techniques adding a mechanism to identify crosscutting features.

The rest of paper is structured as follows: in Section 2 we show some background related to the work presented in this paper. Section 3 shows the framework presented to identify the

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

EA'08, May 12, 2008, Leipzig, Germany.

Copyright 2008 ACM 978-1-60558-032-6/08/05...$5.00.

crosscutting features. The framework is illustrated by applying it to a well-know case study, the Arcade Game Maker product line company [1]. Finally in Sections 4 and 5 we show the related works and conclusions of the paper respectively.

2. BACKGROUND

In this section we briefly explain the details of our approach presented in [5].

2.1 The crosscutting pattern

In [5] we presented a conceptual framework where crosscutting can be clearly distinguished from scattering and tangling. The framework is based on a crosscutting pattern which is used to represent frequently encountered situations where two levels are related to each other (see Figure 1). We called these levels source and target respectively.

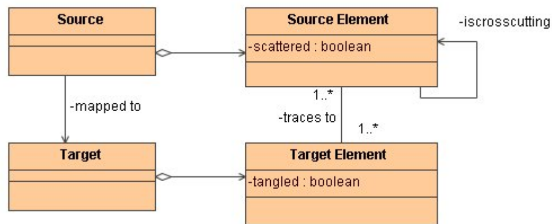

Figure 1. The crosscutting pattern

Although the terms of source and target could represent two different domains, levels, or phases of a software development process, the crosscutting pattern abstracts from specific phases, such as concern modeling, requirements elicitation, architectural design and so on. The only proposition is that crosscutting is defined with respect to two levels of abstraction, the source and the target. Examples of source and target could be concerns of the system and the use cases or classes which implements such concerns respectively.

The terms scattering, tangling and crosscutting are defined as specific cases of the mapping between "source" and "target". We say that scattering occurs when, in a mapping between source and target, a source element is related to multiple target elements. Similarly, we can focus on the relation between target elements and source elements. We say that tangling occurs when, in a mapping between source and target, a target element is related to multiple source elements. There is a specific combination of tangling and scattering which we call crosscutting, defined as follows: Crosscutting occurs when, in a mapping between source and target, a source element is scattered over target elements and where in at least one of these target elements, some other source element is tangled. In [7] we show a formal comparison between our definition of crosscutting and other ones existing in the literature (e.g. the introduced by Masuhara and Kiczales in [17]). In next section we show how the mappings between source and target may be represented by means of traceability matrices.

2.2 Identification of crosscutting

In terms of linear algebra, traceability matrices show the mappings between source and target. In [5] these mappings (source x target) are shown in a special kind of traceability matrix that we called dependency matrix. 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). Scattering and tangling can easily be visualized in this matrix. In Table 1 we show an example of dependency matrix with three and four source and target elements respectively. In this matrix, we show a 1 in a cell when the target element of the corresponding column contributes or addresses the source element of the corresponding row (in Table 1, t[1] and t[4] contribute to the functionality of s[1]). Based on this matrix, crosscutting source elements may be identified. In order to make the process computable, we defined some simple matrix operations which allow automating the process.

Based on the dependency matrix, two different matrices called scattering matrix and tangling matrix are derived, which show the scattered and tangled elements in a system respectively:

- In a 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 a 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).

Table 1. Example dependency matrix

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

The scattering and tangling matrices for dependency matrix shown in Table 1 are presented in Table 2.

Table 2. Scattering and tangling matrices for Dependency matrix shown in Table 1

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

Then the crosscutting product matrix is defined, showing the quantity of crosscutting relations. The crosscutting product matrix ccpm can be obtained through the matrix multiplication of the scattering matrix sm and the tangling matrix tm: ccpm = sm x tm where ccpm [i][k] = Sum{j = 1..n} sm[i][j] * tm[j][k]. This matrix is used to derive the final crosscutting matrix. In the crosscutting

matrix, a matrix cell denotes the occurrence of crosscutting; it abstracts from the quantity of crosscutting. The crosscutting matrix ccm can be derived from the crosscutting product matrix ccpm using a simple conversion: ccm[i][k] = if (ccpm[i][k] > 0) $\wedge$ (i $\neq$ j) then 1 else 0. In Table 3 we show just the final crosscutting matrix for the example. As we can see in this table, we do not consider crosscutting as a symmetric property (e.g. s[1] crosscuts to s[3] but s[3] does not crosscut to s[1]).

Table 3. Crosscutting matrix for the example

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

3. IDENTIFYING CROSSCUTTING FEATURES

As we have mentioned in Section 1, flexibility and configurability in product lines may be improved by aspect-oriented techniques. The more independent the different features are, the less effort we must spend for adding new products to the line. The desired situation is represented in Figure 2, where the different products are built by combining common and different variable features. However, usually we do not find this situation in real systems, having many dependencies between features. These dependencies usually imply that crosscutting features emerge.

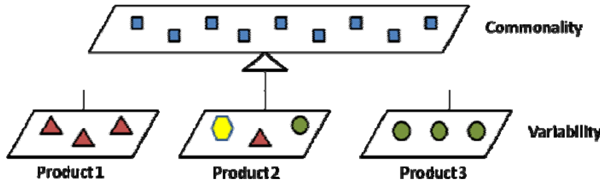

Figure 2. Combination of commonality and variability in product lines

In this section we explain how to perform the analysis to obtain the crosscutting features in a product line development. By the identification of these features, the developer may reduce the dependencies between them allowing a better reutilization of the common assets for the product family.

The analysis of crosscutting features consists of several steps which we detail next:

1) We perform a Feature-Oriented Analysis to obtain the main features of the product family including both common and variable assets.
2) Non-functional requirements are identified. In order to perform this analysis we use a non-functional concerns catalogue.
3) Requirements of the product or products are represented by a modeling language. We use UML as this language; in particular we utilize use case diagrams.
4) Taking the system features and the non-functional requirements as source and the use cases of the system as target, we build a dependency matrix as we explained in Section 2.2.
5) Based on the dependency matrix, we perform the matrix operations explained in the previous section and obtain the final crosscutting matrix which identifies the crosscutting features (and also the non-functional crosscutting requirements).

In order to illustrate the whole process, we apply it to a well-know case study, the Arcade Game Maker (AGM) [1] product line company. This company has been also created (fictional) to illustrate the product line concepts and is focused on the development of three simple arcade games, each in three variations. The company uses an iterative approach to develop a set of games in the three variations: freeware set, wireless set and customizable set. In this paper we focus just on the first variation, freeware set. The games that the company develops in this variation are: Brickles, Pong and Bowling. Although the development of other games, like traditional "board" games, could be feasible in the product line, they are out of the scope of the company goals by now.

3.1 Feature-Oriented Analysis (FOA)

Domain Analysis allows the developer to improve the understanding of software requirements. There are several domain analysis approaches such as the Feature-Oriented Domain Analysis (FODA) [13]. The FODA methodology includes several steps to perform a whole domain analysis like the construction of a context model, a features model or an entity-relationships model. In this section, we focus on the features model for the Arcade Game Maker product line. We have adapted the feature model shown in [1]. In this paper we mainly focus on the functional and operational features and skip out the configuration issues such as the mouse driver utilization and the display quality. The feature model used for the example is shown in Figure 3.

In Figure 3 we have represented the features that the product line has. We have used the XFeature [24] Eclipse plugin to make this model. As we can see in the figure, variability between products is mainly concentrated in the different rules that the games must fulfill. In [1], the authors of the case study performed a commonality and variability analysis. Next, we summarize what is common and variable for the AGM product line:

Commonalities:

- Every game will have a set of Sprites (elements which the players see and interact to).
- Every game has a set of rules.
- All the games involve movement.

Main variations:

- Rules. The games may have some common rules but also other specific to each game.
- Motion initiation. In some games, the player must initiate the motion. In other games the initiation is driven by time and it is inherent to the operation of the game.

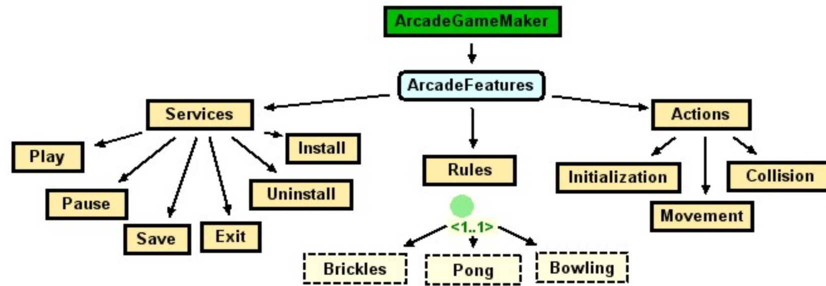

Figure 3. Features model for the Arcade Game Maker product line

3.2 Non-functional requirements elicitation

In the next step in the process we must identify the non-functional requirements involved in the product line. These non-functional requirements may constrain the rest of features of the system. In order to identify such requirements, we use a catalogue which extends the catalogue used in [21] with new non-functional concerns. The catalogue consists of a XML file where common non-functional concerns are presented and related to different words that usually appears in requirements documents. We use these words to analyze the stakeholder requirements so that non-functional concerns are identified when one of these words appears in the text. In Figure 4 we may observe an example of some words related to the Persistence concern defined in the catalogue.

```
<?xml version="1.0" encoding="UTF-8"?>
<catalogue>
   <word content="store" nfc="Persistence"/>
   <word content="database" nfc="Persistence"/>
</catalogue>
```

Figure 4. Part of the catalogue of non-functional concerns

In [1], the authors identified some non-functional requirements such as Performance, Display Quality, Evolvability and Maintainability. In addition to these non-functional concerns we identified other ones like Persistence (the game must be loaded or saved from file) and Data Representation (the game board and the sprites must be represented). Some of these non-functional requirements may be just related to design or hardware decisions, e.g. Evolvability and Maintainability may be addressed by means of using different developing approaches (in this paper, we propose aspect-oriented ones) or Display Quality by using different screens. Then, we take the rest of non-functional concerns (Performance, Persistence and Data Representation) for the analysis of crosscutting since they may constrain the rest of features.

3.3 Requirements modeling

In this activity the requirements engineer must build the first representation of the system using some requirements language or notation. In our example, we use UML use case diagrams [23] to represent the requirements. The use case diagram for the AGM product line is shown in Figure 5.

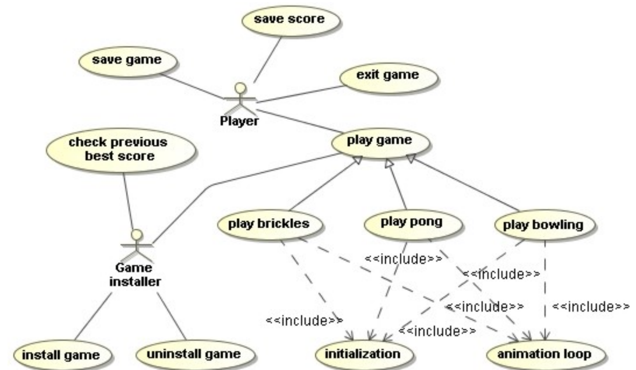

Figure 5. Use case diagram for the AGM product line (taken from [1])

Once we have identified the features of the products (including the non-functional requirements) and we have modeled the requirements, we use all this information to build a dependency matrix.

3.4 Building the dependency matrix

The next activity in the process presented consists of the construction of the dependency matrix (see Section 2.2). This matrix serves as starting point for the analysis of crosscutting performed. In order to build the dependency matrix we must first decide the source and target elements. In this example, we take as source the features and non-functional requirements presented in Sections 3.1 and 3.2 respectively. The requirements modeled and presented in the previous section are taken as target for the analysis. Since dependency matrix represents the relations between source and target, in the dependency matrix we show the use cases (columns) which address the features presented in the feature model (rows). Likewise, we also relate the non-functional requirements (also in rows) with either the use cases which contribute to the realization of such requirements or which are constrained by such requirements. The dependency matrix built is show in Table 4.

Table 4. Dependency matrix for the Arcade Game Maker product line case study

		Use cases										
		Save Game	Exit Game	Save Score	Play Brickles	Play Bowling	Play Pong	Check Previous Best Score	Initialization	Animation Loop	Install	Uninstall
Features	Play				1	1	1					
	Pause		1									
	Save	1	1	1				1				
	Exit		1									
	Install										1	
	Uninstall											1
	Rules_Brickles				1					1		
	Rules_Bowling					1				1		
	Rules_Pong						1			1		
	Initialization				1	1	1		1			
	Movement				1	1	1			1		
	Collision				1	1	1			1		
NFR's	Performance				1	1	1			1		
	Data_Representation				1	1	1			1		
	Persistence	1		1					1		1	1

As an example of the relations represented in the matrix, we can see how the Animation Loop use case is contributing to different features of the product line. For instance, since the Animation Loop use case performs the movement algorithms of the different games, it is related to the Movement and Collision features. Moreover, this use case is constrained by the Performance requirement because of the response of the different games. This use case is also addressing the different rules of the games (main variable features of the product line). In [8] we have introduced some ideas on how to identify the relations between source and target elements based on syntactic and dependencies based analyses. By means of these analyses, the engineer may automatically fill in the dependency matrix. We do not introduce here such ideas for space reason.

In order to fill in the matrix, we use the relationships existing in the use case diagram. In particular, we focus on the <<include>> relations. Whenever a <<include>> relation is found, we apply a rule similar to what other authors have called the Alignment dependency principle [6]. If we find an <<include>> relation from the use case A to the use case B, then use case A is contributing to the concern being addressed by the use case B. For instance, in the use case diagram shown in Figure 5, the Play Brickles use case includes the functionality of the Initialization use case. Since the Initialization use case is addressing the Initialization feature, we say that the Play Brickles use case is also contributing or addressing the Initialization feature. This is why we placed a 1 in the corresponding cell of Table 4.

3.5 Matrix operations

In this activity, we take as input the dependency matrix built in the previous step and perform the matrix operations explained in Section 2.2 to obtain a crosscutting matrix where crosscutting features are identified and visualized. As we showed in Section 2.2, the crosscutting matrix is derived from the product of two different matrices called scattering and tangling matrix. For space reasons, we do not show these two matrices and we just show the crosscutting matrix (see Table 5).

Table 5. Crosscutting matrix obtained from the dependency matrix shown in Table 4

		Features												NFR's		
		Play	Pause	Save	Exit	Install	Uninstall	Rules_Brickles	Rules_Bowling	Rules_Pong	Initialization	Movement	Collision	Performance	Data_Representation	Persistence
Features	Play							1	1	1	1	1	1	1	1	
	Pause															
	Save		1		1											1
	Exit															
	Install															
	Uninstall															
	Rules_Brickles	1							1	1	1	1	1	1	1	
	Rules_Bowling	1						1		1	1	1	1	1	1	
	Rules_Pong	1						1	1		1	1	1	1	1	
	Initialization	1						1	1			1	1	1	1	1
	Movement	1						1	1		1		1	1	1	
	Collision	1						1	1		1	1		1	1	
NFR's	Performance	1						1	1		1	1	1		1	
	Data_Representation	1						1	1		1	1	1	1		
	Persistence			1		1	1				1					

In Table 5 we can see how there are several features and requirements that may be candidates to be modularized by aspects. A cell with 1 in this matrix indicates that the feature or requirement of this row is crosscutting to the feature of the corresponding column. In some cases, the crosscutting features are part of the variability of the product line such as the different rules of the games. The modularization of variable parts of the system by aspect oriented techniques allows the developer to be able to add or remove features just applying different aspects to the system. The addition of aspects to model such crosscutting features allows considerably reducing the number of dependencies between features. This improvement is even better when the crosscutting features belong to variability of the product line, however since the framework may also identify as crosscutting features some common parts of the product line, we do not just focus on variability.

In [18], the authors present a method to modularize volatile concerns at requirements level by aspect-oriented techniques. They utilize a Use Case Pattern Specification and some templates to "mark" the use cases which address some volatile concerns. We used the same technique to refactor the crosscutting concerns in [5]. In our AGM case study we could apply the same approach in order to refactor the crosscutting features. For space reasons, we do not show the refactorization of the AGM system here.

Of course, as we mentioned in the introduction, there are other techniques (different from aspect-oriented ones) that we may use

to get a high level of flexibility or configurability. For instance, in order to address the different rules that the games have, we could have used the Command design pattern, introduced in [9]. However, as it has been demonstrated in several publications like [12] and [10], the utilization of aspect-oriented techniques considerably improves the benefits obtained by the utilization of some design pattern, e.g. the Command one.

4. RELATED WORKS

There are several works that have introduced the need of modelling features in software product lines by aspect-oriented techniques. In [11], the author claims that several technologies may be integrated into a product line development, improving the time-to-market, evolvability and flexibility of products developed. The author discusses on the different techniques that may help in the task of reducing dependencies between components. Some of these approaches are the utilization of traceability tools, design patterns, or aspect-oriented tools. However, the author just explains the need for incorporating these new methodologies into the product line development, and he does not focus on how to face the problem of combining such technologies.

In [6] the authors also propose aspect-oriented techniques to improve flexibility and configurability in software systems, especially in product lines, where these characteristics are even more important. The authors introduce an aspect categorization into three different categories: orthogonal, weakly orthogonal and non-orthogonal aspects. The orthogonal aspects are those which may be added or removed from a system without implying any change in the rest of components of the system. Weakly orthogonal may be added but not removed without any change in the rest of components. Finally, non-orthogonal aspects imply changes in the components of a system whenever they are added or removed from the system. The more orthogonal the aspects are the less dependencies among aspects and components the system has. So product line developments may be improved by aspect-oriented techniques and the improvement is higher when orthogonal aspects are used. Nevertheless, although in [6] the authors give an idea on how to introduce AOSD in product lines, they do not introduce a particular approach to support this contribution.

In [16] the authors propose NAPLES, an interesting method to obtain viewpoints, crosscutting concerns and variability from requirements documents. The method uses the EA-Miner [21] and WMATRIX [22] tools. These tools allow making a syntactic and semantic analysis of the documents searching for key words. Although it reduces considerably the time needed to understand the system requirements, the identification of variability is performed by means of reading the surrounding text of the key words. However, this identification highly depends on the experience of the engineer and it may lose some variability not present in the surrounding text. Moreover, this approach only may be applied in requirements elicitation, whereas our approach may be used in any situation where the crosscutting pattern matches (e.g, other phases of the development process).

The work presented in [14] focuses on the application of aspects in product lines to model variability. The approach uses AspectJ [2] as programming language to implement such features. However, the authors do not explain how the crosscutting features (common or variable) may be identified at early stages and they just focus on the programming level. The framework presented in this paper deals with this identification at early stages of the development. In [19], the authors propose a framework for weaving models using aspect oriented techniques. The framework allows modelling variability so that these features may be added to the system by weaving different models. However, the approach just focuses on variability and does not explain how crosscutting features may be identified.

Several authors use matrices (design structure matrices, DSM) to analyze modularity in software design [3]. Lopes and Bajracharya [15] 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) and not the inter-level dependencies as in the dependency matrices used for our analysis of crosscutting.

5. CONCLUSIONS

It has been demonstrated by several publications that software product line developments may be improved by means of the introduction of aspect-oriented techniques in such developments. In particular, AOSD addresses some problems of flexibility, configurability and reutilization that other paradigms may not solve. The framework presented in this paper allows identifying crosscutting features in SPL in order to be modularized by aspect-oriented techniques. This framework consists of several activities that assist the developer in the process. These activities include a feature-oriented analysis, a non-functional requirements analysis, and a requirements modeling (by use cases). Taking the results of these analyses as input, we build a traceability matrix which represents the relations among features and non-functional requirements and the use cases which contributes to them. By simple matrix operations, we derive a crosscutting matrix where crosscutting features and requirements are identified. Therefore, this matrix provides important information that may be used to take decisions at later stages, e.g. using aspect-oriented modeling techniques to modularize such features or requirements.

6. REFERENCES

[1] *Arcade Game Maker Pedagogical Product Line.* http://www.sei.cmu.edu/productlines/ppl/

[2] AspectJ Team, *AspectJ Project*, http://www.eclipse.org/aspectj/

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

[4] Berg, K. van den, Conejero, J. and Hernández, J. (2006). *Identification of crosscutting in software design*. In Aspect-Oriented Modeling Workshop at 5th AOSD, Bonn.

[5] Berg, K. van den, Conejero, J. and Hernández, J. (2007) *Analysis of Crosscutting in Early Software Development Phases based on Traceability*. Transactions on AOSD III, LNCS 4620, pp. 73–104, Springer-Verlag. Special issue on Early-Aspects.

[6] Colyer, A., Rashid, A. and Blair, G. (2004). *On the Separation of Concerns in Program Families*. Lancaster University Technical Report Number: COMP-001-2004

[7] Conejero, J.,Hernandez, J., Jurado, E. and Berg, K.van den (2007). Crosscutting, what is and what is not?: A Formal definition based on a Crosscutting Pattern. *Technical Report*

TR28/07, University of Extremadura, March 2007. http://quercusseg.unex.es/chemacm/research/TR3_07.pdf

[8] Conejero, J.,Hernandez, J., Jurado, E. and Berg, K.van den (2008). *Aspect Mining in Requirements based on Syntactical Analysis and Dependencies.* Submitted to the 16th IEEE International Conference on Requirements Engineering, Barcelona, Spain, 2008.

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

[10] Garcia, A., Sant'Anna, C., Figueiredo, E., Kulesza, U., Lucena, C. and Staa, A. (2005) *Modularizing Design Patterns with Aspects: A Quantitative Study*. LNCS Transactions on Aspect-Oriented Software Development, Springer, 2005

[11] Griss, M. (2000) *Implementing product-line features by composing aspects.* In proceedings of First International Software Product Line Conference, pp. 271—288, Denver, USA

[12] Hannemann, J. and Kiczales, G. (2002). *Design pattern implementation in Java and AspectJ.* In proceedings of 17th ACM conference on Object-oriented programming, systems, languages, and applications pp. 161—173. Seattle, USA

[13] Kang, K., Cohen, S., Hess, J., Novak, W. and Spencer A. (1990). *Feature Oriented Domain Analysis (FODA). Feasibility Study.* Carnegie Mellon University Technical Report CMU/SEI-90-TR-21

[14] Lee, K., Kang, K., Kim, M. and Park, S. (2006). *Combining Feature-Oriented Analysis and Aspect-Oriented Programming for Product Line Asset Development.* In proceedings of the 10th International Software Product Line Conference (SPLC), Baltimore, USA

[15] 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

[16] Loughran N., Sampaio, A. and Rashid, A. (2005). *From Requirements Documents to Feature Models for Aspect Oriented Product Line Implementation*. In proceedings of workshop on MDD for Product Lines at MODELS 2005. Montego Bay, Jamaica

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

[18] Moreira, A., Araujo, J. & Whittle, J. (2006). *Modeling Volatile Concerns as Aspects*. In 18th Conference on Advanced Information Systems Engineering. LNCS 4001/2006: 544-558. ISBN: 978-3-540-34652-4, Luxembourg.

[19] Morin, B., Barais, O. and Jézéquel, J.M., (2008). *Weaving Aspect Configurations for Managing System Variability*. In Proceedings of Second International Workshop on Variability Modelling of Software-intensive Systems (VaMoS'08), Essen, Germany

[20] Pohl, K., Böckle, P. and Linden, F. van der (2005). *Software Product Line Engineering: Foundations, Principles and Techniques*. Springer, ISBN-10 3-540-24372-0

[21] Sampaio, A., Chitchyan, R., Rashid, A. and Rayson, P. (2005) *EA-Miner: A Tool for Automating Aspect-Oriented Requirements Identification*. In Proceedings of International Conference on Automated Software Engineering (ASE'05), Long Beach, California, USA

[22] Sawyer, P., Rayson, P. and Garside, R. (2002*). REVERE: Support for Requirements Synthesis from Documents.* Information Systems Frontiers, vol. 4, p. 343-353

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

[24] XFeature, Feature Modelling Tool. http://www.pnp-software.com/XFeature/Home.html