THE CROSSCUTTING PATTERN: A CONCEPTUAL FRAMEWORK FOR THE ANALYSIS OF MODULARITY ACROSS SOFTWARE DEVELOPMENT PHASES.

by

José María Conejero Manzano

Computer Science Department

Polytechnic School

UNIVERSITY OF EXTREMADURA

Submitted in partial fulfilment of the requirements for the degree of Doctor of Philosophy in Computer Science

(*Doctor Europeous* mention)

Supervised by Prof. Juan María Hernández Núñez

Cáceres, April 2010

Edita: Universidad de Extremadura
Servicio de Publicaciones

Caldereros 2. Planta 3ª
Cáceres 10071
Correo e.: publicac@unex.es
http://www.unex.es/publicaciones

D. **Juan María Hernández Núñez**, Catedrático de Universidad del Área de Lenguajes y Sistemas Informáticos del Departamento de Ingeniería de Sistemas Informáticos y Telemáticos de la Universidad de Extremadura y Doctor en Informática por la Universidad Politécnica de Madrid

certifica que,

- D. **José María Conejero Manzano**, Ingeniero en Informática y Profesor Ayudante del Departamento de Ingeniería de Sistemas Informáticos y Telemáticos de la Universidad de Extremadura, ha realizado bajo su dirección el trabajo de investigación presentado en esta Tesis Doctoral, titulada:

  THE CROSSCUTTING PATTERN: A CONCEPTUAL FRAMEWORK FOR THE ANALYSIS OF MODULARITY ACROSS SOFTWARE DEVELOPMENT PHASES

  (EL PATRÓN DE CROSSCUTTING: UN MARCO DE TRABAJO CONCEPTUAL PARA EL ANÁLISIS DE LA MODULARIDAD A TRAVÉS DE LAS FASES DEL DESARROLLO DE SOFTWARE)

- Asimismo, una vez revisado este trabajo, estima que puede ser presentado al tribunal que debe evaluarlo y autoriza la presentación de esta Tesis Doctoral, con mención de Doctor Europeo, en la Universidad de Extremadura.

Cáceres, abril de 2010

Fdo. *D. Juan María Hernández Núñez*
Catedrático de Universidad del Departamento
de Ingeniería de Sistemas Informáticos y Telemáticos.

Advisor:

Prof. **Juan María Hernández Núñez,** University of Extremadura, Spain.

Dissertation committe members:

Prof. **Antonio Vallecillo Moreno**, University of Malaga, Spain.
Prof. **Mónica Pinto Alarcón**, University of Malaga, Spain.
Prof. **Xavier Franch Gutiérrez**, Universitat Politècnica de Catalunya (UPC), Spain
Prof. **Ana Moreira**, Universidade Nova de Lisboa, Portugal.
Prof. **Fernando Sánchez Figueroa**, University of Extremadura, Spain.

External reviewers:

Prof. **Klaas van den Berg**, University of Twente, the Netherlands.
Prof. **João Araújo**, Universidade Nova de Lisboa, Portugal.

To the memory of my grandparents...

A la memoria de mis abuelos...

At that time I had a dream. I dreamt that decomposing the whole system into blocks with clearly defined signal interfaces would be an accepted technology...

... I still have another dream. I dream that we will get a component marketplace where different players can work. Some will play the role of selling components; others will buy components.

- Ivar Jacobson

Abstract

Enhancing business performance in contemporary domains requires systems whose size and intricacy challenge most of the current software engineering methods and tools. In this setting, a wide spectrum of methodologies, models, languages and tools have been adopted to deal with the increasing complexity of software systems. Aspect-Oriented Software Development (AOSD) is one of these methodologies that have emerged to tackle the design and development of complex software systems.

One of the key principles in AOSD is Separation of Concerns (SOC). Related with this principle is the problem of crosscutting concerns. Crosscutting is usually described in terms of scattering and tangling. However, the distinction between these concepts has been traditionally left to developers' intuition, sometimes leading to ambiguous statements and confusion. In that sense, precise definitions are required for certain research areas, e.g. for the identification of crosscutting concerns or the definition of software metrics. This thesis proposes a conceptual framework that allows the formal definition of the terms of scattering, tangling and crosscutting. The conceptual framework is based on the concept of crosscutting pattern which denotes to the situation where two different domains, called source and target, are related by a traceability link or mapping. The terms of scattering, tangling and crosscutting are defined, thus, as special cases of this mapping. The utilization of this formal definition aims at identifying situations of crosscutting. Since the crosscutting pattern is not tied to any specific deployment artefact, it may be applied to any development phase, allowing the identification of crosscutting at any abstraction level (from requirements to implementation). Moreover, the crosscutting pattern can be applied across several refinement levels enabling traceability of crosscutting concerns.

Usability of the framework is illustrated by means of applying it to several research areas such as aspect mining, software assessment, identification of crosscutting features in Software Product Lines (SPL) or maintainability analysis. As it is aforementioned, the utilization of the conceptual framework helps to identify crosscutting at any abstraction level. Aspect mining is its main application area. Aspect mining approaches have traditionally focused on the programming level, where architectural decisions have already been made. In this setting, in this thesis an aspect mining process to identify crosscutting is presented. The process extends the conceptual framework by using syntactical and dependency-based analyses to automatically identify trace relations between source and target elements. Although the process may be used at any development phase, its utilization is illustrated at the requirements level. The identification of crosscutting concerns at early stages of development aims at incorporating the benefits of aspect-orientation at the very beginning of the development process. Moreover, early aspect refactoring is given for UML use cases diagrams, improving modularity of the system. Using this refactoring, the system may be easily evolved just using simple composing rules which allow the weaving of base and crosscutting concerns.

The conceptual framework also allows the definition of concern driven metrics. These metrics may be used at indications for modularity assessment. Then, using these metrics, the aspect mining process previously presented may be completed. In particular, the metrics enable the application of an empirical analysis of modularity measuring the degree of crosscutting in a system. Again, the metrics presented are language-agnostic so that they are not tied to any development artefact. However, in order to illustrate its applicability, canonical instantiations of the crosscutting metrics are given for use cases in this thesis. The metrics are theoretical and empirically validated. By the theoretical validation, its accuracy for measuring crosscutting properties is demonstrated. This validation is performed by comparing the results obtained by the metrics with those obtained by similar metrics previously introduced in the literature. The results show how the metrics presented generalises existing ones. They also provide evidences of the need of an specific metric for crosscutting that other metrics suites lack of.

On the other hand, by the empirical validaton of the metrics, its utility is demonstated in terms of being related to other software quality attributes in expected ways. In this thesis, this utility has been demonstrated by relating them to two ISO/IEC 9126 maintainability attributes, namely stability and changeability. In particular, a first and original exploratory study is shown which investigates the correlation between early crosscutting metrics and stability or changeability. The results obtained empirically demonstrate how crosscutting negatively affects to software stability and changeability at early stages of development. These results empirically support the ideas introduced by the aspect community (through several years) claiming that crosscutting is usually harmful to software quality.

The applicability of the conceptual framework is, finally, illustrated by applying the aspect mining process to identify crosscutting features in the Software Product Line (SPL) domain. Crosscutting is a special kind of dependency that compromises the composition and reutilization of SPL. Then, the identification of these dependencies helps to reduce them by refactoring these crosscutting features using aspect-oriented techniques. The need for introducing aspect-oriented techniques to model variable features in SPL has been introduced by several approaches in the literature. However, most of these approaches lack of a process to identify crosscutting features so that they consider as crosscutting features either all the variable features or the well-know crosscutting features widely identified in the literature. The utilization of the aspect mining process presented in this document automates the identification of the crosscutting features allowing their isolation and refactoring. The concern driven metrics proposed have been also applied to the SPL domain showing the benefits obtained by the assessment of crosscutting in SPL as well, e.g. anticipating the impact of a feature change before it occurs.

Resumen

La creciente complejidad de los entornos de negocio actuales requiere sistemas software cada vez más complejos. Esta complejidad plantea un reto para los métodos y técnicas de ingeniería de software actuales. En este contexto, recientemente han aparecido nuevas metodologías, modelos, lenguajes y herramientas para hacer frente a esta complejidad de los sistemas software. El Desarrollo de Software Orientado a Aspectos (AOSD) es una de estas nuevas metodologías que han surgido para facilitar el diseño y desarrollo de sistemas software complejos.

Uno de los principios clave de AOSD es la separación de asuntos (Separation of Concerns) que lleva acarreado el problema conocido de la aparición de asuntos transversales (*crosscutting concerns*). El concepto de *crosscutting* es descrito normalmente en términos de otros dos conceptos, conocidos como *scattering* (esparcido) y *tangling* (mezclado). Sin embargo, normalmente la identificación y diferenciación de estos conceptos se ha basado en la propia experiencia e intuición del desarrollador, implicando, en muchas ocasiones, que estos conceptos sean confundidos. La utilización de definiciones precisas y formales es fundamental para ciertas áreas de investigación, como la identificación de asuntos transversales o la definición de métricas de software. En este sentido, en esta tesis se propone un marco de trabajo conceptual que permite la definición formal de los conceptos *scattering, tangling* y *crosscutting.* Este marco de trabajo conceptual está basado en lo que se ha denominado el patrón de crosscutting (*crosscutting pattern*). El patrón de crosscutting es un concepto utilizado para representar la situación en la que dos dominios identificados, denominados genéricamente como fuente (*source*) y destino (*target*), están unidos mediante algún tipo de relación de traza (*traceability relation* o *mapping*). Así, los conceptos de *scattering, tangling* y *crosscutting* se definen en base a casos especiales de esta relación de traza. La utilización de la definición formal presentada permite la identificación de situaciones de *crosscutting* en sistemas software. Incluso, dado que el patrón de crosscutting no ha sido definido en términos de entidades o artefactos de desarrollos específicos, éste puede ser aplicado en cualquier fase del desarrollo, permitiendo la identificación de situaciones de *crosscutting* a diferentes niveles de abstracción. Además, el patrón de crosscutting puede aplicarse a través de varios niveles de refinamiento, permitiendo la trazabilidad de *crosscutting concerns* a lo largo de las diferentes etapas del ciclo de vida.

Para ilustrar la usabilidad del marco de trabajo definido, éste ha sido aplicado en diferentes áreas de investigación como la minería de aspectos, el análisis empírico de software, la identificación de características transversales en el dominio de las Líneas de Producto Software (SPL) o el análisis de características relacionadas con el mantenimiento del software. Como ya se ha comentado anteriormente, la utilización del patrón de crosscutting permite la identificación de *crosscutting* a diferentes niveles de abstracción. Las propuestas de minería de aspectos existentes en la literatura se han centrado tradicionalmente en la identificación de situaciones de *crosscutting* en fases de implementación, cuando las principales decisiones a nivel de arquitectura ya han sido tomadas. En este contexto, esta tesis presenta un proceso de minería de aspectos. Este proceso extiende el marco de trabajo presentado mediante la utilización de diferentes tipos de análisis, como análisis sintácticos o basados en dependencias. Mediante estos análisis, las relaciones de traza existentes entre los elementos de los dominios fuente y destino son identificadas de manera automática. Este proceso de minería de aspectos puede ser utilizado en cualquier fase del desarrollo. Sin embargo, en esta tesis se ha ilustrado su utilización aplicándolo a nivel de requisitos. La identificación de *crosscutting concerns* en fases tempranas del desarrollo permite incorporar los beneficios obtenidos por la orientación a aspectos desde el principio del proceso de desarrollo. En este sentido, en esta tesis se utiliza un proceso de refactorización orientado a aspectos para casos de uso UML, de modo que se mejora considerablemente la modularidad de los sistemas modelados. Además, esta refactorización permite evolucionar los sistemas de

una manera muy sencilla, sin más que utilizar simples reglas que permiten la composición entre los *concerns* base y transversales.

El patrón de crosscutting puede también utilizarse en la definición de un conjunto de métricas para los diferentes concerns del sistema. Estas métricas proporcionan medidas de la modularidad de un sistema, de modo que nuestro proceso de minería de aspectos se complementa con la utilización de estas métricas. En concreto, las métricas permiten realizar un análisis de la modularidad en fases tempranas proporcionando datos empíricos sobre el grado de *crosscutting* de los diferentes *concerns* de un sistema. De nuevo, las métricas definidas no dependen de ningún lenguaje concreto, de modo que no están ligadas a entidades concretas de las fases del desarrollo. En cualquier caso, en esta tesis se ilustra la utilización de las métricas mediante una instanciación concreta para casos de uso. Además, las métricas son validadas de manera teórica y empírica. Mediante la validación teórica, se demuestra la capacidad de las métricas para medir propiedades relacionadas con el *crosscutting*. Para realizar esta validación, los resultados obtenidos por las métricas son comparados con los obtenidos por otras métricas previamente presentadas en la literatura. Estos resultados muestran que nuestras métricas generalizan a las métricas orientadas a concenrs actuales y demuestran la necesidad de incoporar una métrica específica para *crosscutting* (médida no aportada por otras métricas).

Por otro lado, la validación empírica de las métricas demuestra la utilidad de las mismas, en términos de su capacidad de predecir resultados relacionados con otras características o atributos del software. En esta tesis se muestra la utilidad de las métricas con respecto a dos características del mantenimiento software y definidas por la ISO/IEC 9126. Estos dos atributos son la estabilidad (stability) y la facilidad de ejecutar cambios (changeability). En concreto se presenta el primer análisis que demuestra la correlación existente entre el grado de *crosscutting* y los atributos *stability* y *changeability* en etapas de requisitos. Los resultados obtenidos por este análisis demuestran que el grado de *crosscutting* en etapas tempranas afecta de manera negativa a las características mencionadas, dificultando, por tanto, la capacidad de mantenimiento de los sistemas. Estos resultados ayudan a justificar de manera empírica las ideas que propugna la comunidad de la orientación a aspectos desde hace ya varios años, demostrando que la presencia de *crosscutting concerns* es, normalmente, perjudicial para la calidad del software construido.

Por útlimo, la utilidad del patrón de crosscutting es ilustrada mediante la aplicación del proceso de minería de aspectos para identificar características transversales (*crosscutting features*) en el dominio de las Líneas de Producto Software (SPL). La existencia de *crosscutting* representa un tipo especial de dependencia entre las diferentes características de una línea de producto. Esta dependencia compromete y dificulta la composición y reutilización de dichas características en el domino de SPL. Por tanto, la identificación de estas *crosscutting features* permite realizar una refactorización orientada a aspectos de las mismas, reduciendo así las dependencias existentes entre dichas características. Los beneficios obtenidos por la utilización de técnicas orientas a aspectos para modelar la *variablidad* en SPL han sido presentados ampliamente en la literatura. Sin embargo, la mayoría de estas propuestas carecen de un proceso de identificación de las *crosscutting features,* de modo que estas propuestas aplican técnicas orientadas a aspectos para modelar todas las características variables de la línea de producto o bien, únicamente, aquellas características que han sido previa y ampliamente identificadas como *crosscutting* por la comunidad orientada a aspectos. En este contexto, la utilización del proceso de identificación de crosscutting presentado permite automatizar la identificación de las *crosscutting features*, de modo que éstas pueden ser refactorizadas y aisladas. Además, las métricas de *concerns* presentadas han sido utilizadas también en el dominio SPL, mostrando que el análisis empírico de la modularidad en SPL

puede aportar importantes beneficios, como la capacidad de anticipar el impacto de un cambio en una característica antes de que dicho cambio se produzca.

Acknowledgements / Agradecimientos

La realización de una tesis doctoral es un proceso largo y complejo, un proceso lleno de experiencias, unas buenas y otras no tanto. Sin embargo, en mi opinión, este proceso carecería de sentido si no hubiera personas con las que compartir esas experiencias. Esta sección pretende mostrar mi agradecimiento a todas aquellas personas y entidades que, de una u otra forma, han hecho posible la realización de este trabajo y la escritura de este documento. En especial, me gustaría mostrar mi gratitud con aquellas personas que me han mostrado su apoyo durante todo este tiempo. Han sido tantas personas que posiblemente olvide nombrar a algunas, por lo que pido disculpas a aquellas personas que puedan faltar y hago extensible mi agradecimiento a ellas también:

- En primer lugar, me gustaría agradecer el apoyo facilitado por el **Ministerio de Ciencia y Tecnología** mediante la financiación de los proyectos de investigación TIC2002-04309-C02-01, TIN2005-09405-C02-02 y TIN2008-02985. Sin la existencia de ese primer proyecto de investigación TIC2002, mi carrera investigadora no podría haber comenzado. Me gustaría también agradecer a la **Universidad de Extremadura**, al **Departamento de Ingeniería de Sistemas Informáticos y Telemáticos,** al **Grupo Quercus de Ingeniería de Software** y a la empresa **Homeria Open Solutions** el apoyo mostrado durante todo este tiempo, desde que se hizo efectiva mi vinculación laboral con esta universidad. Por último, quería agardecer a la **Red Europea de Excelencia en AOSD (AOSD-Europe)** y a las **Universidades de Twente y de Lancaster** la ayuda facilitada durante la realización de varias estancias de investigación en estas universidades.
- I would also like to thank to the **members of the committee** the effort and time dedicated in order to make the presentation of this PhD possible. Likewise, I must thank to **Klaas** and **João** for having accepted to be external reviewers of this work. Thank you very much to all of you.
- Por supuesto, este trabajo no podría haber sido realizado sin la inestimable ayuda de mi director de tesis, **Juan Hernández**. Juan me ha enseñado todo lo que sé sobre investigación, entre otras muchas cosas. Sin embargo, en este párrafo no me gustaría resaltar su ayuda como director de tesis, sino su ayuda como amigo. Tengo que agradecerle muchas cosas, como la confianza que depositó en mí desde el principio y la paciencia que siempre ha mostrado conmigo, incluso, a veces, a pesar de mi empeño por complicar las cosas con mis miedos. Pero sobre todo, me gustaría agradecerle el esfuerzo que siempre ha hecho por intentar inculcarme un valor fundamental para desarrollar cualquier actividad en la vida, la confianza en uno mismo. Espero ser capaz algún día de corresponder de algún modo a todo este apoyo.
- Also, this work wouldn't be possible without the help and collaboration of the **Prof. Klaas van den Berg**. I must thank Klaas his hospitality and his invitation to work with him and the **TRESE** group for a large period of time. However, I must also thank him all the lessons that I learned during this period of time. The ideas which this thesis is based on emerged in this period of time so that a big part of the work presented in this document is also due to Klaas. I also really appreciate the help provided by **Bedir Tekinerdogan**, allowing us to use the CFVS as an example in several publications and some parts of this document, but also for being *a really good soccer mate*.
- Igualmente, gostaria de dar graças a **Alessandro Garcia** e **Eduardo Figueiredo**. Além da sua hospitalidade em Lancaster, partilhei com eles longas conversas e discussões que me serviram para compreender a importância da experimentação e a análise empírica em qualquer processo da ciência. Também quero agradecer a ajuda fornecida por **Ana Moreira** e **João Araújo**, com quem igualmente partilhei conversas, reuniões e sessões de trabalho que foram de imensa utilidade para o meu trabalho e que me permitiram afeiçoar-me com o carinho e a hospitalidade portuguesa.

- No podría faltar en estos agradecimientos una mención especial para mis compañeros del grupo **Quercus** de Ingeniería del Software. En especial, me gustaría agradecer el apoyo de **Pedro Clemente**, no sólo por sus revisiones y consejos sobre investigación, sino también por demostrarme el valor de la amistad. También me gustaría agradecer la ayuda proporcionada por **Elena Jurado**, que me enseñó el *arte* de la formalización matemática y de la teoría. Aunque sobre todo debo pedirle disculpas por todas las horas que le "robé" a Juan. No puedo olvidar a **Fernando Sánchez**, por ser alguien a quien admiro y tomo como ejemplo, pero también por empeñarse en que mi sitio siga siendo esta universidad. Asimisimo, debo agradecer a **Roberto Rodríguez** las largas charlas mantenidas sobre ingeniería de software y programación, en las que me ha enseñado tanto. Por último, me gustaría agradecer la amistad y apoyo que durante todo este tiempo me ha ofrecido **Alberto Gómez**, que fue capaz de convertir una pequeña casa de Lancaster en el sitio más *ibérico* de todo el Reino Unido. Podría seguir agradeciendo personalmente el apoyo de cada uno de mis compañeros, pero este documento sería interminable, ya que tengo mucho que agradecer a todos: **Adolfo, Álvaro, Encarna, Juanma, Juan Carlos, Marino**, **Miguel Ángel**, … Muchas gracias a todos.
- Tengo que agradecer también la ayuda y el apoyo de compañeros de la Escuela Politécnica con quienes he compartido tantos ratos y comidas en la propia escuela. En especial, me gustaría destacar la ayuda, el apoyo y, sobre todo, el cariño recibido en los últimos meses por **Carmen Ortiz y Julia González** (que no sólo han sido buenas amigas, sino también buenas asesoras).
- Entre los amigos con los que he compartido estos años en la escuela, no puedo olvidar a **Luz**, que siempre ha hecho todo lo que estaba en sus manos para ayudarme con los aspectos burocráticos, incluso a veces, acarreándole algún que otro quebradero de cabeza. Me gustaría mostrar un recuerdo especial para las tres personas con las que compartí mis primeros pasos como investigador en la Universidad de Extremadura: **Pablo Amaya, Carlos González** y **Javier Pedrero**. Guardo unos estupendos recuerdos del tiempo compartido y espero poder seguir compartiendo tiempo en el futuro, siempre que nos sea posible. Asimismo, guardo también un especial recuerdo y cariño de aquellos amigos con los que he compartido mis años de titulación: **Bachi, Lito, Paco, Jose, …**
- He dejado para el final de estos necesarios, y merecidos, agradecimientos a las personas que han sido más importante para mí durante la realización de este trabajo. En realidad, a estas personas no sólo tengo que agradecerles la ayuda y el apoyo durante la realización de este trabajo, sino durante toda mi vida. Esas personas que esté donde esté (Enschede, Lancaster, Chicago, Shangai,…) siempre han estado presente y a mi lado: **mi familia**. Me gustaría comenzar agradeciendo el apoyo de mis hermanos y sus respectivas familias: **Javi, Miguel, Marifé, Alicia**. Por su apoyo incondicional en todo lo que he hecho y por su comprensión en las cosas que *debería* haber hecho. Pero sobre todo, gracias por haberme ofrecido el mejor de los regalos, un regalo que ha traido una inmensa alegría a mi vida: **Javi, David** y **Helena**.
- Debo seguir esta última parte de agradecimientos con una persona muy especial en mi vida. Se trata, seguramente, de la persona que más ha sufrido las consecuencias de la realización de este trabajo: tiempo robado, distancias, agobios y algún que otro disgusto. A pesar de todo ello, la respuesta ofrecida por ella siempre ha sido la misma: **una inmensa sonrisa**. Por ello, y por ofrecerme su amor, su apoyo, su comprensión, su paciencia y sobre todo, por enseñarme la importancia del optimismo, le estaré siempre agradecido, ¡muchísimas gracias, **Sara**! Espero poder corresponder a este amor y seguir compartiendo esas experiencias contigo en la vida que tenemos por delante.

- Por último, no podían faltar en esta sección las personas responsables de que hoy pueda estar escribiendo este documento. Creo que cualquier argumento que exponga para este agradecimiento se quedaría corto para expresar de manera correcta todo el amor, apoyo y comprensión que ellos me han ofrecido durante toda mi vida. También han estado en los momentos buenos, en los malos (especialmente en los últimos tiempos, que no han sido sencillos), en los regulares, en la distancia, en la cercanía,... Estas personas son mis **padres**: **Paco** y **Marisol**. Tengo tantas cosas que agradecerles que me resulta muy complicado expresarlas en estas líneas. Pero quizás, me gustaría destacar una virtud (que valoro de manera especial) que ellos han intentado inculcarme siempre: ser una persona humilde. No sé si lo habré conseguido, aunque sé que ellos, independientemente de lo que diga, se sentirán orgullosos al leer estas líneas. Aprovecho, entonces, para decirles que **yo también estoy muy orgulloso de ellos.**

Durante el tiempo que he trabajado en la realización de esta tesis, he imaginado en muchas ocasiones cómo sería esta sección de agredecimientos. ¿Qué escribiría? ¿Cómo expresaría estos sentimientos? Tengo que reconocer que ha sido más difícil de lo que imaginaba, especialmente **en los tres últimos puntos**. Aún hoy, creo que no he conseguido demostrar o expresar todo el agradeciminento que les debo a estas personas, espero que sepan perdonarme por ello. Nunca he sabido expresar mis sentimientos muy bien. Por eso, intentaré compensarles y demostrarles este agradecimiento de la mejor forma que sé, estando a su lado día a día...

- Chema -

Table of Contents

List of Figures

List of Tables

1

Introduction

Enhancing business performance in contemporary domains (e.g. e-commerce, financial environments, home automation) requires systems whose size and intricacy challenge most of the current software engineering methods and tools. From early stages in the development of enterprise computing systems to their maintenance and evolution, a wide spectrum of methodologies, models, languages, tools and platforms are adopted. Aspect-Oriented Software Development (AOSD) is one of these methodologies that have emerged as a strong alternative in order to tackle the design and development of complex software systems [76]. Modularity and abstraction are essential techniques for managing such complexity and aspect orientation has appear with the goal of supporting improved modularity of software systems, emphasising on modular structures that cut across traditional abstraction boundaries [109].

One of the key principles in Aspect-Oriented Software Development (AOSD) is Separation of Concerns (SOC) [57]. A concern can be defined very generally as an item in an engineering process about which it cares [76]. Related with this principle is the problem of crosscutting concerns. Crosscutting is usually described in terms of scattering and tangling, e.g. crosscutting is the scattering and tangling of concerns arising due to poor support for their modularization. However, the distinction between these concepts is vague, sometimes leading to ambiguous statements and confusion, as stated in [112]:

.. the term "crosscutting concerns" is often misused in two ways: To talk about a single concern, and to talk about concerns rather than representations of concerns. Consider "synchronization is a crosscutting concern": we don't know that synchronization is crosscutting unless we know what it crosscuts. And there may be representations of the concerns involved that are not crosscutting.

When talking about aspect orientation, we use concepts for which we have some intuition based on our specific experience. We share these concepts with others who may have a similar intuition usually based on other experience. However, the definitions of the concepts are sometimes not consistent with other concepts. Vague definitions imply that it is not always possible to decide when a certain concept applies. When do we have just scattering, when do we have just tangling, when do we have crosscutting and when not? Whenever possible, we should give definitions that are more precise. In some cases, precise definitions are mandatory in order to allow tool support, e.g. for mining of (crosscutting) concerns (c.f. [92]) or also for the definition of software metrics.

In that sense, the project *AOSD Europe Network of Excellence* (an important project with 9 European universities and 2 worldly known companies involved) tried to set the foundations of the AOSD by defining an ontology where these and other concepts were defined [21]. This ontology provided definitions for common concepts underlying the AOSD principles but also for concepts specific of particular approaches (e.g. AspectJ [10] or Composition Filters [26]). I actively worked on this ontology and it was the starting point for the work presented here. The goal of this thesis is to propose a conceptual framework where consistent and precise definitions of scattering, tangling and crosscutting are provided.

As it is mentioned above, sometimes precise definitions are mandatory for several application areas such as the identification of crosscutting concerns at any phase of the software life cycle. In this setting, the conceptual framework presented has been applied into several application areas such as: (i) the aforementioned identification of crosscutting concerns (aspect mining) [23][24], (ii) the definition of a set of concern driven metrics used to assess modularity [51], (iii) the traceability of crosscutting concerns [24], (iv) reducing dependencies in Software Product Lines by identifying the crosscutting features [52].

The rest of this chapter is organized as follows: firstly an introduction to all the areas related to the work presented in this thesis is shown. This introduction includes Aspect Oriented Software Development, aspect mining, software assessment and Software Product

Line Engineering. Secondly, the motivations of the work and the main goals are detailed. Next, the contributions of the dissertation are presented. Finally, in order to drive the reader while reading the thesis, the structure of the whole document is presented.

1.1. BACKGROUND

In this section, a brief introduction to the main research areas related to this work is presented. On one hand, the main concepts underlying AOSD are presented. On the other hand, the main application areas where the conceptual framework have been applied (Chapter 5) are presented. The main application areas are: aspect mining, software assessment and Software Product Line Engineering.

1.1.1. Aspect-Oriented Software Development

The progressive increment of personal computers and networks connecting these computers has motivated the increasing complexity of the software systems built to exploit their functionality. It also implies that software architects and programmers have to deal with more concerns in order to develop these systems. In order to reduce this complexity, Separation of Concerns (SoC) was introduced by Parnas [144] and Dijkstra [57]. A concern is defined as an item in an engineering process about which it cares [76].

Separation of Concerns simplifies system development by allowing the development of specialized expertise and by producing an overall more comprehensible arrangement of elements [76]. In other words, Separation of Concerns allows the separation of a system into functional units that encapsulate the behaviour of the system. These functional units were translated initially into procedures and functions (using structured programming) and secondly into objects and classes (with the introduction of object-oriented paradigm). However, there are certain characteristics that may not be encapsulated into these units and they keep spread all over the system. Typical examples of these characteristics are non-functional requirements such as synchronization, coordination, security or logging. One of the most well-know examples of this situation is the Tomcat project [176], where the logging functionality is spread over almost all the modules of the system. Figure 1 represents the different modules of the Tomcat project. Observe that the red lines represent lines of code related to the logging functionality.

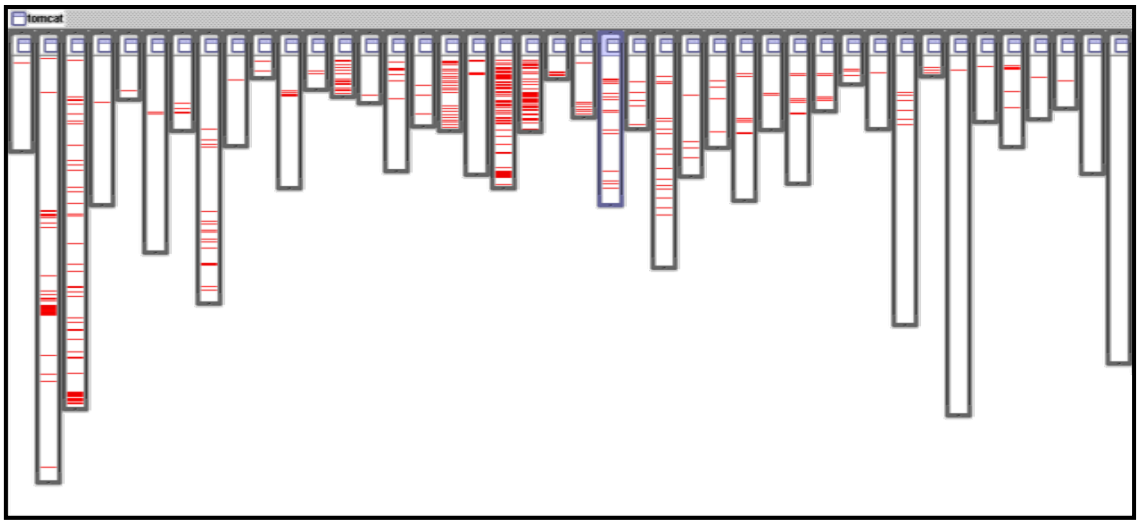


Figure 1. Logging functionality in Jakarta Tomcat project (extracted from [111])

This situation is mainly caused by the phenomenon known as the *Tyranny of the Dominant Decomposition*. The Tyranny of the Dominant Decomposition refers to *restrictions (or tyranny) imposed by the selected decomposition technique (i.e. the dominant decomposition) on software engineer's ability to modularly represent particular concerns* [21]. This situation makes that crosscutting concerns emerge. A crosscutting concern is *a concern, which cannot be modularly represented within the selected decomposition. Consequently, the elements of crosscutting concerns are scattered and tangled within elements of other concerns* [21]. Scattering and tangling are the indications used to detect crosscutting concerns. While scattering refers to the occurrence of elements that belong to one concern in modules encapsulating other concerns, tangling refers to the occurrence of multiple concerns mixed together in one module. Crosscutting concerns compromise important quality attributes of software systems, such as understanding, maintainability, flexibility or reusability. Then, new techniques to cope with these characteristics were needed.

In the last years, several technologies have appeared to solve the problems imposed by the Tyranny of Dominant Decomposition. These technologies are known as Advanced Separation of Concerns techniques. Examples of these approaches are Adaptive Programming [122], Composition Filters [2][26], Multidimensional Separation of Concerns (MDSOC) [171] or Aspect-Oriented Programming (AOP) [109]. Aspect-Oriented Programming has become the most famous one of the approaches and the term *"aspect"*, coined in [109], has become a standard to denote crosscutting concerns. In fact, currently, most of the Advanced Separation of Concerns approaches are considered as specific techniques for Aspect-Oriented Programming.

Aspect-Oriented Programming proposes the utilization of a new entity, called aspect, in programming languages. This entity is responsible for encapsulating the functionality belonging to crosscutting concerns so that it is removed from the modules where it was tangled with functionality of other concerns. This functionality is added to the final system later in a process called *weaving*. M. Aksit, G. Kiczales, K. Lieberherr and H. Osser introduce in [68] the five key concepts that characterize aspect-oriented programming languages:

- **Join point**. A join point is a well defined execution point of a program where the implementation of a requirement or concern is crosscut by the implementation of another concern. A join point may be, thus, any structural artefact of the programming language used, such as method calls, field accesses, object instantiation or exception handlings in object oriented programming. Thus, *a join point model defines the "hooks" where enhancements may be added* [68].
- **Pointcut**. The pointcuts is the mechanism provided by aspect-oriented programming languages to refer to a particular join point in the execution of a program. In other words, *a pointcut is a means of identifying join points* [68]. As an example, using a particular pointcut, we could make reference to the invocation of a method *a* from a different method *b*. They are also a way to group a set of join points, allowing to capture, for instance, all the invocations of methods starting by "*foo*".
- **Advice**. The advice structure is the unit used to encapsulate the behaviour of the crosscutting concern (*a means of specifying behaviour at join points* [68]). This entity allows the definition of the code that will be weaved together with the base code later on. The code defined in the advice will be injected in the base system when the condition defined in an associated pointcut is met.
- **Aspect**. *An aspect is the encapsulated unit combining join point specifications and behaviour enhancements* [68]. Basically, an aspect is a class-like structure with a set of advices and pointcuts defined. Note that these structures were defined in a particular aspect-oriented programming language, AspectJ. However, they are widely accepted

concepts by the community. In other aspect-oriented programming languages, these concepts may be defined using a different terminology.

- **Weaving**. The weaving is the process used to combine the unit used as enhancement with the base code where these units must be added. There are two different kinds of weaving: dynamic (at runtime) and static (at compile time).

Aspect-Oriented Programming has important contributions that have been already mentioned. However, the aspect-oriented community identified some limitations that made needed the extension of this technology. The community realized that the utilization of these techniques at the programming level had important problems, mainly: the utilization of aspects at programming level leaded to inconsistencies with the design, which did not include aspects; these inconsistencies made maintainability and evolution difficult, reducing thus the benefits introduced by AOP; the programmer is responsible for tasks that often belong to the designer; the identification of aspects is difficult when the system is already coded. All these limitations caused that more and more approaches introducing the benefits of aspect-orientation at earlier stages of development appear. Finally, a new whole methodology had emerged, the Aspect-Oriented Software Development (AOSD) [76].

AOSD introduces the benefits of aspect orientation throughout the development phases of a software system, from requirements elicitation to programming and testing. In the last years, there have appeared approaches to deal with crosscutting concerns at the requirements level [14][99][151], to model crosscutting concerns at the architectural design [82][146] [173] or detailed design [14][48], and of course, there have still appeared approaches focused on the programming level [136][170]. Even, some approaches such as [91] have presented techniques to cover all the phases of development, introducing whole aspect-oriented methodologies.

1.1.2. Aspect Mining

One of the main challenges in aspect-orientation relies on aspect identification. AOSD is meaningless unless crosscutting concerns are properly identified in software systems. Logging, tracing, security are known to be crosscutting concerns but, certainly, as Gregor Kickzales states in [112] we don't know that they are crosscutting unless we know what they crosscut. In that sense, aspect mining refers to the process of identifying crosscutting concerns throughout an existing software system which can be then refactored using aspect-oriented techniques [107]. Other definitions existing in the literature state that aspect mining is a specialized reverse engineering process, which aim at investigate legacy systems (source code) in order to discover which parts of the system can be a crosscutting concern [141]. Then, we can distinguish between aspect mining that is the activity of discovering the crosscutting concerns and refactoring to aspects which is the activity of actually transforming these potential aspects into real aspects in software systems [107].

After more than ten years from its emergence, AOP has started to be widely adopted by the industry. This fact implies an important challenge for aspect mining today. Aspect mining has important benefits for the building of new software systems; however, one of the main application areas of this technique is the refactoring of legacy systems, usually developed without considering aspect-orientation. Then, similarly to the process occurred with OOP, industrial software systems should be now migrated into the AOP paradigm [107]. And this is the point where aspect mining comes into play. The reasons for this change are widely justified by the benefits introduced by aspect-orientation: reducing duplicated code, improving cohesion of software modules, enhancing software understanding, easing maintainability and evolution, etc.

However, the manual identification of crosscutting concerns is a tedious task and time-consuming and the effort needed by the developer really compromise the utilization of aspect-

oriented techniques in legacy systems. Note that sometimes the developers have to face up with complex systems and they lack of documentation and knowledge about the system. In this setting, new tools and techniques are needed to reduce this complexity. This is why, in the last years, several approaches have emerged to help in the (semi-)automatic of crosscutting concerns in legacy (or new) systems, e.g. [30] [128] [134][177]. These, and most of the aspect mining techniques, are mainly focused on the analysis of source code. Nevertheless, in the last years some techniques dealing with the early identification of crosscutting concerns have appeared (e.g. [14][158]).

1.1.3. Software Product Lines

As it has been previously mentioned, the increasing complexity of software systems has motivated that new software engineering paradigms and techniques emerge. Software market demands techniques to reduce development cost and effort. Software Product Line Engineering (SPLE) is one of these new technologies introduced to help developers in reducing time-to-market, development efforts and costs. In fact, *the improvement of costs and time-to-market are strongly correlated in software product line engineering: the approach support large-scale reuse during software development* [123]. However, the improvement in reutilization is not for free and it requires some investment at the beginning of the developments. Note that reusability in SPLE is achieved by the utilization of a set of core and variable assets (usually named features). Then, the developer needs to dedicate more effort to obtain reusable assets. However, this investment is justified by the economic advantages obtained for the products built. Note that SPLE deals with the development of products families related to a particular domain. Then, the more reusable the assets are, the less effort needed to develop the family. There are different approaches for dealing with this initial investment: from big-bang approaches to incremental [162]. However, as it may be observed in Figure 2, there is always a break-even point where the investment made becomes profitable.

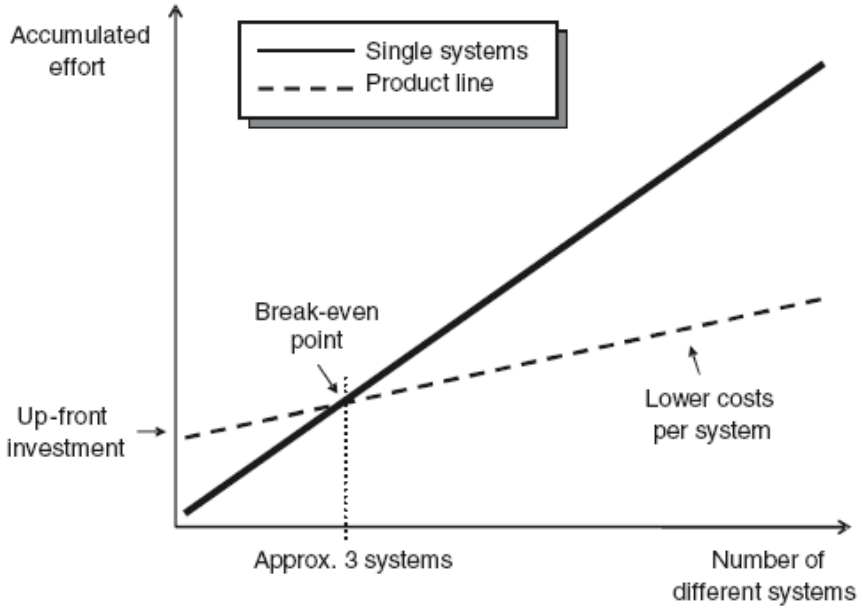


Figure 2. Economic details of SPLE with respect to traditional process (extracted from [131])

The reductions of time-to-market (see Figure 3) and development costs come usually together with an improvement of maintainability. Observe that since the assets that make up the product line are highly reused, the amount of code and documentation to be maintained is also reduced. This reduction also produces a decrease in project risks. All these benefits

obtained by improving internal quality attributes of family products cause also important benefits with respect to external quality attributes. For instance, reliability and safety are highly improved. Then, in addition to the economical benefits obtained by the use of SPLE, important software quality contributions are also achieved (this is even more important for many software development companies).

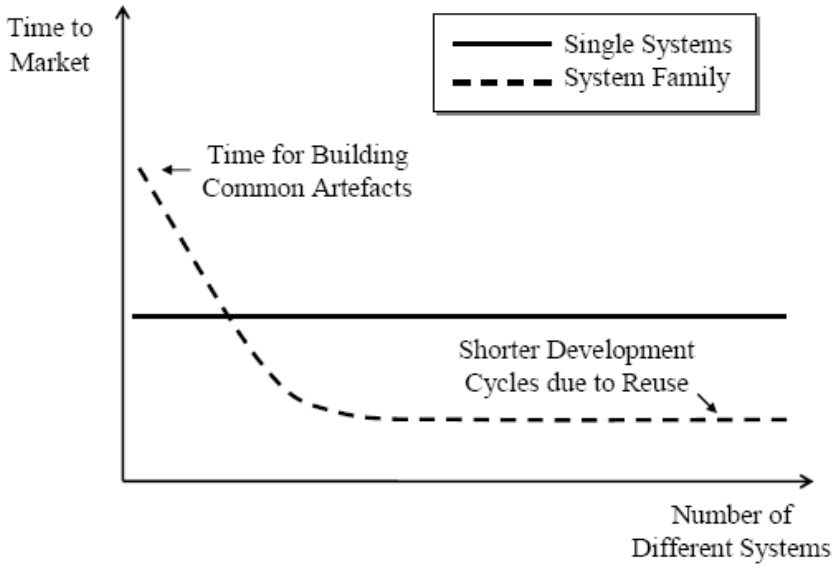


Figure 3. Comparison of time-to-market in SPLE and time to market for single systems [147]

SPLE supposes an important change in the way that software systems are built. This change is not only focused on the development process but it also requires a change in a business field integrating a new strategy. The main difference between traditional software development process and SPLE is that the latter relies on the distinction between two parts of development: *development for reuse* and *development with reuse* [123]. This distinction in the software development process is also aligned with the distinction between domain engineering and application engineering. Domain engineering (development for reuse) deals with the implementation of the product line infrastructure that encompasses all the assets needed to develop any product of the family (or the domain). These assets cover all the stages of the development process, from requirements elicitation to testing. An important contribution of SPLE with respect to other reuse approaches is that the assets contain information about variability. For instance, a particular model used in the development (e.g. a features model) may explicitly contain information of specific requirements only applicable to certain products.

On the other hand, application engineering (development with reuse) is responsible for the development of the individual products. This development is strongly driven by the product line infrastructure built by the domain engineering so that the individual products are developed on top of this infrastructure. In other words, the individual products are built by reusing the features implemented in the product line infrastructure. Differences between the products are obtained by the variability modelled in the domain engineering. In that sense, requirements of the domain are categorised as being part of the product line (commonality or variability) or product-specific. Then, in order to obtain a product, these requirements are just instantiated right way obtaining up to 90% of reutilization in the development of the product. Next, the key concepts in SPLE are defined:

- **Variability management**. The individual products built in product line are developed as variations of a common theme. Thus, this variability represents the differences

between the products. There are specific models and artefacts to explicitly represent this variability and to systematically manage it.

- **Business centric.** This term refers to the connection existing in product lines between the engineering of the product line and the strategy of the business. As it is aforementioned, SPLE implies a change in engineering but also in the way that business are managed by the companies.
- **Architecture centric.** This is maybe the most important concepts since represents the technical development of the core part of the product line. The product line must be developed in a way that similarities between individual products are implemented in this core part. Then, the individual products will take advantages of this development.
- **Two-life-cycle approach.** As it has been already mentioned, the development of the individual products is driven by an infrastructure or software platform. This software platform must be previously implemented and, thus, it has its own life cycle for the development (different from the process used to develop the individual products).

1.1.4. Software Measurement

As it is stated in [103], measurement is crucial to the progress of all sciences: scientific progress is made through observations and generalisations based on data and measurements, the derivation of theories as a result, and in turn the confirmation or refutation of theories via hypothesis testing based on further empirical data. In software engineering, the process of assessment is also performed in the same way. Whenever we want to test an hypothesis, we need to define the set of key concepts involved in this hypothesis. As an example, if we claim that *"the more rigorously the front end of the software development process is executed, the better the quality at the back end"*, then we need to define concepts like software development process, front end and back end [103]. For instance, we should define the steps and activity included in both the front end (e.g. design, implementation, debugging, etc.) and the back end (e.g. formal machine testing and early customer trials).

The process of implementing the system could be defined since it is related to the front end: *whatever is described in the process documentation that needs to be executed, we execute*. However, this definition is not enough for the assessment purpose. The concepts defined should be operationalized in order to be measured. In that sense, some indicators are needed. For instance, an operationalization of the implementation concept could be done by using Lines of Code (LOC) or function points (FP) as indicators to be explored (using code inspection). Scoring the measures obtained by this inspection is usually a good indicator as well (e.g., 5 = very effective, 4 = effective, 3 = somewhat effective, 2 = not effective, 1 = poor inspection). For the back end, we also need to define quality and select the best indicators to measure this quality. Again, we need to operationalize the concepts. In order to assess quality at the back end part, defects per KLOC found could be a good indicator.

Then, assuming the indicators mentioned, some hypotheses could be formulated, e.g. *the higher the percentage of the designs and code that are inspected, the lower the defect rate at the later phase of formal machine testing; the more effective the design reviews and the code inspections as scored by the inspection team, the lower the defect rate at the later phase of formal machine testing* [103]. Once the hypotheses have been formulated, the next step in order to test them is to gather empirical data. For this purpose, the unit of measurement should be defined. In the example shown, the units could be projects or components. Then, obtaining a considerable number of measurements, statistical analysis could be performed. For instance, we could classify the projects into groups according to the obtained data; compare the results based on the variables assessed, in order to extract conclusions; perform correlation analyses; etc. In summary, while concepts in theoretical definitions are defined in

terms of other concepts that are well understood, and sometimes undefined; operational definitions spell out the metrics and procedures needed to obtain the data that measures the indicators selected.

Once we have obtained the data after inspection, the level of measurement should be defined. This level of measurement is the base for being able to compare the results obtained. For instance, the quality of inspection could be measured using a five-points scale but also using percentages. In [103], the authors defined four levels of measurements: nominal scale, ordinal scale, interval scale and ratio scale.

- **Nominal scale**. Using this level of measurement, the elements measured are classified into categories according to an observable attribute. For instance, a classification of software products, according to the software process used to develop them, could categorize the products by waterfall development process, spiral development process or iterative development process. The categories defined by the nominal scale must meet two important requirements; they must be jointly exhaustive and mutually exclusive. While jointly exhaustive means that all categories together must cover all possible values for the attributes (otherwise, an "other" category should be included), mutually exclusive means that a subject may be only included in one category.
- **Ordinal scale.** This measurement level is similar to the nominal scale but it imposes an order between the different categories. This order allows the comparison of the subjects belonging to these categories. A simple example of this category has been already mentioned above, the classification of inspection into the categories: very effective, effective, somewhat effective, not effective, poor inspection. Note that an ordinal scale has the properties of being asymmetric (if A>B, then B>A is false) and transitive (if A>B and B>C then A>C). Ordinal scales are at a higher level than nominal scales. However, they still have limitations. For instance, ordinal scales do not offer information on the differences between elements, i.e. we cannot say how much better an inspection very effective is comparing to an effective. In general, ordinal relations may not be translated into mathematical operations.
- **Interval scale.** This measurement level is able to provide the exact differences between measurements performed for different elements. For instance, we can measures the difference between two products A and B with 5 and 9 defects per KLOC, respectively. In this case, the developer could ensure that defects level for product B is 4 defects per KLOC higher than the corresponding level for A. Note that the mathematical operations of addition and subtraction are possible in this kind of measurement level. Of course, this kind of scale needs of a measurement unit to properly work.
- **Ratio scale.** The ratio scale is an extension to the interval scale that has an important characteristic; in a ratio scale an absolute or non-arbitrary zero point can be located. Besides of addition and subtraction, division and multiplication may be also applied to the ratio scale. For instance, using a ratio scale we could say that a measurement for an element is twice as much as the measurement for other elements. This assumption could be done for defects rate because a zero point in this scale is possible (a product without any single defect). Ratio scales are the highest level of measurements.

In software engineering, measurements are usually performed by using metrics that allow the assessment of internal and external quality attributes (by both internal and external metrics). In [96], the International Organization for Standardization (ISO) established a quality model where these quality attributes were defined (ISO/IEC 9126). On one hand, internal attributes refer to static characteristics of the software intermediate products (from an internal view and using internal metrics). Internal quality attributes may be used to validate the products at various stages of development. Examples of internal attributes defined in [96]

are maintainability and efficiency. Observe that internal attributes are general characteristics that a developer should care about.

On the other hand, external attributes typically measures the behaviour of the code when executed (from an external view and using external metrics). They are important for developer, but also for the final user, and they are usually measured while testing the system in a simulation scenario. Instances of external quality attributes are reliability and usability. *Appropriate internal attributes of the software are a pre-requisite for achieving the required external behaviour, and appropriate external behaviour is a pre-requisite for achieving quality in use* [96].

Software metrics have been widely used in the software engineering area. Examples of these metrics are coupling and cohesion [189], introduced with the success of object oriented programming. However, before being broadly accepted by the community, software metrics must be validated. Traditionally, software metrics have been evaluated using a double validation: theoretical and empirical. Theoretical validation is used to demonstrate that *a measure is really measuring the attribute it is purporting to measure* [96]. In other words, this validation measures the accuracy of the metrics for the purpose defined. Empirical validation is used to check that *the metric is useful in the sense that it is related to other variables in expected ways* [96]. Then, the metric may be used to anticipate important information about the related variables (i.e. other quality attributes).

1.2. MOTIVATION AND GOALS

As it has been stated in the introduction of this chapter, when talking about aspect orientation, we use concepts for which we have some intuition based on our specific experience. We share these concepts with others who may have a similar intuition usually based on other experience. Nevertheless, the definitions of the concepts are sometimes not consistent with other concepts. In that sense, crosscutting is usually described in terms of scattering and tangling. However, the distinction between these concepts is vague and sometimes leading to ambiguous statements and confusion [112]. As it has been previously mentioned, a precise definition of crosscutting is mandatory for certain research areas, such as aspect mining or concern oriented metrics.

Goal 1) Provide a conceptual framework for the formal definition of the concepts related to crosscutting, namely, scattering, tangling and crosscutting. These definitions should be independent of any abstraction level or deployment artefact. A general definition of crosscutting ensures that it may be applied at different application domains. This general definition is the base for other application areas, such as crosscutting identification or the definition of crosscutting metrics.

The identification of crosscutting concerns is known in the community by aspect mining. Aspect mining refers to the process of identifying crosscutting concerns throughout an existing software system which can be then refactored using aspect-oriented techniques [32]. However, most of the research on aspect mining has traditionally focused on the implementation level, when architectural decisions have already been made. It has been widely demonstrated by the AOSD community that aspects also emerge in the early stages of development, leading to what it was called Early Aspects [65]. There have appeared some approaches to apply the same concepts of aspect mining at these early stages (e.g. at the requirements level), incorporating the benefits of aspect orientation from early stages of software development. The sooner the crosscutting concerns are identified, the better

modularity in subsequent stages the software may have. In particular, the utilization of early aspects and their managing at later phases can [16]:

- *increase the consistency of requirements and architecture designs with each other and with the implementation;*
- *provide a rationale and traceability for aspects across life-cycle activities; and*
- *help ensure that crosscutting concerns evident in a system's problem domain or solution space are captured as aspects in the implementation.*

Nevertheless, the Early Aspects community has just focused on dealing with crosscutting properties at early phases. Only few works have addressed the automatic identification of crosscutting concerns at the requirements level. EA-Miner [158] and Theme/DOC [14] are two of these approaches, and they rely on the use of natural language processing techniques for identifying crosscutting concerns at this level. Although these proposals contribute to aspect mining at requirements level, they cannot be applied to other requirements artifacts different than text. However, an important application area of mining early aspect is the refactoring of legacy systems, which are usually described using other requirements artefacts, such as UML use cases or viewpoints.

Goal 2) Definition of an aspect mining process to identify crosscutting concerns at early stages of development. The process should consider different requirements artefacts in order to be applied in legacy systems. However, it should not be tied to a particular level so that the identification of crosscutting concerns in different domains or abstraction levels is possible. The process should be based on the formal definition of crosscutting described by Goal 1.

There are other domains where aspect mining may have important benefits. Software Product Line Engineering is one of these domains. SPL approaches [45][147] aim at improving maintainability and reducing maintenance costs, while improving the design stability and changeability of software systems [96]. In this setting, feature-oriented modelling techniques support the analysis of commonalities and variabilities among products of a family [55] [104]. In addition, feature dependency analysis identifies the dependencies among features of a SPL [120]. The effectiveness of a software product line approach highly depends on how well managed features are throughout both development and maintenance stages [181]. The more independent the assets are, the easier the products are likely to be built [46]. However, features may crosscut each other, making them rigidly dependent and reducing thus the stability and changeability of the SPL assets [46] [87] [121] [181].

Several works have introduced the benefits of using aspect-oriented techniques to deal with crosscutting features, reducing dependencies between them [46] [87] [121] [126] [138] [181]. However, these proposals only focus on the modelling of variable features in SPL using aspect-orientation (and not on the identification of crosscutting features). In addition, some of these approaches (e.g. [87] [121] [138]) focus on programming or design stages, relegating the benefits of aspect-orientation to the latest phases of the development. However, crosscutting features manifest in early development artifacts, such as requirements descriptions [71] and architectural models [82] [160], due to their widely-scoped influence in software decompositions.

Besides, as it is stated in [121], common features could be also modelled using aspect-oriented techniques (e.g. aspectual components) if they crosscut other features. Analogously, variable features need not to be defined always as crosscutting concerns. They may be effectively implemented in modular components if they do not crosscut other features. Accordingly, although the need of identifying crosscutting features in SPL has been demonstrated in previous works, all the aforementioned approaches just analyse the benefits

of incorporating aspect-oriented techniques in SPL and they do not deal with the identification of the crosscutting features (common or variable). Moreover, the incorporation of aspect-oriented techniques at early phases of development improves flexibility and reutilization of the product assets from beginning of the development.

Goal 3) Provide a process to identify crosscutting features at early stages of product line developments. This process should identify crosscutting features independently of their nature, mandatory or variable. The process should be based on the formal definition of crosscutting described by Goal 1.

Aspect oriented community has widely claimed that crosscutting concerns often lead to harmful software instabilities and decrease software quality, e.g. increasing modularity anomalies [71][86] and number of introduced faults [62]. This is the main reason why aspect mining or early aspects approaches emerged with the goal of supporting improved modularity and stability of crosscutting concerns throughout the software lifecycle. However, the use of aspect-oriented decompositions cannot be straightforwardly applied without proper assessment mechanisms. As an example, there is growing empirical evidence that, for instance, software stability is often inversely proportional to the presence of crosscutting concerns [62] [71] [83] [86]. However, to the date, most of the systematic studies of crosscutting concerns (e.g. [62] [71] [83] [86]) concentrate on the analysis of source code, when architectural decisions have already been made. There is a lack of empirical analyses at early stages of development. This became more evident according to recent empirical studies of AOSD based on source-code analysis (e.g. [71] [83] [86]). First, not all types of crosscutting concerns were found to be harmful to some quality attributes, such as design stability. Second, there are certain measurable characteristics of crosscutting concerns that seem to recurrently lead to design instabilities [71] [86].

Even worse, a survey of existing crosscutting metrics has pointed out that they are defined in terms of specific OO and aspect-oriented (AO) programming languages [72]. However, inferring quality attributes after investing in OO or AO implementations can be expensive and impractical. In addition, crosscutting metrics defined for early design representation are very specific to certain models, such as component-and-connector models [160]. These metrics are overly limited as many crosscutting concerns are visible in certain system representations, but not in others [72].

Moreover, the incorporation of these crosscutting metrics to other software domains may provide important benefits for them. As an example, although the need for incorporating aspect-oriented techniques into the SPL domain has been broadly asserted, not much effort has been dedicated to empirically demonstrate the benefits obtained. The utilization of concern driven metrics to assess modularity in these domains would help to find these evidences. Moreover, the empirical validation of the metrics should show their benefits for anticipating information about other related variables (e.g. instability, changeability or feature dependencies, significant attributes in SPL). In that sense, the empirical measurement of crosscutting could complement existing empirical analysis performed in SPL. As an example, in [3] the authors applied the *Maintainability Index* metric to measure the capability of several product lines to evolve to changes. However, again, this measurement process is focused on the programming level.

Goal 4) Provide a measurements framework driven by generic metrics for early quantification of crosscutting. Although the framework must be applied at early stages of development, it should be generic enough to be applied at different abstraction levels. The metrics should be validated both theoretically and empirically. The internal validation must show that the metrics are measuring what it is supposed. The external

validation must check out whether the metrics are useful for inferring information about other internal or external quality attributes (e.g. maintainability characteristics such as instability or changeability [96]).

1.3. CONTRIBUTIONS

In this section, the main contributions of the work presented in this document are described. These contributions are related to the different goals explained in the previous section. Then, the contributions will be explained following the order established by the goals described:

Goal 1. Provide a formal definition of crosscutting.

This thesis proposes a formal definition of crosscutting based on the study of trace dependencies through an extension to traceability matrices. This definition allows developers both to identify crosscutting concerns in early phases and to trace crosscutting concerns from early stages to subsequent phases of the software life cycle [24]. This definition is based on a conceptual framework that it is called the crosscutting pattern. The crosscutting pattern denotes the situation where two different domains, source and target, are related by a traceability relation. Source and target domains could be, for instance, concerns and requirement statements or concerns and use cases. Although there are other definitions of crosscutting in the literature, these definitions are usually very tied to the implementation level, such as [129]. The definition of crosscutting presented is generic so that it is not tied to any abstraction level or deployment artefact. A formal comparison of similarities and differences between other definitions and ours is also shown, demonstrating that, in most of cases our definition generalises the others.

Goal 2. Definition of an aspect mining process to identify crosscutting concerns at early stages of development.

The conceptual framework defined has many application areas. One of the most important areas is the identification of crosscutting concerns. Although the framework may be applied at any development stage, unlike other previous works, in this dissertation it has been mainly applied to the requirements level in order to incorporate the benefits of aspect-orientation as soon as possible. In that sense, an aspect mining process based on syntactical and dependency analyses at the requirements level has been proposed [54].

The aspect mining process includes early aspect refactoring given for UML use cases diagrams. This refactoring allows early aspects to be properly modularized from the requirements level, improving modularity of the system since the crosscutting concerns are isolated. Moreover, the system may be easily evolved just using simple composing rules which allow the weaving of base and crosscutting concerns.

The process is validated by a comparison with other early aspect mining proposals. The comparative study is particularly useful as a benchmark for other aspect mining approaches.

Goal 3. Provide a process to identify crosscutting features at early stages of product line developments.

The aspect mining process to identify crosscutting concerns have been also applied to the SPL domain [52]. This process allows the identification of early crosscutting features in SPL requirements artefacts. Crosscutting features are then refactored using aspect-oriented techniques, thus complementing other works in the literature such as [1]. Then, dependencies between features of the product family are reduced. This is particular useful in the SPL context, where analysis of crosscutting features should be undertaken in early SPL representations. The use of the aspect mining process in the SPL domain only needs a simple

adaptation consisting in using specific models for SPLE (e.g. features models used for Feature Oriented Domain Analysis [104]).

Goal 4. Provide an empirical assessment of modularity at early stages of development.

A different application area of the crosscutting pattern is the assessment of modularity. In particular, a language-agnostic metrics suite for early quantification of crosscutting is presented [51]. This is particular useful with the transition to Model-Driven Software Engineering [132] gaining momentum, where analysis of crosscutting concerns should also be undertaken in early system representations. Again, since the definition of the metrics is based on the conceptual framework, the metrics are not tied to any deployment artefact and they are independent of specific requirements and architectural models. Nevertheless, canonical instantiations of the crosscutting metrics are given for usecases.

Moreover, the metrics have been theoretical and empirically validated. For the former, a comparison with other similar metrics introduced by other authors has been performed. This comparison has been done by applying all the metrics to a real application. The latter has been done by showing a first and original exploratory study investigating how two ISO/IEC 9126 maintainability attributes, namely stability and changeability [96], are correlated to early crosscutting measures. Moreover, these results are also used to compare crosscutting properties with concern dependencies, showing their correlation. The results obtained help developers to anticipate important decisions regarding to maintainability at early stages of development.

Finally, since the metrics complement the aspect mining process, they have been also applied in the SPL domain, demonstrating that crosscutting in SPL is harmful for features stability and software changeability.

1.4. ROADMAP FOR THIS DOCUMENT

In this section the structure of this thesis is presented. The layout of this structure is shown in Figure 4.

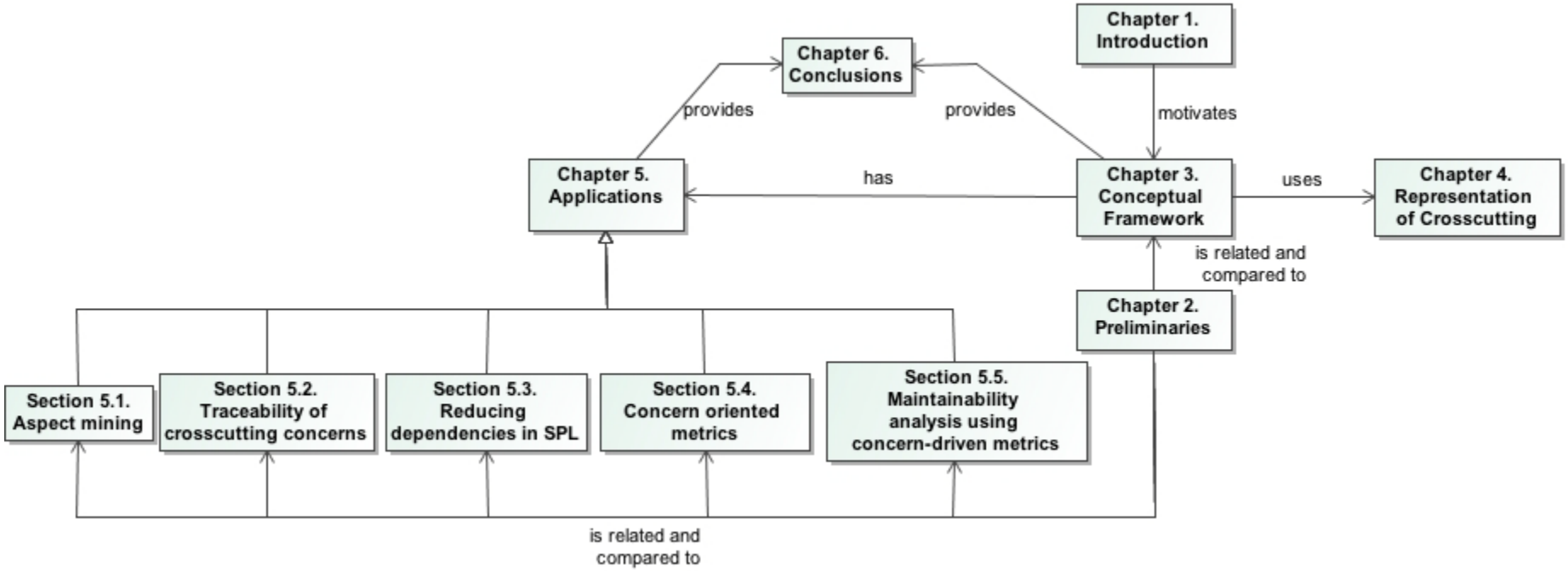


Figure 4. Layout of this thesis.

Next the content of each chapter is described:

- **Chapter 1** contains an introduction of the thesis (current chapter) where a brief background on the main research areas related to this thesis is shown. This chapter also provides sections motivating the work presented, showing the goals of the thesis and its main contributions.
- **Chapter 2** presents a survey of the state of the art in the different topics related to this thesis. This chapter firstly presents several definitions of crosscutting existing in the literature. Secondly it presents a survey on different aspect mining approaches. Then, the chapter shows current approaches that apply aspect oriented techniques in the Software Product Line domain. Finally, concern oriented metrics similar to the presented in this thesis are described.
- **Chapter 3** introduces the conceptual framework presented. This conceptual framework is based on what we called the crosscutting pattern. Based on the crosscutting pattern, the terms of scattering, tangling and crosscutting are formally defined. Finally, a formal comparison between our definition of crosscutting and others existing in the literature is shown.
- **Chapter 4** presents different representation of crosscutting that may be used to complement the conceptual framework presented in Chapter 3. Examples of these representations are dependency graphs or traceability matrices. In particular, traceability matrices are used to allow the identification of crosscutting by using simple matrix operations. This chapter also shows how to use transitivity of traceability matrices to allow the traceability of crosscutting concerns through several subsequent development phases.
- **Chapter 5** shows different application areas where the conceptual framework has been presented. One of the most important application areas is the identification of crosscutting concerns at early stages of development. Traceability of crosscutting concerns is also shown. A set of generic concern-oriented metrics based on the conceptual framework is also presented. These metrics improve the modularity analysis performed by the aspect mining process. Next, the identification of crosscutting features in early SPL stages is described. Finally, theoretical and empirical validations of the metrics demonstrate the accuracy of the metrics and its utility with respect to other quality attributes, respectively.
- **Chapter 6** finalizes this thesis describing the main conclusions extracted not only from it contributions but also from the research activities carried out in this period of time, during the development time of this thesis.
- **Chapter 7** shows the main publications obtained where the ideas of this thesis have been previously presented.

2

Preliminaries

This chapter shows background in the different topics that this thesis deals with. In particular, the state of the art is described by showing the different approaches existing in the literature for each topic. A more detailed comparison between our approach and some of those described in this chapter is shown in later chapters where our approach is presented.

Firstly, this chapter shows a survey of different definitions of crosscutting that we may find in the aspect-oriented literature. A formal comparison between these definitions and ours is provided in Chapter 3, where we show how these definitions could be considered as special cases of the definition based on the crosscutting pattern (shown in Section 3.1).

Secondly, we describe the state of the art in the aspect mining area. In particular, the most used techniques are described such as the fan-in or the identifiers based analyses. Moreover, different approaches for mining aspects at early stages of development are shown (e.g. at requirements or architectural descriptions). A more detailed comparison between the results obtained by these approaches and ours is shown in Chapter 5 (Section 5.1.7).

Next, the chapter shows the application of aspect-oriented techniques to the Software Product Lines (SPL) domain. There are several works which have introduced the need for using aspect-oriented techniques in SPL in order to improve flexibility and reusability of products built. These works are described here.

Finally, several works which introduce aspect-oriented metrics are presented. These metrics are used to validate our concern-oriented metrics in Chapter 5. Thus, the results obtained by these metrics in a real case study are compared to those obtained by our metrics (Section 5.4.5).

In case the reader finds the chapter too large, he/she may focus on the section more interesting according to his/her topics of interest. In that sense, in Figure 5 a layout of the different topics and/or approaches described in this chapter is shown.

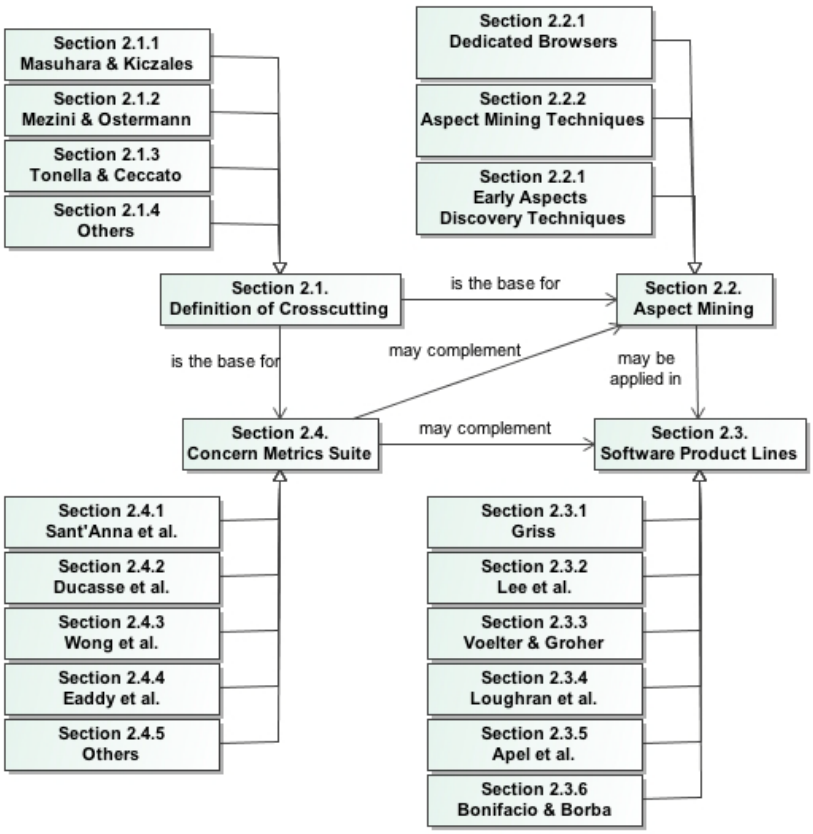


Figure 5. Layout of the approaches described in throughout Preliminaries chapter

2.1. DEFINITIONS OF CROSSCUTTING

In Chapter 3, a formal definition of the terms of scattering, tangling and crosscutting is provided. A comparison with other definitions existing in the literature is also shown. In order to carry out this comparison, the definitions existing are presented in this section. In particular, this section describes the definitions provided by Masuhara and Kiczales in [129], the definition introduced by Mezini and Ostermann in [135], the one presented in [177] by Tonella and Ceccato and finally some other definitions just mentioned in several publications.

2.1.1. Definition by Masuhara and Kiczales

In [129] the authors provide an interesting model for defining how four different AOP mechanisms support modular implementation of crosscutting concerns. These mechanisms are based on a common framework which allows the authors to define what makes a technique aspect-oriented. The framework models the AO mechanisms as a weaver that combines two different programs and produces as result either a program or a computation. This weaver is based on an 11-tuple:

$$\{X, X_{JP}, A, A_{ID}, A_{EFF}, A_{MOD}, B, B_{ID}, B_{EFF}, B_{MOD}, META\}$$

where the different elements of the tuple are defined as follows:

- A and B are the languages in which two different programs p_A and p_B are written.
- X is the result domain of the weaving of p_A and p_B. This result is usually a computation although it could be a third language to model systems (like Hyper/J [95]).
- X_{JP} is the set of join points in X.
- A_{ID} and B_{ID} are the means, in the languages A and B, of identifying elements of X_{JP} (the join points in X).
- A_{EFF} and B_{EFF} are the means, in the languages A and B, of effecting semantics at identified join points.
- A_{MOD} and B_{MOD} are the units of modularity in the languages A and B.
- META is an optional language to parameterize the weaving process (left it out of the model in case it is not needed).

The weaving process is then defined as the signature: $A \times B \times META \rightarrow X$

Based on the 11-tuple and the weaving process, the authors also provide an interesting definition of crosscutting. The notion of crosscutting provided in [129] is focused on programming level, and it is based on the two source languages A and B (one of them being aspect-oriented) and the target one called X (resulting in the weaving process of A and B). The authors take as input two different programs p_A and p_B written in A and B respectively. Then they define the term projection as follows: *"for a module m_A (from p_A), we say that the projections of m_A into X is the set of join points identified by the A_{ID} elements within m_A"*. A_{ID} refers to the means in A for identifying the join points in X (in object-oriented languages methods and field signatures). For more details see [129]. The authors use the canonical figures-display example [110] in the poincut-and-advice mechanisms to show these concepts in AspectJ (see Figure 6).

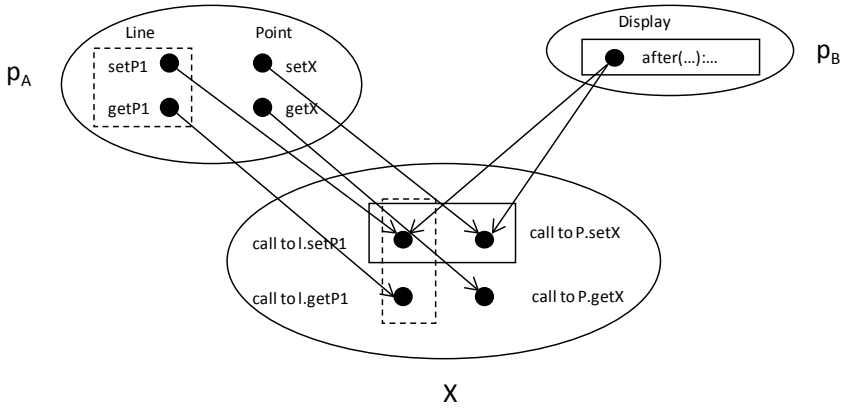


Figure 6. Point class and display updating advice crosscut each other in resulting domain X [129]

Then crosscutting is defined as follows: *For a pair of modules m_A and m_B we say that m_A crosscuts m_B with respect to X (the result domain) if and only if their projections onto X intersect, and neither of the projections is a subset of the other*. According to this definition crosscutting is a symmetric property.

As we will demonstrate in Chapter 3, this definition could be considered as a special case of the definition presented in that chapter. In fact, unlike the definition presented in Chapter 3, this definition is not generic ant it is tied to the programming level.

2.1.2. Definition by Mezini and Ostermann

In [135] the authors describe an approach which allows multiple decompositions simultaneously. The framework presented in [135] is called CAESAR. By means of this framework, the authors avoid problems of arbitrariness of the decomposition hierarchy, which the authors identify as the main problem for code tangling and scattering.

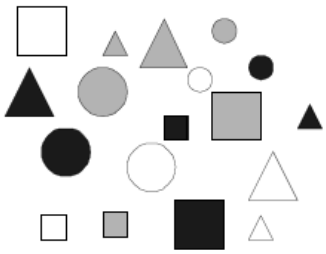


Figure 7. Example of Abstract Concern Space extracted from [135]

Moreover, in [135] a definition of crosscutting is provided. This definition is based on the terms of model, projections and concern space. A concern space represents the artefacts which are composed to obtain the final system. In [135], the authors use an abstract example with different figures to represent the concern space. They called this set of figures the abstract concern space (see Figure 7). Then, a model is defined as the result of a decomposition of the system according to particular criteria. Following the abstract example, a model is defined as the result of classifying the set of figures according to three different criteria: color, shape and size. As an example, in Figure 8 we show the model resulting from decomposing the abstract space according to firstly color concern, secondly to shape and finally to size.

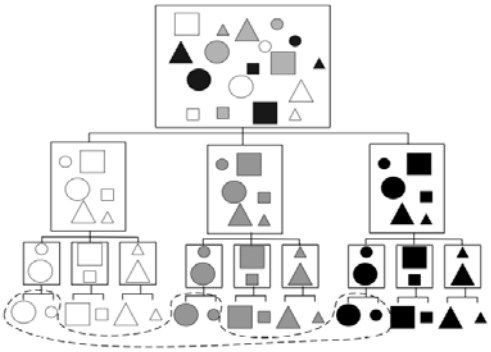


Figure 8. Concern space classified by color, shape and size

Then, the authors define projection: a projection of a model M is a partition of the concern space into subsets $o_1,\ldots, o_n$ such that each subset o_i corresponds to a leaf in the model. An example is shown in Figure 9 with projection of the color model.

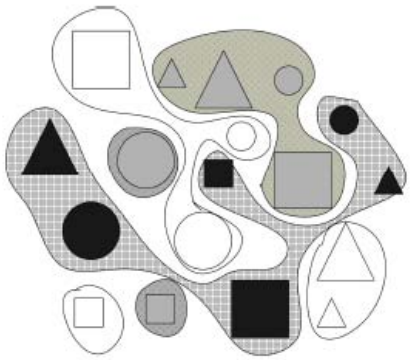


Figure 9. Projection of color model

Using the aforementioned terms, the authors claim that models resulting from simultaneous decomposition of the system according to different criteria are usually crosscutting with respect to the execution of the system [135]. They define crosscutting as a relation between two models with respect to the abstract concern space as follows: two models, M and M', are said to be crosscutting, if there exist at least two sets o and o' from their respective projections, such that $o \cap o' \neq \emptyset$, and neither $o \subseteq o'$, nor $o' \subseteq o$. In Figure 10 we can see an example of several modules crosscutting each other. In this figure, we can see how the black concern of the color model crosscuts the big concern of the size model. Note that the projections of both models have sets that intersect and they are not subsets of each other.

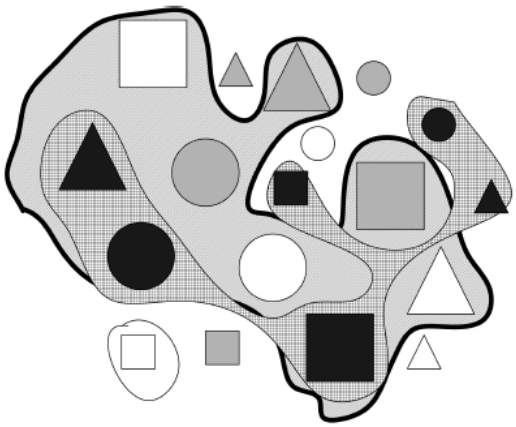


Figure 10. Color and size concerns crosscutting each other

As we can see, this definition is very similar to Mashuara and Kiczales's definition presented in [129]. Then, again, we can consider that this definition is a special case of the definition presented in Chapter 3 as we will show in that chapter.

2.1.3. Definition by Tonella and Ceccato

In [177] Tonella and Ceccato use a mathematical tool to represent the relation between concerns and source code units; it is the formal concept analysis that will be introduced in following section.

2.1.3.1. Formal concept analysis.

Formal concept analysis (FCA) [81] is a branch of lattice theory that can be used to identify meaningful groupings of elements that have common properties [177]. FCA takes as input a so-called context, which consists of a set of elements E, a set of properties P on those elements, and a Boolean incidence relation between E and P.

An example of such a context is given in Table 1, which relates different properties defined on integer numbers. Consider *E* = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10}, and *P* = {composite, even, odd, prime, square}.

	1	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**
Composite				X		X		X	X	X
Even		X		X		X		X		X
Odd	X		X		X		X		X	
Prime		X	X		X		X			
Square	X			X					X	

Table 1. Matrix that represents the relation between E and P.

Starting from such a context, FCA determines maximal groups of elements and properties, called *concepts*, such that each element of the group shares the properties, every property of the group holds for all of its elements, no other element outside the group has those same properties, nor does any property outside the group hold for all elements in the group. Graphically, a concept corresponds to a maximal 'rectangle' containing only marks in the table, considering any permutation of the table's rows and columns.

A concept lattice can be built from a context. In the concept lattice, every node is a concept**,** that is a pair containing both a property cluster and its corresponding object cluster.

The concept lattice shown in Figure 11 has been built from the context described in Table 1. Obviously, it is only a different way to represent the relation between E and P.

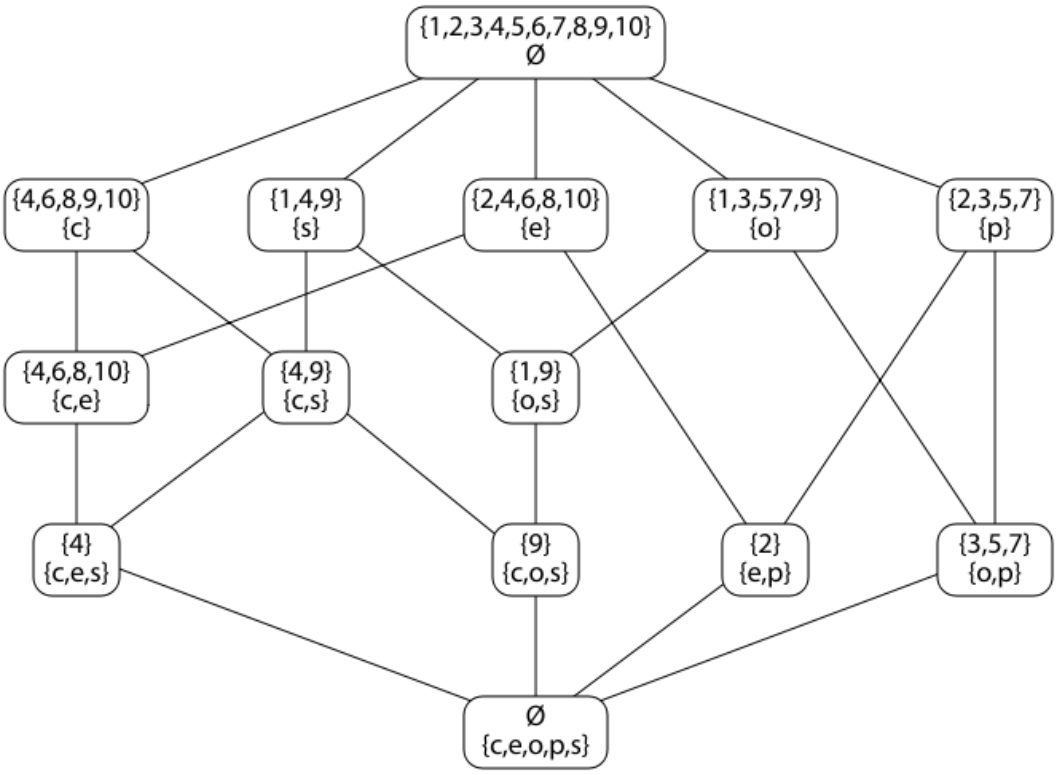


Figure 11. Concept lattice for the context describe in table 3

Formally, a *concept* is defined to be a pair (E_i, P_i) such that

1. $E_i \in \mathscr{P}(E)$
2. $P_i \in \mathscr{P}(P)$
3. every element in E_i has every attribute in P_i
4. for every element in E that is not in E_i, there is a property in P_i that the element does not have
5. for every property in P that is not in P_i, there is an element in E_i that does not have that property

E_i is called the *extent* of the concept, and P_i is the *intent*.

Nodes in the concept lattice can be partially ordered by inclusion: if (E_i, P_i) and (E_j, P_j) are concepts, we define a partial order by saying that (E_i, P_i) ≤ (E_j, P_j) whenever $E_i \subseteq E_j$. Equivalently, (E_i, P_i) ≤ (E_j, P_j) whenever $P_j \subseteq P_i$. Every pair of concepts in this partial order has a unique greatest lower bound and a unique least upper bound. The greatest lower bound of (E_i, P_i) and (E_j, P_j) is the concept with elements $E_i \cap E_j$; it has as its properties $P_i \cup P_j$. The least upper bound of (E_i, P_i) and (E_j, P_j) is the concept with properties $P_i \cap P_j$; it has as its elements the set $E_i \cup E_j$.

2.1.3.2. Labels in the concept lattice

It is very interesting the idea of selecting elements and properties that label a given concept, they are those that characterize the concept most specifically.

More precisely, a concept **c** is labelled with an element **e** only if **c** is the most specific (i.e., lowest) concept having **e** in the extent. A concept **c** is labelled with a property **p** only if **c** is the most general (i.e., highest) concept having **p** in its intent.

We formally introduce this idea with the following functions:

$\alpha(c) = \{p \in P \mid c \text{ is the largest lower bound of the set of concepts that have } \mathbf{p} \text{ in its intent}\}$

$\beta(c) = \{e \in E \mid c \text{ is the least upper bound of the set of concepts that have } \mathbf{e} \text{ in its extent}\}$

Considering the previous example, the smallest concept including the number 3 is the one with objects {3, 5, 7}, and attributes {odd, prime}, then 3 is a label for this concept. The largest concept involving the attribute of being square is the one with objects {1,4,9} and attributes {square}, then square is a label for this concept. Thus:

$\alpha(\{3,5,7\}\{odd, prime\}) = \varnothing$ $\beta(\{3,5,7\}\{odd, prime\}) = \{3,5,7\}$

$\alpha(\{1, 4, 9\}\{square\}) = \{square\}$ $\beta(\{1,4,9\}\{square\}) = \varnothing$

2.1.3.3. Applying FCA to identify crosscutting situations

In [177] formal concept analysis has been used to identify the computational units (i.e., procedures) that specifically implement a feature (i.e., requirement) of interest.

Execution traces obtained by running the program under given scenarios provided the input data. The executed methods are the elements of the concept analysis context, while execution traces associated with the use-cases are the properties[1]. In the resulting concept lattice, the use-case specific concepts are those labelled by at least one trace for some use-case (i.e. α (c) contains at least one element), while the concepts with zero or more properties as labels (those with an empty α (c)) are regarded as generic concepts. Thus, use-case specific concepts are a subset of the generic ones.

Both use-case specific concepts and generic concepts carry information potentially useful for aspect mining, since they group specific methods that are always executed under the same scenarios (Section 2.2.2.1 shows how to use this information in that sense).

2.1.3.4. Definition of crosscutting

A concern seed is a single source-code entity, such as a method, or a collection of such entities, that strongly connotes a crosscutting concern. A candidate seed is a potential concern seed.

Formally, a concept **c** is considered a candidate seed iff [36]:

- Scattering: $\exists$ m, m' $\in \beta(c) \mid pref(m) \neq pref(m')$
- Tangling: $\exists$ m $\in \beta(c)$, $\exists$ m' $\in \beta(c')$ $\mid c \neq c' \wedge$ pref(m)= pref(m')

where pref(m) is the fully scoped name of the class containing the method m.

The first condition (scattering) requires that more than one class contributes to the functionality associated with the given concept. The second condition (tangling) requires that the same class addresses more than one concern. As we shown in Chapter 3, this definition is equivalent to the presented in that chapter. However, this definition is provided in terms of programming artefacts whilst the definition presented in Chapter 3 is not tied to any development artefact so that it may be used at any abstraction level.

[1] In [36], the authors claim that the executed methods are the properties of the concept analysis context. However, in the formal definition they define p and p' $\in \beta(c)$ (set of elements). We think this inconsistence is a typo. In order to clarify this issue, we considered here executed methods as the elements of the concept analysis context.

2.1.4. Other definitions

There are other works where the authors mention or introduce other definitions of crosscutting or scattering. Most of these definitions are not formally defined. However, some times the use of a formal definition is mandatory for having tool support (e.g. in the aspect mining area). Here, we detail some of these definitions mentioned in the literature.

In [78], Fox provides a definition of aspect based on a syntactical and semantic corpus. Based on this definition, he classifies the aspects into two different categories: Systemic and Fine Granular (FG). Systemic refers to the aspects that influence the decisions shaping or modifying the architectural behaviour (e.g. multiple views or quality of service). Fine Granular denotes to aspects that may be implemented at the level of the methods in classes (e.g. security, synchronization, logging). While Systemic aspects must be solved by using composition mechanisms such as Hyper/J [95] or Composition Filters [1], Fine Granular aspects may be solved using Design Patterns [80] or AspectJ [10]. In this work, the author also provides a formal definition of crosscutting. This definition establishes that crosscutting occurs when a requirement is partially implemented in more than one class. Then, an aspect is defined as follows:

Definition (Aspect): Given a problem space $\mathcal{P}$

a solution space $SOLSP = \bigcup_{\rho} SolSp(\rho)$

Let $A =_{def} \{r \in \mathbb{RE} \mid \exists\ o_1, o_2 \in \mathbb{CL} : Implements_{\varphi}(r, o_1) \wedge Implements_{\varphi}(r, o_2) \wedge o1 \neq o2$

where $Implements_{\varphi}(r, \theta) \iff \forall_{\varphi'} \subseteq \varphi\ .(\varphi' \vDash r \implies \theta \subseteq \varphi')$

$\varphi \vDash r$ φ *satisfies* r

$\mathbb{RE} =$ *𝒫(SOLSP)*

$r \subseteq r \bigcup_{\rho} SolSp(\rho)$

$\mathbb{CL}_{\varphi}$ is the set of classes in the solution space.

As can be seen in this definition, an aspect is defined in terms of scattering so that scattering is a needed and sufficient condition to have crosscutting. The definition presented in Chapter 3 considers crosscutting as a special combination of scattering and tangling and it allows the distinction of situations with just scattering, just tangling or crosscutting.

There are other informal definitions mentioned in several publications. As an example, in [68] Karl Lieberherr gives a definition of crosscutting: "*Two concerns crosscut if the methods related to those concerns intersect.(…) We say a method is related to a concern if the method contributes to the description, design, or implementation of the concern*". This definition can be considered as a particular case of Mashuara and Kiczales's definition presented in [129].

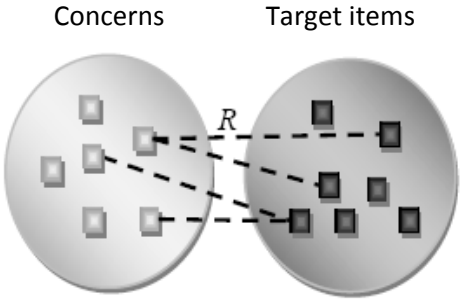


Figure 12. Concern model defined and extracted from [63]

In [63], Marc Eaddy focuses on the automatic identification of concern locations. This work is based on the conceptual framework that we present in Chapter 3. In particular, he defines a concern model which identifies relations between two domains (see Figure 12): a concern domain and a target domain (being a special case of the crosscutting pattern presented in Chapter 3). Eaddy uses the definitions of scattering and tangling presented in Chapter 3 to identify crosscutting situations. He considers scattering as a needed and sufficient condition to have crosscutting.

In [16], the authors describe the term of early aspect being a crosscutting concern at early development phases (i.e. requirements and architectural levels). In particular, they define early aspects as *"concerns that crosscut an artefact's dominant decomposition*[2]*, or base modules derived from the dominant separation of concerns"*.

In [21], an ontology is proposed where some common terms of the aspect-oriented community are defined. In this ontology we can find the definitions of the terms crosscutting or crosscutting concerns. On one hand, crosscutting is defined as *"crosscutting is the scattering and/or tangling of concerns arising from the inability of the selected decomposition to modularise them effectively".* On the other hand, crosscutting concern is defined as *"a concern which cannot be modularly represented within the selected decomposition. Consequently, the elements of crosscutting concerns are scattered and tangled within elements of other concerns"*.

2.2. ASPECT MINING

One of the main challenges in aspect-orientation relies on aspect identification. AOSD is meaningless unless crosscutting concerns are properly identified in software systems. Logging, tracing, security are known to be crosscutting concerns but, certainly, as Gregor Kickzales states in [112] we don't know that they are crosscutting unless we know what they crosscut. In that sense, aspect mining refers to the process of identifying crosscutting concerns throughout an existing software system which can be then refactored using aspect-oriented techniques [107]. Other definitions existing in the literature state that aspect mining is a specialized reverse engineering process, which aim at investigate legacy systems (source code) in order to discover which parts of the system can be a crosscutting concern [141]. Then, we can distinguish between aspect mining that is the activity of discovering the crosscutting concerns and refactoring to aspects which is the activity of actually transforming these potential aspects into real aspects in software system [107].

In the last years, most of the systematic studies of aspect mining have concentrated on source code (e.g. [32] [36] [164] [177]). However, crosscutting concerns manifest in early development artifacts, such as requirements descriptions [24] and architectural models [82][160], due to their widely-scoped influence in software decompositions. As at the implementation level, aspect mining techniques have been also introduced at early phases to be able to identify and modularize crosscutting concerns earlier, incorporating the benefits of aspect orientation from early stages of software development.

According to [107] the approaches to discover aspects may be divided into three main categories: dedicated browsers, aspect mining and early aspects approaches.

---

[2] Remember that in [21], the term *Tyranny of Dominant Decomposition* is defined as: *"The Tyranny of the Dominant Decomposition refers to restrictions (or tyranny) imposed by the selected decomposition technique (i.e. the dominant decomposition) on software engineer's ability to modularly represent particular concerns."*

Dedicated browsers. The techniques classified as dedicated browsers are advanced special purpose code browsers that aid the developer in navigating through the code in order to find crosscutting concerns. As it is stated in [107], although the primary goal of these techniques is not to mine for aspects, these dedicated browsers may be used to identify aspects besides of documenting and localising the crosscutting concerns in order to maintain and evolve the system. Most of the approaches of this category require that the user establishes a "seed" of a concern in order to have a starting point to explore throughout the code. Examples of these approaches are Concern Graphs [152], Aspect Browser [187], Aspect Mining Tool [92] or Prism [192].

Aspect mining techniques. These approaches are based on the automation of the aspect discovery process and they usually propose their users one or more aspect candidates. These techniques usually rely on the analysis of the source code or some data obtained by the manipulation of the code. There are different techniques used to identify the crosscutting concerns. Examples of these techniques are Formal Concept Analysis (e.g. *Dynamo* [177] or *DelfSTof* [134]), execution traces (*DynAMiT* [30]), clustering of methods (AMAV [163]), natural language processing of source code (e.g. the approach by Shepherd et al. [164]), clone detection techniques (in [32] some of these techniques are deeply analysed), fan-in analysis (e.g the approach by Marin et al. [128]), and so on [107].

Early aspects discovery techniques. Early aspects approaches deal with the identification of crosscutting concerns at early stages of development. As it has been already mentioned, an early aspect a *concern that crosscuts an artefact's dominant decomposition, or base modules derived from the dominant separation of concerns* [16]. Examples of early aspects approaches are those focused on the identification of crosscutting concerns at the requirements level or domain analysis (e.g. Theme/Doc [14], EA-Miner [158]) but also at architectural design (e.g. ASAAM [173], CAM/DAOP [146], architectural reasoning [17]). The identification of crosscutting concerns at early stages of development introduces the benefits of the aspect-orientation at these early stages of life cycle so that developer may achieve a better modularization of the system from the early beginning of development.

Most of these approaches will be explained in next subsections. The approaches explained are divided according to the classification explained above.

2.2.1. Dedicated browsers.

In this section, the techniques classified as dedicated browsers are explained.

2.2.1.1. Concern graphs

In [152], Robillard and Murphy introduced an approach to localize the source code related to a particular concern. This approach is based on a Concern Graph representation which abstracts the implementation details of a concern and makes explicit the relationships between the different parts of the concern. As the authors claim in [152], Concerns Graphs are more effective than lines of source code for the purpose of documenting and analysing concerns.

A Concern Graph represents the relationships between the relevant program elements of object oriented programming (namely classes, methods and fields). Then, based on the relations between these elements, the authors establish a program model which abstracts and augments source code. The Structural Program Model built by the authors is based on a graph expressed as $P = (V_p, E_p)$ where $V_p$is the set of vertices and $E_p$is the set of labelled directed edges $e = (l, v_1, v_2)$.

Taking into account the graph defined, a vertex in *P* may be one of the next three elements:

- **Class vertex (C)**, which represents a class without its members.
- **Method vertex (M)**, representing a method of class.
- **Field vertex (F)**, which represents a field member of a class.

On the other hand, the edges in the graph are labelled with the semantic relationship that they represent:

- **(calls, m_1, m_2)**, method m_1 has a call that bind to method m_2.
- **(reads, m, f)**, method m has an instruction that reads the field f.
- **(writes, m, f)**, method m has an instruction that writes o defines a value to the field f.
- **(checks, m, c)**, method m checks the class of an object or casts an object to c.
- **(creates, m, c)**, method m creates an object or c type.
- **(declares, c, {f|m})**, class c declares a field f or a method m.
- **(superclass, c_1, c_2),** class c_2 is superclass of class c_1.

Figure 13 shows the program model for a simple example which contains a multiplier class (extracted from [152]). In this figure we can see how the multiplier class declares a field member called product and two methods called sum and product.

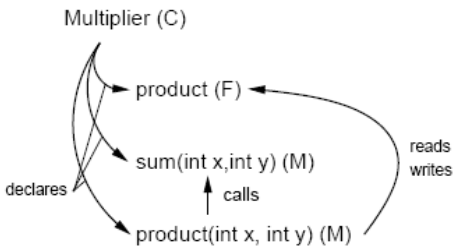


Figure 13. Program model for a multiplier class.

Finally, based on the program model defined above, the authors define the Concern Graph for a particular concern *i* and the program $P = (V_p, E_p)$ as $C_{p,i} = (V_{p,i}, V^*_{p,i}, E_{p,i})$, being the compacted subset of P documenting the implementation of the concern, where:

- $\boldsymbol{V_{p,i}}$ represents the set of *part-of* vertices of V_p. The elements classified as *part-of* partly implements the concern *i*.
- $\boldsymbol{V^*_{p,i}}$ represents the set of all-of vertices of V_p. These elements are entirely devoted to the implementation of the concern *i*.

Using the same example illustrated in Figure 13 the Concern Graph for summing concern using the *part-of* and *all-of* concepts may be observed in Figure 14 (also extracted from [152]).

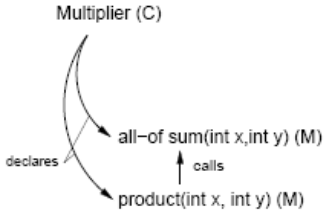


Figure 14. Concern Graph for summing concern.

In [152], the authors also introduced FEAT (Feature Exploration and Analysis Tool). This tool allows the localization and analysis of the source code contributing to a concern in Java programs. The tool provides three main functions:

- Display Concern Graphs in a suitable way for software developers
- Access to vertices and edges of the structural program model related to the vertices of the Concern Graph. This action allows the iterative construction and modification of the Concern Graph.
- Mapping from the vertices of the Concern Graph to the source code.

This tool has been improved, extended and presented in later publications such as [154] [153]. The tool has been finally implemented as an Eclipse plug-in with a FEAT perspective [142] which includes all the views for managing concern graphs. In Figure 15 the general layout of the tool is shown. As an example, the Concern Graph View (Figure 15-(1)) allows the user to create new subconcerns, to delete existing concerns and to move concerns in the hierarchy. Once a concern has been selected (the *active* concern), the Participants View (Figure 15-(2)) shows the participants (elements contributing) for the corresponding concern. The Relations View (Figure 15-(3)) shows the relations between the participants of the *active* concern.

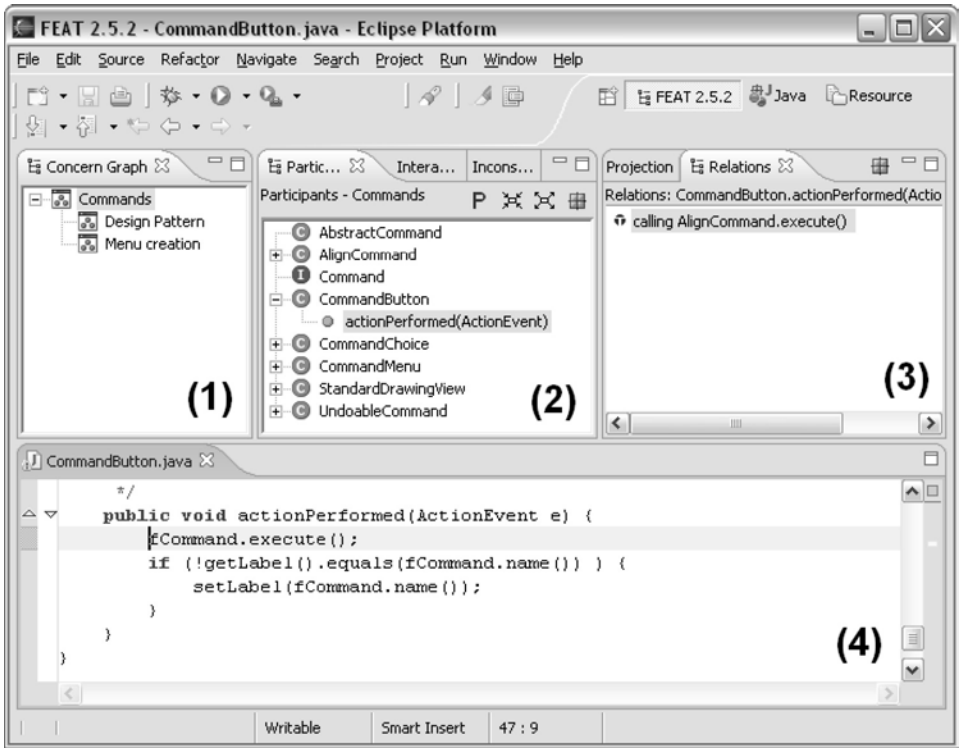


Figure 15. Layout of the FEAT Eclipse plug-in

Since this tool provides an abstract representation of the different parts of a system contributing to a particular concern, scattered concern may be identified just observing this abstract representation. However, the authors neither explicitly provide a view for dealing with crosscutting concerns nor demonstrate how to isolate the crosscutting concerns identified. In addition, the tool may be only used at programming level so that the benefits introduced are relegated to the later phases of the development.

2.2.1.2. Aspect Browser.

In [187], Yoshikiyo et al. presented a tool for localizing and visualizing the crosscutting concerns in a system. As authors state in [187], Aspect Browser is actually a set of tools with a twofold goal: on one hand, to provide and integrate tools to assist a programmer to reason and recall with regard to crosscutting aspects and on the other hand, to encourage programmers to think in terms of aspects as real entities much like other object-oriented artefacts (e.g. classes).

The tool is based on pattern matching technology so that any aspect contains a regular expression and the code matching the regular expression is highlighted with a color. In particular any aspect is highlighted with a unique color wherever it appears in the code [187]. The tool also allows the developer to perform several operations on the aspects such as name, annotate, edit, visualize, etc. Basically, the Aspect Browser is based on two internal tools which perform these actions: Aspect Emacs and Nebulous.

Aspect Emacs provides the core functionality of Aspect Browser. It allows the definition of aspects as a pair: a regular expression and a color. Then, whenever an aspect is activated, Aspect Emacs analyses the code searching for the lines matching the regular expression. These lines are highlighted with the color associated to the aspect. However, Aspect Emacs also searches for new aspects in the code. This functionality is provided by two internal tools: *redundancyfinder* and *aspectfinder*. The tool *redundancyfinder* analyse the source code searching for redundant lines of code (lines of code appearing more than once). The *aspectfinder* tool analyses the identifiers in the code separating them into tags. As an example, the name *delete_source_file* would be separated into the tags *delete*, *source* and *file.* Both tools provide a list of candidate aspects. The user must decide whether the candidate aspects are really crosscutting concerns.

Nebulous provides the visual representation of the crosscutting concerns in the source code. For this purpose, the tool presents the different files of the system as vertical bars. The source lines of code are represented in these bars as rows of pixels. Then, whenever a line of code is related to a particular aspect, the tools shows a row of pixels in the color associated to the aspect. This visual representation allows the developer to have a global vision of how scattered the concerns of the system are or how many concerns a file in the system is addressing. When two different aspects are being addressed in the same line of code, the tool presents this line in red indicating that there is an *aspect collision*. Nebulous also allows the developer to navigate through the source code so that a double click in the lines of a vertical bar causes the file is opened and the part of code is shown in the editor.

Although this tool was initially developed as an standalone application, in [165] the authors explain how the tool was refactored to be used as an Eclipse plug-in. In Figure 16 we can see the AspectBrowser application running as an Eclipse plug-in. The *Aspect Tree* view (at the left side of the figure) may be appreciated where the user enter the aspects in form of regular expressions. The *Visualization* and *Navigation* view (at the right side) shows the vertical bars representing the different files of the system and the aspects highlighted as colored lines.

The AspectBrowser tool may be considered as a perfect complement for languages such as AspectJ since the tool provides a visual representation for the different aspects defined in this aspect-oriented language. However, the aspect mining part of the tool is highly dependant on the programmer's expertise and this could highly constrain the results. As an example, the *aspectfinder* tool needs that the developer uses an specific naming convention in the source code so that the different tags of the identifiers are extracted (tags separated by "_"). On the other hand, this tool is very tied to the programming level so that it could be only used at the later phases of the development, relegating the benefits of AOSD to these stages (as we have already mentioned in previous approach).

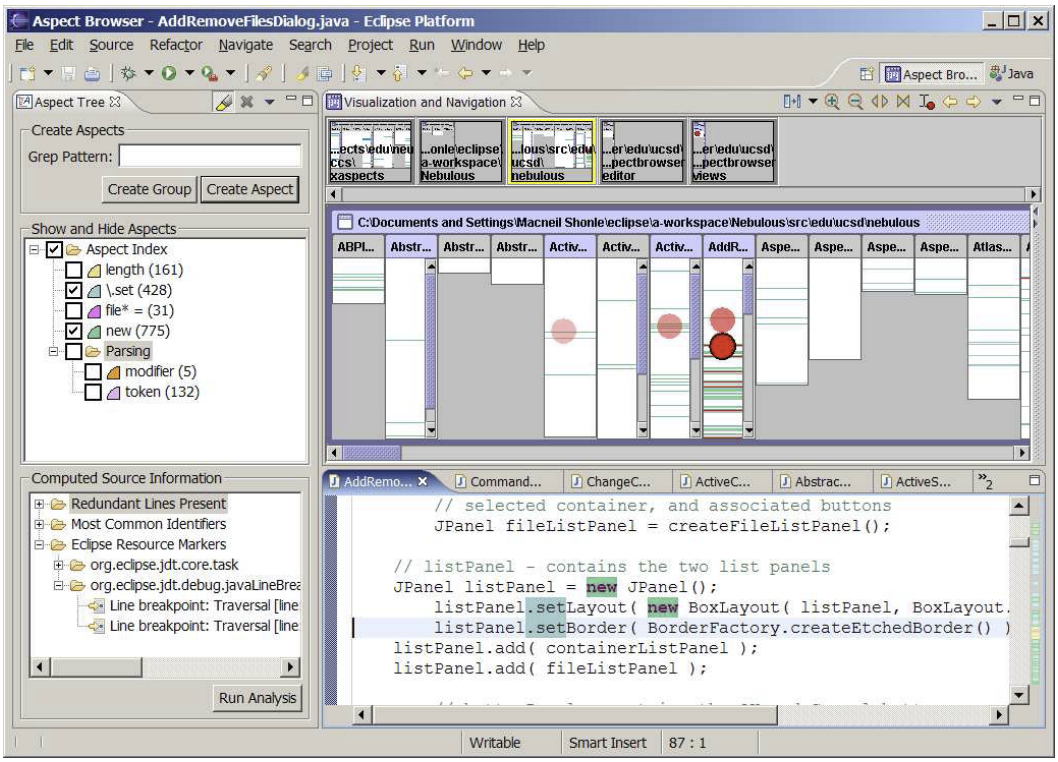


Figure 16. AspectBrowser running as an Eclipse plug-in (extracted from [165]).

2.2.1.3. Aspect Mining Tool.

The Aspect Mining Tool is an application very similar to the Aspect Browser which allows the developer to search for candidate aspects and also visualize the effect of these aspects in the source files using vertical bars. This application was presented by Hannemann and Kiczales in [92]. It is based on the utilization of two internal tools: an analyser and a visualizer.

The analyser provides the developer with two different kind of analysis: a syntactical analysis and a type-based one. Then the application uses the combination of both analyses to obtain the results. The syntactical analysis consists of the application of a pattern-based matching so that any line of code matching with a regular expression is marked. As authors claim in [92] this kind of analysis requires that the source code is written following strong naming conventions, otherwise the technique may fail in many cases. In order to avoid this problem, the authors propose the combination of this technique with a type-based analysis. By using this analysis, the developer may search for all the occurrences of a particular type in a system. The recurrent use of a type indicates the possible existence of *hidden concerns*[3]. The utilization of both techniques avoids the problem of having false positives (hidden concerns wrongly identified) and false negatives (hidden concerns not identified).

The visualizer is a tool very similar to Nebulous tool (presented in previous section) since it allows a graphical representation of the different files of the system showing where the hidden concerns are addressed. Again, each source file is represented as a vertical bar and the lines of code contribution to the realization of a particular concern are highlighted in different colors.

[3] Concerns not represented (i.e. not well modularized) in the current system decomposition and they are hidden into the primary decomposition.

This tool has the same limitations explained for the Aspect Browser Tool. Moreover, as authors explain in [94], *the Aspect Mining Tool is no longer in active development* so that it is not planned to improve the tool solving its limitations.

2.2.1.4. Prism.

Prism is an Eclipse plug-in developed to discover non-localized units of modularity in large software systems [192]. The non-localized units of modularity manifest in software systems as crosscutting concerns such as logging, tracing, synchronization, etc. The Prism tool is based on an aspect mining process consisting of several techniques that are applied in order to identify different kind of crosscutting concerns. Next, the different steps needed to use the tool are described.

As Zhang and Jacobsen explain in [192], the aspect mining process used by this tool is guided by a human miner who knows the existence of aspects in the source code. This human miner is responsible for providing an initial description of the crosscutting structure of an aspect. In order to provide this structure, the miner may use lexical expressions or type-based patterns. This description of the aspect in form of syntactical or type patterns is denoted as the *characterization* of an aspect. This first step is used to identify common crosscutting concerns existing in most of software systems, such as logging, tracing, security or persistence. These crosscutting concerns are well identified by the syntactical and type-based analysis. As an example, logging or tracing concerns are usually described using identifiers containing words like *log* or *trace* or types such as *Logger* or *Tracer*. Thus, this method is used to discover crosscutting concerns with structures at the statements granularity level.

For those crosscutting concerns specific for the domain application, the identification process depends much more on the understanding of the domain specific semantic by the human miner. In that sense, the process now is performed in several refinements level where each iteration provides information to the miner and he send feedback to the tool to refine the results. This process is firstly driven by a lexical pattern where the human miner establishes the crosscutting structure of the aspect being mined. This first structure is based on the miner's intuition. The process is also assisted by a *type ranking* feature where the tool ranks the different types (class types) by its usage in the system. Then, the more widely used types are good indicators of candidate or potential aspects.

Finally a different analysis performed by the authors consists of the study of control flows used to coordinate the interaction between different concerns. As an example, some functionality related to synchronization or concurrency is usually describe by means of conditional branches which are mixed with the functionality related to other orthogonal concerns. Then, the process consists of analysing the values used in these conditional statements and traces all the code where these values are used (e.g. assignments, parameters passing, accessor methods). These conditional statements are used to establish code slices so that when these code slices are not localized or well encapsulated, they are good indicators of candidate crosscutting concerns. However, to date and to the best of our knowledge, this last analysis is not supported by the tool and the authors perform it by manually analysing the code using Eclipse and FEAT [152].

The crosscutting structure defined in Prism to characterize an aspect is defined in terms of two different concepts: *aspect fingerprint* and *aspect footprint*. On one hand, an *aspect fingerprint* abstracts the pattern which is used to identify the aspect in the source code. On the other hand, an *aspect footprint* is an abstraction of the location of a particular *aspect fingerprint*. Then, an *aspect fingerprint* usually localizes many instances of *footprints* across the code base. The concepts represented by *Aspect fingerprint* and *aspect footprint* are similar to the abstraction represented by other two well-know concepts in the AspectJ terminology [10]: *pointcut* and *jointpoint,* respectively. In Figure 17 we can see a screenshot of the tool where

different views are shown. In particular we can see the FingerPrint and FootPrint views at left and right sides of the figure (both at bottom), respectively.

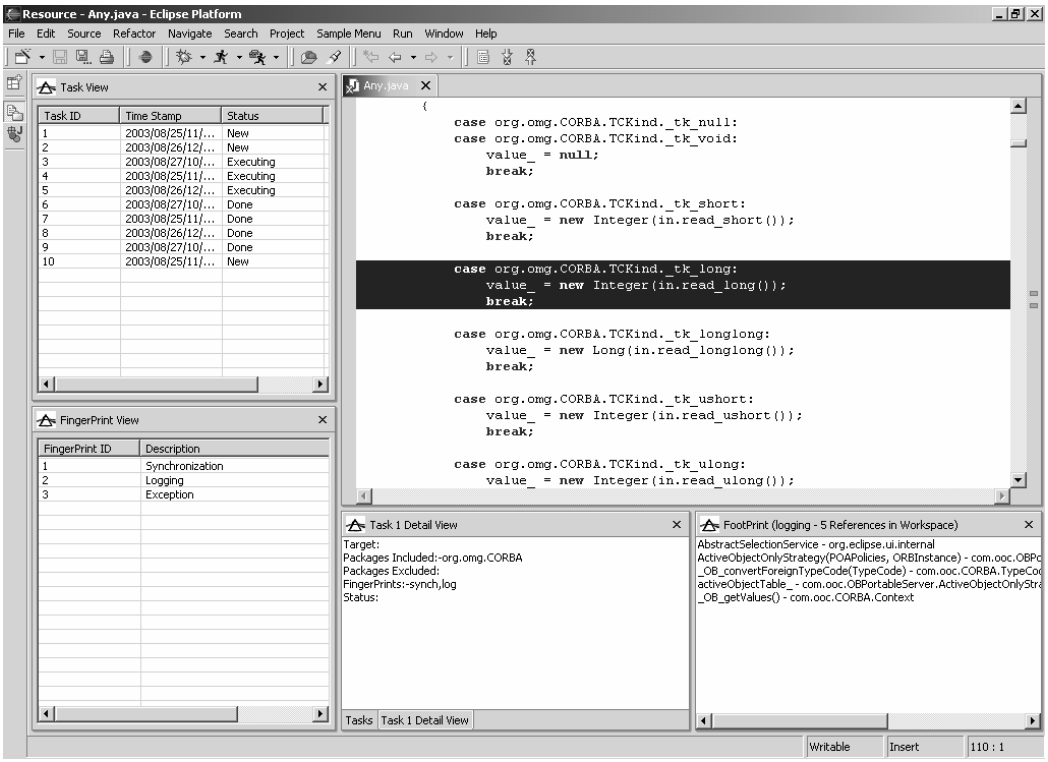


Figure 17. Screenshot of the Prism Eclipse plug-in (extracted from [192]).

This approach basically relies on the same techniques used by the previous ones presented in this section (dedicated browsers). All of them are tied to the programming level so that they may be only applied at the programming level. However, along the last years, the early aspects [65] community have demonstrated the need for using aspect-oriented techniques at early stages of development.

Before presenting this approach, the authors also presented a different tool to visualize aspects based on the Aspect Mining Tool introduced in [92]. This tool was called *Multi-Visualizer,* although the authors also refer to this application as the Extended Aspect Mining Tool (AMEXT) [191]. This tool tried to solve some problems identified in the Aspect Mining Tool improving mainly its scalability so that the aspect mining process could be applied to larger software systems consisting of thousands of classes with millions of lines of code. In that sense, the main differences between both versions were: 1) the extended version was able to use a pattern matching using a coarser granularity level (e.g. packages) so that the process could be applied to bigger systems and 2) the authors introduce some metrics to define the purpose of the mining activity (collection of pattern matching class types or ranking of more used classes).

2.2.2. Aspect mining techniques.

In this section a detailed survey of different aspect mining techniques is shown. At the end of the section, a table summarisying the distinct approaches is shown (Table 3). The reader with background on the aspect mining topic may just refer to this table in order to have a quick and global view of the approaches analysed in this section.

2.2.2.1. Dynamo

In [177] Tonella and Ceccato presented Dynamo which is an aspect mining tool based on the application of Formal Concept Analysis (explained in Section 2.1.3.1). This tool analyses execution traces in order to relate two different domains: use cases (or scenarios) and methods involved in the execution of these use cases. Then, the authors establish a concept lattice which shows the relation between these two different domains and identifies the crosscutting concerns by observing use cases addressed by different classes or classes involved in the execution of more than one use case.

As it is explained in Section 2.1.3.1, Formal Concept Analysis is a branch of lattice theory that provides a way to identify maximal groups of objects that have common attributes [177]. In that sense, given a *context (O, A, R)* where R represents a binary relationship between a set of objects O and a set of attributes A, a concept C is defined as a pair of sets (X,Y) so that:

$$X = \{o \in O \mid \forall\, a \in Y : (o, a) \in R\}$$

$$Y = \{a \in A \mid \forall\, o \in X : (o, a) \in R\}$$

where *X* is said to be the *extent* (*Ext[c]*) of the concept *c* and *Y* is called the *intent* (*Int[c]*) of *c*.

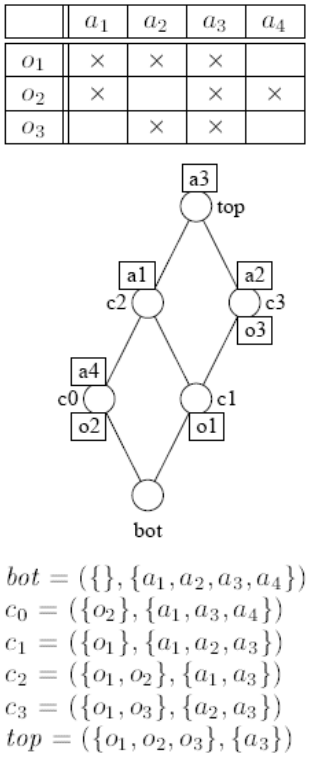


Figure 18. Concept lattice extracted from [177]

Given the set of concepts for a concept lattice L, the authors also establish a labelling technique where a concept is labelled with the more specific object or attribute that characterizes it. More precisely, a concept *c* is labelled with an object *o* only if *c* is the most specific (i.e., lowest) concept having *o* in the extent. A concept *c* is labelled with an attribute *a* only if *c* is the most general (i.e., highest) concept having *a* in its intent. Then, when a concept c is labelled, the objects or attributes that label it indicates the most specific properties that characterize this object. In Figure 18 we can see an example of concept lattice extracted from [177]. In this figure the relationship between three objects and four attributes is shown (using both a relationship matrix and a concept lattice). At bottom part, the different concepts of the lattice are shown.

The approach uses a feature location technique to relate computational units (procedures or class methods) to the features (requirements or scenarios) implemented by these computational units. The technique used is based on execution traces so that the relations are obtained by a dynamic analysis. In this analysis, the developer provides a set of scenarios to be executed (the set of objects in the concept lattice) and the tool provides the set of computational units involved in the execution of these scenarios (the set of attributes in the concept lattice). Then, the relation *R* contain the pair *(o,a)* if the computational unit *a* is executed when scenario *o* is performed [177]. In this setting, computational units label a concept if it is the most specific computational unit for the scenario and an scenario labels a concept is this concept has only this scenario in its extent.

Once the authors have obtained a concept lattice with the computational units addressing each scenario they manually review this lattice in order to obtain the candidate crosscutting concerns. In particular, they identify two kinds of situations as an indication of the presence of crosscutting concerns:

- use case specific concepts labelled by computational units belonging to different classes.
- different computational units (belonging to the same class) label more than one use case specific concept.

The two situations analysed by the authors suggest that they search for the presence of scattering (first situation) and tangling (second one). In that sense, the authors consider that crosscutting is a combination of scattering and tangling so that any of the situations alone are not sufficient to have crosscutting (both situations are needed).

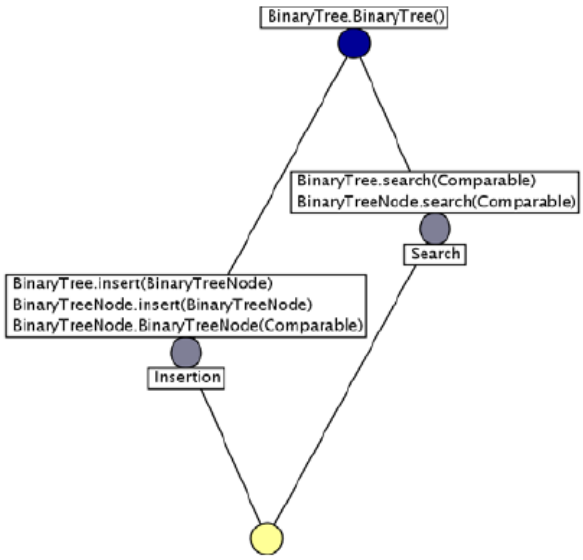


Figure 19. A real example of concept lattice (also extracted from [177]).

In Figure 19 we can see an example of concept lattice for a real application. This example implements a binary search tree application with two main functionalities (features): insertion and search. As the authors concluded in [177], in this application both insertion and search features are considered as crosscutting concerns since the two use case specific concepts (insertion and search) are labelled by methods that belong to more than one class (scattering). On the other hand, it can be also observed how each class contributes to more than one feature (tangling).

This utilization of concept lattices to represent the relations between two different domains could be compared to the use of our traceability matrices (shown in Chapter 4). Both tools are able to represent these relations; however traceability matrices are easier to use and

the user do not need to know the underlying mathematical theory to establish the relations between two domains. Anyway, the aspect mining process used by Dynamo is use cases driven so that the developer could try to identify the crosscutting concerns earlier in the development. However, the use of execution traces to obtain the crosscutting concerns implies to postpone this identification to a state where the system must be, at least, partially developed.

2.2.2.2. DelfSTof

In [37][134] Tourwé et al. presented an approach that also uses Formal Concept Analysis to mine source code by searching for what they called *source-code regularities*. These *source-code regularities* may indicate, among other concepts, what concerns are addressed in the source code, what patterns, program idioms, conventions or where the different concerns are implemented. One of the *source-code regularities* identified by the approach is the *crosscutting concept* which denotes the presence of a crosscutting concern.

As it is aforementioned, this approach uses Formal Concept Analysis (see Section 2.1.3.1 or 2.2.2.1 to obtain more details on this technique) to mine the source code. In order to perform this task, the authors establish a relation between a set of elements and a set of attributes related to these elements (to build the concept lattice). The set of elements used to build the concept lattice is composed by the source code entities used in the system being analysed: classes, methods and methods parameter. The attributes associated to these entities are automatically generated by using a syntactical analysis. This analysis is based on the identifiers of the different entities. In this analysis, the identifier of the entity is divided into the different tokens or words that compose the identifier. As an example, a class with the name *QuotedCodeConstant* is divided into the tokens *quoted*, *code* and *constant*. These three tokens are attributes associated to the class *QuotedCodeConstant.* Then, using this analysis, the authors build a concept lattice where the different concepts represent the source code entities (elements) with the related tokens or words (attributes).

Once the concept lattice has been built, the authors filter this lattice in order to reduce the number of concepts obtained. This task is performed by removing some concepts that do not provide too much information. As an example, the concepts with two or less elements are removed since the authors claim that *these concepts do not provide relevant information* [134]. All the concepts that share just one property are also removed. Finally a third filter removes all the concepts which contain classes with a similar name in the same hierarchy. The authors claim that these concepts do not provide very useful information (*except if we want to know the naming convention used in the hierarchy* [134]).

When the different filters have been applied to the concept lattice, the authors use a manual filter where they classify the concepts according to the elements and the related properties. Although the authors use a large classification for the concepts, in [134] they mainly group the concepts into three different categories:

a) *Single class concepts* which are those containing just methods belonging to the same class.

b) *Hierarchy concepts* are concepts which contain classes, methods and methods parameters that belong to the same class hierarchy.

c) *Crosscutting concepts* are concepts where two different class hierarchies are involved. In order to verify this situation, the authors check that the most specific common superclass is *Object* and that none of the methods in the concept are defined on the *Object* class.

Then, by using this classification, the authors annotate the different concepts helping to identify the crosscutting situations. This fact allows the developer a better understanding of the system and enables the possibility of using refactorings.

This approach has tool support, DelfSTof. The goals of the tool are several: performing the FCA algorithm, building the concept lattice, filtering the results, classifying and annotating the concepts. In Figure 20 a screenshot of the tool is shown (extracted from [134]).

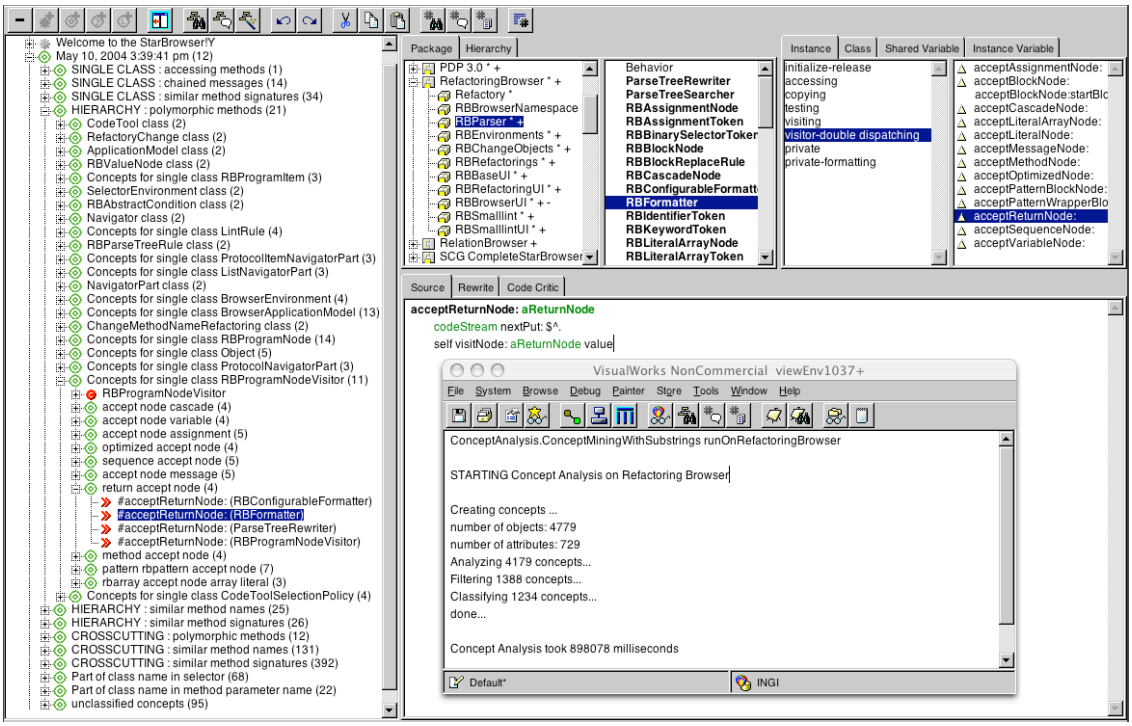


Figure 20. DelfSTof tool: applying FCA to mine source-code regularities

However, as authors state in [37], the approach has important limitations. One of the most important limitations is that the FCA algorithm has been designed for being applied to Smalltalk applications so that it could be highly dependant on the naming conventions used in this language. Although the authors claim that the algorithm could be easily adapted to other languages (e.g. Java), the use of such a lightweight technique (using substrings of classes and method names) may lead to obtain a high number of false positives and negatives (as authors indicate in [37]).

2.2.2.3. DynAMiT

DynAMiT was introduced by Breu and Krinke in [30]. This approach is based on a dynamic analysis of execution or program traces to obtain crosscutting concerns. Basically, the program traces analysis tries to establish relations between method calls. These relations are called *execution relations* and they describe in which relation two method executions are.

The *execution relations* defined in [30] are based on program traces. Formally, a Program Trace T_p of a program P with method signatures $\mathcal{N}_P$ is defined as a list $[t_1, \ldots, t_n]$ of pairs $t_i \in (\mathcal{N}_P \times \{ent, ext\})$ where *ent* and *ext* denote entering and exiting a method execution, respectively.

Then, the authors search for two kind of *execution relations* in order to identify crosscutting situations. In particular, the *execution relations* that they are interested are: a method may be executed either after the preceding method execution is ended or inside the execution of the preceding method call. These *execution relations* are called *outside-* and *inside-execution relations*. However, these two kinds of relations are not sufficient for aspect

mining and the authors refine the relations into 4 different categories: *outside-before-execution*, *outside-after-execution*, *inside-first-execution* and *inside-last-execution relations*. On one hand, *Outside-after-* and *outside-before-execution relations* represent the situation where a method is called before and after a different call method, respectively. On the other hand, *inside-first-* and *inside-last-execution relations* denote the situation where a call is executed as the first and the last invocation within a method, respectively.

The *ent* and *ext* situations are represented in [30] as "{" and "}" symbols respectively so that a program trace may be easily represented in a visual way. In Figure 21 we can see an example of a program trace. Using the program trace of the figure, we can observe several examples of the aforementioned types of execution relations. For instance, we can observe how method C in line 2 is the first invocation inside method B in line 1. Then, there is an *inside-first-execution relation* between these two methods. Analogously method A in line 7 has an *outside-after-execution* relation with method B in line 1.

```
1 B() {          17   J() {}      33   }
2   C() {        18 }             34 }
3     G() {}     19 F() {         35 D() {
4     H() {}     20   K() {}      36   C() {}
5   }            21   I() {}      37   A() {}
6 }              22 }             38   B() {
7 A() {}         23 J() {}        39     C() {}
8 B() {          24 G() {}        40   }
9   C() {}       25 H() {}        41   K() {}
10 }             26 A() {}        42   I() {
11 A() {}        27 B() {         43     J() {}
12 B() {         28   C() {}      44   }
13   C() {       29   G() {}      45   G() {}
14     G() {}    30   F() {       46   E() {}
15     H() {}    31     K() {}    47 }
16   }           32     I() {}
```

Figure 21. Example of program trace [30]

Once the authors have the set of execution relations existing in a system, they analyse the recurring execution relations in order to identify crosscutting situations. But, they do not only focus on the identification of recurring execution relations and they define several constrains to decide whether these recurring relations are an indication of a crosscutting concern. Basically, the constrains defined establish: 1) a execution relation is recurring if and only it exists in always the same composition (as an example, an outside-before execution relation between u and v is defined as recurring if each execution of v is preceded by an execution of u), 2) a recurring execution relation is defined as crosscutting if and only if it occurs in more than a single calling context (following the same example with u and v, there is other method invocations where u is also always preceding). Basically, the second constrain states that the crosscutting method must crosscut more than one method to be defined as crosscutting.

The main contribution of this approach is the utilization of dynamic analysis to identify the crosscutting concerns. By now, most of the approaches explained rely on the analysis of static source code to identify the crosscutting concerns. However, the utilization of run time analyses may identify situations not contemplated by the static ones. Nevertheless, the approach used by DynAMiT would fail in identifying other crosscutting concerns using a granularity level different from methods (e.g. in exception handling). Moreover, the approach could work properly with well-know crosscutting concerns such as logging, tracing or concurrency since they usually follow a same pattern to interact to the code base (homogeneous crosscutting, e.g. logging is always performed after doing some functionality). However, there could be domain specific crosscutting concerns that do not follow this same pattern to interact to the base code (heterogeneous crosscutting). In those cases, the method would not identify such crosscutting concerns.

In fact, in [29] the authors extended the tool by adding static analysis in order to solve some of the problems existing in the dynamic analysis. As authors claim in [29], the main problem with the dynamic analysis was the identification of false positives due to the dynamic binding of methods analysed. That means that a call to a method could produce invocations to different methods when the method is abstract (polymorphic). The author identified that the analysis systematically produced false positives whenever the code contains abstract methods with several concrete implementations. In addition, since the DynAMiT tool internally uses AspectJ to trace the method execution, the authors realised that the utilization of execution pointcut (instead of method call pointcut) also contributed to identify false positives due to dynamic binding. The reason is that execution pointcut uses the type of the actual method invoked (after the dynamic binding has been done). In this setting, the DynAMiT tool was extended using a static analysis where the static signature of a method at the call site (not only the signature of the method actually invoked) was traced and analysed. The authors claim that, after the extension, the number of false positives highly decreased and the number of crosscutting patterns identified increased between 167 and 200%.

2.2.2.4. AMAV

The Aspect Mining and Viewer (AMAV) tool presented in [163] is an aspect mining application which relies on the clustering of methods based on similarities between their identifiers. Basically, the application provides to the user with a view where all the methods related to a particular concern (assuming that the concern is characterized by some words appearing in the name of the methods) are presented. In order to group the methods, the authors utilize a Natural Language Processing (NLP) technique which establishes a distance function between these methods. Then, the groups contain methods with a small distance among them. The author, Shepherd and Pollock, claim that these groups often represent crosscutting concerns.

The approach uses agglomerative hierarchical clustering (ALC) [105] to group the methods related to a concern. The algorithm used is described as follows:

1. Every method is included into a different group.
2. All the groups are compared in pairs. The pair of groups with the smallest distance is marked.
3. If the distance for the marked pair of groups is smaller than a threshold, the groups are merged in a new group. Otherwise the algorithm ends.

Steps 2 and 3 are repeated until the condition of *smallest distance between groups is bigger than the threshold value* is fulfilled. Finally, the algorithm rules all the groups with a unique member out and it returns the rest of groups. Note that when two groups are merged, the group resulting is presented as the root node of a tree where the children are the merged groups. The identifier of the root node in the new group is formed by the common-substring in the name of the children.

Since AMAV groups the methods by relating their identifiers, the distance function is based on these identifiers. In particular, the authors calculate the distance function between two methods as follows: the distance function for two methods m and n is $1/commonSubStringLength(m.name, n.name)$. When the distance function is calculated between two groups, the identifier of the root node of the group is used to compare. As authors claim in [163], one of the advantages of using this distance function is that it is pluggable so that it could be changed by a different function.

Once the clusters of methods are calculated, the tool presents them to the user in a visual way. Actually, the viewer part of the tool shows three different panes: the crosscutting pane, the cluster pane and the editor pane. The crosscutting pane is the view where all the methods related to a particular concern are shown. We can see that this view presents to the user a

"virtual source file" with all the related methods (without their actual context since the classes which contain these methods are not shown). The cluster pane displays all the clusters or groups identified by the tool. Finally, the editor pane presents the source file (a class or interface) where a particular method is defined (i.e. its context). Then, when the user selects a particular cluster in the cluster pane, he obtains a crosscutting view of all the implementations of the concern.

Moreover, the approach is used to discover new crosscutting concerns from scratch, however, as authors state in [163], it could be easily adapted to ask the user for a seed in order to use it for the mining process. Nevertheless, the main problem this tool presents is that it only works at a particular granularity level (method level) and some crosscutting concerns that use a finer granularity level could not be identified (e.g. exception handling mechanisms).

2.2.2.5. Natural Language Processing based Clusters

In [164], a Natural Language Processing (NLP) technique is also used to form groups of source code entities related to a particular concern. Based on these groups, Shepherd et al. identify crosscutting concerns by exploring the groups containing words appearing in different source code files. Unlike the approach presented in previous section, the NLP technique used here is not only based on syntactical information (identifiers) but also on semantic (the context of the words and the meaning is also analysed). As authors claim in [164], the tool developed improves the AMAV [163] results (explained in Section 2.2.2.4) since in AMAV *any slight syntactic variations in the names have a large impact on the results.* In addition, authors state that AMAV is not able to relate methods semantically equivalent but syntactically different.

This approach uses a technique called *lexical chaining*. This technique is responsible for grouping the words semantically related. The tool takes as input a text and builds chains with all the related words. The chains are built based on a distance function used to assess the degree of relation between two words. In order to calculate this distance, the WordNet [33] database is used (a database of known relationships between natural language words). The distance is then obtained by assessing the lengths of the relationships path between the two words. The longer the length the less related the two words are. The authors also explain that the context of the word could be analysed by using tools such as QTAG [179]. However, they do not clarify whether the tool presented perform this kind of analysis.

Finally, the tool (an Eclipse plug-in) presents to the user the results using the Lexical Chain Viewer where all the clusters identified are shown. Note that the tool analyses source code entities such as method, field and classes names but also comments. Then, once the results are presented, the developer analyses the clusters containing several items (and belonging to more than one class) and decide whether they represent a crosscutting concern's implementation. Then, an aspect-oriented refactoring may be performed to improve the system modularity.

It is clear that this approach may considerably improve the results presented by AMAV, however the approach is still using a prefixed granularity level (method or field) so that a simple statement could not be analysed. This fact implies that some crosscutting concerns could not be identified. Maybe the analysis of the comments in the source code could solve this situation, however the tool should trust that the system is well documented (and this is not always the actual situation).

2.2.2.6. Clone detection techniques.

It has been demonstrated in the literature that crosscutting concerns usually manifest in form of source code duplication. Since these concerns may not be well modularized and encapsulated into separated entities, developers are often forced to write the same code over

and over again, using finally the practice of copying, pasting and slightly adapting the code to the needs [32]. Tracing or logging concerns are good examples of these concerns with an implementation duplicated all over the system. Then, in the last years there have appeared several code duplication techniques to identify crosscutting concerns allowing, thus, to apply an aspect oriented refactoring to remove this duplicated code.

The clone detection techniques used in the literature vary from those looking for text-based duplicated structures to those using program dependencies graph to identify the clones. Basically these techniques may be classified as follows [32]:

- **Text-based approaches** (e.g. [59] [101]). These techniques directly use the textual representation of the source code to search for identical or slightly different sections of code.
- **Token-based approaches** (e.g. [11] [102]). They pre-process the source code and tokenize the source code so that these tokens are used to detect the clones.
- **AST-based techniques** (e.g. [19]). Before searching for the clones, a parser analyses the source code and builds an abstract syntax tree (AST) representation of this code. Then the AST is analysed in order to detect similar subtrees.
- **PDG-based approaches** (e.g. [114] [116]). These approaches use a more abstract representation of the source code, they use program dependencies graphs (PDG). These graphs contain not only syntactical information but also semantic. Then, the approaches try to detect similar graphs in order to identify the clones.
- **Metrics-based approaches** (e.g. [130]). In these approaches several metrics are calculated in order to assess some attributes of code fragments (e.g. fan-in). An example of these approaches is deeper explained in Section 2.2.2.7.
- **Information retrieval-based techniques** (e.g. [127]). They try to identify concerns by exploiting semantic similarities in the source code itself.

The authors in [32] selected 3 tools belonging to three of these techniques and applied them to the same case study. Then, they compared the results in order to assess the accuracy of these kinds of techniques. The tools used for the analysis were: the *ccdiml* tool contained in the *Project Bauhaus* [148], belonging to the AST-based techniques; *CCFinder* [102], a token-based tool; and finally a PDG-based technique denoted as *PDG-DUP* [114].

All the tools selected were applied to a particular component (called CC) of a real application denoted as ASML. This component consists of 16406 LOC (written in C) and the whole application has over 10 million LOC). In order to assess the accuracy of the tools analysed, the component was annotated by an original developer so that any line of code belonging to 5 well-know crosscutting concerns was marked. The 5 crosscutting concerns were: *Memory Error Handling, NULL-value Checking, Range Checking, Error Handling* and *Tracing*. Then, the annotations made by the developer were compared with the results obtained by the tools.

The results obtained by the analysis showed that the tools really identified the crosscutting concerns so that these crosscutting concerns were characterized by code duplication (or other representation such as AST, tokens or PDG). Of course, the results obtained by the different tools were, in all cases, worse than the results obtained by the human. But, in general, the results obtained were satisfactory enough to use these automatic tools. Note that manually annotating the source code would be unfeasible in real developments. Some interesting results obtained by the authors showed that *ccidml* tool obtained the best results for some of the crosscutting concerns (*Range Checking, NULL-value Checking* and *Error Handling*). *CCFinder* obtained similar results for *NULL-value Checking* and *Error Handling* and finally *PDG-DUP* behaves as the best tool for *Tracing* and *Memory Error Handling*. Even the authors compared the results obtained by the application of a combination of the three techniques and

concluding that these results were the most convincing. Refers to [32] to obtain all the data and the whole analysis. Although the results obtained by the authors were convincing about the useful of these techniques to detect crosscutting concerns, it seems that most of the concerns used for the analysis were highly coupled to the implementation level, as an example consider the *NULL-value Checking* or *Range Checking* concerns. Then, although being aware of the contributions of these kinds of techniques, they could be limited to their utilization at late phases of development.

2.2.2.7. Fan-in analysis.

Some of the most common design structure metrics are fan-in and fan-out. These metrics are based on the ideas of coupling presented by Yourdon and Constantine in [189]. In particular, fan-in and fan-out metrics are defined as [103]:

- Fan-in: it counts the number of modules that call a given module.
- Fan-out: it measures the number of modules that are called by a given module.

In [128] Marin et al. presented an aspect mining process based on the utilization of the fan-in metric. The process consists of measuring the fan-in metric for all the methods of a system. Then, those methods with highest fan-in values are considered as functionality belonging to a crosscutting concern and they are candidates to be modelled using aspect-oriented techniques. The authors provide a particular definition of the fan-in metric as: *fan-in of a method m is the number of distinct methods bodies that can invoke m.* Since the approach is applied to object-oriented systems (and because of polymorphism), the authors state that a method call could affect the fan-in of several other methods. As an example, a call to method *m* will affect the fan-in metric of all methods refining *m* but also to the methods refined by *m*.

The approach is based on the application of three systematic steps that may be automated:

1) In the first step, the authors calculate the fan-in metric for all the methods in the source code of the application. This step is automatically performed by using an Eclipse plugin developed using some Eclipse functionalities such as the *search for references* feature. After this analysis, the results are stored and ordered by fan-in values in a special structure. This structure allows the developer to navigate through the methods called and the calling context.
2) Secondly, a filtering step is applied where the methods obtained by the previous step are analysed. The main goal of this step is to discard the methods that should not be considered as candidate aspects. As an example, the authors discard methods like:
    a. Methods which have a fan-in metric lower than a threshold value. This threshold value allows the authors to relax or to harden the condition to identify candidate aspects depending on the targeted system. The authors explain that 10 is a generally used threshold value since this value represents the 5% of the total number of methods. However, the results presented in [128] demonstrated that using a threshold value of 8 or 9 could also provide interesting results (not identified with 10).
    b. Getters and setters method are also discarded. In this case, the authors use a double iteration where firstly they discard the methods based on their signature and secondly in their implementation. The only getters and setters methods considered for the analysis are those accessing to static fields. These methods usually represent functionality related to some design patterns, such as Singleton. As it is demonstrated in [128], the

implementation of many design patterns may be improved using aspect-oriented techniques. Thus, they are well-know candidates to be identified using aspect-mining techniques.

c. Some well-know utility methods are also discarded. Examples of these utility methods are *toString*, methods provided by collections or other methods provided by common APIs.

3) Finally a third step consists of manually reviewing the resulting set of methods. In this step, the developer must analyse the results after filtering the discarded methods. In this analysis, the authors encourage not only to analyse the callers and call sites but also the methods implementation and comments in the code.

Authors show the results obtained by the application of the aspect mining process to several case studies [128] (namely PetStore [145], JHotDraw [100] and Tomcat [176] systems). By using these results they demonstrate the utility of the process to identify well-know crosscutting concerns (e.g. tracing or logging). However, they also identified other crosscutting concerns not previously presented in the AOSD literature like the *undo* concern in JHotDraw or the *lifecycle* concern in Tomcat. The authors also claim that the process may be used to identify crosscutting concerns by providing a *seed*. This *seed* represents a starting point for identifying the functionality of a particular concern.

The fan-in and fan-out analyses are some of the most used techniques in aspect mining for identifying crosscutting concerns. The combination of these techniques with others shown in previous sections could highly improve the results obtained by the aspect mining process (as authors mention in [128]). In that sense, the combination of different techniques could help to avoid some problems that fan-in techniques present. As an example, the fan-in threshold value used to determine the methods candidates to be aspectized is a limitation that could highly constrain the results obtained. In addition, after filtering of methods to be discarded, the resulting set of methods could be still very huge. Then, the manual analysis of these results is really time-consuming and the developer could dedicate a big effort to this task. In next section an approach combining three different aspect mining techniques (being fan-in analysis one of the techniques used) is presented.

2.2.2.8. Combining different aspect mining techniques.

As it is aforementioned the combination of different aspect mining techniques may improve the results obtained by the application of each technique separately. That is demonstrated in [36] where Ceccato et al. applied three different aspect mining techniques to the same case study. The results obtained where, then, compared with those obtained by the application of different combinations of the techniques to the same system.

The example system used for the application of the aspect mining techniques was the JHotDraw graphical editor framework [100]. This system was developed following good object-oriented design patterns [80]. This fact implies that the number of crosscutting concerns identified should be lower than in other systems since the system is well designed. Nevertheless, at is shown in [36], several crosscutting concerns are identified in the system demonstrating limitations of object-oriented designs to encapsulate certain properties or functionality of the systems.

The authors selected a set of three aspect mining techniques to perform the comparative analysis: (i) the fan-in technique, summarised in Section 2.2.2.7 [128]; (ii) an identifier analysis based on the utilization of Formal Concept Analysis, described in Section 2.2.2.2 [37]; (iii) a dynamic analysis that also uses Formal Concept Analysis to locate features' implementation in execution traces, also described in Section 2.2.2.1 [177].

The results obtained by the application of the three aspect mining techniques to the JHotDraw system were used to extract important conclusions. The first conclusion extracted was that all the techniques used had limitations. This conclusion was previously suspected, however it was confirmed by the data obtained. The reason was really simple; the sets of crosscutting concerns identified by the different techniques were different to each other. That implies that all the techniques presented false negatives (the crosscutting concerns that the techniques does not identify and the others do).

In particular, the results obtained indicated that the fan-in technique was suited to identify crosscutting concerns highly scattered and which highly influence in the software decomposition. However, crosscutting concerns which not manifest under these conditions are not identified. These concerns with a small code footprint and, thus, with low fan-in are missed. In [36], the authors of the survey give as an example the Observer pattern. The identification of this pattern as a crosscutting concern using the fan-in technique is dependent of the number of classes implementing the observer role. Note that the number of calls to the registration operation of the subject role depends on the number of observer classes.

On the other hand, the identifier analysis usually provides too much information and results. However, the results obtained require to be manually filtered since they often contain false positives. Thus, the developer must spend too much effort and time in this task. Moreover, the results obtained are often incomplete so that sometimes the crosscutting concern implementation is not totally identified. This is due to the fact that concerns are usually described by more than one concept. Using the same example, the Observer pattern could be described by terms like *register* or *update*.

Finally, the results obtained by the dynamic analysis were also incomplete. Sometimes the execution of a particular scenario does not imply that all the methods related to a concern are affected. In that sense, the results are highly influenced by the execution traces designed for the different concerns.

In order to illustrate the results obtained, the authors showed a table where the results obtained by the dynamic and the fan-in analyses are compared. Note that only 4 of the 30 different concerns identified by these techniques were identified by both of them (see Table 2). The identifier analysis was considered as the least discriminating of the three techniques. The overlapping between this technique and the others was quite large and a concern identified by dynamic or fan-in analysis was usually identified by the identifiers technique. This is mainly due to the fact that a common lexicon is usually used to define methods related to a particular concern. As an example, when the *registration* operation of the Observer pattern is identified by fan-in analysis, it is also identified by the identifier analysis since *registration* is an identifier related to the Observer pattern.

Technique	Concerns
Dynamic analysis	18
Fan-in analysis	16
Dynamic analysis ∪ Fan-in analysis	30
Dynamic analysis ∩ Fan-in analysis	4

Table 2. Crosscutting concerns obtained by the fan-in and the dynamic analyses in the JHotDraw system

In this setting, the authors considered the combination of the three aspect mining techniques. The main goal is to analyse whether the results obtained improve those provided by the different techniques individually. In that sense, they analysed different combination possibilities. The easiest combination was to use the different techniques individually and consider as results the union of the results obtained by each technique. This combination would have an important contribution: the crosscutting concerns identified by the three techniques are likely the best aspect candidates.

However, the authors observed two parameters that should be taken into account to consider a particular combination. Firstly, the identifier analysis requires a manual filtering of the results that consumes too much time and effort. That makes the applicability of this technique in large systems difficult. Secondly, they observed that the fan-in and dynamic analyses just identify a starting point of the concern implementation (the seed). The rest of the concern implementation must be explored by expanding this starting point. They also realized that the classes and methods obtained by manually exploring the extension have similar identifiers. Then, the evidences showed that a good combination could be the utilization of the identifier analysis as a seed expansion technique to analyse the crosscutting concerns identified by the fan-in and dynamic analyses (used individually).

Then, the algorithm proposed to combine the three techniques was the next:

1) Identify the candidate seeds by using fan-in analysis, dynamic analysis or both of them.
2) For the methods belonging to the candidate seeds obtain the identifiers occurring in the method name. These identifiers are obtained using the algorithm defined by the identifiers technique (separating words in tokens; e.g. *QuotedCodeConstant* is separated into *quote, code and constant).* The identifiers existing in the enclosing class of each method identified are also taken into account in this analysis.
3) Apply the identifier analysis to the entire application and search for the *nearest* concepts to the identifiers obtained in the previous step. The *nearest* concept is the one that contains more identifiers related to a seed.
4) Add the methods contained by the *nearest* concept to the seed expansion.
5) Review the results obtained in order to avoid false positives (wrongly identified) and false negatives (missed seeds).

The data obtained by the algorithm showed that, in general, the utilization of the identifier analysis after applying the other techniques (either one of them or both of them) improves the results obtained without this combination. The only case where the algorithm behaved worse was for the Observer pattern. The results obtained by applying the identifiers analysis were worse than those obtained without this analysis. After a manual exploration of the results, the authors concluded that identifiers analysis did not properly worked with this concern because of the naming convention used to implement it. Examples of identifiers used to implement the Observer pattern were figure or update. However, these identifiers were very common in all the JHotDraw application (remember that it is a graphical editor framework). Then, the number of methods identified as belonging to the Observer pattern was huge increasing the number of false negatives. Anyway, the results obtained demonstrated that, in general, the combination of different techniques or analyses helps to improve the aspect mining process.

2.2.2.9. Summarisying aspect mining techniques

Finally, in this section the different approaches for mining aspects are compared using the criteria established in [107], where the authors presented a deep survey of different aspect mining techniques at the programming level. The criteria used to compare the approaches are the next:

1. **Static versus dynamic data**. This criterion distinguishes whether the approaches use static data to perform the analysis (at compiling time) or they work using dynamic data (executing the program).
2. **Token-based versus structural behaviour analysis**. While some approaches use lexical analysis of the program using regular expressions and sequences of

characters, others use structural behaviour such as parse trees, or type information.

3. **Granularity**. By this criterion, the approaches are differentiated in terms of the granularity level used for the analysis: methods, individual statements, code fragments, and so on.
4. **Tangling and scattering**. Crosscutting concerns are characterized by scattering and tangling. However not all the approaches search for symptoms of both characteristics, sometimes the only presence of scattering is used to identify crosscutting concerns.
5. **User involvement**. In some approaches, the results provided by the approach must be manually analysed in order to filter the results.
6. **Larger system**. By this criterion, the larger systems where the approaches have been applied are compared.
7. **Empirical validation**. The approaches are compared in terms of the validation performed in real-life cases.
8. **Preconditions.** Some aspect mining techniques require that the concerns in the program satisfy a set of conditions in order to identify candidate aspects.

Based on the criteria shown, the approaches are compared adn summarised in Table 3. In this table, the different approaches are shown in rows whilst the different criteria are represented in columns. The cells detail the property that a particular approach has regarding to the criterion of the corresponding column. A column showing the main limitations of each approach has been also included.

		Sect.	Criteria 1: Static	1: Dynamic	2: Token	2: Structural	3: Method	3: Code Fragments	4: Scattering	4: Tangling	5: User involvement	6: Larger system	7: Empirical validation	8: Preconditions	Limitations
Approaches	Dynamo	2.2.2.1	-	X	-	X	X	-	X	X	Selection of the use cases and manual interpretion of results	JHotDraw (18KLOC)	-	At least a use case existing exposes the crosscutting concern and another one does not	- User needs to know the underlying mathematical theory. - Benefits of AO relegated to the final phases of development.
	DelfSTof	2.2.2.2	X	-	X	-	X	-	X	-	Browsing of mined aspects using IDE integration	JHotDraw (18KLOC)	-	Names of the methods implementing the concerns are alike	- Very dependant of the implementation language used. - It may obtain a high number of false postitives and negatives.
	DynAMIT	2.2.2.3	X	X	-	X	X	-	X	-	Inspection of the resulting recurring patterns	Graffiti (82KLOC)	-	Order of calls in context of crosscutting concerns is always the same	- Fail in detecting crosscutting concerns with a granularity level different from method or domain specific ones. - Abstract methods produce a high number of false positives.
	AMAV	2.2.2.4	X	-	X	-	X	-	X	-	Browsing of mined aspects using IDE integration	JHotDraw (18KLOC)	-	Names of the methods implementing the concerns are alike	- Fail in detecting crosscutting concerns with a granularity level different from method.
	NLP based clusters	2.2.2.5	X	-	X	-	X	-	X	-	Manual interpretation of resulting lexical chains	PetStore (10KLOC)	-	Context of concern contains keywords which are synonyms for the crosscutting concern	- Very dependant of the writing style used.
	Clone Detection Techniques: PDG based	2.2.2.6	X	-	-	X	-	X	X	-	Browsing and manual interpretation of the discovered clones	TomCat (38KLOC)	-	Concern is implemented by reusing a certain code fragment	- Crosscutting concerns identified are very tied to the programming level. - Identification of crosscutting concerns just characterized by code duplication.
	Clone Detection Techniques: AST based		X	-	-	X	-	X	X	-		ASML-C-Code (20KLOC)	X		
	Fan-in analysis	2.2.2.7	X	-	-	X	X	-	X	-	Selection of candidates from list of methods, sorted on highest fan-in	JHotDraw (18KLOC) TomCat 5.5 API (172KLOC)	-	Concern is implemented in separated method which is called a high number of times, or many methods implementing the concern call the same method	- The set of results after filtering by the threshold values is very big. - The analysis of the results is time-consuming.

Table 3. Comparison of the aspect mining approaches using several criteria

Using the criteria commented above, we may extract the following (some of them already pointed out in [107] and throughout the corresponding sections of this document).

A first observation is that most of the approaches used static analysis. This kind of analysis is useful when the user wants to apply the aspect mining approach to a system which is not still finished (or just compiled). In addition, the results provided by dynamic analysis are sometimes very large (the data provided by tracing execution is huge) and the user cannot deal with so much information. An ideal aspect mining technique could use both analyses, although the advantages observed by using both techniques are not really important.

Secondly, most of the approaches use method as the granularity level. This granularity level is suitable when the methods completely implement the crosscutting concern. However, there are some crosscutting concerns which are implemented by a pattern in the code, e.g, parameter checking. In these cases, the technique could not identify these crosscutting concerns.

Regarding to the utilization of scattering and tangling, most of the approaches just use scattering. This is contrary to our definition of crosscutting (presented in Chapter 3), which considers scattering and tangling as needed conditions to have crosscutting. In most of cases, the reason for not considering tangling is that it would need to have information about the different concerns in the system. However, this problem could be solved when information of artefacts of the software engineering process are available (e.g at earlier stages).

It can be also observed that AST-based clone detection technique is the only one that has an empirical validation. Empirical validation is really important for having an actual comparison. However, empirical validation has an intrinsic difficulty and it requires the intervention of many people (e.g. end user of the aspect mining technique or the programmers of the systems being analysed). In addition, it should consider the replication of the studies not only in industrial contexts but also with students in classrooms. The use of concern-metrics may help and drive this empirical validation as it is shown in Chapter 5.

In order to extract conclusions about scalability of the techniques, the larger systems where they are applied provide some information on this issue. However, it may not be ensured the scalability of the different techniques at least they are applied to several different systems.

The preconditions used by the techniques constrain the kind of crosscutting concern identified by them. For this reason, the combination of different aspect mining technique for a same system may considerably improve the results obtained (the kind of crosscutting concern not identified by a particular technique may be identified by others).

2.2.3. Early aspects discovery techniques

A requirement defines a property or capability that must be exhibited by a system in order to solve the business problem for which it was conceived [39]. Requirements are usually classified into two different categories: functional and non-functional requirements. While functional requirements are related to features that the system must provide, non-functional requirements describe quality or constrain of the system. In that sense, constrains limit the set of possible solutions to the system. Constrain may be of technical nature (e.g. bandwidth in a network application) or related to the problem domain (e.g. legislation or standards) [39].

Requirements of the system are often well localised in software artefacts that implement them, producing a well modularized system. However, in many situations, requirements are not well localized in a single entity or artefact used by traditional software paradigms. Whenever requirements are not effectively modularised in single entities, the quality of the system is drastically decreased since they have an important influence in the rest of

requirements. These requirements are know as aspectual requirements or, as they are called by the community, Early Aspects [65]. In the last years, several approaches have appeared to deal with crosscutting concerns at early stages of development. The term of Early Aspects was coined to denote to aspect-oriented approaches at requirements level but also at architectural level. In particular, Aspect Oriented Requirements Engineering (AORE) is a subset of the Requirements Engineering area that provides support for separating crosscutting functional and non-functional requirements during requirements stages. In addition, AORE techniques also deal with the identification and management of conflicts arising due to tangled representations of the crosscutting requirements [39].

In [39], a deep survey of Requirements Engineering approaches was presented. This study was conducted by a set of well-know researchers in the Early Aspects area. Moreover, the study is included within the tasks carried out by the AOSD Europe Network of Excellence [5]. In this survey, the authors introduced the main approaches from requirements engineering to design stages. The approaches were divided into non-aspect oriented (Non-AO) and aspect-oriented (AO) categories. In this section, some aspect-oriented requirements engineering approaches are described (the most related to the work presented here).

The AO approaches described in [39] are classified according to the general method, the artefacts and the process used. Then, the same classification is used in this section showing examples of each category:

- **Viewpoint-based group**. This group represents the approaches which use viewpoint techniques [77] to deal with requirements (e.g. AORE with Arcade [151]).
- **Goal-oriented group**. In this group, requirements are represented by goals techniques [118] (e.g. Aspects in Requirements Goals Models [140]).
- **Use cases-based group**. The approaches belonging to this group use UML use case diagrams [180] to represent the requirements of the system (e.g. AOSD with Use Cases [99]).
- **Multi-dimensional separation of concerns group**. These approaches use a particular advanced separation of concerns technique, multi-dimensional separation of concerns [171], to deal with crosscutting concerns at requirements level (this group is represented by Cosmos [169]).
- **AO Component-based group**. The approaches classified into this category combine Component-based Software Development (CBSD) and AOSD at the requirements level (e.g. AORE for CBSE [91]).
- In a final category, some approaches which use **Natural Language Processing techniques** to identify early aspects are considered. These approaches are mainly focused on the analysis of the requirement documents, without using a particular requirements notation (different from just text) for the identification (e.g. EA-Miner [155] and Theme/DOC [15] approaches).

All the approaches presented in this section are summarised in Table 5 (at the end of the section). Those readers that have a deep background on the early aspects topic may refer to this table if they wish.

2.2.3.1. AORE with Arcade.

Although this approach is actually called “aspect-oriented requirements engineering”, the authors in [39] called it AORE with Arcade in order to not confuse the approach with the AORE topic. AORE with Arcade was presented in [151] by Rashid et al. The approach focuses on the separation and composition of aspectual and non-aspectual requirements. The authors

proposed a general process to deal with aspectual requirements which could be instantiated by using any requirements engineering technique. In particular, while in [151] they used Viewpoints to provide a concrete instantiation of the process, an instantiation using UML use cases is shown in [8].

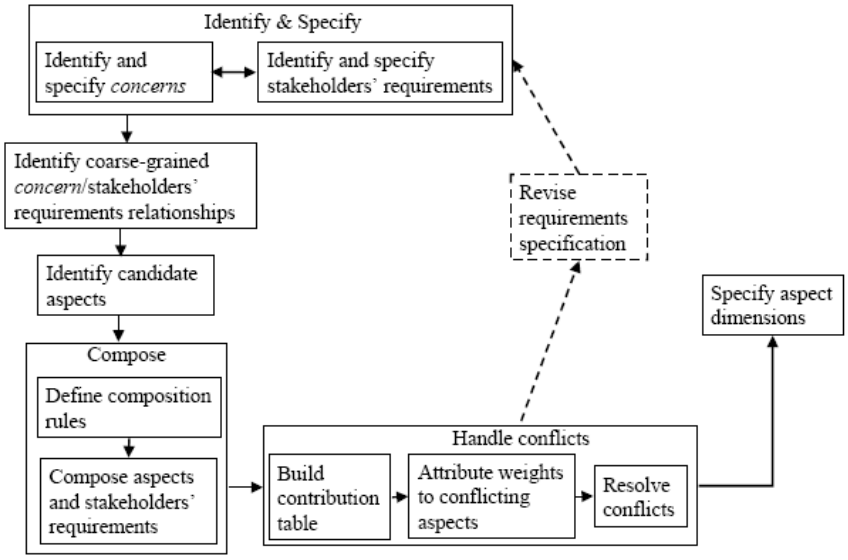


Figure 22. AORE general process (extracted from [39]).

The general process introduced in [151] consists of several steps that are outlined in Figure 22. As we can see in the figure, the process is firstly driven by the identification of concerns and stakeholders' requirements. Stakeholders' requirements are specified using any existing specific technique such as viewpoints [77], use cases [180], goals [118] or problem frames [97]. Once concerns and requirements are identified, they are represented in a matrix where the concerns which constrain to the requirements are marked. Concerns which are related to more than one stakeholders' requirement are considered as candidate aspects. Then, XML composition rules are defined where the authors specify the way that aspectual requirements influence or constrain to non-aspectual ones. Finally, these composition rules are used to compose the final requirements of the system and to identify possible conflicts between them. In particular, the authors build a contribution matrix where different relations between aspects and non-aspectual requirements are shown. When an aspect contributes positively to the non-aspectual requirement (e.g. adding some functionality needed to perform extra-actions), a "+" is shown in the cell. Negative contributions are represented by the "-" symbol (e.g. security or tracing usually contributes negatively to performance requirements). Using the contributions table, the authors add weights to the different aspects in order to solve the conflicts. In cases where the conflicts may not be solved by weights (aspects with equal weights), stakeholders must decide. In Figure 23 and Figure 24 we can see an example of the matrix built to relate concerns and requirements and the contribution matrix respectively.

SR / *Concerns*	SR_1	SR_2	…	SR_n
$Concern_1$				✓
$Concern_2$		✓		✓
…				
$Concern_n$	✓	✓		

Figure 23. Matrix used to relate concerns and stakeholders' requirements.

Aspects / Aspects	$Aspect_1$	$Aspect_2$	…	$Aspect_n$
$Aspect_1$		+		
$Aspect_2$				–
…				
$Aspect_n$				

Figure 24. Contribution matrix.

As it is aforementioned, in [151] the authors provided a concrete instantiation of the general AORE process using Viewpoints. In this instantiation, the main artefacts used by the

process are viewpoints and concerns. All these artefacts are represented using XML so that the compatibility with the composition rules defined is guaranteed. In Figure 25 examples of a viewpoint and a concern are represented – in a) and b), respectively -. These examples are extracted from a case study where the authors applied the approach. This example application is a Highways Toll System used to automatically charge the money to be paid to vehicles circulating through highways. The main concepts (in this case corresponding to physical devices) involved in the system are gizmo and ATM. Gizmo is the device installed into the vehicle to send/receive information from/to the ATM device which is installed in the toll gate.

```
<?xml version="1.0" ?>
  <Viewpoint name="ATM">
     <Requirement id="1">
       The ATM sends the customer's card number,
        account number and  gizmo
       identifier to the system for activation
       and reactivation.
       <Requirement id="1.1">
            The ATM is notified if the activation or
          reactivation was successful or not.
          <Requirement id="1.1.1">In case of
            unsuccessful activation or
            reactivation the ATM is
            notified of the reasons for failure.
        </Requirement>
       </Requirement>
    </Requirement>
  </Viewpoint>
```

(a)

```
<?xml version="1.0" ?>
  <Concern name="Compatibility">
    <Requirement id="1">
     The system must be compatible
     with systems used to:
       <Requirement id="1.1">activate
        and reactivate gizmos;
       </Requirement>
       <Requirement id="1.2">deal with
        infraction incidents;
       </Requirement>
       <Requirement id="1.3">charge
         for usage.
       </Requirement>
    </Requirement>
  </Concern>
```

(b)

Figure 25. Viewpoint and concern represented using XML.

The viewpoints and concerns are defined using the tool called ARCADE. This tool uses XML templates to define both the viewpoints (and concerns) and the composition rules. These XML templates are then stored in eXists, a native XML database system [133]. Then, ARCADE is also used to validate the composition rules and to compose the aspects with the viewpoints identifying the existing conflicts. A composition rule is shown in Figure 26. The composition rule uses a *Constrain* tag which is a key concept to perform the composition. This tag defines how the aspectual and non-aspectual elements are composed by specifying actions, operators and outcomes. In the example shown in Figure 26, the action *ensure* is used. This action is used to specify that a particular condition must be satisfied by the viewpoints. In this example, the viewpoint ATM must be compatible with the set of devices used as gizmo. The *with operator* defined in *Constrain* tag is used to specify that the condition must be satisfied by the two set of requirements with respect to each other (this is, ATM must be compatible with gizmo and vice versa). Finally, the *outcome* tag is used to describe the results expected from the composition. Note that the composition rules are defined at the granularity of individual requirements. In case the developer wants to relate an aspect with all the requirements belonging to a viewpoint, the *all* value must be specified in the *id* attribute of the corresponding *Requirement* tag.

```
<?xml version="1.0" ?>
  <Composition>
    <Requirement aspect="Compatibility" id="1.1">
      <Constraint action="ensure" operator="with">
        <Requirement viewpoint="ATM" id="all" />
      </Constraint>
      <Outcome action="fullfilled" />
    </Requirement>
  </Composition>
```

Figure 26. Composition rule using action *ensure* and operator *with*

Finally, the process provides information on traceability of the identified aspects to later stages. In particular, a set of dimensions are defined to specify whether the aspect should be mapped to a function, decision or an aspect in later stages. Moreover, the results obtained by

the framework are used as input for the PROBE framework [106]. This framework is responsible for adding links between aspectual requirements and their implementation at later stages. Then, the crosscutting concerns may be traced from requirements engineering to architecture and design.

The approach described in this section was one of the first approaches introducing the term of Early Aspect. It was also one of the first dealing with the identification, separation and composition of crosscutting concerns at the requirements level. Then, its contributions were really important. However, this approach had some limitations that could be solved to improve the results obtained. Firstly, the approach only deals with the identification and separation of non-functional crosscutting concerns. However, as it is claimed and demonstrated in [39], functional concerns may be also crosscutting concerns and candidates to be modularized using AO techniques. Secondly, the authors did not provide information showing how the matrix used to relate concerns and stakeholders' requirements is obtained. Then, it seems that the mappings between concerns and viewpoints are obtained by developers' expertise or intuition. This fact highly influences the results obtained. An automatic or semi-automatic way to obtain the mappings is mandatory in order to be able to apply the approach in complex systems. The approach presented in section 2.2.3.6 (EA-Miner [155], a tool supervised by one of the authors of the approach presented in this section) tries to solve these problems by using Natural Language Processing techniques to automate the identification of the aspectual requirements.

2.2.3.2. Aspects in Requirements Goals Models

Goal-based models support the description and analysis of intentions that underlie a new software system [140]. In particular, goal models usually represent functional and non-functional requirements by means of *goals* and *softgoals* artefacts, respectively. However, traditional goal models do not support the representation of crosscutting concerns so that new approaches have appeared to deal with these situations [140] [190]. Most of these works focus on the analysis of relations between *softgoals* and *goals* to identify those with a high fan–in which are considered as candidate aspects.

In [190], Yu et al. presented one of the first approaches to deal with crosscutting concerns in goal-based models, in particular in V-graphs. V-graph is an specific type of goal model where *goals* and *softgoals* are related resulting in a graph with overall shape of letter V. The two top vertices of the graph represent a *goal* and *softgoal* related by a correlation link. These links may be of several types (besides the traditional *AND* and *OR* used by goals techniques): *break, hurt, make* and *help*. The bottom vertice represents a task that must be carried out in order to satisfy the two goals related. In this kind of model *softgoal* usually represents a non-functional requirement. The *task* is related to the *goal* and *softgoal* by a contribution link, which may be of two different types: *satisfy (S)* or *deny (D)*. Therefore, the nodes in the graph are classified into two different categories: intentional (*goals* and *softgoals*) and operational (*tasks*). Moreover each node is labelled with a *type* and a *topic*. The *topic* contains contextual information whilst the *type* represents the generic functional or non-functional requirement related to the *goal* or *softgoal*, respectively. In Figure 27 an example of a generic V-graph may be observed. In this figure, the *goal Function* is correlated to a *softgoal*. They are also decomposed in several *sub-goals* and *sub-softgoals* respectively and these *sub-goals* and *sub-softgoals* are satisfied by means of carrying out several *tasks*. Note that *goals* are represented by octagons, *tasks* by hexagons and *softgoals* by clouds.

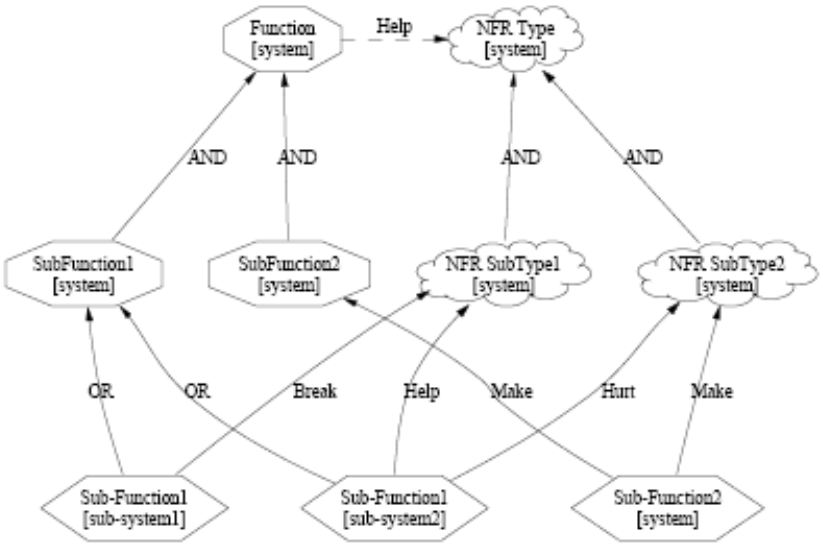


Figure 27. Example of a generic V-graph

Then, in [190] the authors propose a method to build V-graphs and to identify candidate aspects in these graphs. The process is iteratively applied and it consists of several procedures, being the most important: *Correlate, Decompose, ResolveConflict, CorrelationDecompose* and *ListAspects*. In order to illustrate the process of applying these procedures, in [190] the authors used a case study, a Media Shop, where they applied the whole process. From Figure 28 to Figure 31 we can see the results of the main procedures.

The *Correlate* procedure is responsible for adding the initial relationships (correlations) between root *goals* and *softgoals*. Since a *goal* or a *softgoal* could be decomposed into several *sub-goals* or *sub-softgoals*, the children propagate their contributions up to the parent goal satisfaction [190] (remember that the process is iterative). As result of the propragation, the parent must be fully satisfied as they were at the beginning of the process. A goal fully satisfied is denoted by a goal with the label [S=1.000, D=0.000]. That means that the goal is satisfied at 100% and not denied. In Figure 28 we can see the V-graph for the Media Shop after applying the correlate procedure.

The *Decompose* procedure is used to add new *sub-goals* or *sub-softgoals* to the V-graph. Thus, the V-graph is refined in the iterations by adding these *sub-goals*. If a *sub-goal* provides a negative contribution to the parent, the procedure *ResolveConflict* is performed which basically must remove the contribution link between the *sub-goal* and the parent. In Figure 29 we can see how there is a *softgoal* (Responsiveness [Transaction]) that is partially satisfied. This situation represents a conflict. Then, a different decomposition must be provided. A different conflict could occur when a *softgoal* is correlated to different goals by opposite correlations links, e.g. *make* and *hurt*. As an example, in Figure 30 the Responsiveness *softgoal* has *make* and *hurts* correlations. This type of conflict is solved by the *CorrelationDecompose* procedure.

Finally, the *ListAspects* procedure is used to identify the candidate aspects. The identification process is driven by the number of correlation links between a *softgoal* and *goals*. In particular, when a *task t* has a contribution link to a *softgoal s* and it has more than one chain of contributions links $\{t \rightarrow, \ldots, f\}$ where *f* is a functional *goal*, we say that the *goal f* is crosscut by the *task t*. Then, the *tasks* contributing the *softgoal* are encapsulated in a separated entity called *goal aspect*. This entity is represented by a *goal* with three peripheries (see Figure 31).

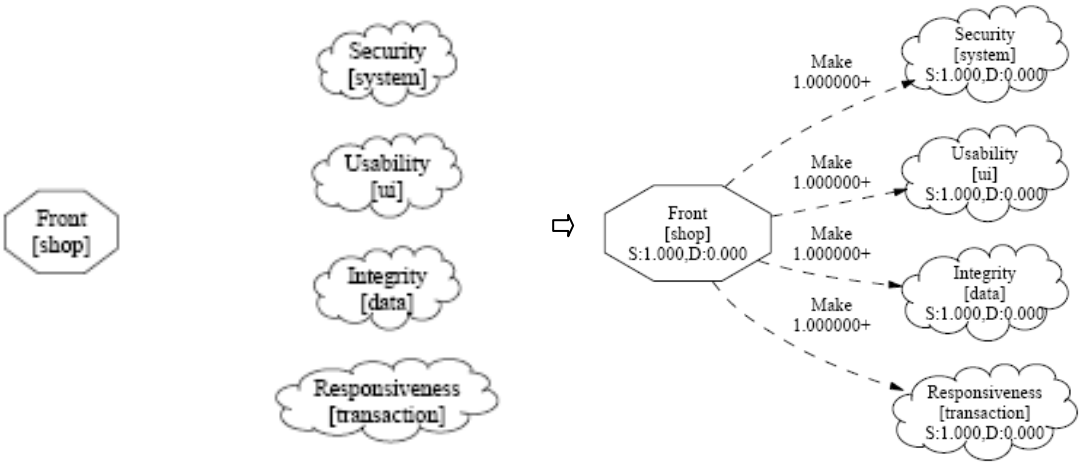


Figure 28. Result of the *Correlate* procedure

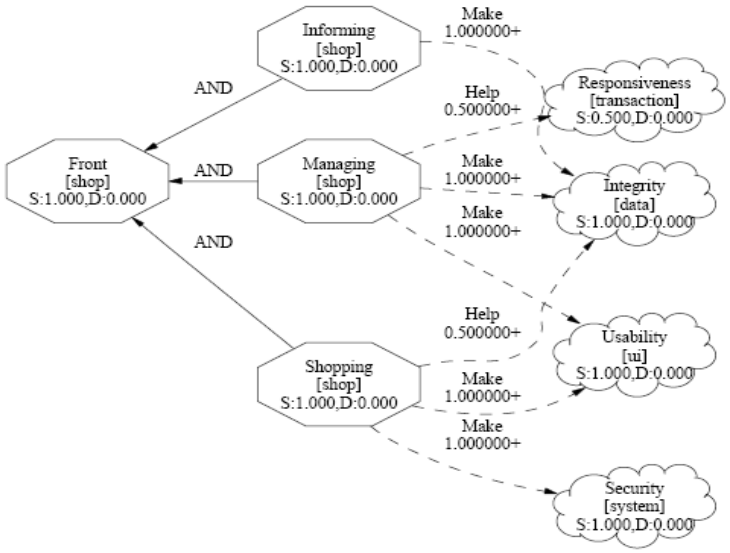


Figure 29. Detecting conflicts caused by a softgoal partially satisfied

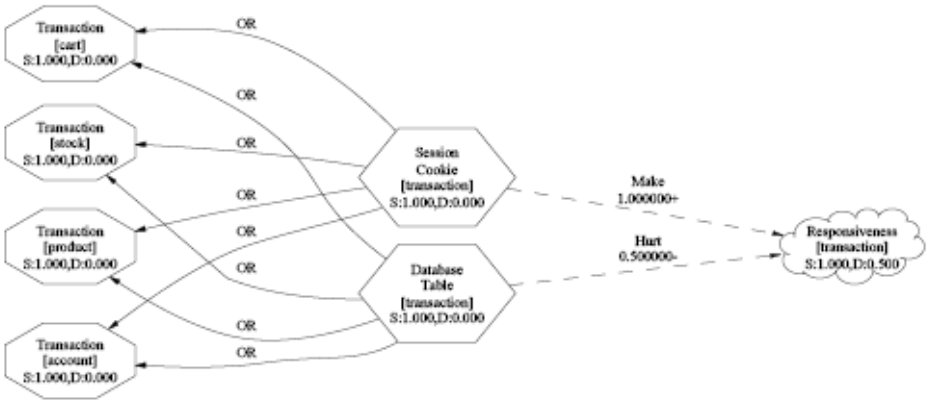


Figure 30. Detecting conflicts by contray correlation links

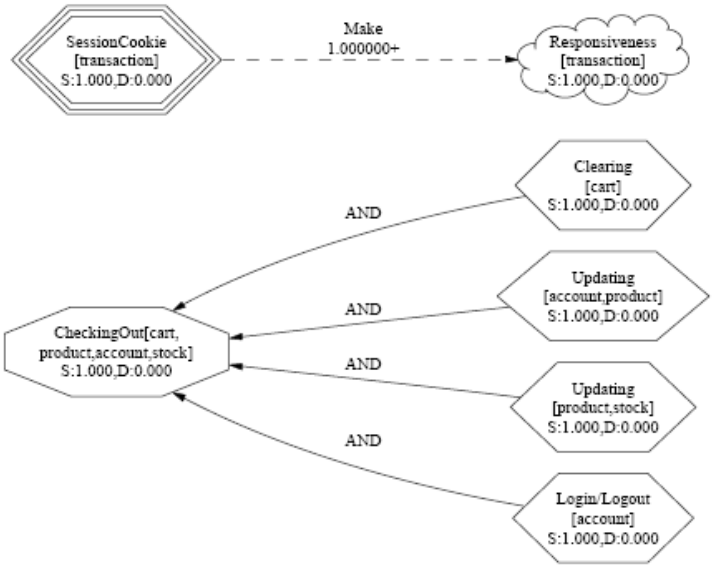


Figure 31. Encapsulating tasks related to the crosscutting softgoals into goal aspects

As it is aforementioned in this section, this approach is one of the first dealing with crosscutting concerns in goal models. It has important limitations that should be solved in order to be applied in real systems. First of all, as it is described in [190], most of the procedures used in the process must be manually performed. This should be really time-consuming and not feasible in complex systems. Secondly, the authors do not explain how the *goals* and *softgoals* are identified. Again, in complex system, the use of any semi-automatic tool to extract this information could help to reduce the effort needed to apply the approach. Analogously, the authors do not explain how the correlation and contributions links are established. However this decision could highly influence the results obtained by the process.

Finally, as it is explained in [39], there could be situations where the *decompose procedure* could fail in satisfying all the root goals conditions. For instance, in Figure 32 we can see several goals. In this model, the goal A3 contributes negatively to the goal A1. Then, the Decompose procedure would remove this goal causing that the goal A2 is not satisfied anymore since it requires A3 and A4. Obviously, goal A wouldn't be satisfied either.

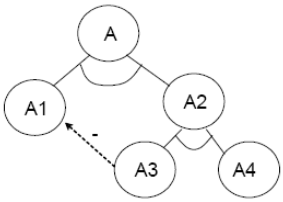


Figure 32. Example where Decompose procedure would fail (extracted from [39])

2.2.3.3. Aspect-Oriented Software Development with Use Cases

Jacobson and Ng have proposed a method to integrate aspect-orientation into Use Case Driven Developments [98][99]. Actually, what the authors claim is that Use Case Driven Development is an aspect-oriented technique since it provides extensions mechanisms that are similar to the extensibility achieved by AO. The approach was called Aspect-Oriented Software Development with Use Cases.

In AOSD with Use Cases, the authors assume that a use case represents the functionality of a particular concern. Since they use a Use Case Driven Development, use cases are used from the requirements stages to testing phases. In particular, the process is iteratively applied and

in each iteration of the software lifecycle, developers must go through the following sequence of activities [99]:

1. find the use cases and specify each use case
2. design each use case
3. design and implement each component, and finally
4. test each use case

Then, the authors realized that the realization of the different use cases were usually spread thorough the implementation components so that they were not well localized in simple components. Therefore, the authors claim that the use cases are usually crosscutting concerns since the concerns being addressed by the use case is often scattered and tangled. Thus, the process introduced in [99] consists of the separation of crosscutting use cases in well-encapsulated entities and the use of *extend* relationships to compose these crosscutting use cases with the base ones. Then, the crosscutting use cases could keep separated from the crosscut artefacts during the whole development process, until implementation phases where AO languages could be used to implement the crosscutting use cases.

In order to achieve the separation between crosscutting and not crosscutting elements, the authors adopted an implementation-language-like view on the approach. In particular, they incorporated AspectJ [10] constructs (namely jointpoints and pointcuts) and HyperJ-type [95] decomposition modules (slices) [39].

The authors distinguish in the approach between two kinds of use cases: *peer* and *extension*. Peer use cases are those which are independent of each other. That implies that these use cases may be used separately without any reference to the rest of use cases. Ideally, when these use cases are composed, their operations are composed without intervention in their execution. Extension use cases are used to implement additional features that are added to base use cases. These extensions may be defined independently of the base cases, however their utilization do not make sense without the base cases. The composition of the extension and base use cases usually implies that the operations related to the base use cases are interfered by the extensions added. The authors also suggest the utilization of a special type of use case to encapsulate functionality related to non-functional requirements. This special use case is called *infrastructure use cases*.

They also introduce MDSOC concepts like use case slice and use case module. A use case slice contains all the artefacts regarding to the realization of a use case in a particular development phase. In Figure 33 an example of use case slice at the requirements level is represented. The use case module contains all the information of a use case throughout all the development lifecycle. Then, the use case module contains all the artefacts related to the use case independently of the development phase. Note that a use case module will contain several use case slices. In Figure 34 an example of use case module is also represented.

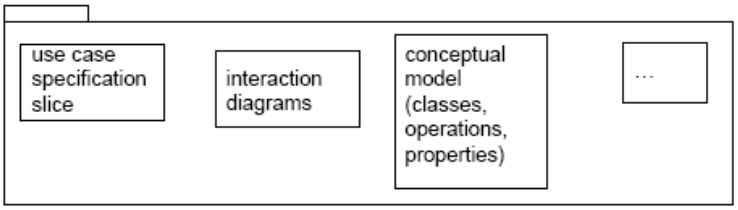


Figure 33. Representation of a use case slice at the requirements level (extracted from [39])

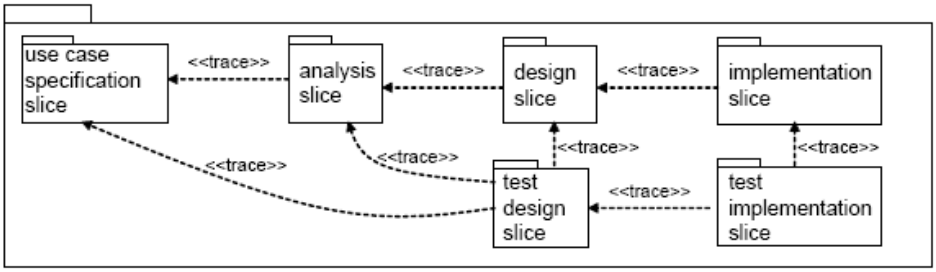


Figure 34. An example of use case module (extracted from [39]).

The approach incorporates the jointpoint and pointcut mechanisms by using extension points artefact defined in the UML use case diagrams. Then, in this case an extension point not only represents a point where the control flow of a use case is extended but it is also used to identify these execution points at later phases in order to achieve software composition. The definition of the composition between base and extension use cases is performed by defining the *pointcuts* and *advice* constructs within the use case templates or classifiers. Note that the authors use the classifiers instead of the ellipse graphical representation since the classifier provides more information of the control flows performed by the use case. In Figure 35 an example of an extension definition may be observed.

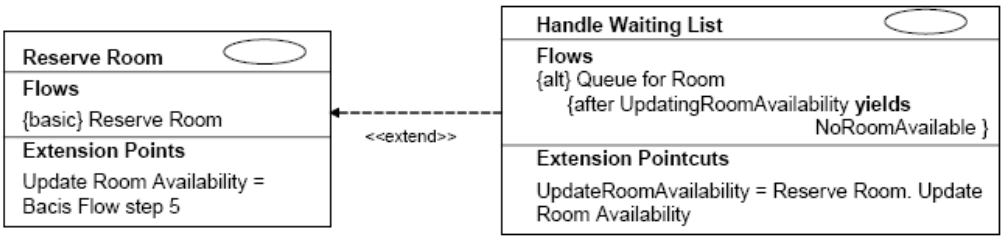


Figure 35. Base use case extended by using an *extension use case* (also extracted from [39])

We can see in Figure 35 how the *Handle Waiting List* use case extends the behaviour of the *Reserve Room* base use case. Note that the UML standard *<<extend>>* relationships are used to represent these kind of extensions. The authors also introduced new tags in the use case description. For instance, the tags *{basic}, {sub}* and *{alt}* are used to express different options in the control flows. *{basic}* tag represents a flow that must be triggered by an actor, *{sub}* tag denotes a flow that can be only included or referenced by another flow and, finally, *{alt}* tag is define to represent extension behaviour. Then, the extended use case may be referenced by the extender use case using the flows defined in the Extension Points section defined in the use case description. Moreover, the definition of the extension behaviour may include conditions to express whether the behaviour should be added *after, before* or *around* the extended behaviour. Note also that the behaviour defined by the base use case is independent of the extension behaviour added later on (ensuring the AOSD obliviousness principle).

One of the main advantages of the utilization of a use case driven development in this approach is that the system is sliced into the different use cases and they keep separated through the whole development lifecycle. Then, traceability of both base and crosscutting concerns may be improved since the use cases drive the whole development. In addition, the utilization of the use case module artefact allows the encapsulation of all the information (from requirements to implementation and testing phases) related to a particular concern in a single entity.

However, this approach still has limitations that could compromise its utilization. As an example, the authors assume in the approach that a use case is an aspect that crosscut many implementation components. From the Early Aspects perspective, this assumption is difficult to assume since a crosscutting concern at the implementation level must not be necessary a

crosscutting concern at the requirements level. In other words, taking the equivalence between concern and use cases made by the authors, such concerns will be crosscutting concerns at the design level, but this might not be sufficient to be a crosscutting concern at the requirements level (as it is argued in [39]). On the other hand, the authors presented a way to model and compose crosscutting concerns in use case diagrams, however, the approach does not deal with the identification of such crosscutting concerns. Then, either it is assumed that the crosscutting concerns have been already identified in a previous step or the authors just consider as crosscutting concerns the extension use cases. However, on one hand the task to previously identify the crosscutting concerns is not always simple and obvious. On the other hand, the definition of an extension use case as a candidate crosscutting concern would really depend on the concrete language used to model the concerns (at the concrete development phase). As at it is aforementioned, the crosscutting concerns at the requirements level could be different to those identified at the programming level.

2.2.3.4. Concern modelling with COSMOS

Sutton and Rouvellou presented a concern modelling schema (called COSMOS) that allows to structure concerns of a system independently of the software artefacts used to implement them in any particular development phase [167][168][169]. The authors claim that concern modelling at requirements level has important benefits for the requirement engineer that should be considered as [168]: *it leads to better understanding* and *more systematic treatment of concerns from the first stages of development*; *it may help to identify reusable elements during requirements analysis*; *it facilitates the formulation of alternative views and representations of requirements*; and *it supports compositional technologies for requirements*. In addition, the utilization of concern modelling at requirements allows the systematic use of concern modelling and aspect-oriented techniques at later stages of development.

In COSMOS a concern is any matter of interest in a software system. Concerns are considered as first class entities. The authors assume that concerns arise at all stages of development throughout the software lifecycle. Nevertheless, requirements modelling and analyses stages are particular important for concerns since they are usually a source of concerns and the concerns identified at these stages motivate, guide, and constrain much of the rest of development. Thus, a single concern considered in COSMOS can spam multiple phases of the lifecycle, be related to multiple types of artefacts and affects the different phases and artefacts in different ways [168]. Even, a concern may change over the time with the corresponding change in the software artefacts related to this concern. Similarly to concerns, aspects in COSMOS are considered as representations of concerns that also occur during the whole lifecycle and they may be also related to different artefacts (depending on the development phase).

The concern modelling schema defined in Cosmos comprises of three different types of elements: *concerns, relationships* and *predicates.* Concerns are also divided into two different categories: logical and physical. Logical concerns are related to concepts entities of the systems, e.g. issues, problems or "ilities"[4]. Physical concerns refer to the actual things that compose our software, such as software units, classes, services or hardware units. Logical concerns are classified into 5 different categories as well: *Classifications, Classes, Instances, Properties* and *Types*. *Classifications* are for modelling systems of classes and allow for multiple alternative classifications of concerns. *Classes* allow the developer to categorise the concerns.

[4] *Ilities* refers to typical quality attributes of software systems, e.g. usability, maintainability, scalability and so on. A more detail explanation of these and other software quality attributes may be observed in [96]

Instances represent concrete concerns. *Properties* are characteristics of the software (mainly related to "ilities"). Finally, *topics* are a way to group the *classes*, *instances* and *properties* related to a same theme, e.g. "error handling".

Relationships are used to represent how the concerns interact to each other and they are also divided into different categories: *categorical, interpretive, physical* and *mapping*. *Categorical* relationships are used to link elements semantically related (e.g. a concern and a subconcern are linked by a generalisation relationship). *Interpretive* relationships allow to link concerns for which the user has added a semantic relationship (e.g. when a concern contributes to the realization of another one, they are related by a contribution relationship). Finally, *physical* relationships are used just for relating physical concerns, whilst *mappings* represent relationships between logical and physical concerns (e.g. the implementation of a logical function by a Java class).

Predicates are used to establish integrity conditions over relationships and are classified according to the relationships for which they are defined. An example of predicate could state that a concern cannot be both a class and an instance. All the artefacts used by the approach were defined in a metamodel for application level concerns so that classes and instances of application concerns are defined.

The COSMOS process is driven by an analysis of the requirements documents in order to identify the concerns involved in the system and obtain a taxonomy according to the categories mentioned above. COSMOS is a MDSOC-like approach so that there are not concerns more important than other ones (all are equally treated) [171]. Then, the main goal of the approach is to allow a decomposition of the system into the concerns identified and a later composition using different criteria. Since multiple compositions of the concerns are possible, different versions of the system could be built (in a software product lines-like way). Moreover, by relating concerns with software artefacts that realise them (at any abstraction or development level), traceability is really improved and change impact may be easily assessed.

Finally, the authors suggest that concerns shared by different use cases or requirements statement should be considered as candidate aspects. Even, concerns reused in different parts of the same use case could be considered as crosscutting concerns. Nevertheless, since the concern model used in COSMOS is multidimensional, different dimensions and compositions of the concerns are possible. This fact, together with the possibility for using particular implementation languages or technologies, could solve the problem of crosscutting concerns.

In Figure 36, a part of the concern modelling schema for an example application is shown. This model belongs to the application of the approach to a warehouse management system which contains an initial requirements document (one single page) and two use case descriptions (of less than two and one pages respectively).

In this example application, the approach identified over 300 concerns. Considering that the size of the system was really small, the application of the approach to more complex systems seems unfeasible. Note that the results obtained in real systems could be so large that the developer could not manage them. The authors also realized this problem and argued that the problem of having so many concerns identified arises from two different factors: the nature of the requirement documents and the nature of the concern modelling schema and method. The former refers to the fact that any thing of interest showed in the document must be also a concern for the system. In other words, *if it's in the document, it's probably a concern* [168]. Taking into account this assumption, the authors claim that they identified one concern per 6 to 8 words in the documents (resulting in a so high number of concerns). The latter is related to the capability of COSMOS for modelling extensive concern models. As an example, it supports the modelling of instances of concerns but also of classes and classifications of the

instances. Moreover, since COSMOS is multidimensional, it supports multiple classifications of the same instances, increasing, thus, the final number of concerns identified.

Classifications

- Of technologies
  - By current vs. solution
  - By hardware vs. software vs. combination
  - By domain: communications vs. IT vs.inventory control
- Of solution technologies
  - By means of acquisition: retain versus develop
  - versus purchase
  - By mandated vs. optional
- Of roles
  - By company
  - By technologies used

Properties

- Decentralization
- Reachability
- Synchronization
- Parallelism
- Distribution (physical)
- Efficiency
- Continuity
- Separation
- Growth

Classes with subclasses

- Activities
  - Company activities
    - Warehouse management activities
    - Storing items activities
    - Redistributing items activities
    - Managing items activities
    - Customer activities
  - User -role activities [elaborated in use cases]
- Services
  - Warehousing services
    - Storage services
    - Redistribution services
- Business goals
  - Provide services goals
  - Grow business goals
  - Acquire supporting technology system goals
- ACME customers
  - Companies needing storage before shipping
  - Companies needing storage without offices

Instances under selected classes

- ACME system-user roles
  - Foreman role
  - Warehouse worker role
  - Truck driver role
  - Forklift operator role
  - Office personnel role
- Customer system-user roles
  - Customer role

Figure 36. Part of the concerns model for the warehouse management system presented in [168]

Note also that COSMOS is mainly focused on the classification and modelling of concerns but it does not deal with the identification process. The only suggestion about aspects made in the approach is that those reused in several documents could be considered as crosscutting concerns. However, as it has been previously mentioned, the identification of (crosscutting) concerns is not always simple. Therefore, the utilization of semiautomatic tools becomes mandatory to apply these approaches in real systems.

2.2.3.5. Aspect-Oriented Requirements Engineering for Component Based Systems

Grundy presented an approach that introduces aspect-oriented techniques at Component Based Software Development [90][91]. Although the approach was initially called Aspect-Oriented Requirements Engineering for Component Based Systems (AOREC), it was extended to cover all the phases of the development process (from requirements to implementation) [91]. Then, a whole methodology was presented where components and aspects are treated as first entities classes from the very beginning to the end of the software development

process. Nevertheless, in this section the AOREC part of the approach is considered and it focuses on the concepts related to the requirements engineering.

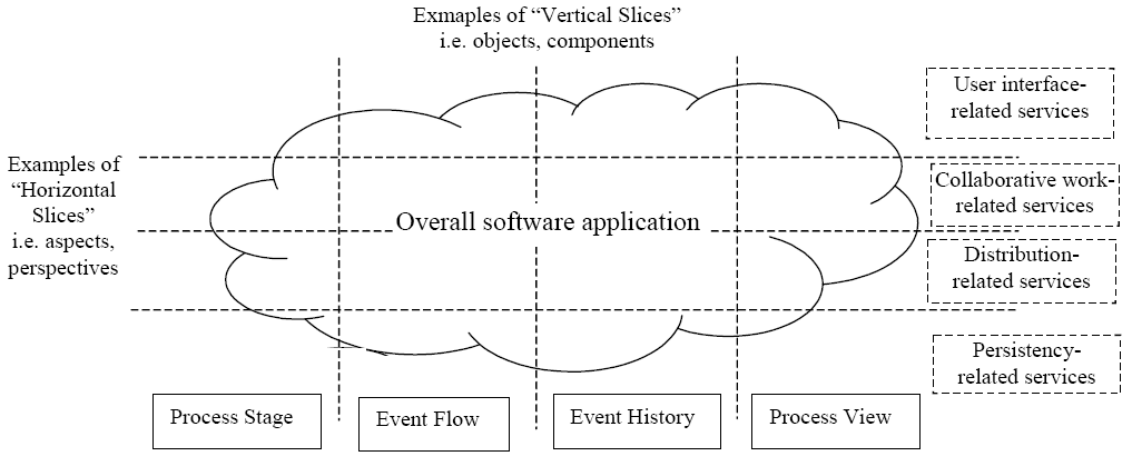


Figure 37. Vertical vs. horizontal slices (extracted from [91])

Basically, Grundy proposed an extension to Component Systems allowing the introduction of "horizontal slices". He claimed that traditional component based developments were only able to design and implement systems taking "vertical slices" of the system functionality, braking systems into services grouped by data and operations on these data [91]. Then, these traditional approaches were not able to capture, reason about and encode higher level concepts that are usually spread all over the vertical slices. These higher level concepts refers to crosscutting concerns (most of times) related to non-functional requirements or services often present in domain specific applications. Most concrete, an aspect (a crosscutting concern) in AOREC is a characteristic of a system for which components provide or require services. As it is mentioned in [91], these aspects are "horizontal slices" of system's functionality and non-functional constraints, and include user interfaces, collaborative work facilities, persistency and distribution management, and security services. A graphical representation of the two different dimensions (vertical and horizontal) involved in the component based development proposed by Grundy is presented in Figure 37.

Aspect	Aspect Detail	Description
User interface	Views Affordances Feedback Extensible parts	Components supporting or requiring user interfaces, including extensible & composable interfaces for several components
Persistency	Store/retrieve data Locate data Lock data	Components supporting or requiring data persistency management facilities
Distribution	Object Identification Method Invocation Transactions Event propagation Concurrency control	Components supporting or requiring distributed object management facilities

Table 4. Aspect details for the aspects User Interface, Persistency and Distribution

Aspects in AOREC are also characterized by a set of aspect details. Aspect details are used to more precisely describe component characteristics related to an aspect. These aspect details are used as provided and required operations in order to bind components with aspects and vice versa. The process to establish the relations between components and aspects is described later on. In Table 4 some examples of aspect details for several aspects are presented. The approach also considers the grouping of interrelated components into a new

entity called *aggregate aspect*. This entity allows global or system wide requirements to be captured.

The process introduced by Grundy starts with the analysis and study of the requirement documents. In this analysis the developer must identify the main components that comprise the system. The requirements for the identified components are then elaborated and they are used to identify the component aspects for each component. The aspect details for each component aspect must be also identified. The aspects are, thus, refined until the aspect details are obtained. The aspects identified are used now to establish the components composition and configuration (the links between required and provided interfaces). In case some aspects or components are interrelated (e.g. they belong to the same theme) they are grouped into aggregation aspects that are also considered as components by the approach. Finally, once the components, aspects and aggregate aspects are identified and composed, the components model is verified according to the system requirements in order to ensure that all requirements are met. In Figure 38 an outline with the main steps of the process may be observed.

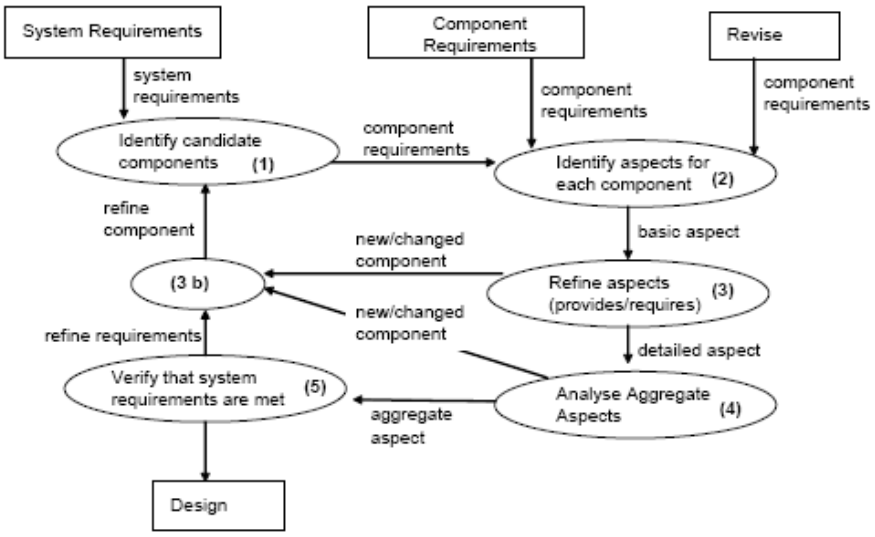


Figure 38. AOREC process (extracted from [39])

In order to establish the relations between the required and provided interfaces of aspects and components, a textual representation of the components is also used. A textual specification language is used to define components, aspects, aggregate aspects, aspect details and details properties. These properties are characteristics of aspect details used to more formally define the relationships between components and aspects. An example of these properties is (NUM_PER_SECOND >=3) which expresses that a sender (in an event based notification system) must be able to propagate, at least, 3 messages per second. This last property, among others, may be observed in Figure 39. This figure represents the composition (in the textual specification language) for a component and an aspect.

The approach presented by Grundy was one of the first approaches dealing with aspect orientation throughout the whole development lifecycle (from requirements to implementation). This is one of the key contributions of the approach. Even better, since the whole development is considered, a high level of traceability between the different artefacts may be easily achieved. Moreover, the introduction of aspect-oriented techniques into component based developments allows to merge the benefits obtained by the two paradigms by separately.

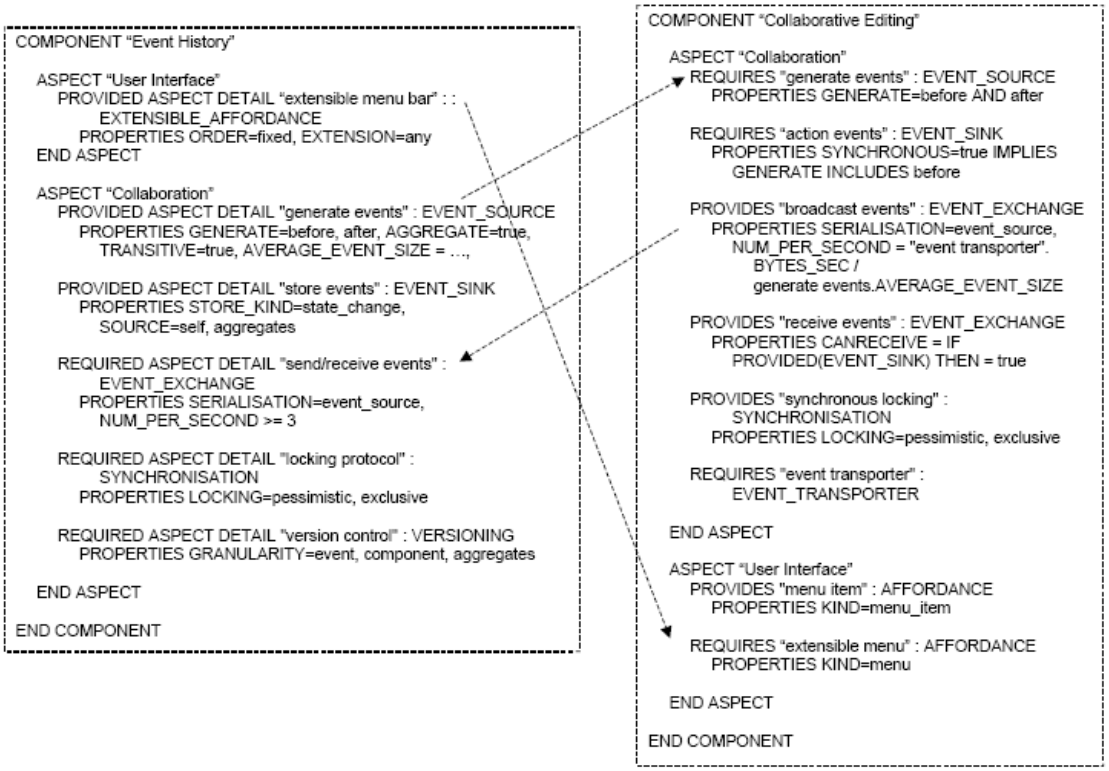


Figure 39. Textual representation of component Event History and aspect Collaboration in a software process management tool called Serendipity-II (extracted from [91])

However, the approach has some limitations that should be solved in order to improve the process. First of all, aspects are defined just to implement non-functional concerns behaviour. It has been widely assumed by the community (e.g. [39]) that functional concerns could be also considered as candidate aspects. In addition, the approach lacks a methodological process for identifying candidate aspects and it is lead to developers' expertise in the specific domain application. Then the approach could properly work with well-know domain specific crosscutting concerns, but it would fail in the identification and modelling of not so known crosscutting concerns.

2.2.3.6. EA-Miner

The EA-Miner tool, developed by Sampaio et al., was initially presented in [156] and it has been largely extended in [155] and [158] later on. The tool is based on the application of Natural Language Processing (NLP) techniques to different types of requirement documents in order to identify key concerns or early aspects. The tool is semi-automatically applied so that, as authors claim, the time and effort spent in understanding a system is drastically reduced. The approach relies on the use of not only lexical-based analysis but also on the utilization of semantic information. The tool takes as input the requirement documents of the system and is able to produce different kinds of requirement models based on the concepts identified (by the NLP techniques). As a result, the tool allows the identification and structuring of the model abstractions of different requirements models such as use cases or viewpoints.

More precisely, EA-Miner relies on the WMATRIX NLP tool [161]. WMATRIX tool implements different types of NLP analyses varying from syntactical to semantical. Examples of these analyses are frequency analysis, part-of-speech and semantic tagging. Frequency analysis presents statistical data of the usage of the different words in the requirement documents. These data are used to establish what words are the most relevant in the text. Part-of-speech tagging is a syntactical analysis to obtain the grammatical function of each

word in a text, e.g. noun, infinitive verb, comparative adjective, etc. Semantic tagging consists of a semantical analysis where the words of a text are analysed in order to obtain their meaning. Then, the analysis groups words in categories in terms of their meaning relations. The semantic tagger uses a large dictionary of over 73984 words and multiwords. EA-Miner utilizes the WMATRIX tool to pre-process the requirement documents in order to tag the words identified in these documents. These tags are used later by EA-Miner to identify concerns and early aspects. As an example of the identification process, viewpoints are identified among words with "*noun*" tag or use cases are obtained from verbs in the text.

The early aspects identification process performed by EA-Miner consists of several steps and activities. These steps are represented in Figure 40. This process is summarised as follows (the steps will be more detailed explained by their activities later on):

1) **Elicit requirements from stakeholders and documents**. In this task the main requirements of the system must be elicited. It may be carried out by interacting with stakeholders or by reading available documents.

2) **Identification of requirements model concepts**. This task is automated by EA-Miner since it takes the documents obtained in step 1) and performs the NLP analysis to identify model concepts (including early aspects). The relationships between the model concepts are also automatically obtained.

3) **Structuring the requirement documents**. In this step, the results obtained by the previous step are filtered by removing, adding or editing the concepts identified.

4) **Validating requirements and resolving conflicts.** The output of step 3 may be used by existing tools (e.g. ARCADE [150]) to validate the consistency of the requirements model built.

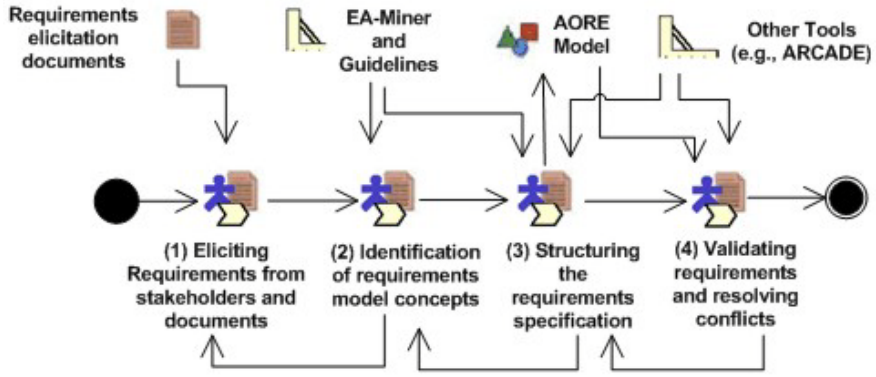


Figure 40. Outline of the different steps performed by EA-Miner

Now that the main steps performed by the tool are know, more detailed activities are described in order to show how EA-Miner works. The activities explained are graphically represented in Figure 41. These activities are explained assuming that a viewpoint requirements model has been selected.

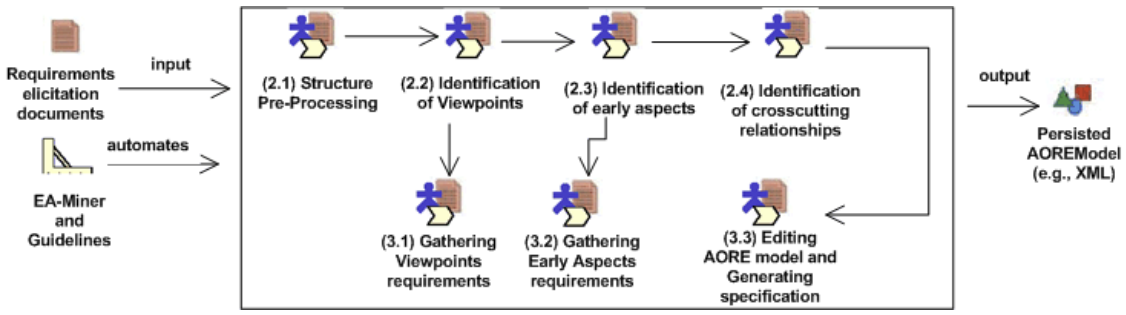


Figure 41. More detailed activities performed by EA-Miner

- **Structure pre-processing (2.1)**. The first activity that the tool carries out is a pre-processing of the requirements documents. This pre-processing is mainly focused on defining the minimal unit that the parser that analyses the document will used. This minimal unit is called a parser unit. By default, the minimal unit used by WMATRIX is a sentence so that EA-Miner also uses sentence as the default minimal unit. This minimal unit will represent a requirement in the system.
- **Viewpoint identification, presentation and structuring (2.2 and 3.1).** In this activity, the tool parsers the requirement documents using the minimal unit defined in previous step. In particular, if sentence is selected as the minimal unit, each sentence in the requirements is tagged with <s> and </s>. The words contained by the sentence are also tagged with syntactical and semantical information (<w pos="value" sem="value"> word </w>). The analysis then (i) identifies each sentence as a requirement; (ii) identifies the viewpoints and early aspects between the words contained in the sentence; (ii) the collection of requirements where a viewpoint or early aspect appears are related to this viewpoint or early aspect. The requirements associated to the different concepts are used to identify the relationships between viewpoints and early aspects based on the requirements shared by more than one concept. The rule used to identify viewpoints is based on the presence on nouns (identified by Part-of-speech). The rule used to identify early aspects is explained below. The different concepts identified may be structured by the developer adding, removing or editing them.
- **Early aspects identification, presentation and structuring (2.3 and 3.2).** The rule used to identify the non-functional early aspects is based on the utilization of the semantic tag provided by the semantic tagging task (performed by WMATRIX). The algorithm used consists of the comparison of the semantical value associated to a word with a catalogue of non-functional concerns where similar words are defined with their associated semantical values. If a match is found, the word is identified as a candidate aspect and the requirement sentence where the word appears is associated to the early aspect. Since functional concerns could be also considered as candidate crosscutting concerns, the tool also uses a fan-in analysis (that explained in [128]) to obtain the more frequently used words and to classify them as candidate aspects.
- **Crosscutting relationship identification (2.4).** In this activity the relationships between viewpoints and the early aspects are established. These relationships are based on the requirements statements shared by these concepts. In that sense, as it is explained in [158], the requirements statements act as join points where the early aspects and the viewpoints intersect. In fact, intersection operations are defined to compare collection of requirements.
- **Editing the AORE model and specification generation (3.3).** After the model is built, filtering and editing features can be applied. The user has the possibility of removing, adding or editing any identified concept. In addition, the user may apply different filters that help to reduce the results obtained avoiding false positives. Examples of these filters

are the utilization of frequency to obtain only the most significant words (used more than a threshold value), the utilization of a synonym list to avoid repeated concepts, or the utilization of a stemmer which groups words syntactically related (e.g. vehicle and vehicles) by considering the morpheme of the word.

The EA-Miner tool is implemented as an Eclipse plug-in. In Figure 42 and Figure 43 two screenshots of the tool are shown. While Figure 42 shows the view that presents the viewpoints identified (together with the related requirements), Figure 43 shows the view which presents the early aspects identified (also with the related requirements).

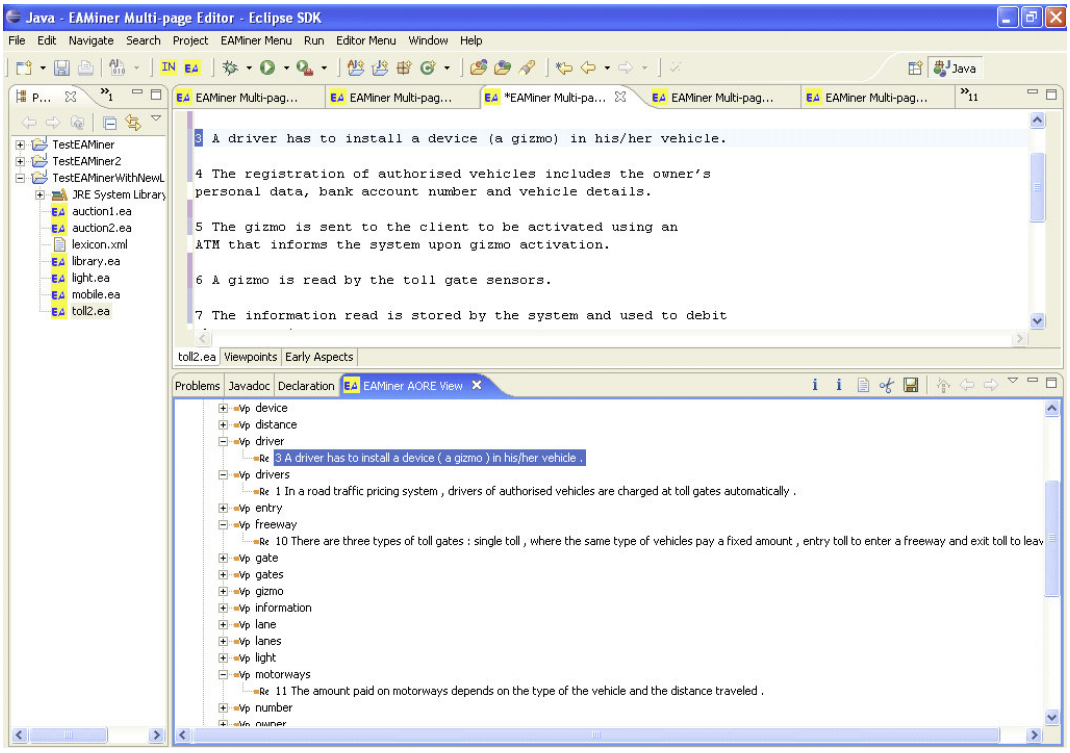


Figure 42. Viewpoints identified in a sample application

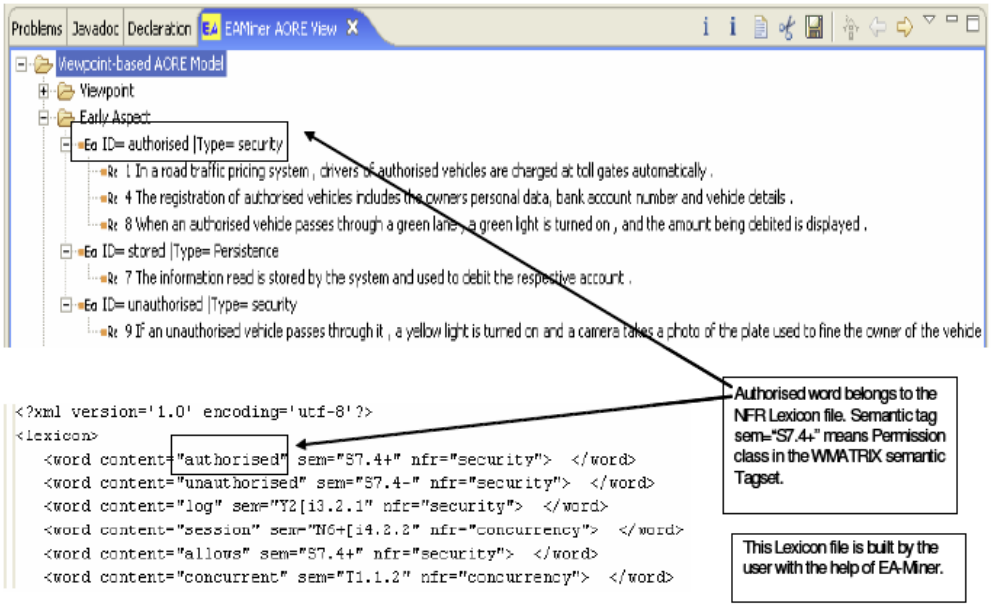


Figure 43. Early aspects identified in a sample application

The EA-Miner tool has important benefits and advantages for the software developer. Its main contribution is the automated process for identifying concepts in the very early

requirement stages. Authors also applied the tool to an industrial application (a Toll Gate system) showing promising results [157]. However, after analysing the tool and applying it to the same case study, we realized that the tool requires a big user involvement after detecting the candidate concepts. The initial number of candidate concepts is very much bigger than the results shown by the authors in [157]. Then, the time and effort needed by the developer to adapt the initial results obtained to the actual results is higher than the desired.

The authors also realized this problem and they described a set of guidelines to be used by the developer to reduce the time and effort needed to analyse the results. In particular, in order to identify the right viewpoints in a system the guidelines suggest: avoid viewpoints that are very general, check whether their semantic tags belong to the most frequent categories, sort the viewpoints by relative frequency, apply stemming and synonyms filtering. On the other hand, for the identification of early aspects, the authors suggest: edit the requirements if it is needed in order to avoid early aspects, expand the lexicon (catalogue) used by the tool whenever new words or new non-functional concerns are identified.

Moreover, the tool may be mapped to any requirements model in the sense that it is able to identify concepts of these models (depending on the selected one). However, the tool could not be applied to other requirement artefacts different from just text. Nevertheless one of the main application areas of aspect mining is the refactoring of legacy systems sometimes using different requirements models such as viewpoints or use cases. The tool could not be used in these systems. Finally, the identification of early aspects by using the requirements documents as input could lead to false positives or negatives since the existence of crosscutting concerns really depends on the decomposition capability of the selected language to model these requirements. Maybe an identified early aspect in text could be properly modelled using a particular requirements language removing the scattering and tangling. In Section 5.1.7 the results obtained by EA-Miner and compared with the obtained by the aspect mining approach presented in Section 5.1.

2.2.3.7. THEME/DOC

The Theme/DOC is the first part of a larger approach called Theme [14][15] and developed by Baniassad and Clarke. Theme is divided into two approaches: Theme/DOC and Theme/UML. While Theme/DOC is dedicated to the analysis of requirements in order to identify crosscutting concerns, Theme/UML is responsible for modelling these crosscutting concerns at design level using UML artefacts.

As it is explained in [14], the task of encapsulating behaviour into aspects needs a previous step where these aspects should be identified. Note that using intuition or domain knowledge is not sufficient to identify the potential set of aspects appearing in applications. While most of the approaches presented in this section are just focused on the modelling of early aspects, the last two ones presented give more importance to the identification phase and they are more focused on this task. Theme/DOC performs this identification by analysing the requirements documents searching for base and crosscutting themes. A theme is defined as an element of design: a collection of structures and behaviours that represent a feature. Then, it could be seen as a concern realization. Theme/DOC provides a process not only to identify the crosscutting features in the system but also to model the relationships between these features and the base ones. In addition, the resulting model facilitated by the approach is used to drive the design using Theme/UML.

The process used by Theme/DOC starts by a manual analysis of the requirement documents by the developer. This analysis provides a list of key actions identified in the text. These key *actions* are obtained by detecting sensible verbs appearing in the requirements. Then, the approach performs a lexical analysis to build an action view. An action view represents the relations between the actions and the requirement statements where they

appear. The action view is obtained using two inputs: a list of actions identified by the developer and the original requirement documents.

Each action identified in Theme/DOC is a potential theme. Ideally, each theme should be designed separately, however, they are usually scattered over several requirements and tangled with the functionality of other themes. Then, the authors divided the themes into two categories: base themes and crosscutting themes. Whilst the former is composed by the actions that are related to a single requirement, the latter contains the actions related to more than one requirement. The relations between actions and requirements are represented in the action view. In this view, each action is represented by a diamond and the requirements where it appears are represented by rounded boxes (sentence records). In Figure 44 we can see an example of an action view for a Course Management System (CMS, introduced in [14]). In this system the students may register/unregister for courses and professors may give marks and also unregister students. Observe that the rounded box related to logged and register actions (requirement R3, *when a student registers then it must be logged in their record*) has been expanded to see an example of requirement statement.

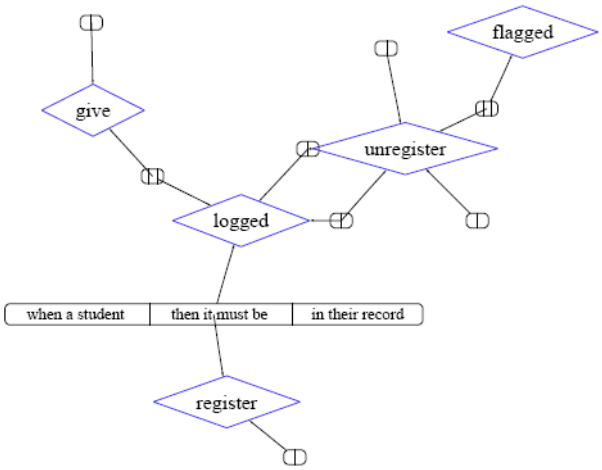


Figure 44. Action view for the CMS [14]

As it was expected (and observed in the action view), there are actions that share requirements and actions related to several requirements. The identification of crosscutting themes is driven by the analysis of the shared requirements. When a requirement is shared by two different actions, the developer must analyse what the key action realised in the requirement is. The final goal is to have an action view where each action is isolated with the requirements only related to it. This is achieved by using a clipped view. In the clipped view, each requirement statement should be just linked to the action more coupled with it. As an example, in Figure 44 we observed that the R3 requirement was shared by logged and register. The analysis of R3 concluded that this requirement was mainly implementing the logging functionality. Then it was related to logged action. Logged action was considered as a crosscutting theme and register as the base theme crosscut by logged. This analysis was repeated with all the actions and the clipped view shown in Figure 45 was built. In this view, the grey arrows represent the relations between crosscutting and base themes. These arrows have a dot in the crosscutting action and point to the crosscut action. In this figure we can still observe a requirement shared by two different actions: unregister and flagged. The authors claimed that after analysing this requirement, they concluded that flagged functionality could be included into unregister and this is why they share the requirement. However, the flagged action could have been also modelled as a crosscutting theme.

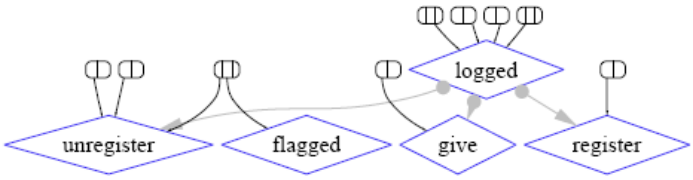


Figure 45. Clipped view for the CMS [14]

Once the clipped view has been obtained, the approach uses this view to plan the translation of the identified themes to the corresponding Theme/UML design model. An intermediate model is build to achieve this translation, the theme view. The theme view is an augmented version of action views since they do not only show requirements and actions but also the key elements of the system needed to define the different themes in Theme/UML. The theme view is also obtained by a syntactical analysis of the requirement documents and it also requires as input the list of key actions identified in the first step. Figure 46 shows the theme view for the register theme of the CMS system. The artefacts used in the clipped view are placed in the needed order to enable the reading of the requirements statements. For instance, in Figure 46 the requirement statement is read as *"student can register for course"*.

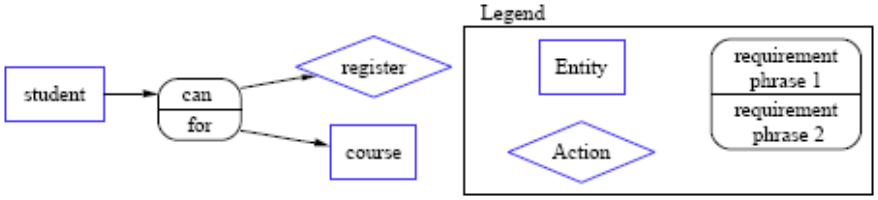


Figure 46. Theme view for register theme in the CMS

The theme view for a crosscutting theme also represents the elements crosscut by the theme, so that the view has a twofold goal: identify the elements that must be included into the Theme/UML theme, identify the relations with the crosscut themes (they will be included in the binding of the Theme/UML themes). Refer to [15] or [175] to obtain more details on how to build the Theme/UML themes. Figure 47 shows the theme view for the logged action.

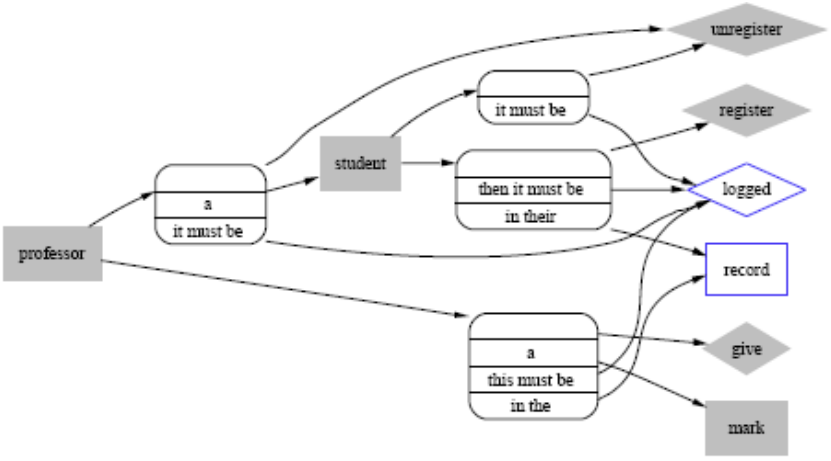


Figure 47. Theme view for the logged action in the CMS

Finally, once the Theme/UML themes have been designed, they are used to validate and verify whether the functionality represented by the design is aligned with the requirements. Then, the theme views are reviewed and they are usually completed with methods and associations. In particular, *"has"* (represented by an inverted grey arrow) and *"calls"* (represented by a dashed arrow) relationships are added. In this new view, methods are represented as actions and classes as entities, both with corners marked. The theme view

shown in Figure 46 for the register action has been augmented resulting in the theme view of Figure 48.

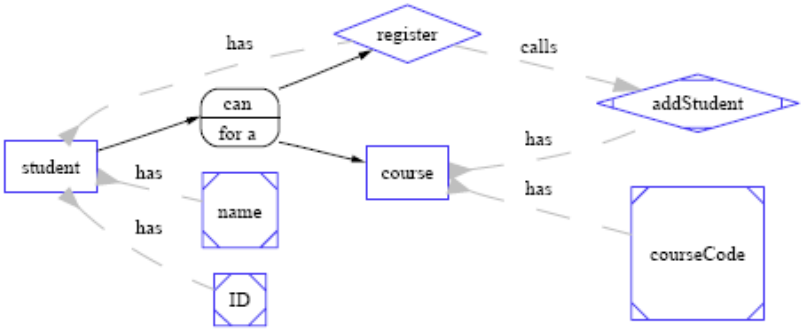


Figure 48. Augmented theme view after aligning the theme view with the Theme/UML design

Observe that traceability in this approach is achieved by relating the themes at requirements level (Theme/DOC) with those defined at the design level (Theme/UML). In that sense, the alignment task described above allows the identification of possible inconsistencies between both versions of the themes. Then, the whole approach ensures that the crosscutting concerns may keep separated from the early stages of development to the latest.

One limitation found in Theme/DOC is that it does not deal with non-functional concerns or non-functional requirements. The identification of these requirements is lead to the requirements engineer; however, as it is claimed in [39], sometimes this is the problem that the identification needs to address. The use of a catalogue of non-functional requirements may be really useful for improving the results obtained in that sense. Moreover, the approach focuses on the identification of crosscutting concerns at requirements statements level. This fact makes the approach very useful since the crosscutting concerns are identified before any software-based representation is built. However, an important application area of aspect mining approaches is the refactoring of legacy systems, which are usually described using other requirements artefacts, such as UML use cases or viewpoints. The Theme/DOC approach could not be applied to any deployment artefact different from just text.

In [113], the Theme/DOC approach has been also extended by using Latent Semantic Analysis (LSA) [119]. The authors stated that the lexical nature of the analysis performed by Theme/DOC makes that some aspectual requirements are missed or wrongly identified. Also, some requirements could be wrongly clustered if they contain ambiguous terms. Then, the authors proposed to complement the lexical analysis used by Theme/DOC with a LSA to improve the results obtained by the approach in those senses. In particular, LSA is used to cluster the requirements in order to group them according to similarities between what they called *corpus* (fragment of text). Note that LSA is an approach for determining similarity of terms or portions of text based on statistical analysis of large text corpora. Then, by using this technique, the authors claimed that the results obtained by Theme/DOC where in some cases improved. For instance, LSA approach falsely identified fewer aspectual requirements than lexical approach. However, lexical approach misses fewer aspectual requirements than LSA [113]. Then, the authors encourage the utilization of a combination of both approaches since they are complementary. A comparison of the results obtained by Theme/DOC and the approach presented in Section 5.1 is shown in Section 5.1.7.

2.2.3.8. Summarisying the early aspects discovery techniques

This section shows a comparison of the early aspects approaches presented in previous sections. The criteria used to compare the approaches were introduced in [39], where a deep survey of early aspects approaches was presented. The main criteria used to compare the

approaches are traceability through software lifecycle, composability, evolvability and scalabilty. These criteria are defined as:

- **Traceability** through software lifecycle. Traceability is the degree to which dependencies are made explicit between different artifacts such as stakeholder needs, requirements, design, system components, source code, etc. [149]. Traceability facilitates the understanding of software by relating the software artefacts with their pervious and next representations in an unbroken chain of requirement to code production process. Then, once an artefact is defined (e.g. a requirement) traceability helps to follow this artefact through different representations (e.g. design or code).
- **Composability** is the process performed to compose artefacts of a system. This concept allows the view and understanding of the complete set of artefacts of a system and to identify the interrelationships and conflicts between the artefacts. Composability is opposite to modularity. Note that modularity aims at decomposing a system into arbitrary kinds of modules; however, this decomposition is not valuable unless a later composition is possible.
- **Evolvability** is the capability of a system to accomplish changes in the requirements/architecture/design models. Evolvability is often translated to a change/addition/removal of existing artefacts in the models. The evolution of artefacts is usually due to a change in business rules or requirements of the systems but it may be also due to maintenance or improvements tasks. Evolvability is highly related to other software quality attributes defined by ISO 9126 [96] such as stability and changeability.
- **Scalability** is the ability of an approach to be applied equally to small and large projects. This characteristic is often reduced to tool support, however sometimes the tool support is not sufficient.

In [39] these general criteria are also refined into more sub-characteristics, such as trade-off analysis, decision support for mapping or verification and validation. However, since these sub-characteristics are somehow contained in the general terms defined above, here the comparison is only driven by the four general criteria defined.

Table 5 summarises all the early aspects approaches described in this section. A column with the main limitations of each approach has been also shown.

		Criteria				Limitations
		Traceability through software lifecycle	Composability	Evolvability	Scalability	
Approaches	AORE with Arcade	Each requirement is nested with the viewpoint that it originates from Records what type of architecture or design artefact a concern transforms to and it may be traced (by PROBE)	Supports requirements composability through a clear join point model and well defined composition rules and operators	A change in a concern only affects the evolving concern representation and the related composition specification for affected requirements	Utilization of XML for representation and composition makes the approach very scalable. Although the tables used would become very large.	- It just identifies non-functional concerns as crosscutting. - The matrix is manually obtained, difficulting its application in real systems.
	Aspects in RGM	Records the origin of sub-goals from softgoals but not record the source of softgoals themselves Relates softgoals with the functional goals. The operationalizations are linked to design decisions	Matchs the topics of the functional goals and the non-functional tasks that apply to them. The sysntax and semantic of composition rules require further research	Due to the link between functional and non-functional goals and tasks, the change in one can be clearly related to others, using composition specifications	Has limitations regarding to the size of the graphs built.	- Most of the procedures manually applied. - Correlation and contributions links are obtained by the developers' intuition.
	AOSD with UseCases	Records the origin of functional requirements on the use case diagram. Records non-functinoal requirements through *Perform Transaction* artefact Use case slices allow to preserve information on operations and states of objects	Supported by the addition of the new extension pointcut. Also specially desing level composition semantics are provided	The new infrastructure added for the non-functional use cases implies that a change in a use case requires to check whether it affects to non-functional use cases	Limited by the growing size of use case diagrams. The utilization of packages may solve the problem. The packages should contain a diagram that may be saw at glance	- Lacks of identification process. - Assumes thatcrosscutting concerns at requirements keep being crosscutting as design (not always true).
	Cosmos	Source of concerns is not deemed important Physical concerns may provide some information about traceability	Does not support composition althought the relationships between concerns may be used to infer how they should be treated together	Change in a concern just affects this concern and the related information (relationships, constrains,…). No other concerns are affected	Very scalable, an increase in the size just implies an increase in the number of modelled concerns	- The number of concerns identified is sometimes unmanageable. - Lacks of a method to identify the concerns classified.
	AOREC	It does not address origin of requirements Aspectual requirements are directly propagated to design level	Component requirements are composed through the aspects by used and required aspect details	Although it does not assist in aspectual evolution, aspect details contain the information to know the affected components	Graphical representations are not scalable. However aggregate aspect provides coarser granularity level relations to deal with large systems	- Only deals with non-functional behaviour. - Identification of crosscutting left to developers' expertise (it may fail in detecting not well-known crosscutting concerns).
	EA-Miner	Records the requirement statement that a viewpoint or aspect comes from Provides mapping from requirements statements to several requirements representations but it does not care about subsequent phases	The tool generates a model with information about the join points where aspects crosscut. But no explicit composition rules are defined	Changes in requirements may be solved by re-generating all models identifying the new concepts, e.g. viewpoints and aspects	Theutilization of XML ensures the scalability of the approach. However in large systems the number of identified concepts become unmanageable	- Requires a big user involvement. - It may not be applied to other artefacts different from text.
	Theme/ DOC	It does not care about origin of requirements Direct link between requirements and design models using *major action* and *theme view*	Clipped view proposes the order for the composition, but the actual view where the composition is done is provided at the design level	The only theme affected by a change is the one that generated the change. Re-generation of views is supported by the tool	The graphical representations used become unmanageably large	- Does not deal with non-functional requirements. It may not be applied to other artefacts different from text.

Table 5. Comparative table of Early Aspects approaches

The criteria selected to compare the approaches allow some conclusions to be pointed out.

As a first observation, regarding to traceability, we observed that most of the approaches just care about the origin of the different concerns and suggest a direct mapping from aspectual requirements to aspects at the design level. However, as it has been mentioned in this section, an aspect at the requirements level is not always a crosscutting concern at the design. As an example, imagine that a refactoring is performed at the requirements level so that the crosscutting concern is isolated. This could remove scattering and tangling for this concern in the rest of the phases. In the case of Theme/Doc, the authors suggest the isolation of the crosscutting concerns at the design level using Them/UML. Both tools are part of the Theme approach. Theme and AOREC are the approaches that provide a separation of the crosscutting concerns from requirements to the latest phases of development.

Secondly, as it may be observed in Table 5, most of the approaches pay special attention to the composability of aspectual and base requirements. There is just an approach, COSMOS, which does not deal with this composability. Composability is really important since it allows to evolve a system just applying different composition rules. As an example, imagine a system where logging is performed using databases. The logging policy could be changed to use files just composing the system with a different aspect. In our aspect mining process, presented in Section 5.1, composition is afforded by using Pattern Specification. Anyway, although we have provided a way to compose base and aspectual requirements, our main purpose is the process to identify crosscutting concerns.

Since most of the approaches focus on the separation of the crosscutting concerns using aspectual entities, evolvability is highly improved. In that sense, in most of the approaches, a change in a concern only affects to the corresponding artefacts implementing this concern and their relations to other elements. Note that the removal of the crosscutting behaviour allows to improve cohesion of the different modules (at the requirements or whatever abstraction level). Note again, the evolvability is also improved by the composition rules used by the different approaches since they provide a way to change parts of the system.

Regarding to scalability, the main limitations that most of the approaches present is related to the size of the graphical representations or notations (for those approaches that use graphical models). However, in most of cases, this limitation is solved by using entities with a coarser granularity level (such as packages). In general, scalability of the approaches is ensured for those that use XML for representing the different entities used. In that sense, in our aspect mining process, XML has been also selected to represent both source and target elements (shown in Section 5.1).

A final observation of the different approaches is that most of them focus on the modelling of aspectual requirements using separated entities. However, only a few deal with the identification of the crosscutting concerns in early stages. In that sense, EA-Miner and Theme/Doc are the approaches that a present a method to perform this identification. Nevertheless, these approaches rely on the analysis of natural language and they lack a unified process that generalises its application to other requirements artefacts such as use case diagrams or viewpoints. In this setting, the aspect mining process presented in this thesis focuses mainly in the identification of crosscutting concerns. Moreover, although it has been applied at the requirements level, since it is based on a generic conceptual framework, is application to other abstraction level using different artefacts is also possible.

2.3. Aspect-Oriented Software Product Lines approaches

As it has been stated in the Section 1.1.3, Software Product Line Engineering supposes an important change in the way that software systems are built. In particular, SPL approaches [45]

[147] aim at improving maintainability and reducing maintenance costs. In this setting, feature-oriented modelling techniques support the analysis of commonalities and variabilities among products of a family [55] [104]. In addition, feature dependency analysis identifies the dependencies among features of a SPL [120]. The effectiveness of a software product line approach highly depends on how well features are managed throughout both development and maintenance stages [181]. The more independent the assets are, the easier the products are likely to be built [46]. However, features may crosscut each other, making them rigidly dependent and reducing thus the stability and changeability of the SPL assets [46] [87] [121] [181].

There are several works which have introduced the need for incorporating the benefits of aspect-oriented techniques to deal with crosscutting features in SPL domains. Examples of these works are those presented in [46] [87] [121] [126] [138] [181]. However, most of these approaches rely only on the modelling of variable features in SPL using aspect-orientation. But, as it is stated in [121], common features could be also modelled using aspect-oriented techniques (e.g. aspectual components) if they crosscut other features. Analogously, variable features need not to be defined always as crosscutting concerns. They may be effectively implemented in modular components if they do not crosscut other features [52].

In this section, several works which use aspect-oriented techniques in the SPL domain are described. The section focuses just on approaches that combine AOSD with SPL techniques. Thus, other important SPL approaches (e.g. [45] [84] [147] [178]) or techniques (e.g. Feature Oriented Programming (FOP) based techniques [18]) that do not use aspect-orientation are omitted because they are beyond the scope of this Thesis. Some of the approaches presented also combine the utilization of AOSD with Model Driven techniques to model product families. The reader may find deeper analyses of many SPL approaches in AMPLE project [4]. This project provides several surveys of SPL approaches classified according to the development phase where the approach is applied, e.g. at requirements [126], architecture [6], design [88] or implementation [87] level.

Table 6 (at the end of the section) summarises the approaches presented in this section so that the readers may refer to this table if they wish.

2.3.1. Feature-driven, aspect-oriented product-line CBSE

Martin L. Griss was one of the first authors introducing the benefits obtained by the use of aspect-oriented techniques in product line developments. In particular, he describes in [87] how to improve software reuse and modularity in SPLs by using different technologies, being aspect-oriented techniques within the choosen. In this work, the author briefly describes a process to build product lines combining feature driven techniques and aspect-oriented ones. This process consists of the next steps [87]:

1. The process starts by building a feature oriented model and high-level architecture and design with explicit variability and traceability. The feature model may be obtained by using different techniques such as FeatuRSEB. The features obtained are also traced to collaborations, variation points and variants in the design and implementation.
2. An aspect-oriented technique must be selected in order to implement the features that span multiple components of the product line. The aspect-oriented language must be selected according to the granularity level used to express the variable design and implementation features and the pattern of their combinations.
3. Aspects are implemented according to the mechanisms provided by the aspect-oriented technique selected.

4. The complete products of the family are designed and implemented by selecting and composing the features. The aspects implemented are also selected and weaved with the base features to obtain the components that implement the final products.

This work was one of the first establishing the foundations for the application of aspect-oriented techniques in SPL. However, the work lacks of a concrete process to be applied and it was just a roadmap for the combinations of both methodologies (SPLE and AOSD). Moreover, the process suggests the application of aspect-oriented refactorings once the product line has been developed. However, that implies a double effort and increase time-to-market since the system must be previously developed in order to apply the aspect-oriented techniques. In that sense, the utilization of aspect-oriented techniques in previous phases of development aims at solving this situation, as it has been shown by several approaches [16].

2.3.2. Feature relation and dependency management using aspect-orientation

Lee et al. introduced in [121] the need for using aspect-oriented techniques to deal with crosscutting features in SPL domains. In this work, the authors focus mainly on the modelling of variable features using aspect-orientation, although they claim that common features could be also modelled using aspects when they crosscut other features of the product line. The work presents also the need for modelling feature dependencies using aspect-oriented techniques. For instance, the introduction of a particular feature as an aspect into a product line could imply that other aspects (implementing other variable features) are changed according to the new feature added. A different example of dependency between features is *usage*. The *usage* dependency means that a feature A requires a feature B for its correct functioning [121]. Then, the authors follow a dependency-driven approach to identify situations where aspects should be used. The work is more elaborated in [41], where they presented different aspect-oriented patterns to model several types of feature dependencies.

```
class Car {
  Engine engine = new Engine();
   AirConditioner ac = new AirConditioner();
   ...
  public void start() {
     ac.start();
    ...
  }
}
```

Figure 49. Example of aggregation between several features (extracted from [41])

```
class Bird extends Animal implements IFlyingObject {
   public void cry() { ... } // from Animal
   public void fly() { ... } // from IFlyingObject
   ...
}
```

Figure 50. Example of aggregation between several features [41]

```
aspect AggregationWithAirCond {
   AirConditioner Car.ac = new AirConditioner();
   ...
   pointcut ACstart(Car c):
     target(c) && execution(void Car.start());

   before(Car c): ACstart(c) {
     c.ac.start();
   }
 }
```

Figure 51. Aggregation between features [41] implemented using AspectJ

```
class Bird extends Animal { ... }

aspect SpecializationOfFlyingObject {
   declare parents: Bird implements IFlyingObject;
   public void Bird.fly() { ... }
   ...
 }
}
```

Figure 52. Generalisation between features [41] implemented using AspectJ

In [41] the authors identified several types of dependencies between features. These dependencies were classified into four groups: structural, configuration, operational and activation dependencies. **Structural dependencies** are the relationships used in a feature model to organize the features, namely aggregation and generalisation. Aggregation between two features A and B indicates that feature B is a part of feature A. Using this relation, composition may be expressed in feature models. Generalisation is used to represent the relations where a feature is specialized into more specific features that inherit all the

properties of the former. Examples of aggregation and generalisation relationships are shown in Figure 49 and Figure 50 respectively. The former represents a Car class implementing a Car feature and adding the AirConditioner and Engine features. The latter represents a Bird feature that inherits the behaviour of two different features Animal and FlyingBehaviour. In both cases, the modification of certaing features implies changes in other features. Imagine examples of cars produced without air conditioning. The aggregation relationship implies that Car class must be modified to remove the AirConditioning instance and method calls. In this setting, the utilization of an aspect to model the AirConditioning feature would resolve this situation since the aspect could be added or removed to the Car feature without altering this feature. The aspect implementing the AirConditioning feature may be observed in Figure 51. In Figure 52, an aspect to add the FlyingBehaviuor to Bird feature is also shown. Note that in the object-oriented version, the existence of birds that are not able to fly compromises the design since modifications to the Bird class would be needed.

The second group of relations is called **Configuration dependencies**. This group represents the relations existing when different features are selected to compose a product. Then, they are used in the configuration during the feature selection process. In this group, the authors also identified two different relations: required configuration dependency and excluded configuration dependency. The required dependency is used when a feature requires a second feature for performing its functionality. On the other hand, the excluded dependency is used when a feature should be omitted to produce a particular product of the family. Both kinds of dependencies may be modelled using aspect-orientation in a similar way to the shown for structural dependencies. In this case, the authors use the compile-errors introduction mechanism of AspectJ to implement the aspects (more details may be obtained in [41]).

Operational dependencies group also includes two types of relations: usage and modification. The usage dependency is identified when a feature depends on a different feature for its correct functioning. Again, the modification of the dependee feature implies that all the dependeer features must be changed accordingly. The modification dependency is used to denote the situation when the activation of a feature in a product implies the modification of other features. These two kinds of dependencies are also identified as possible candidates to be modelled using aspects. Examples of these aspects are shown in [41].

The last group of relations identified is **Activation dependencies**. This group is used to represent the situations where the activation of a feature depends on the activation of other features. This group includes four different dependencies: exclusive activation, subordinate activation, concurrent activation and sequential activation. Exclusive activation is used when two different features are mutually exclusive each other. In this case, the activation of a feature implies the no-activation of the other one. Subordinate activation refers to the situation where a feature (the subordinate) may be only activated when other feature (the superior) is activated. In concurrent activation, the subordinate feature is always activated at the same time that superior feature. Finally, sequential activation introduces an order between the features activation so that all the subordinate features of a superior must be activated following the established order (sequentially). Again, the authors provide examples of aspects to implement all these kinds of dependencies [41].

Finally, the authors analyse the benefits obtained by the application of these aspects to the product lines development. They concluded that these benefits provide important improvements of different quality characteristics of software such as adaptability, reusability, capability of evolution and incremental development, and configurability. They argued that adaptability is improved since the addition or removal of features to a product is just carried out adding or removing aspectual components, respectively. The use of aspects to model dependencies between features also helps to reusability since these dependencies are removed from the core components. Evolution and incremental development are also

achieved just adding new aspects to the products. Finally configurability in product lines is achieved by selecting the features that make up a particular product. Ideally, the units of features should correspond to units of components. Then, the utilization of aspects to model variable features and dependencies helps to make the corresponding between units of feature variations and units of components.

However, despite the aforementioned benefits introduced by the appraoch, it still has several limitations that compromise its utilization. First of all, although the authors have identified the need for modelling any type of feature (variable or not) using aspects, they just focus on variability leaving mandatory features out. In addition, they also identified that not all variable features need to be modelled using aspects: *variable features that can be encapsulated into modular units, called modular features, can be implemented as modular components* [121]. These assumptions emphasise the need for using a process to identify crosscutting features in order to model these features using aspect-oriented techniques. Moreover, this approach relies on the use of aspect-oriented techniques at the programming level, relegating the benefits of reducing dependencies between features to the latest phases of development.

2.3.3. xWeave: modelling product line variability using aspects

In [181] Voelter and Groher presented an approach to deal with the development of software product families. This approach is based on the combination of Model Driven (MDSD) and Aspect-Oriented Software Development (AOSD) techniques to tackle the complexity of the developments. MDSD is used to ensure traceability between model elements. In particular, the authors suggest the utilization of model transformations to automatically generate application engineering models taking domain engineering models as starting point [181]. This transformation aims at bringing the gap between problem space and solution space, an important improvement in the SPL domain. This mapping between problem and solution spaces is formally defined and automatically performed by a model-to-model transformation. Note that core functionality of the product family, as well as the flexibility to adapt to different products requirements are captured and modelled as reusable assets during domain engineering. These reusable assets are assembled to generate the different products which are completed with product-specific artefacts. This is the point where aspect-orientation comes into play. By using model weaving and an additive approach, the authors propose a tool to build the final products of the family by weaving the variable features into the core models generated by the model transformation.

As authors claim, the combination of MDSD and AOSD into SPL developments has important benefits summarised as:

- Variability is described on model level so that it is more concisely described than approaches using just traditional mechanisms (mainly used at programming level, e.g. pre-compilers, polymorphism, reflection and so on).
- As it is aforementioned, the mapping between problem and solution spaces is formally defined and automated by model transformations.
- Aspect-oriented techniques are used to model variability not only at codel level but also at model and generator level.
- Fine-grained traceability is achieved since traceability is supported at model level rather than only at code artefacts level.

Since the focus of this section is aspect-oriented product lines, in the rest of this section the aspect-oriented part of the approach is described. In particular, the approach presented in [181] is extended in [88] with tool support for modelling product lines variability using aspect-

oriented techniques. The approach uses xWeave, a model weaver to add variable features to the core functionality modelled in the domain engineering phase. A different tool called *pure::variants* is used to automatically generate the products form the feature selection in the configuration step.

xWeave is a tool that supports the expression of variability at model and meta-model level. The tool uses the concept of join point model as the points in the core functionality where the behaviour added by the variable features will be weaved. The join point model used by xWeave is based on the metamodel of the base model. Thus, this join point model is generic and any artefact of the base model may be a potential join point where adding functionality is possible. Both the base model and the aspectual model (which models variability) are defined by meeting the same metamodel so that the model resulting in a weaving could be again weaved with a different model. That allows recursive weaving enabling the composition of different aspectual models into the same base model. Note that this approach is asymmetric since aspectual models do not have sense without a base model to be weaved.

xWeave provides two ways of defining pointcuts at model level: name matching and explicit pointcut expressions. Name matching is used to weave model elements with the same name and type. As an example in Figure 53 a base model (a) and an aspectual model (b) are shown. Since both models have an element called State, these elements are merged by the the weaving resulting in the model shown in (c).

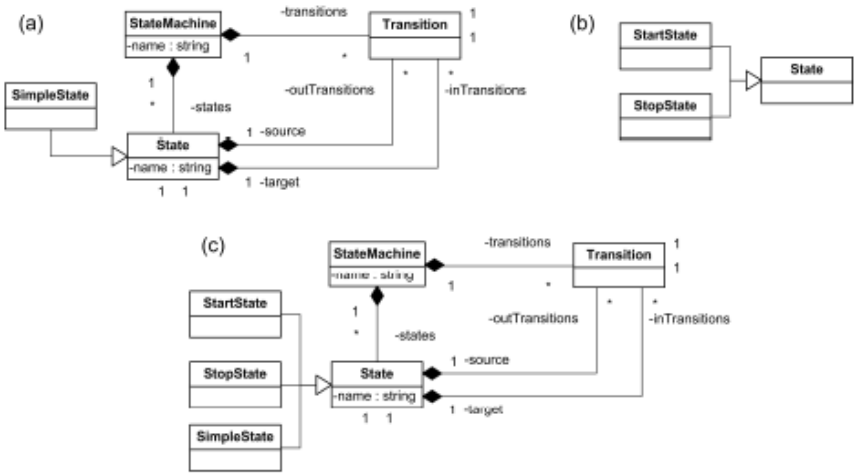


Figure 53. Example of name matching pointcut [88]

A more complex way of defining pointcuts is explicit pointcut expressions. By explicit pointcut expressions, the tool provides a way to identify any model artefact at a join point and to group these join points so that different model elements may be used in the same pointcut. These expressions are externally defined so that they are described in a separate expression file independent from the base and aspectual models. In order to define these expressions, the tool uses an oAW's expression language [143]. In Figure 54 an example of model weaving using this kind of pointcuts is shown. In this example, the state EmergencyStop is added to a base state machine. Note that the aspectual model represented in (b) define a pointcut called *pc* which intercepts all the states of type *SimpleState* defined in the base model (a). The resulting weaved model is shown in (d). The aspectual model in (c) also shows an alternative way of selecting all the instance of a particular type by using the asterisk character (*).

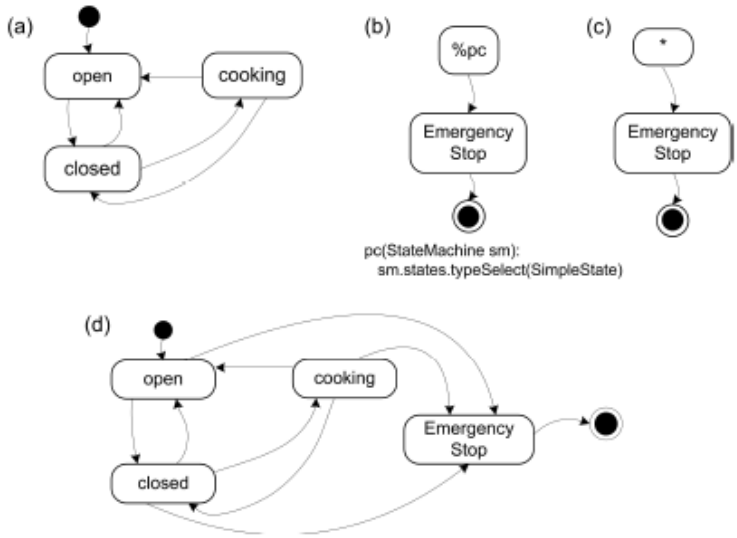


Figure 54. Examples of explicit pointcut expressions to weave models [88]

The weaving process performed by xWeave is driven by the selection of the features for the different products of the family. Based on this selection, a model transformation automatically generates the specific product models. This transformation takes into account the variable features selected to peform the weaving of the models. The approach uses a tool called *pure::variants* to link the domain engineering models to the application engineering ones supported by the model transformation. The basic idea of *pure::variants* is to model product lines using *feature models* and *family models*. Feature models are used to define the whole functionality of the product line. This model lists all the features (common or variable) and the interdependencies between them. On the other hand, family models are used to represent the features that implement a particular product. Family models use the concepts of components which represent elements of the software solution (classes, objects, functions, etc.). Then, the developer just uses pure::variant to select the features for a product. This selection of features is called *variant model.* The tool checks the validity of the variant model and resolve conflicts when needed. The result of the validation process is an abstract representation of the selected product in terms of components. The application of the weaving process provided by xWeave is done by establishing a link between the features in the feature model and the aspectual models that implement these features. In particular, aspects that implement optional parts of structural models (based on components) are linked to the features defined in the configuration model. This process is represented in Figure 55.

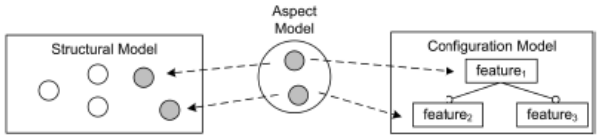


Figure 55. Aspect models are linked to the features that they implement [88]

The benefits provided by this approach have been already mentioned throughout this section: the application of aspect-oriented techniques at earlier development stages, and the utilization of MDSD to generate the particular products of a family just configuring the wished features. However, this approach just models variability using aspect-oriented techniques. Although this is an important contribution because the selection of variability may be used to automatically generate the different products, the core functionality should be also analysed in order to detect possible crosscutting concerns that would affect the modularity of all the specific products generated.

2.3.4. NAPLES

In [126] Loughran et al. introduce an approach, called NAPLES, to identify common and variable concepts in product line requirements using Natural Language Processing techniques. Actually, the approach is an extension of the EA-Miner tool presented in [156] (and described in Section 2.2.3.6) to be applied at the SPL domain. As it was described in Section 2.2.3.6, the tool allows the identification of viewpoints (or use cases) and early aspects using requirement documents as input. Then, based on the concepts identified different requirement models may be derived. In this case, the identification process has been adapted to also identify commonality and variability. Therefore, the concepts identified are used to generate AORE and feature models. These models aim at deriving a *framed aspects* model that allows the automatic generation of source code, using the tools described in [125]. The extended process is shown in Figure 56 where the main steps performed are outlined.

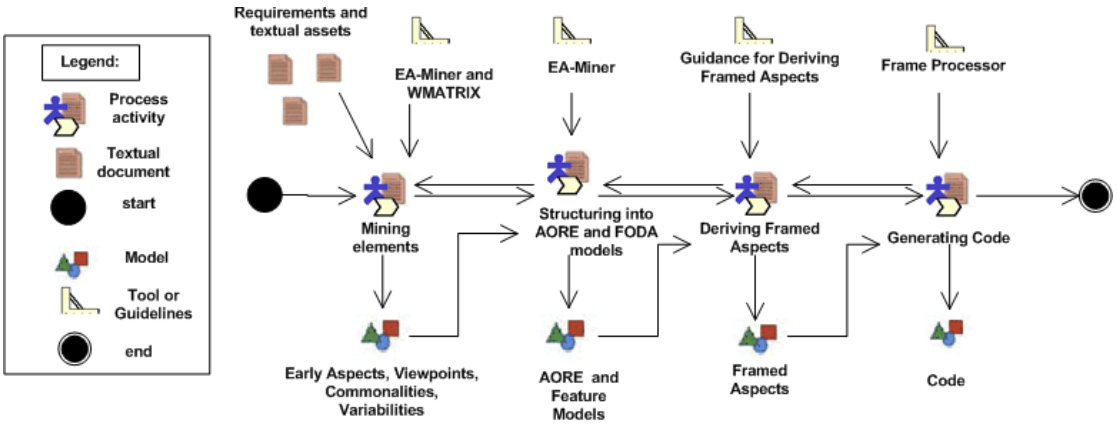


Figure 56. EA-Miner tool extended to be used at SPL domain

As it may be observed in Figure 56, the main differences with respect to the original tool are concentrated in the identification of key concepts (including commonality and variability), the generation of a feature model and the *framed aspects* model derived from the models generated by the tool. The identification of common concepts for the product line is performed by using a catalogue where domain specific concepts are defined. For instance, in [126] the authors introduce an example of product line for the mobile phones domain. Then, the catalogue contains words like game, contacts, chat, calendar, etc. When these words are identified in the requirement documents of the product line, then they are considered as common features of the product line. The identification is based on syntactical and semantical analyses (by using WMATRIX [161]).

Once commonality has been identified, the tool encourages the developer to focus on the surrounding text of the common concepts to indentify variability. In particular it shows to the developer the context of a particular common concept (the sentences where it appears). Then, the developer must manually identify the concepts that make different to the products of the family. Finally, using the set of concepts identified, both an aspect-oriented requirements model (note that EA-Miner identifies viewpoints and aspectual requirements) and a feature model are obtained. These two models are also used to derive the *framed aspects* model. *Framed Aspects* is an approach that combines variability management with aspect-oriented programming. In particular, the approach uses an independent meta-language to implement variability and an aspect-oriented language to implement crosscutting concerns. The aspects implemented are adapted and configured to meet the different requirements of the products (including, thus, variability into aspects). The *Framed Aspects* approach also includes composition rules that allow the integration of the aspects into the base code so that the final code that implements the product line may be generated.

An important contribution of the NAPLES approach is the semi-automation of the process to identify variability and commonality in SPL applications. This is combined with the utilization of aspect-oriented techniques to model the features that are really crosscutting others and not just variability. Moreover, the approach is used at the really beginning of the development process, taking requirement documents as starting point. Finally, the utilization of the *Framed Aspects* approach aims at improving the implementation of crosscutting concerns as aspects. They are parameterised to be used in different context, allowing the variability management. However, since NAPLES is based on the EA-Miner tool, the former presents the same limitations that those described for the latter. Firstly, although the approach is semi-automatically applied, the user involvement is still high and the identification of variability is completely left to the developer's domain knowledge. Secondly, the tool may be only used in textual requirements artefacts; consigning legacy systems (usually described by using other artefacts) to oblivious. The utilization of the catalogue of domain specific common concepts provides important benefits since these core concepts may be automatically identified (also variability which is identified in the surrounded text). However, this fact implies: (i) the need for having the particular catalogue for any domain where we want to apply the approach; (ii) that any common or variable concept not explicitly mentioned in the catalogue will not be identified.

Similarly to the *framed aspects* approach, Morin et al. also introduced an approach to make aspects reusable by means of parameterizing them to be used in different contexts [138]. In particular, in this work the authors use a previous aspect-oriented modelling approach that allows the weaving of models implementing aspects and base code. The aspects modelled are used to implement variability in product lines so that the weaving at model level adds variability to the core product line, resulting in the different products. The authors argue that the approach improves variability management in two different dimensions: the weaving of aspects represents the first variability dimension; the second dimension is achieved by the introduction of variability into aspects so that they may be parameterized and reused. The latter is done at the meta-model level by means of extending a previous meta-model defined to allow aspect modelling. The extension consists of the introduction of different adapter meta-classes. Again, this approach also deals with variability using aspect-oriented adaptations, however, the approach lacks of a process to identify crosscutting situations so that aspects are used to model well-know crosscutting concerns or to add specific behaviour to the system.

2.3.5. Aspectual Mixin Layers

Aspectual Mixin Layers (AML) is the approach introduced by Apel et al. in [6]. This approach proposes the combination of Feature-Oriented Programming (FOP) and Aspect-Oriented Programming (AOP) to obtain more flexible and reusable systems in the SPL domain, where flexibility and configurability are especially important. AML is based on a previous work in FOP called Mixin Layers. Mixin Layers is based on the utilization of a technique to horizontally decompose systems in form of layers. These layers are composed of fragments of different classes that implement features and collaborations between these classes. In Figure 57 an example of three mixin layers (L_1 – L_3) is shown. In this figure we can see how the layers span over three different classes (C_A - C_C) and each class is composed by a set of mixins (the squares). The mixin layer, thus, is obtained by the combination of a set of mixins.

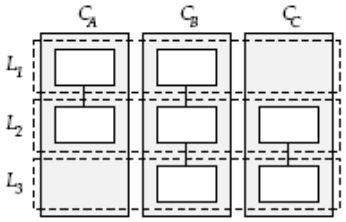


Figure 57. Three mixin layers (extracted from [6])

The whole set of mixins that belongs to a same class is called a *refinement chain*. Mixin may be of two types: *constants* and *refinements*. While *constants* represent the roots of a *refinement chain*, *refinements* are the mixin applied to a *constant* in order to extend it and they form the *refinement chain*. In Figure 58, a fragment of a class is shown with two mixins, a *constant* (line 1) and a *refinement* of this *constant* (line 5). This mixins are extracted from an example implementing a buffer.

```
1  class Buffer {
2     char *buf;
3     void put(char *s) {}
4  };
5  refines class Buffer {
6     int len; int getLength () {}
7     void put(char *s) {
8        if (strlen(s) + len < MAX) super::put(s);
9     }
10 };
```

Figure 58. Mixin with a buffer and its refinement [6]

The AML approach proposes the extension of the mixin layers in order to also include aspects into the layers. Then, an aspect is a new mixin type that collaborates with others to implement the functionality of a particular layer (a feature). But, aspects may be also refined so that they may be adapted and reconfigured in order to be used in different contexts where different products may be produced. The combination of these two approaches may be carried out in two different ways: (i) composing the mixins, applying the aspects subsequently and then selecting the layers; (ii) compose mixins and aspects layer by layer. The authors selected the first option. That implies that the implemented aspects should be reusable enough to be used in different layers (this is achieved by adding *refinements* to aspects). In Figure 59 the aspectual mixin layers implementing the buffer example are shown. In this case, a logging feature has been added. Figure 60 shows the same system where the aspect implementing logging has been refined to be used in a different context (in ExtLog layer). Note that the evolution of the aspects just implies to add a new layer.

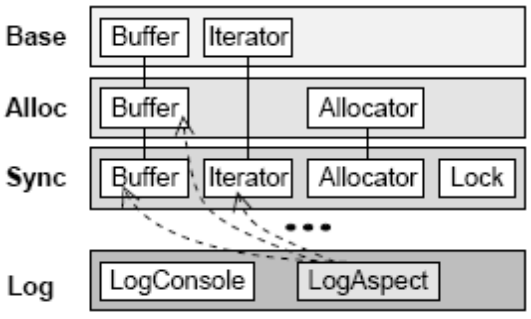


Figure 59. Aspectual mixin layers for the buffer example [6]

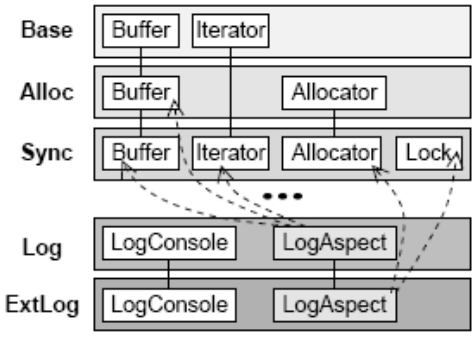


Figure 60. Buffer example with a refinement of logging aspect [6]

The authors also evaluated the situations where it was better to use mixin refinements or aspects to implement crosscutting behaviour. In [6] they concluded that *refinements* should be

used to implement static crosscutting (adding methods and attributes to a class) and heterogenous crosscutting (when different code is added to different join points related to the same crosscutting concern) and also for hierarchy-conforming crosscutting (crosscutting behaviour that affects to classes belonging to the same hierarchy). On the other hand, aspects should be used to implement homogeneous crosscutting (i.e. an aspect adds the same behaviour to the different join points), features that depend highly on the runtime control flow and non-hierarchy-conforming crosscutting (i.e. aspects connecting structural independent features) [6]. This work has been also extended in [7] where the authors applied the AML to more complex case studies and performed a deeper analysis of the advantages of its utilization with respect to traditional FOP techniques. In this extension, the name of Aspectual Mixin Layers is changed by Aspectual Feature Models (AFM) to denote the approach combining FOP and AOP.

The utilization of refinements to extend aspects has important contributions for the SPL domain. Note that aspects implemented by refinements are more reusable and configurable so that they may be easily adapted to be used in different products or implementing different features. Moreover, the process to develop software using AML seems to start at architectural level by designing an aspect-oriented architecture that is decomposed by layers later on. Then, the benefits of aspect-orientation would not be relegated to later phases of development. However, most of the tasks shown in the process are really tied to the programming level so that refinements and aspects are really implemented at source code. In addition, the aspectual mixin layer model is more tied to the detailed design than to a more abstract level like architecture. Moreover, like most of the approaches presented in this section, the approach lacks of a process to identify crosscutting features.

2.3.6. Modelling scenario variability as crosscutting mechanisms

Bonifiacio and Borba have recently proposed an approach to model variability in use case scenarios as crosscutting [28]. The approach is based on the separation of variability and scenario specification concerns. These concerns are later composed by a weaving (driven by different kinds of artefacts) that allows to obtain the final products of the family. These artefacts are: feature models, SPL use case models, product configuration and configuration knowledge. They will be described below. The authors claim that the approach has important benefits with respect to previous approaches (e.g. PLUC [27] and PLUSS [69]) where variability is modelled within the scenarios so that the addition of a new product to the line implies to change too many of them. The approach allows the separation of three dimensions of variability: variability in function, variability in data and variability in control flow. Variability in function is achieved by allowing different products to have different features. In other words, variability in function "*occurs when a particular function might exist in some products and not in others"* [28]. Variability in data is achieved by allowing different products to have the same behaviour with different values or paramters for specific concepts: it "*occurs whenever two or more scenarios share the same behaviour (the sequence of steps) and differ in relation only to values of a same concept"* [28]. Finally, variability in control flow denotes the situations when two or more products differ on the behaviour implemented by the use case scenarios: it "*occurs when a particular pattern of interaction (a use case scensario) varies from one product to another"* [28].

The process presented starts by the building of the features model of the product line. This feature model represents the main concerns that the different products may implement. However, it also has an important function for the weaving process performed later. It is used to validate the products obtained checking whether they are correct according to the features represented in this model. In [28] the authors introduce an eShop example which has the next features model (Figure 61):

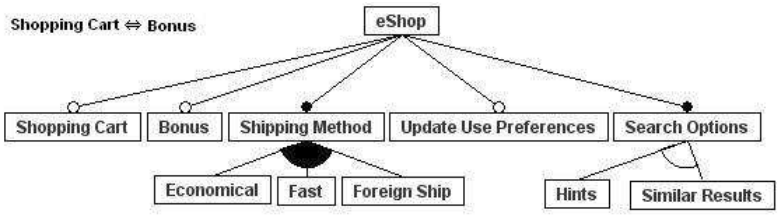


Figure 61. Features model for an eShop example (extracted from [28])

Once the features of the product line are modelled, the authors utilize use cases to model the requirements of the different products. This is mainly the step where they separate variability from the base functionality of the use cases scenarios. In order to achieve this separation a distinction between use cases and aspectual use cases s done. A use case consists of a name, a description and a list of scenarios (pairs of *User action x System response*). The aspectual use cases are used to model variability in control flow since they implement advices to be added to the core features of the products. The advices implemented by aspectual scenarios may be executed before or after any step of a particular scenario. Variability in data is also achieved by introducing parameters into the scenarios. In Figure 62 and Figure 63 an scenario and an aspectual scenario are shown. The former represents the functionality of *Proceed to purchase* in the eShop system. The latter adds the functionality of *Buy a product* and it is added only when the *Shoping cart* and *Bonus* features are not selected.

Id: SC01
Description: Proceed to purchase

Id	User Action	System Response
P1	Fill in the requested information and select the proceed option.	Request the shipping method and address.
P2	Select one of the available ship methods (<SM>), fill in the destination address and proceed.	Calculate the shipping costs.
P3	Confirm the purchase.	Execute the order and send a request to the Delivery System to dispatch the products. [RegisterPreference]

Figure 62. Scenario for the Proceed to purchase use case [28]

Id: ADV01
Description: Buy a specific product
Before: P1

Id	User Action	System Response
B1	Select the buy product option.	Present the selected product. The user can change the quantity of items he wants to buy. Calculate and show the amount to be paid.
B2	Select the confirm option.	Request payment information.

Figure 63. Scenario for implementing the Buy a specific product advice [28]

The behaviour defined in the advice *Buy a specific product* is added before the step P1 of the *Proceed to purchase* scenario. Note also that step P2 in *Proceed to purchase* scenario defines a parameter (*<SM>*) to specify the shipping method selected by the user. This is how the scenarios allow to bind parameters and to model variability in data. This parameter will be instantiated later during the product configuration.

The product configuration is the process where the features of the different products of the family are selected. The authors use a tree-like structure to represent the different features selected for each product (see Figure 64). The configuration of each product is validated by using the features model built in the first step of the approach. This is done during the weaving process.

Figure 64. Configuration tree for two different products [28]

Finally, the approach introduces a configuration knowledge step or artifact which relates features expressions (written in propositional logic) to the tasks used to generate specific products. An example of feature expression used in the configuration knowledge is shown in Figure 65.

Feature Expression	Tasks
eShop	select scenario **SC01** select scenario **SC02**
not (Shopping Cart **and** Bonus)	evaluate advice **ADV01**
(Shopping Cart **and** Bonus)	evaluate advice **ADV02**
Update User Preferences	evaluate advice **ADV03**
Shipping Method	bind parameter **SM**

Figure 65. An example of feature expression in configuration knowlegde [28]

Note that the different tasks shown in the feature expression are used to model the different dimensions of variability. The *select scenario* task is used to select the different features for a product (variability in function), the *evaluate advice* task is used to model variability in control flow and, finally, the *bind parameter* is used to model variability in data. The final weaving process performed is outlined in Figure 66. This weaving is used to obtain the specific use case models for each final product.

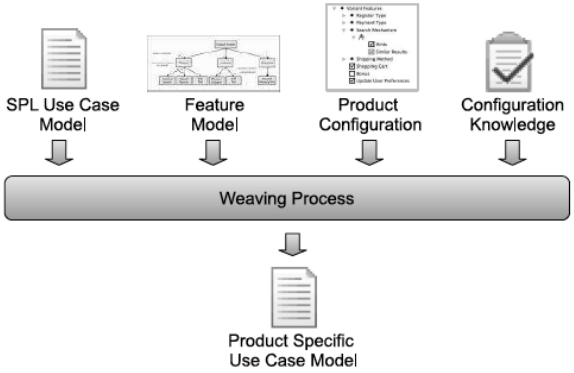


Figure 66. Weaving process to obtain the final products [28]

The main contribution of this approach is the application of aspect-oriented techniques to model variability at the requirements level. In addition, the approach uses requirements model different from just text, which may be really interesting for legacy systems, where aspect refactorings are needed. However, the approach lacks of a process to identify crosscutting behaviour so that, like most of the appraoches described before, it just focuses on the modelling of variable features using aspect-oriented techniques. Regarding to the weaving

process, we also find a problem with the annotation of the core scenarios (for achieving parameters binding and avoiding fragile pointcuts [40]). Althoug the annotation of the core scenarios enhance the power of pointcut and allows the possibility of adding advice's behaviour at any point of execution, it also violates the obliviousness principle, a fundamental principle of AOSD [76].

2.3.7. Summarising the aspect-oriented software product lines approaches

In this section, a brief comparison of the different approaches presented in this section is provided. This comparison has been performed taking into account the generic criteria defined in [115] to compare requirements engineering SPL approaches. In addition, new criteria have been added to the comparison. In particular the criteria added are: abstraction level, crosscutting features modelled and crosscutting identification process. All these criteria considered to compare the approaches are described as follows:

- **Evolvability**. Evolvability denotes to the capability of the approach to deal with changes in any requirement or feature. These changes also include addition or removal of requirements or features.
- **Verification**. This concept is related to the process performed to check whether the different products of the family are aligned to their specifications.
- **Scalability**. Scalability refers to the capability of the process to be equally applied in small and large projects. This requirement is usually met by tool support. However, sometimes tool support is not sufficient.
- **Traceability**. This concept is related to the possibility to establish relationships or links between the artefacts of different phases involved in the approach. As an example, in approaches focused on the requirements level, traceability implies to establish a relation between requirements and architectural artefacts (both forward and backward traceability).
- **Abstraction level**. This concept is related to the development phase where the approach is mainly applied.
- **Tool support**. This point check whether the approach have a tool that implements the concept behind the ideas.
- **Crosscutting features modelled**. This concept represents the difference that exists between some approaches regarding to the kind of features that the aspect-oriented approach provides support for. This criterion has been added since most of the approaches introducing aspect-oriented techniques at the SPL domain just focus on the modelling of variable features. However, mandatory features should be also considered as possible candidate to be modelled using aspect-oriented techniques.
- **Crosscutting identification process.** This last criterion checks whether the approach has a crosscutting identification process or it just deals with well-known crosscutting concerns, such as non-functional concerns.

The summary of the different criteria met by the approaches is shown in Table 6. A column with the main limitations of each approach has been also included in this table.

Authors	Evolvability	Verification	Scalability	Traceability	Abstraction level	Tool support	CC features modelled	CC identification process	Limitations
Griss (2.3.1)	*The aspect-oriented support improves evolution of features*	*No process described*	*No scalable*	*Proposed but no described.*	*From feature modelling to design*	*No*	*Any, mandatory or variable*	*No*	*Lacks of a process to apply the ideas presented. Relegates the application of AO to the latest phases, once the system has been already developed.*
Lee et al. (2.3.2)	*Supported by separating mandatory and variable features and also features dependencies from the core functionality*	*Not supported*	*No scalable*	*No*	*Implementation*	*No*	*Identified the need for modelling any feature. However only variable features modelled*	*No*	*Focuses just on modelling variable features using AO.*
Voelter et al. (2.3.3)	*MDD support improves evolvability using model transformations. AOSD support improves adding new features or modifying existing ones (it is an additive approach)*	*pure::variants checks the validity of variant models and resolves conflicts is needed*	*Supported by xWeave*	*Supported by model transformations and pure::variant*	*From domain engineering to application engineering (implementation)*	*xWeave and pure::variants*	*Variable features*	*No*	*Focuses just on variability to be modelled using AO techniques.*
Loughran et al. (2.3.4)	*Improved by the separation of early aspects Framed Aspects allows to parameterize the aspects to be adapted to different contexts*	*Not supported*	*Scalable to large systems. However user involvement highly increases*	*From requirement models to framed aspects models*	*Requirements. Using framed aspects also at implementation*	*NAPLES*	*Both mandatory or variable if they are identified as Early Aspects*	*Identification of Early Aspects and common and variable features*	*User involvment after applying the process is time-consuming. The tool may be used only in textual requirements artefacts. Common or variable concepts not explicitly mentioned in the catalogue are not identified.*
Apel et al. (2.3.5)	*Supported by separating crpsscutting by aspects. Also supported by refining aspects*	*Not supported*	*No scalable*	*Only horizontal traceability between artefacts of the same layer*	*Design and programming level*	*No*	*Variability is introduced into aspects*	*No*	*The process is very tied to the programming level.*
Bonifacio et al. (2.3.6)	*Improved by separating variability into aspects. The weaving process allows to regenerate a product after a change*	*A Haskell function uses the features model to check the validity of the products*	*Supported by the weaving process*	*Between features and scenarios. Maintained by product configuration and configuration knowledge*	*Requirements*	*Weaving process*	*Only variability*	*No*	*Annotation violates the obliviousness principle.*

Table 6. Comparative table of the different aspect-oriented software product lines approaches

Using the different criteria shown in Table 6, we have extracted some conclusions that are summarised as follows:

Firstly, we observed that evolvability is the characteristic that is more improved by the utilization of aspects in SPL domains. Of course, by separating variable features into well encapsulated entities, the evolution of these features (and in general of the whole system) is highly improved. Note that a change in these features would not imply to change the features crosscut by the former. Moreover, the separation of variable features may help to build systems just composing features by a weaving process.

Secondly, regarding to verification, there are just a few approaches which provide specific methods to verify that the system meets the feature models. This is mainly due to the fact that most of the approaches described just focus on the incorporation of aspect-oriented techniques into the SPL process and the task of verification is left to the own SPL process.

Scalability is mainly supported in all the approaches by the utilization of different tools to support the method. However, sometimes the tool support is not enough to make the approach scalable since the size of the artefacts used become unmanageable or the user involvement is really high. This is why some of the approaches are not scalable.

Regarding to traceability, we observed that most of the approaches just support traceability between two set of artefacts directly related (e.g. features and scenarios where these features are involved). Therefore, there is a lack of approaches that support traceability from throughout the whole development process across the different abstraction levels. Naples [126], the approach presented by Loughran et al., is the only one that supports traceability from requirements to the source code (authors claim that it may be automatically generated by using external tools).

Although we also observed that most of the approaches focus on the programming level, there are also some approaches that deal with aspects in SPL at the requirements level. These approaches are mainly Naples and the presented by Bonifacio and Borba [28]. However, as it has been widely claimed by aspect-oriented community, the sooner aspect-oriented techniques are incorporated into the software process the more benefits are obtained.

As it is aforementioned, scalability of the approaches is mainly achieved by the tool support. In that sense, we observed that the 50% of the approaches analysed have tool support. This criterion is really important since tool support becomes essential for the wide acceptation and utilization of an approach by the community.

An important observation of the analysis was that only two of the approaches studied model any feature (mandatory or variable) using aspect-oriented techniques. Most of the approaches deal just with variability. However, the need for modelling also mandatory features as crosscutting concerns when they are not weel modularized has been already identified (e.g. [121]). We concluded that an approach to model aspects in SPL domains should deal with any kind of crosscutting feature.

An important observation that may be extracted is that most of the approaches lacks of a process to identify crosscutting features. Naples is the only approach that has an specific process to perform this identification. However, as it has been mentioned in the previous paragraph, any feature may be modelled using aspects if it is crosscutting other ones. Then, a process to identify crosscutting features becomes mandatory for applying aspect-oriented techniques in the SPL domain. This is characteristic is one of the main limitations of almost all the approaches analysed. We have not highlighted this limitation in the corresponding column of the table (column showing the limitations of each approach) since there were an specific column for specifying whether each approach has the aspect mining process.

2.4. CONCERN-ORIENTED METRICS

As it has been stated in Chapter 1, measurement is crucial for the progress of all sciences. Software engineering is not an exception and the empirical assessment of variables in experiment that may be repeated is the base for research. In that sense, measurement needs the operationalization of concepts in order to be measured using indicators. The empirical assessment of these indicators has traditionally made by means of metrics. Software metrics have been widely used in the literature. Examples of these metrics are coupling and cohesion [189]. However, with the emergence of new paradigms like AOSD, some of these metrics became obsolete and unable to represent new concepts introduced by these new technologies.

In this setting, several works have introduced new metrics that either adapt or extend the traditional object-oriented metrics to deal with new concepts introduced by aspect-orientation [58][60][124][159][183]. They mainly provide measurements for modularity. Most of these metrics are related to the programming level so that assess attributes in terms of concepts such as lines of code, methods, fields or advices (they are defined in terms of specific design or implementation artefacts). As it is stated in [72], all these metrics suites have an important difference with respect to traditional modularity measures [38]: *they capture information about concerns traversing one or more structural component.* In next sub-sections some of these metrics are described.

In Section 5.4.5 we show an empirical comparison between the most important metrics described here and the metrics defined in this thesis. This empirical analysis is used to extract interesting conclusions about the utilization of the metrics, e.g. the need for adapting some of the metrics since they are tied to an specific deployment artefact or the need for introducing an specific metric for crosscutting.

At the end of this section, Table 7 summarises all the metrics presented in this section. The reader with background on the measurement topic may just refer to this table.

2.4.1. Concern diffusion and Lack of concern cohesion.

In [159], Sant'Anna et al. introduced a set of concern-oriented metrics to assess modularity in terms of fundamental attributes of software such as separation of concerns, coupling, cohesion or size. These metrics are defined upon other traditional software metrics such as Lines of Code (LOC) or OO metrics like the aforementioned terms of coupling, cohesion or size. However, the metrics proposed deal with aspects-like concepts, namely separation of concerns.

The metrics proposed are grouped into 4 categories according to the attributes they measures [159]: separation of concerns, coupling, cohesion and size.

- **Separation of Concerns**.

  Whithin this category, the authors defined the next metrics: Concern Diffusion over Components (CDOC), Concern Diffusion over Operations (CDO) and Concern Diffusion over Lines of Code (CDLOC). CDOC measures the *number of components whose main purpose is to contribute to the implementation of a concern*. It also counts *the number of components that access to the primary components by instance instantiation, method calls, attribute declarations, etc.* CDO is a similar metric but it counts *the number of operations whose primary purpose is to contribute to the functionality of a concern and the methods and*

advices accessing to these primary operations. Finally, CDLOC counts *the number of transition points*[5] *for each concern over the lines of code*. In order to use this last metric, a shadowing technique should be used to mark the lines of code implementing the different concerns. Then, a transition point is identified by a transition from a shadowed area to a non-shadowed area (and vice versa). The only difference between these three metrics is the granularity level used to obtain the masurements.

As it can be observed, these metrics are mainly used to measures the scattering of a concern since they counts the number of software artefacts related to the concern. The higher these metrics, the more scattered the concern is. In addition, the use of CDLOC allows the measurement of concern tangling. Note that the number of transition points in a module provides an indication of the concerns implemented by the module.

- **Coupling**.

  Coupling is a metric that indicates the strength of interconnections between software components. Related to this concept, two different metrics are introduced: *Coupling between Components (CBC)* and *Depth of Inheritance Tree (DIT)*. CBC metrics counts *the number of components (classes or aspects) coupled with a particular component*. It considers attribute declarations or components declared in formal parameters, return types, throw declarations and local variables. This metric considers aspect-like concepts since it takes into account accesses to aspect methods or attributes defined by introductions [10]. DIT is defined as the maximum length from a node to the root of the tree. In other words, it measures how far in the inheritance a class or aspect is declared.

- **Cohesion.**

  Cohesion is defined as the closeness of the relationships between the internal components of a module. In that sense, a new metric is defined to measure cohesion in aspect-oriented domains, Lack of Cohesion in Operations (LCOO). LCOO is defined as:

  Let C1 be a component with n operations (methods and advices) O_1, …, O_n then $\{I_j\}$ is the set of instance variables used by the operation O_j. Let $|P|$ be the number of null intersections between instance variables sets. Let $|Q|$ be the number of non-empty intersections between instance variables sets. Then: LCOO = $|P| - |Q|$, if $|P| > |Q|$, otherwise LCOO = 0.

  The higher this metric, the less cohesion an operation presents. The less cohesive an operation is, the worse modularized the design.

- **Size.**

  The size is a traditional metric used in software systems to measure the length of a system's design or code. To measure this attribute, four different metrics are introduced: Vocabulary size (VS), Lines of Code (LOC), Number of Attributes (NOA), Weighted Operations per Component (WOC). VS counts *the number of components (classes or aspects) into the system*. LOC is actually equal than the traditional LOC metric. The only difference is that it considers aspects. NOA measures the internal vocabulary of each component (without considering inherited attributes). Finally, WOC measures *the complexity of a component in terms of its operations.* This last metric is defined as:

  Let C1 be a component with operations O_1, …, O_n . Let c_1,…, c_n be the complexity of the operations. Then, WOC = $c_1 + … + c_n$. where the complexity operation is defined in terms of

[5] A transition point is a line of code implementing a concern different from the implemented by the previous lines of code. In other words they represent a *"concern switch"*

the number of parameters. It is assumed that the more parameters the operation has, the more complex.

In further works of the same authors, they also adapted these metrics to be used at the architectural level [160]. As an example, the Concern Diffusion over Components metric counts the number of architectural components instead of classes and aspects. They also adapted the LCOO metric to measures the number of concern addressed by an architectural component. This metric was called, Lack of Concern Cohesion (LOCC) Finally a new metric is introduced that measures the interlacing between components. This metric is called Component-level Interlacing Between Concerns (CIBC) and it counts the number of other concerns with which the assessed concerns share at least a component.

The metrics presented in this section has been widely used to perform different modularity analysis, e.g. [71] [83] [86]. Even, the metrics has been used to relate modularity with other quality attributes such as stability [71]. However, these analyses have been mainly performed at programming or design level and the application of the metrics to the requirements level has not been faced up yet.

2.4.2. Size, Touch, Spread and Focus.

In [58], Ducasse et al. introduces a technique to visualize software partitions in form of colored rectangles and squares. This technique is called Distribution Map and it allows two partitions that represent all the software artefacts to be graphically represented. More concretely, given a software system *S* that contains a set of software artefacts s_i, and two partitions *P* and *Q* of that set of artefacts, a Distribution Map is a means to visualize *Q* with respect to *P*. Each system artefact is represented by a small square, the partition *P* is used to group the squares into larger rectangles and the partition *Q* is used to color the squares. Partitions *P* and *Q* are defined as follows:

- **Reference partition**. The partition *P* represents the logical decomposition of the software system. i.e. represents a decomposition of the system using its intrinsic structure. For instance a software system is partitioned by software components or classes.
- **Comparison partition**. The partition *Q* represents a decomposition of the system using properties previously analysed. In other words, this partition groups the artefacts by the properties associated to them. The parts resulting in this partition are called properties.

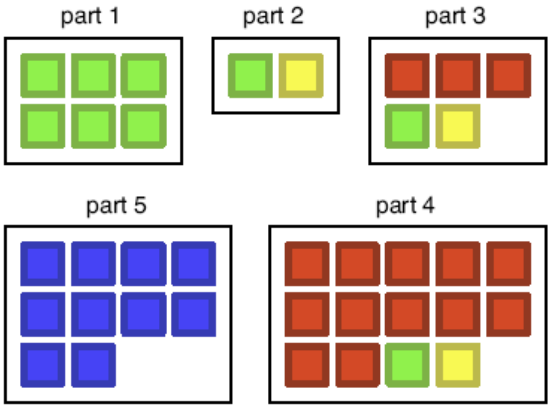


Figure 67. Example of Distribution Map [58]

Then, any software artefact s_i of the system *S* belongs to a particular part p_n of *P* and it is attributed with a property q_n of *Q*. In Figure 67 we can see an example of a Distribution Map. In this map the module 5 contains artefacts only related to the property blue. Then, we can say that this module is well-encapsulated. However, the property yellow is a crosscutting concern

and green could be characterised as an octopus crosscutting concern (see [73] to obtain more details on crosscutting categories).

Based on distribution maps, the authors introduced four concern measures: Size, Touch, Spread and Focus.

- **Size.** This metric counts the number of squares associated to a property.
- **Touch**. The touch metric is a measure of relative size. This relative size is given in terms of the percentage of small squares realising a property. In particular, being $p \in P$ and $q \in Q$ a part and a property, respectively, the relative size of $q \cap p$ is defined as:

$$touch\ (q,p) = \frac{|q \cap p|}{|p|}$$

- **Spread**. It counts the number of modules (classes or components) related to a particular property. This is the number of classes with an internal member associated to this concern. The calculation of this metric would obtain the same results than those obtained by CDOC (described in previous section). This metric is defined as: given a total set *S*, a subset of elements $q \subset S$ and a partition *P* such that $\cup P = S$

$$spread\ (q,P) = \sum_{p_i \in P} \begin{cases} 1, touch(q,p_i) > 0 \\ 0, touch(q,p_i) = 0 \end{cases}$$

- **Focus**. This metric measures the closeness between a particular partition and a property. The more internal members related to the concern the module has, the higher the Focus metric for the module and concern. *That means that well-encapsulated properties have a high focus value, and crosscutting properties have a low focus value* [58].The focus of *q* on the touched parts of *P* is defined as:

$$focus\ (q,P) = \sum_{p_i \in P} touch\ (q,p_i) * touch\ (p_i,q)$$

In [58], the authors illustrated the utilization of the Distribution map technique in combination with other features location techniques. They claim that this technique helps in visualizing the results obtained in a graphical and intuitive way. The metrics presented in this section are the more generic ones so that its application at different abstraction levels could be feasible. However, the authors have not applied them to measure modularity at early stages of development. Moreover, the metrics lacks of an specific metric to measure crosscutting.

2.4.3. Concentration, Dedication and Disparity

Wong et al. also introduced a set of concern oriented metrics to measure the closeness between a feature and a program component [183]. In particular, the authors used a slice-based technique to identify the locations in source code where different features are implemented. Then, using these locations, they calculated three different metrcs: Disparity, Concentration and Dedication. These metrics are defined as follows [183]:

- ***Disparity**: measures how close a feature is to a program component.*
- ***Concentration**: shows how much a feature is concentrated in a program component.*
- ***Dedication**: indicates how much a program component is dedicated to a feature.*

However, in [183] the authors provide more formal definitions of these concepts in order to properly assess the measurements. Before describing the formal definitions of these metrics, the authors introduce the terms of *program features, program components, invoking*

and *excluding inputs* and, finally, *basic blocks*. *Program features* are defined as any functionality defined in the specification. In other words, a program feature refers to a concern or property in the system. *Program components* are entities that comprise the structure of the system, e.g. system files, classes, functions or group of functions. Observe that different granularity levels are possible. An input is used to exercise the system in order to check the features affected by the simulation. In particular, an input *t* is an *invoking input* for feature *F* if, when executed in *P* (the program), it shows the functionality of *F*. Otherwise, the input *t* is defined as a *excluding input*. In order to locate the source code of *P* related to a feature *F*, the authors introduce the notion of blocks. A *basic block* is defined as a consecutive sequence of statements or expressions that they do not contain transfers of control until the end of the block. Then, if an statement is executed, then all are.

Once the previous concepts have been defined, the authors formally define the aforementioned metrics. The definition of the metrics requires establishing some common notation as follows. Let *T* be a set of invoking inputs for feature *F*, where *P* is the program and *C* is a component, then [183]:

- B_{t_i} represents the set of blocks in *P* executed by input $t_i \in T$
- B_F is the union of B_{t_i} such that $t_i \in T$, i.e. B_F is a set of blocks in *P* which are used to implement *F.*
- B_C is the set of blocks in *C*.
- $B_{C \cap F}$ is the intersection of B_C and B_F, i.e., the set of blocks in *C* which are used to implement *F*.
- $B_{C \cup F}$ is the union of B_C and B_F.
- $B_{C \oplus F}$ is the set of blocks in either B_C or B_F, but not both.

Using these concepts, Disparity of a component *C* for a feature *F* is defined as:

$$DISP_{CF} = \frac{|B_{C \oplus F}|}{|B_{C \cup F}|}$$

By simple computation:

$$DISP_{CF} = 1 - \frac{|B_{C \cap F}|}{|B_{C \cup F}|}$$

$DISP_{CF}$ measures the degree to which a feature *F* is close to a component *C*. This metric has the following properties: (i) it is normalized between 0 and 1, the value obtained is inversely proportional to the blocks in $B_{C \cap F}$ (the more blocks shared by *C* and *F*, the smaller the disparity), and it is directly proportional to the blocks in $B_{C \oplus F}$ (the more blocks in B_C or B_F, but not both, the larger the disparity between *C* and *F*).

Next, the metrics of Concentration and Dedication are defined. These metrics represents how much a feature is in a component and how much a component is in a feature, respectively. Concentration is defined as:

$$CONC_{FC} = \frac{|B_{C \cap F}|}{|B_F|}$$

while Dedication is defined as follows:

$$DEDI_{CF} = \frac{|B_{C \cap F}|}{|B_C|}$$

Both metrics, Concentration and Dedication, have the same properties: the values are normalized between 0 and 1, the values obtained must be directly proportional to the blocks in $B_{C \cap F}$ (the more blocks shared, the higher the Concentration of *F* in *C* and the higher the Dedication of *C* to *F*), the values obtained are inversely proportional to B_F for CONC and to $B_C$for DEDI (the more blocks *F* has, the less probabilities that these blocks are in *C*).

Again, these metrics have been applied mainly at the programming level relegating the benefits obtained by the empirical assessment to the latest phases of development. In addition, the metrics also lack of a specific metric for measuring crosscutting.

2.4.4. Degree of scattering and Degree of tangling

In [62], Eaddy et al. presented an empirical analysis showing the correlation existing between scattering and number of faults in software systems. The authors claim that they found a moderate to strong correlation between these two variables so that the more scattered concerns a system has, the more observed defects.

In order to measure the degree of scattering in the system, this work is based on a concern model wich establishes the relations between concerns and program artefacts. This model is shown in Figure 68, and as the authors say in [60], it is based on the crosscutting pattern [24] described in Chapter 3. Using the relations between concerns and target program artefacts, the authors define scattering as the situation where a concern is related to multiple target elements (this definition is also taken from [24]). For the purpose of their analysis, the authors consider that a crosscutting concern is a scattered concern (without considering tangling). The relation between concerns and target artefact is used to locate the source code related to a particular concern.

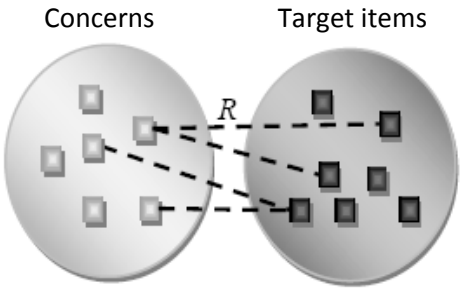


Figure 68. Concern model defined in [60]

The empirical analysis presented in [62] is driven by the utilization of a set of concern metrics to assess modularity. These metrics were firstly introduced in [60] [61]. The first metric defined is *Program Element Contribution (CONT)*. This metric counts the number of lines of code in a program element associated with a concern. When considering methods and fields, the contribution is measured by counting the number of lines in the method or field declaration. In classes, the contribution counts the number of lines dedicated to the class declaration plus the contribution of the methods and fields. For an entire program P, the contribution is computed by adding the contribution of all the classes. The contribution of a concern *s* with respect to a program *P* is considered as a special case and the authors define its own metric for this case: *Lines of Concern Code (LOCC)*, such as *LOCC(s) = CONT(s,P).*

Then, the authors define two new metrics, *Degree of Scattering* and *Degree of Tangling,* for assessing the scattering and tangling in a system, respectively. These metrics are based on the metrics introduced by Wong et al. [183] (described in previous section). In particular, Degree of Scattering is based on *Concentration* whilst *Degree of Tangling* is based on *Dedication*.

The *Degree of Scattering (DOS)* metric provides information about how the concern's code is distributed among the elements. This metric uses *Concentration*. However, this metric is used in terms of Lines of Code. Then, *Concentration* is defined as the quotient of LOC in a component realising a concern by the total LOC realising this concern in the system:

$$CONC(c,t) = \frac{source\ lines\ in\ element\ t\ related\ to\ concern\ c}{source\ lines\ related\ to\ concern\ c} = \frac{CONT(c,t)}{CONT(c,P)}$$

Finally, *DOS* metric is defined as the variance of the *Concentration* of a concern over all program elements with respect to the worst case:

$$DOS\ (c) = \ 1 - \frac{|T|}{|T|-1} \sum_{t \in |T|} \left(CONC(c,t) - \frac{1}{|T|} \right)^2$$

when $|T| > 1$.

The authors distinghuish between DOSC and DOSM which measure the degree of scattering across classes and methods, respectively. DOS has the next characteristics:

- It is normalized between 0 and 1, where 0 indicates a concern completely localized in a program element and 1 indicates that the concern is completely distributed.
- DOS is proportional to the number of program elements related to the concern and inversely proportional to the *Concentration*.
- DOS has a ratio-scale measurement level (see Section 1.1.4). Note that a 0 value indicates no scattering.
- Although the DOS metric has not units of measurment, the concepts used to calculate it have units. In particular, Lines of Code are used to measure CONC and |T| is measured in terms of classes or methods. Then, in order to be able to compare two DOS measurements, the same units should be used.

Using the DOS metric, Eaddy also calculates the *Average of Degree of Scattering (ADOS)* in [63] as:

$$ADOS = \frac{1}{|C|} \sum_{c \in |C|} DOS\ (c)$$

where *C* represents the set of concerns in the system.

In a similar way to the *DOS* metric, the metric *Degree of Tangling (DOT)* is also defined in [63]. This metric is calculated based on the Dedication [183] metric, also considering Lines of Code. Dedication measures how many lines of code of a component *t* are dedicated to a concern *c*. It is calculated as follows:

$$DEDI(c,t) = \frac{source\ lines\ in\ element\ t\ related\ to\ concern\ c}{source\ lines\ of\ component\ t}$$

Then, *DOT* metric is defined to measure the dedication of a program element to one or more concerns of the system as:

$$DOT\ (t) = \ 1 - \frac{|C|}{|C|-1} \sum_{c \in |C|} \left(DEDI(c,t) - \frac{1}{|C|} \right)^2$$

As it was done with DOS, the DOT may be also calculated in terms of classes and methods implying two different measurements *DOTC* and *DOTM* respectively.

This metric also has several characteristics similar to the explained for *DOS*. These characteristics are the next:

- It is also normalized between 0 and 1. Again, 0 indicates an element completely dedicated to a concern and 1 indicates that the program element is addressing all the concerns of the system.
- It is directly proportional to the number of concerns addressed by the program element.
- It is inversely proportional to the Dedication. The more uniformly distributed between the program elements a concern is, the higher the *DOT* is.

Finally, the Average of Degree of Tangling metric across all the program elements is defined as:

$$ADOT = \frac{1}{|T|} \sum_{t \in |T|} DOT\ (t)$$

where *T* is the set of program elements.

All these metrics were used to drive the empirical analysis where the authors demonstrated the correlation between the presence of crosscutting and software defects [62]. This analysis was conducted by a methodology based on three different steps:

1. **Reverse engineering for concern-code mapping**. The first step consisted in analysing the source code of an application in order to locate concerns' implementation.
2. **Mining the bug-code mapping**. Secondly, the bugs in the application are identified and related to the parts of code causing these bugs.
3. **Mining bug-concern mapping**. Having the two previous relations, a transitivity operation is used to obtain the mappings between concerns and code. Having these mappings, the concerns with a higher degree of defects may be measured. This measurement is compared with the aforementioned metrics.

The results obtained showed that the higher the degree of scattering of a concern, the more defects this concern causes. This effect was even independent of the concern's size. The analysis was very important since it empirically demonstrate that crosscutting lead to harmful software quality (as the aspect community has claimed for several years). In that sense, the empirical analyses shown in this thesis (Section 5.5) complements and also supports these ideas. However, unlike the analysis performed by Eaddy et al., the maintainability analyses we present in Section 5.5 are focused on the requirements level. Then, we also demonstrated that the sooner the aspect-orientation is incorporated into the software process, the more benefits.

2.4.5. Features Crosscutting Degree, Aspect Crosscutting Degree, Homogeneity Quotient and Program Homogeneity Quotient.

Lopez-Herrejon and Apel also proposed a set of metrics to classify crosscutting according to the number of classess crosscut and their language constructs [124]. The results obtained allow the developer to classify the crosscutting concerns into two different categories: homogeneous and heterogeneous. These categories have traditionally distinguished crosscutting concerns that add the same functionality throughtout the system classes (homogeneous) from those that add different behavior over the classes (heterogenous).

The metrics proposed are semi-formally defined using a functional programming language and they focus on the measurement of crosscutting characteristics in Software Product Lines. In that sense, the metrics are defined in terms of features. The definition of the metrics is based on the utilization of an abstract program structure that represents a general system. Using this abstract program structure, a program system P is defined as:

P = [F_1, F_2, …, F_n], where F_i are the features of the system.

The abstract program structure used may be observed in Figure 69. A program contains a list of features which also contain elements of type *class*, *interface* or *aspects*. On one hand, *class* and *interface* tokens are also defined by the combination of field and methods and field and signatures, respectively. On the other hand, the *aspect* token may contain two different types of elements: *inter-type declarations* (also known as introductions in AspectJ [10]) and *advices*. Inter-type declarations are defined as structural variations that an aspect adds to a base class (e.g. adding a new field declaration, a method, a constructor, etc.). These introductions are defined as structural crosscutting in [124]. Advices are used to add dynamic behaviour to a system by using join points and pointcuts, then, they are defined as dynamic crosscutting. Note that an advice is defined by the combination of a pce (pointcut expression) and a body.

```
program :: [feature]

feature :: [feature_element]
feature_element :: class | interface | aspect

class :: [class_element]
class_element :: method | constructor | ...

interface :: [interface_element]
interface_element :: methoddecl | field

aspect :: [aspect_element]
aspect_element :: methodITD | constructorITD | fieldITD | advice

methodITD :: (class, method)
constructorITD :: (class, constructor)
fieldITD :: (class, field)

advice :: (pce,body)
pce :: pointcut_expression
shadow :: (program_element, class, pce)
program_element :: class_element | interface_element | aspect_element
```

Figure 69. Abstract Program Structure for a program P (extracted from [124])

Then, based on the abstract program structure defined, the authors define a set of auxiliary functions that aims at calculating the metrics proposed. These auxiliary functions are defined using a functional programming language. The most important functions with respect to the metrics explained later on are described now. There are other functions but they are not shown since the meaning of the functions is quite intuitive and they are similar to those shown:

- **count**. This function returns the number of elements in a list.

  `count :: [a] -> n (where a is any type and n is a number).`

- **loc**. This function returns the number of lines of code.

  `loc :: [a] -> n (where a is any type and n is a number).`

- **sum**. Receives an input list of numbers and calculates the summation.

  `loc :: [n] -> n (where n is a number).`

- **classes**. Receives a feature and returns the list of classes in the feature.

  `classes :: feature -> [class] (where n is a number).`

- **aspects**. Receives a feature and returns the list of aspects in the feature.

  `aspects :: feature -> [aspect] (where n is a number).`

- **advices**. Receives an aspects and returns the list of pieces of advices in the aspect.

  `advices :: aspect -> [advice] (where n is a number).`

Using these functions, the authors introduce the program structure metrics. These metrics mainly measure the contribution of an aspect to the global structure of a program (in terms of LOC). The metrics are the next:

- **Number Of Features (NOF).** It counts the number of features in a program.

  ```
  NOF (P) = count (P)
  ```

- **Number Of Aspects (NOA).** For each feature in program P, this metric extracts its aspects and counts them. Finally, it sums the total aspects for all the features.

  ```
  NOA (P) = sum(foreach(P,λf.(count(aspects(f)))))
  ```

- **Number of Classes and Interfaces (NCI).** Counts the number of interfaces and classes.

  ```
  NCI(P)=sum(foreach(P,λf.(count(union(classes(f))(interfaces(f))))))
  ```

- **Base Code Fraction (BCF).** Number of LOC that comes from standard Java classes relative to the total LOC in the program.

  ```
  BCF(P) = sum (foreach(P,λf.(sum  (loc(classes(f)))
                                   (loc(interfaces(f)))))) / loc(P)
  ```

- **Aspects Code Fraction (ACF).** Similar to the BCF but it counts the number of LOC that come from aspects.

  ```
  ACF(P) = sum (foreach(P,λf.(loc(aspects(f))))) / loc(P)
  ```

- **Introductions Fraction (IF).** Counts the number of LOC that come from introductions or inter-type declarations relative to the total LOC.

  ```
  IF(P)=sum(foreach(P,λf.(sum(loc(fieldITDs(aspects(f))))
                              (loc(methodITDs(aspects(f))))
                              (loc(constructorITDs(aspects(f)))))))/loc(P)
  ```

- **Advice Fraction (AF).** Counts the number of LOC that come from pieces of advice relative to the total LOC.

  ```
  AF(P)=sum(foreach(P,λf.(sum(loc(advices(aspects(f)))))) / loc(P)
  ```

Finally, the authors define feature crosscutting metrics upon the metrics describe above. These new metrics are used to classify the concerns into the homogeneous and heterogenous categories:

- **Feature Crosscutting Degree (FCD).** Counts the number of classes crosscut by all the pieces of advices and introductions of a feature.

  ```
  FCD(f,P)=count(union(ccclasses(methodITDs(aspects(f))),
                       (ccclasses(constructorITDs(aspects(f))),
                       (ccclasses(fieldITDs(aspects(f))),
                        (sclasses(shadows(pointcuts(advices(aspects(f))),P))))
  ```

- **Advice Crosscutting Degree (ACD).** Number of classes crosscut only by pieces of advices of a feature.

  ```
  ACD(f,P)=count(sclasses(shadows(pointcuts(advices(aspects(f))),P)))
  ```

- **Homogeneity Quotient (HQ).** Calculates the division of ACD between FCD.

  ```
  HQ(f,P)= ACD(f,P)/FCD(f,P)      if FCD(f,P) !=0
         = 0                      otherwise
  ```

- **Program Homogeneity Quotient (HQ).** Computes the average of the HQ for all the features in the program P.

```
PHQ(P)= sum(foreach(P, λg.HQ(g,P)))/NOF(P)
```

The authors, thus, use the HQ to classify the concerns into two categories whose the upper ends are:

- **Fully Homogeneous Feature.** This category corresponds to features whose pieces of advice crosscut all the classes crosscut by the feature. For these features, ACD=FCD so that HQ=1.
- **Fully Heterogenous Feature.** A feature is included in this category when its HQ=0. That means that the feature either implements base code or it crosscuts other features only using ITDs (introductions).

The Program Homogeneity Quotient is also used to categorise: (i) systems that use advices as the major way to implement crosscutting concerns (PHQ close to 1); (ii) systems that either mainly use ITDs as the way to implement crosscutting concerns or do not have crosscutting concerns (PHQ close to 0). The authors illustrate the application of the metrics by using several case studies. However, although the metrics have important benefits in order to classify crosscutting concerns, they also have important limitations. Firstly, the metrics are very tied to specific programming artefacts. The problem is even more evident taking into account the Aspect-J like constructs used to compute the metrics. Secondly, the metrics could be only used in systems modelled using aspect-oriented techniques. However, as it has been widely commented in this document, legacy systems are also a challenge for aspect-orientation where it should show its contributions (and modularity analysis should be one of them).

In [35], Ceccato and Tonella claim that the utilization of aspect-oriented programming implies the addition of coupling between aspects and base components, compromising the advantages obtained by its utilization. Then, they argue that aspect-oriented metrics should be used to analyse the trade-off between the advantages obtained by the separation of concerns and the disadvantages produced by the coupling introduced. They claim that "*empirical studies should be conducted to evaluate costs and benefits offered by the AOP solution with respect to the more traditional, Object-Oriented (OO) one, in terms of code understandability, evolvability, modularity and testability"* [35]. In this setting, they introduce a set of metrics similar to those presented by Lopez-Herrejon and Apel. These metrics measure the effects of software aspectization in terms of coupling. These metrics are defined upon traditional object-oriented metrics (e.g. *Depth of Inheritance Tree*), or adaptation of these metrics to be used in aspect-oriented programs (e.g. *Number of Children*, which counts the number of sub-classes or sub-aspects of a module). The most interesting metrics defined are the next:

- ***Coupling on Advice Execution (CAE)***: it counts the *number of aspects containing advices possibly triggered by the execution of operations in a given module*.
- ***Coupling on Intercepting Modules (CIM)***: counts the *number of modules or interfaces explicitly named in the pointcuts belonging to a given aspect.*
- ***Response for a Module (RFM):*** counts *methods and advices potentially executed in response to a message received by a given module.*
- ***Crosscutting Degree of an Aspect (CDA):*** meaures *the number of modules affected by the pointcuts and by the introductions in a given aspect*.

It is clear that the utilization of these metrics aims at discerning when aspect-orientation should be used. As an example, in [83] an empirical analysis using similar metrics demonstrated that not all the implementation of the design patterns where improved by using aspect-oriented techniques. However, since these metrics must be applied to aspect-oriented systems, they have the same limitations described for the metrics introduced by Lopez-Herrejon and Apel.

2.4.6. Summarising the concern-oriented metrics

Now, the most significative metrics of the presented in this section are summarised. All these metrics are described in Table 7. In this table, a column “observations” has been added to describe the relation between the metrics, e.g. when a metric is equivalent or proportional to other metrics presented. Also, a column showing the main limitations of the metrics has been added.

As it may be observed in the table, most of the metrics are focused on the programming level or they have been only applied at the implementation. This fact implies that the improvements obtained by using them and identifying modularity problems are relegated to the latest phases of development. Even, some of the metrics are very tied to specific programming artefacts of particular languages, maken the reutilization of the metrics at different abstraction levels difficult.

We also observed that most of the metrics measure concept related to scattering and tangling concepts, however, there are no specific metrics measuring the concept of crosscutting. In Section 5.4, the metrics suite presented includes an specific metric for dealing with crosscutting. This is an important observation since our definition of crosscutting (presented in Chapter 3) distinguishes between the concepts of scattering, tangling and also crosscutting.

Authors	Metric	Definition	Observations	Limitations
Sant'Anna et al. (2.4.1)	*Concern Diffusion over Components, Operations or LOC (CDOC, CDO, CDLOC)*	It counts the number of components, operations or LOC addressing a concern	CDOC equivalent to Spread. CDO equivalent to Size.	Used at the implementation and design but not at early stages (such as requirements). Lack of an specific metric for crosscutting.
	Coupling between Components (CBC)	Counts the number of components coupled with a particular component	Proportional to FCD, CDA and CAE.	
	Use Cases Level Interlacing Between Concerns (UCLIBC)	It counts the number of other concerns with which the assessed concerns share at least a component		
	Lack of Concern Cohesion (LOCC)	It counts the number of concerns addressed by the assessed component	Proportional to Degree of Tangling	
Ducasse et al. (2.4.2)	*Size*	It counts the number of internal members of classes associated to a concern.	Equivalent to CDO	Although being the more generic metrics of those described, they have not been applied at any stage of development different from implementation. Lack of an specific metric for crosscutting.
	Touch	It assesses the relative size of a concern or a property (Size of a property divided by the total size of the system)	Proportional to Size and CDO	
	Spread	It counts the number of modules (classes or components) related to a particular concern	Equivalent to CDOC	
	Focus	It measures the closeness between a module and a property or concern	Proportional to Size and Touch. Proportional to DEDI and CONC. Inversely proportional to DISP	
Wong et al. (2.4.3)	*Disparity (DISP)*	It measures how many blocks related to a particular property or feature (or concern) are localised in a particular component	Inversely proportional to CONC, DEDI and Focus	Applied only at the programming level. Lack of an specific metric for crosscutting.
	Concentration (CONC)	It measures how much a feature is concentrated in a component	Proportional to DEDI and Focus. Inversely proportional to DISP	
	Dedication (DEDI)	It quantifies how much a component is dedicated to a feature	Proportional to CONC and Focus. Inversely proportional to DISP	
Eaddy et al. (2.4.4)	*Degree of Scattering (DOS)*	It is defined as the variance of the Concentration of a concern over all program elements with respect to the worst case	Proportional to CDOC and Spread. Inversely proportional to CONC, DEDI and Focus	The analyses performed using these metrics are just focused on the implementation. Lack of an specific metric for crosscutting.
	Degree of Tangling (DOT)	It is defined as the variance of the Dedication of a component for all the concern with respect to the worst case	Proportional to LOCC. Inversely proportional to CONC, DEDI and Focus	
Lopez-Herrejon et al.	*Feature Crosscutting Degree (FCD)*	Counts the number of classes crosscut by all the pieces of advices and introductions of a feature	Equivalent to CDA. Proportional to CBC and CDA.	Metrics very tied to specific programming artefacts. Metrics must be applied to systems modelled using AO techniques.
	Homogeneity Quotient (HQ)	Calculates the parts of code crosscut by advices in relation with the total parts of code crosscut (by advices or introductions)		
Ceccato et al. (2.4.5)	*Coupling on Advice Execution (CAE)*	Counts the number of aspects containing advices possibly triggered by the execution of operations in a given module	Proportional to CBC, FCD and CDA	Metrics focused on the programming level. Metrics must be applied to systems modelled using AO techniques.
	Crosscutting Degree of an Aspect (CDA):	Meaures the modules affected by the pointcuts and by the introductions in a given aspect	Equivalent to FCD. Proportional to CBC and CAE	

Table 7. Comparative table showing the metrics presented in this section

2.5. CONCLUSIONS

In this section the main conclusions extracted from the analysis of the different approaches studied in this chapter are described. This conclusions have drived the work presented throughout this Thesis and that conform its main contributions.

- Firstly, as it is mentioned in the introduction, scattering, tangling and crosscutting are concepts usually described based on our intuition or experience. However, formal definitions of these concepts are mandatory for certain research areas such as the identification of crosscutting or its empirical assessment. In that sense, by analysing the definitions found in the literature, we observed: (i) most of them were focused on a particular abstraction level, namely programming level; (ii) they are not general enough to cover other abstraction levels; (iii) the definition presented by Tonella and Ceccato [36] is the only one that is really used to apply an aspect mining process based on it, (iv) The definition introduced by M. Eaddy and A. Aho in [60] is the only one that have been used to empirically measure concepts related to modularity [62], but as authors explicitelly mention in [60] their definition is based on our definition presented in Chapter 3.
- Secondly, we observed that most of the traditional approaches for mining aspects were focused on the programming level when important architectural decisions have been already made. Moreover, we observed that the combination of different mining techniques obtains better results than the use of an isolated technique. In the last years, there have also appeared several approaches for dealing with aspects at earlier stages of development, e.g. requirements and architecture. However, by the analysis of these techniques, we observed: (i) there were just a few techniques that provide a way to identify early crosscutting concerns (namely EA-Miner [155], Theme-Doc [14]) and they rely on the use of natural language processing techniques so that they may not be applied to other requirements artefacts different from text; (ii) most of the approaches may not be applied at different abstraction levels since they are not based on generic definitions of crosscutting; (iii) most of the approaches analysed deal just with non-functional concerns as crosscutting concerns; however, functional concerns may be also modelled as aspects if they crosscut other concerns; (iv) there was a lack of empirical support that demonstrates that the crosscutting concerns identified by the approaches at the requirements level are harmful for software quality.
- Regarding to the application of aspect-oriented techniques in Product Lines domain, we mainly observed that most of the approaches just deal with variability using an aspect-oriented modelling or programming techniques. However, there was a lack of a process to identify crosscutting features in SPL. Even, the problem was more evident since most of the approaches were focused on the implementation level. There was just an approach (NAPLES [126]) which provides a way to identify crosscutting features (based on the aforementioned EA-Miner tool) at the requirements level. Moreover, the benefits obtained by the incorporation of aspect-oriented techniques to the SPL domain have not been empirically demonstrated in the literature.
- Finally, an analysis of different approaches to empirically assess modularity was performed. These approaches are mainly based on the definition of a set of concern-driven metrics to measure concepts related to scattering and tangling. By studying these approaches, we observed: (i) again, most of them were mainly focused on the programming level (even, sometimes they were very tied to specific programming or language artefacts); (ii) there was just a set of metrics (that defined by Ducasse et al. [58]) defined in a generic way so that they were not tied to the programming level; however,

there was not evidence of the application of these metrics at the requirements level; (iii) although the metrics studied were proposed to assess modularity, there was a lack of an specific metric for crosscutting.

3

Conceptual Framework for the Definition of Crosscutting

Separation of Concerns is one of the key principles in AOSD. This concept was firstly introduced by Dijkstra in [57]. A concern can be defined very generally as a thing in an engineering process about which it cares [76]. Separation of concerns simplifies system development by allowing the development of specialized expertise and by producing an overall more comprehensible arrangement of elements [76]. Then, Separation of Concerns allows a developer to cope with complexity by providing a way to focus on specific parts of the system. As a result, an increase in quality of the systems is obtained since maintainability, reusability, extendibility and flexibility are improved.

Related with the principle of Separation of Concerns is the problem of crosscutting concerns. Crosscutting is usually described in terms of scattering and tangling, e.g. crosscutting is the scattering and tangling of concerns arising due to poor support for their modularization, or a crosscutting concern is a concern for which the implementation is scattered throughout the rest of an implementation [76]. However, sometimes the distinction between these concepts is not obvious and it could lead to misleading situations. As an example, in [109], the authors state: *... the term "crosscutting concerns" is often misused in two ways: To talk about a single concern, and to talk about concerns rather than representations of concerns. Consider "synchronization is a crosscutting concern": we don't know that synchronization is crosscutting unless we know what it crosscuts. And there may be representations of the concerns involved that are not crosscutting.* We use terms in AOSD for which we have a general intuition shared by all the community. This intuition is usually based on our experience. However, sometimes the concepts are not consistent with other ones and the intuition of different people could vary depending on their personal experience.

In this chapter a formal definition of scattering, tangling and crosscutting is provided based on the definition of what we called the Crosscutting Pattern. By a formal distinction between these terms, we can identify when each concept applies. Moreover, in some cases, precise definitions are mandatory in order to provide tool support, for instance in aspect mining domain (i.e. crosscutting identification) [24][52] or for the definition of crosscutting metrics [51].

There are other definitions of crosscutting that have been presented in the literature (shown in Section 2.1). A formal comparison between our definition and those existing is shown in this chapter. This comparison shows that the definition presented here generalises the other ones.

3.1. DEFINITIONS BASED ON CROSSCUTTING PATTERN

In this section, an intuitive notion of crosscutting, which will be generalised in a crosscutting pattern, is firstly introduced. Next, based on this pattern, precise definitions of scattering, tangling and crosscutting and their relation are provided.

For example, assume we have three concerns shown as elements of a source in Figure 70, and four requirements artefacts (e.g. viewpoints or use cases) shown as elements of a target. In this figure, the requirements are related by arrows with the concerns that they address. This picture is consistent with the quotation in the Introduction of this chapter. Intuitively, we could say that *s1* crosscuts *s3* for the given relation between source and target elements. In this figure, we only show two abstraction levels. Multiple intermediate levels between source and target may exist. This picture also fits other intuitive notions of crosscutting, scattering and tangling which we can find in the literature such as *"an aspect in requirements is a concern that crosscuts requirements artifacts; an aspect in architecture is a concern that crosscuts architectural artifacts"* [16] or *"scattering occurs when the design of a single requirement is necessarily scattered across multiple classes and operations in the object-oriented design",*

"tangling occurs when a single class or operation in the object-oriented design contains design details of multiple requirements" both in [14]. As we can see in these citations, the notion of crosscutting, scattering and tangling is based on the relationship of elements at two levels or domains, depicted here as source and target.

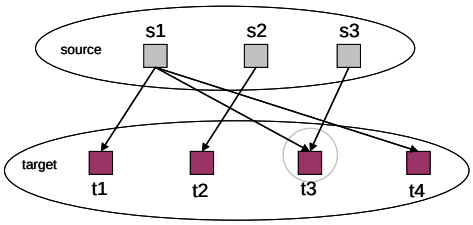


Figure 70. Trace relations between source and target elements

In the following section, we generalise this intuition by means of a crosscutting pattern. Furthermore, we focus on definitions of crosscutting, tangling and scattering.

3.1.1. Crosscutting pattern

The proposition here is that crosscutting can only be defined in terms of *'one thing' with respect to 'another thing'*. Accordingly and from a mathematical point of view, what this means is that we have two domains related to each other through a mapping. The general terms *source* and *target* (as in [132]) are used to denote these two domains and the trace relationship is the mapping relating these domains (Figure 71).

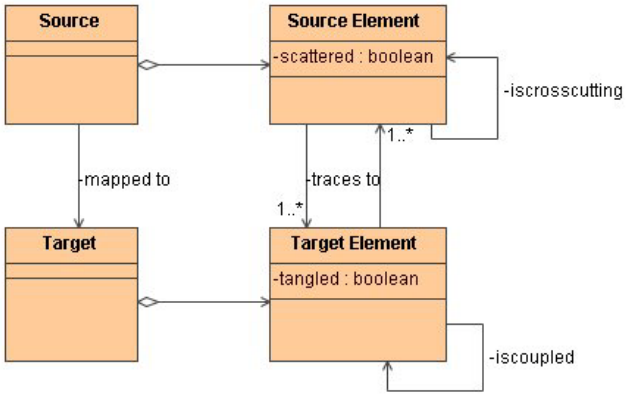


Figure 71. Crosscutting pattern

The term of Crosscutting Pattern [24] is used to denote the situation where source and target are related to each other through trace dependencies. *Pattern* is used as in design patterns [80], in the sense of being a general description of frequently encountered situations [129], [135]. Although the terms 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 modelling, requirements elicitation, architectural design and so on. The only proposition is that crosscutting is defined for two levels, called source and target. This approach can be applied to early phases in software development, e.g. concerns and requirements, but also to other phases near implementation, e.g., a UML design and Java code. In each case the trace relations between the respective source elements and target elements must be defined. In order to illustrate examples of the occurrence of the Crosscutting Pattern, Table 8 shows some situations where it can be applied, with examples of source and target elements.

Examples	Source	Target	We may identify
Ex. 1	Concerns	Requirements Statements	Crosscutting Concerns with respect to mapping to Requirements Statements
Ex. 2	Concerns	Use Cases	Crosscutting Concerns with respect to mapping to Use Cases
Ex. 3	Concerns	Design Modules	Crosscutting Concerns with respect to mapping to Design Modules
Ex. 4	Use Cases	Architectural Components	Crosscutting Use Cases with respect to mapping to Architectural Components
Ex. 5	Use Cases	Design Modules	Crosscutting Use Cases with respect to mapping to Design Modules
Ex. 6	Design Modules	Programming Artifacts	Crosscutting Design Modules with respect to mapping to Programming Artifacts
Ex. 7	PIM artifacts (MDA)	PIM artefacts (MDA)	Crosscutting in PIM artifacts with respect to mapping to PSM artefacts

Table 8. Some examples of source and target domains

In the Crosscutting Pattern, the mappings between source and target elements are captured in trace dependency relationships. In Figure 72, a model of these relationships is shown. Ramesh and Jarke [149] show a more detailed model about traceability where these and other more specific relations are explained. The UML 2.0 specification [27] also covers such relationships. In [18] the authors show a different taxonomy of traceability relationships. Then, in order to determine when two elements from source and target are related to each other, we introduced the trace dependency model shown in Figure 72. This model is based on the previous ones ([149], [27], [18]) covering some important trace relationships of interest for crosscutting identification. The utilization of the Crosscutting Pattern mainly relies on the identification of mappings between two different abstraction levels or domains (Refinement and Representation relations).

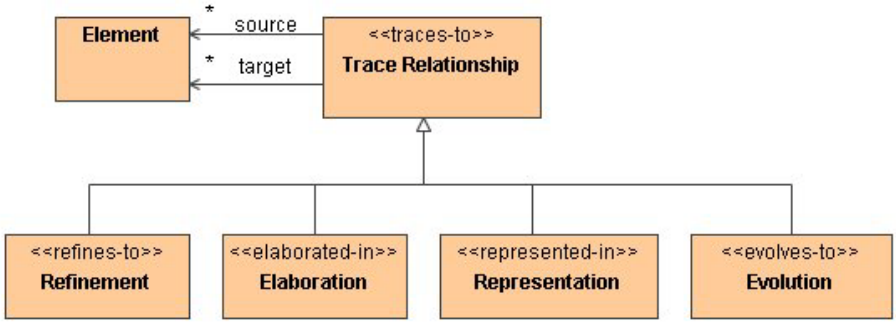


Figure 72. Traceability relationships model

As shown in Figure 72 the model focuses just on the following types of trace relationships: refinement, elaboration, evolution and representation. Other types can be defined depending on the goal of traceability to be achieved. These relationships may be applied to different domains where we can find them. For example:

- ***Refinement***. In software development we usually find refinements between different abstraction levels. For instance, the first abstraction could refer to the concerns a system must deal with and the second one to the software artifacts which address such concerns (this could be extended to any phase in software development). As another example, the Model Driven Architecture (MDA) [132] provides a way to build software based on different refinements or transformations between models or artifacts belonging to different abstraction levels (e.g. Computational Independent Model (CIM), Platform Independent Model (PIM) and Platform Specific Model (PSM)).
- ***Elaboration***. We can find relationships between models of the same abstraction level. In such situations, we elaborate or add some extra information to a model in order to get a

new model. For instance at requirements level we can elaborate a use case based on a previous one.

- ***Representation***. In requirements engineering it is very common to have different representations of the same user needs. For instance, we can represent the requirements as statements extracted from a requirements elicitation document and we can also represent such requirements as viewpoints or use cases. We can link both kinds of representation by means of trace relationships.
- ***Evolution***. With this type of dependencies we can relate gradual changes of software artifacts over time (as in adaptive maintenance). The <<evolves-to>> relationship exists between modified (structural and/or behavioral) elements in artifacts.

The definitions of tangling, scattering and crosscutting are relative to the source and target in the Crosscutting Pattern. Therefore, scattering, tangling and crosscutting are defined as specific cases of the mapping between source and target (denoted as Source x Target). This is explained in the following section.

3.1.2. Concepts based on Crosscutting Pattern

As we can see in Figure 71 there is a multivalued function from source elements to target elements. $f': S \longrightarrow T$ such that if $f'(s) = t$ then there exists a trace relation between s and t (where s and t are source and target elements respectively).

Analogously, we can define another multivalued function g' that can be considered as the inverse of *f'*. $g': T \longrightarrow S$ such that if $g'(t) = s$ then there exists a trace relation between s and t. If *f'* is not a surjection, we consider that Target is the range of *f'*.

Obviously, *f '* and *g'* can be also represented as single-value functions considering that the codomains are the set of non-empty subsets of Target and the set of non-empty subsets of Source respectively.

Let $f: Source \longrightarrow \mathscr{P}(Target)$ and $g: Target \longrightarrow \mathscr{P}(Source)$ be these functions defined by:

$\forall s \in Source, f(s) = \{t \in Target$ / *there exists a trace relation between s and t*$\}$ and

$\forall t \in Target, g(t) = \{s \in Source$ / *there exists a trace relation between s and t*$\}$.

The concepts of scattering, tangling and crosscutting are defined as specific cases of these functions.

When *g'* is not an injective function, we can define the concept scattering. So, scattering occurs when, in a mapping between source and target, a source element is related to multiple target elements.

Definition 1. [Scattering] We say that an element s ϵ Source is **scattered** if card(f(s))>1.

When f' is not an injective function, we can use the term tangling. So, Tangling occurs when a target element is related to multiple source elements.

Definition 2. [Tangling] We say that an element t ϵ Target is **tangled** if card(g(t))> 1.

There is a specific combination of scattering and tangling which we call crosscutting. Crosscutting occurs when a source element is scattered over various target elements and at least one of these target elements is tangled.

Definition 3. [Crosscutting] Let s1, s2 ϵ Source, s1 ≠ s2, we say that s1 **crosscuts** s2 (s1 cc s2) if:

a) card (f(s1)) > 1

b) $\exists t \in f(s1)$: $s2 \in g(t)$

We do not require that the second source element (s2) is scattered. In that sense, our definition is not symmetric as definition in [129] (see Section 2.1.1).

Using **Definition 3** (definition of crosscutting), and the next **Lemma**, we can express crosscutting just in terms of function f as follows:

Lemma 1. Let s1, s2 ∈ Source, s1 ≠ s2, then

s1 crosscuts s2 if and only if card(f(s1)) > 1 and f(s1)∩ f(s2) ≠∅.

Proof.

$$s1\ crosscuts\ s2$$

$$\Leftrightarrow card\left(f(s1)\right) > 1\ \wedge\ \exists\, t\ \in f(s1) : s2 \in g(t)$$

$$\Leftrightarrow card\left(f(s1)\right) > 1\ \wedge\ \exists\, t\ \in f(s1) : t \in f(s2)$$

$$\Leftrightarrow card\left(f(s1)\right) > 1\ \wedge\ f(s1) \cap f(s2) \neq\ \emptyset$$

□

In next sections, the name BCH-definition (Berg, Conejero and Hernández) is used to denote the definition presented in this section. We use this name to distinguish the definition of others presented in Section 2.1.1.

3.2. COMPARISON WITH OTHER DEFINITIONS

In this section we formally compare BCH definition with others existing in the literature. These other definitions have been introduced in Section 2.1.1.

3.2.1. Definition by Masuhara and Kiczales

The notion of crosscutting provided in [129] is focused on programming level, and it is based on two source languages A and B (one of them being aspect-oriented) and a target one called X (resulting in the weaving process of A and B). The authors take as input two different programs p_A and p_B written in A and B respectively. Then they define the term projection as follows: *"for a module m_A (from p_A), we say that the projection of m_A into X is the set of join points identified by the A_{ID} elements within m_A"*. Then, crosscutting is defined as follows: *For a pair of modules m_A and m_B we say that m_A crosscuts m_B with respect to X (the result domain) if and only if their projections onto X intersect, and neither of the projections is a subset of the other*. According to this definition crosscutting is a symmetric property. For the rest of section, the term MK-definition (Mezini and Kiczales) is used to denote this definition of crosscutting.

In the following paragraphs, a comparison between BCH and MK definitions is shown. This comparison proves that MK-definition of crosscutting is a particular case of BCH-definition (presented in Section 3.1.2).

According to MK-definition, let assume that

- *Source* = {m_A : m_A is a module of program p_A} ∪ {m_B : m_B is a module of program p_B}
- *Target = {join points of X}*
- *f: Source* $\longrightarrow$ $\mathscr{P}$*(Target)* defined by *f(s)* is the projection of *s* onto *X*

This definition of *f* is independent of the fact that *s* will be a module of p_A or a module of p_B.

Next, a theorem is defined showing that any crosscutting situation detected by MK-definition in the context defined in [129] can be also detected with BCH-definition. The corresponding demonstration of the theorem is also provided.

Theorem 1. If there is a crosscutting situation using the MK-definition then there is also crosscutting using BCH-definition.

Proof. If there is a crosscutting situation using the MK-definition then there is a pair of modules m_A and m_B such that:

1. $f(m_A) \cap f(m_B) \neq \varnothing$
2. $f(m_A) \not\subset f(m_B)$
3. $f(m_B) \not\subset f(m_A)$

Obviously $card(f(m_A)) > 1$.

If $card(f(m_A)) \leq 1$, as $f(m_A) \cap f(m_B) \neq \varnothing$ then $f(m_A) \subset f(m_B)$ and it is not true (since the hypothesis is that there is crosscutting according to MK-definition). Thus, $card(f(m_A) > 1$

Analogously, $card(f(m_B) > 1$.

Applying Lemma 1 (in Section 3.1.2), we have that m_A crosscut m_B according Definition 3. □

Theorem 1 shows that MK-definition can be seen as a particular case of BCH-definition, being MK projections the mapping between Source and Target. Since MK-definition is focused on implementation level, we can say that definition based on Crosscutting Pattern is a generalisation of it. This definition can be applied to any level or domain so that crosscutting can be identified in it.

Since BCH-definition does not require that $f(m_A) \not\subset f(m_B) \wedge f(m_B) \not\subset f(m_A)$, this definition is less restrictive than MK[6]. BCH definition only requires that the cardinality of the projection of m_A onto X is larger than 1 (scattering), but the cardinality of m_B onto X can be larger or equal than 1.The implications of this statement are important because of the set of crosscutting cases each definition could cover. That idea implies there would be some cases of crosscutting which BCH-definition identifies whereas MK-definition does not. For example, certain tracing cases cannot be identified as crosscutting with MK-definition but with BCH, as it is shown below.

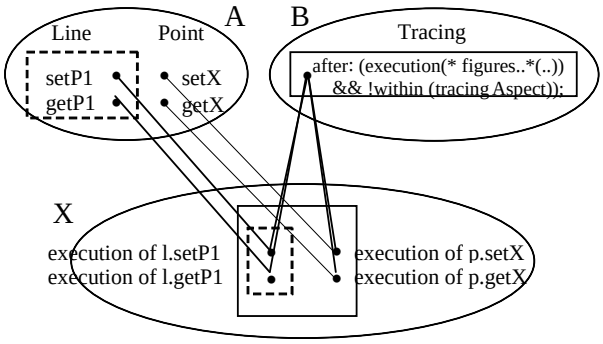


Figure 73. Projections of Line and Display advice according to Masuhara and Kiczales's definition

[6] Note that $f(mA) \not\subset f(mB) \wedge f(mB) \not\subset f(mA)$ is equivalent to $f(mA) \setminus f(mB) \neq \varnothing \wedge f(mB) \setminus f(mA) \neq \varnothing$.

In [129], the authors use the canonical figures-display example [110] to illustrate the application of their definition. This example can be also seen as a concrete application of the Observer Pattern defined in [80]. However, instead of considering the Display concern we may be interested in tracing the execution of all methods of Point or Line classes (Figure 73).

In that case, MK-definition is applied as follows: projection of Line class onto X includes the execution of all methods of Line. The same is true for Point class. On the other hand, projections of advice include execution of all methods of Line and Point classes (in AspectJ execution join points are within the projection of the class that defines the method, as the authors explain in [129]). We can easily observe that projections of Line or Point are a subset of advice's one. Then, according to MK-definition, subset condition is not accomplished in such an example and Line and Tracing do not crosscut each other. However, tracing is a well-know crosscutting concern widely accepted by the community and the literature (e.g. in [76] [85][117]). We could consider other monitoring techniques such as logging or profiling as similar examples [117]. However, as BCH-definition does not require the subset condition, it identifies crosscutting in such a case. Note that BCH definition just focuses on cardinality of mA and intersection of both projections (mA and mB). This will be illustrated in next chapter by means of the different representations of the mappings between source and target.

To sum up the comparison between these definitions, the main differences between them are shown as follows:

- Since BCH-definition may be applied to any model or domain, it generalises MK-definition.
- BCH-definition is less restrictive than MK since it does not require the subset condition.
- BCH-definition does not consider crosscutting being a symmetric property whereas MK-definition does.

The applicability of the definitions above depends on the goal of the crosscutting analysis. As an example, in Chapter 5 different applications of BCH-definition are shown.

3.2.2. Definition by Mezini and Ostermann

In [135] the authors define crosscutting as a relation between two models resulting from a decomposition of a software system (see Section 2.1.2). In particular, they define crosscutting as follows: *two models, M and M', are said to be crosscutting, if there exist at least two sets o and o' from their respective projections, such that* $o \cap o' \neq \varnothing$, *and neither* $o \subseteq o'$, *nor* $o' \subseteq o$. The term MO-definition (Mezini and Ostermann) is used to denote this definition in the rest of document.

Next, we formally compare MO-definition with BCH showing that the former is a particular case of the latter. Again, the canonical figures-display example is used to show an example of the differences between both definitions.

According to MO-definition (see the whole details of the model in Section 2.1.2) let assume that:

- *Source* = Model Space
- *Target = ACS (Abstract concern space)*
- *Source elements = Concerns* $\{c_1 \ldots c_n\}$ / $c_i \in$ MS (e.g. black concern in color model of example shown in Section 2.1.2)
- *Target elements = Artefacts* $\{o_1 \ldots o_m\}$ / $o_j \in$ ACS (e.g. a black big circle of example shown in Section 2.1.2)

- *f: Source* $\longrightarrow \mathcal{P}(Target)$ defined by $f(c_i) = o_i / o_i$ is the projection of concern c_i onto *ACS,* i.e. subset of ACS which addresses the concern c_i (all black figures of example shown in Section 2.1.2).

Again, we define a theorem relating MO-definition and BCH. This theorem is demonstrated in the following paragraphs.

Theorem 2. If there is a crosscutting situation using the MO-definition then there is also crosscutting using BCH-definition.

Proof. If there is a crosscutting situation using the MO-definition then there is a pair of modules M and M' and a pair of concerns $c_i \in M$ and $c_j \in M'$ such that:

1. $f(c_i) \cap f(c_i) \neq \varnothing$
2. $f(c_i) \not\subset f(c_j)$
3. $f(c_j) \not\subset f(c_i)$

Since MK-definition and MO-definition are defined using the same conditions, we may apply the same reasoning (already applied for MK-definition) to demonstrate Theorem 2.

Again, if $card(f(c_i)) \leq 1$, as $f(c_i) \cap f(c_j) \neq \varnothing$ then $f(c_i) \subset f(c_j)$ and it is not true (the hypothesis is that there is crosscutting according to MO-definition). Thus, $card(f(c_i)) > 1$. The same could be said for $card(f(c_j))$ so that $card(f(c_j)) > 1$.

Then, applying Lemma 1 (in Section 3.1.2), we have that c_i crosscut c_j according to Definition 3. □

This fact implies that BCH-definition could identify crosscutting situations that MO does not. In order to show an occurrence of this situation we can focus on the same example shown in Section 3.2.1 (the canonical figures-display example with tracing functionality). Let assume that we have two classes (Line and Point with the methods setP1, setP2 and setX and setY respectively) and we want to trace all the methods executed in the program. In this example, the same notation used to explain MO-definition has been used. In that sense, the methods of the different classes have been represented using figures. In particular, triangles have been used to represent the methods of the Line class whilst methods of the Point class are represented by squares.

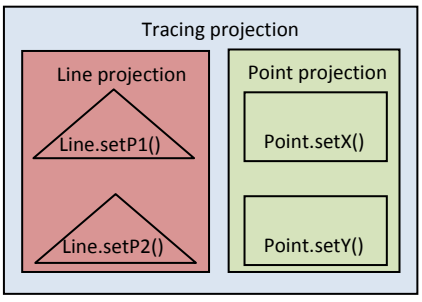


Figure 74. Projections of figures and tracing concerns

Taking as input the example illustrated in

Figure 74, we select three different concerns to define their projections:

- c_1 = Line
- c_2 = Point
- c_3 = Tracing

Based on these concerns, their projections include the method executed to address each concern:

- Projection (c_1) = {setP1, setP2}
- Projection (c_2) = {setX, setY}
- Projection (c_3) = {setP1, setP2, setX, setY}

Using MO-definition we can see that:

1. $f(c_1) \cap f(c_2) \neq \emptyset$
2. $f(c_2) \cap f(c_3) \neq \emptyset$
3. $f(c_2) \cap f(c_3) \neq \emptyset$
4. $f(c_1) \subset f(c_3)$
5. $f(c_2) \subset f(c_3)$

Since the projections of c_1 *and* c_2 are subsets of the projection of c_3, MO-definition is not fulfilled and, thus, we can say that tracing is not crosscutting the rest of concerns. However, using BCH-definition, the conditions needed are the following:

1. $card(f(c_3)) > 1$
2. $f(c_1) \cap f(c_3) \neq \emptyset$
3. $f(c_1) \cap f(c_3) \neq \emptyset$

Then, using BCH-definition we can assure that tracing concern is crosscutting Line and Point. In this case, since we have used a method-based granularity level, we could say that Line and Point are also crosscutting tracing since these concerns are also scattered and tangled with the tracing concern.

We summarise now the main differences between BCH and MO definitions:

- Since BCH-definition may be applied to any model or domain, it generalises MO-definition.
- BCH-definition is less restrictive than MO since it does not require the subset condition.
- BCH-definition does not consider crosscutting being a symmetric property whereas MO-definition does.

3.2.3. Definition by Tonella and Ceccato

In [177] Tonella and Ceccato use Formal Concept Analysis (FCA) to represent the relation between concerns and source code units. Using these relations in [36] they provide a definition of crosscutting (summarised in Section 2.1.3). From now, the term TC-definition (Tonella and Ceccato) is used to denote this definition.

TC-definition is based on the concept of concern seed. A concern seed is a single source-code entity, such as a method, or a collection of such entities, that strongly connotes a crosscutting concern. A candidate seed is a potential concern seed (see Section 2.1.3.4).

Formally, a concept **c** is considered a candidate seed iff [36]:

- Scattering: $\exists m, m' \in \beta(c) \mid pref(m) \neq pref(m')$
- Tangling: $\exists m \in \beta(c), \exists m' \in \beta(c') \mid c \neq c' \wedge pref(m) = pref(m')$

where *pref(p)* is the fully scoped name of the class containing the method p.

The first condition (scattering) requires that more than one class contributes to the functionality associated with the given concept. The second condition (tangling) requires that the same class addresses more than one concern.

In following paragraphs a formal comparison between TC-definition and BCH is shown. This comparison shows how TC-definition is a particular case of BCH-definition.

According to TC-definition, let assume that

- *Source* is the set of concepts
- *Target* is the set of classes
- *f: Source* $\longrightarrow \mathcal{P}(Target)$ defined by $f(c) = \{pref(m) / m \in \beta(c)\}$

f(c) is the set of classes containing methods that labelled the concept **c**.

Next, it is proven that any crosscutting situation detected by TC-definition in the context defined in [36] can be also detected with BCH-definition.

Theorem 3. If there is a crosscutting situation, the use of TC-definition is equivalent to the use of BCH-definition.

Proof.

1. We prove that if there is a crosscutting situation using TC-definition then there is also crosscutting using BCH-definition.

   The TC-definition says that

   1. $\exists$ m, m' $\in \beta$(c) / pref(m) ≠ pref(m')
   2. $\exists$ m $\in \beta$(c), $\exists$ m' $\in \beta$(c') / c ≠ c' $\wedge$ *pref(m)= pref(m')*

   Obviously, *card* (*f(c)*) > 1, because *pref(m), pref(m')* $\in$ *f(c)* and, considering item 1 in TC-definition, we have that pref(m) ≠ pref(m')

   Considering item 2, we have that *pref(m)* $\in$ *f(c)* $\cap$ *f(c')* $\Rightarrow$ *f(c)* $\cap$ *f(c')* $\neq \varnothing$.

   Applying Lemma 1 in Section 3.1.2, we have that *c* crosscut *c'* according to BCH-definition.

2. We prove that if there is a crosscutting situation using BCH-definition then there is also crosscutting using C-definition.

   The BCH-definition says that $\exists s1 \neq s2 \in Source$ such that

   a) card (f(s1)) > 1

   b) $\exists t \in f(s1): s2 \in f^{-1}(t)$

   Considering item a, we have that *f(s1)* has at least two different elements (e.g. classes cl1 and cl2).

   $\exists\, cl1 \in f(s1) \wedge cl2 \in f(s1) \Rightarrow$

   $\Rightarrow \exists m \in \beta(s1): cl1 = pref(m) \wedge \exists m' \in \beta(s1): cl2 = pref(m') \wedge m \neq m' (since\ cl1 \neq cl2)$

   Considering item b,

   $\exists\, cl \in f(s1): s2 \in f^{-1}(cl) \Rightarrow cl \in f(s1) \wedge cl \in f(s2) \Rightarrow$

   $\Rightarrow \exists m, m': cl = pref(m) = pref(m') \wedge m \in \beta(s1) \wedge m' \in \beta(s2)$

   Then, s1 crosscuts s2 according to TC-definition □

Finally, in order to sum up the comparison between TC-definition and BCH, the main differences and similarities are shown:

- Since BCH-definition may be applied to any model or domain, it generalises TC-definition.
- BCH-definition and TC-definition are equivalent since they identify the same crosscutting cases. Whenever a situation of crosscutting is detected by TC-definition, it is also detected by BCH.

- Both definitions consider scattering and tangling as needed but not sufficient conditions to have crosscutting. In other words, a concern scattered among several modules could not indicate the presence of crosscutting. Some of the modules should present tangling to consider crosscutting.
- Both definitions do not consider crosscutting being a symmetric property.
- A minor difference is detected between both definitions. While BCH-definition explicitly indicates that crosscutting occurs whenever scattering and tangling is detected in the same elements, TC-definition do not make explicit this requirement. However, although this requirement is not mentioned in TC-definition, we consider that the requirement is implicitly considered in the definition.

3.2.4. Definition by Lieberherr

In Section 2.1.4, we also showed other definitions of crosscutting mentioned in the literature, such as those introduced by Fox [78], Lieberherr [68] or Eaddy [63]. The notion of crosscutting introduced by Lieberherr is not formally defined. However, as it is mentioned in Section 2.1.4, this definition may be considered as a special case of Mashuara and Kiczales definition so that the analysis performed for MK-definition is also applicable for Lieberherr's definition.

The definition presented by Fox is formally defined in terms of design or programming classes, so that we could say that it is tied to the latest phases of development. Moreover, it considers scattering as the only condition needed to have crosscutting. In that sense, our definition is more restrictive since we consider crosscutting as a special combination of scattering and tangling. This fact implies that there could be situations identified as crosscutting by Fox's definition that our definition could consider just scattering. An example of these situations is shown in 4.2.2.

Finally, Eaddy et al. used the crosscutting pattern presented in current section to define scattering and tangling. However, they also consider scattering as the only condition needed to have crosscutting so that tangling is not considered.

3.3. CONCLUSIONS

Finally, this section summarises the main conclusions extracted from the comparison of our definition with the existing in the literature.

As it has been formally demonstrated, our definition generalises the definition presented by Masuhara and Kiczales [129] and also the presented by Mezini and Ostermann [135]. Note that the subset condition required by these definitions makes that well-know situations of crosscutting may be missed. In addition, these definitions have been only applied at the programming level, even being defined in terms of specific programming artefacts. Finally, unlike these definitions, our definition does not consider crosscutting as being a symmetric property.

Regarding to the definition presented by Tonella and Ceccato [177], we formally demonstrated that both definitions are equivalent since they identified the same crosscutting situations. However, our definition is generic so that it may be applied to any model or abstraction level. Both definitions also consider crosscutting as a not symmetric property.

The rest of definitions analysed may be considered as specific cases of the three aforementioned, so that the main assumptions for these definitions may be made. In addition, these definitions are also tied to the programming level so that its application to other abstraction levels has not been shown in the literature.

4

Representation of Crosscutting

This chapter decribes how the mappings existing between source and target domains (function *f* introduced in the previous chapter) can be represented by means of dependency graphs and an extension to traceability matrices. In the former the trace relationships between source and target elements are represented in form of graph links. In the latter trace relations are captured in a dependency matrix, where each cell represents a mapping between the source and target elements of the corresponding row and column, respectively. The utilization of the dependency matrix allows tool support (just using simple matrix operations) for many applications (see Chapter 5) so that we mainly utilize this representation throughout this Thesis. From the dependency matrix, other different matrices are derived, namely scattering, tangling and crosscutting matrices. This is illustrated with some examples. The final crosscutting matrix denotes the occurrence of crosscutting.

4.1. DEPENDENCY GRAPHS

A dependency graph is a directed graph used to represent dependencies between objects. In this graph, an edge linking two nodes (A and B) usually represents that the node A depends on node B. Dependency graphs have been widely used in software engineering tasks such as program understanding, debugging, testing and maintenance. They have been mainly used at programming level (Program Dependency Graphs, PDG) showing control and data dependencies between code artefacts, like Ferrante et al. do in [70]. Even some approaches have emerged to adapt these graphs to represent aspect-oriented programs, e.g. the work presented by Zhao and Rinard in [193]. In Figure 75 an example of dependency graph is illustrated.

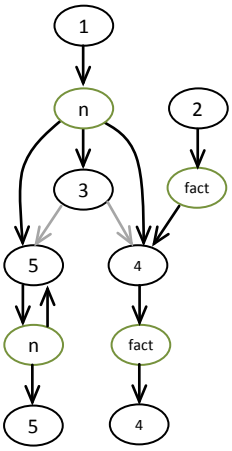


Figure 75. An example of dependency graph (extracted from [13])

In other works, similar graphs have been used to locate features implementation. As an example, in [152] Robillard and Murphy introduce concerns graphs, where each edge relates a concern with a code artefacts addressing or contributing to this concern. The concept of concern graph is also used in [63] where Eaddy also uses the relations between concerns and code artefacts to establish a concern model (see Figure 12) and different modularity metrics (both shown in Chapter 2). Concern graphs are closely related to the Crosscutting Pattern defined in Chapter 3. Note that an edge in the concern graph would be an instance of the trace link which relates source and target domains in the crosscutting pattern.

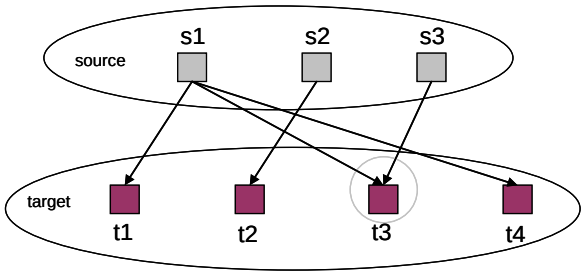


Figure 76. A dependency graph based representation of source and target elements

Then, these graphs may be used to deal with the traceability links introduced in the Crosscutting Pattern. As it is shown in Section 3.1 (and repeated in Figure 76), a very intuitive and simple representation of mappings between source and target can be made by means of dependency graphs, so that crosscutting may be easily identified and represented by using such graphs. We may distinguish several cases of mappings according to their cardinality between source and target [24]:

- Injection: distinct source elements are related to distinct target elements (i.e. a *one-to-one*[7] function). In Figure 76, we can see how s2 is only related to t2 (an example of injection).
- Scattering: a source element is related to multiple target elements (i.e. a one-to-many function). Observe how s1 is related to three different target elements in Figure 76.
- Tangling: a target element is related to multiple source elements (i.e. a *many-to-one* function). In the same example, t3 is related to s1 and s3.
- Crosscutting: a target element is involved both in scattering and tangling (e.g. t3. Thus, s1 crosscuts s3).

However, other representations of crosscutting may be possible. For instance, by means of traceability matrices, we can represent dependencies between source and target elements. In the next section we show such matrices in order to identify and represent crosscutting. Matrix representation allows building automatic tools to find out crosscutting based on simple matrix operations.

4.2. MATRIX REPRESENTATION

In terms of linear algebra, the relation between source elements and target elements can be represented in a special kind of traceability matrix [56] that we called *dependency matrix*. 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 depends on the source element. Scattering and tangling can be easily visualized in this matrix (see the examples below).

A new auxiliary concept called *crosscutpoint* is defined and used in the context of dependency diagrams, to denote a matrix cell involved in both tangling and scattering. The existence of one or more *crosscutpoints* denotes the presence of crosscutting.

[7] This name is best avoided, since some authors understand it to mean a bijective function

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 4.2.1 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 or abstraction (intra-level dependencies). In some sense, the *coupling matrix* is related to the *design structure matrix,* introduced in [12] by Baldwin and Clark. 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). Anyway, coupling information may be also used to extract or infer new mappings between source and target elements. In particular, in Section 5.1.4, an example showing how to use this information using use cases dependencies is provided.

We now give an example and use a *dependency matrix* (Table 9) and *crosscutting matrix* (Table 10) to visualize the definitions. In the *dependency matrix* 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.

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	

Table 9. Example dependency matrix with tangling, scattering and one crosscutpoint

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

Table 10. Crosscutting matrix derived from dependency matrix of Table 9

In this example, we have one scattered source element s[1] and one tangled target element t[3]. Applying our definition of crosscutting, we obtain the *crosscutting matrix*. In this 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 set of one of these elements, namely t[3]). The reverse is not true: the crosscutting relation is not considered as a symmetric property. The example shown in the *dependency matrix* of Table 9 is the same described in the dependency graph of Figure 76.

4.2.1. Building the Crosscutting Matrix

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 *crosscutpoints*, in this example there are 8 points (see Table 11; the dark grey cells).

The *crosscutting matrix* (Table 12) shows 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 is a necessary condition for crosscutting).

		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	

Table 11. Example dependency matrix with tangling, scattering and several crosscutpoints

		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

Table 12. Crosscutting matrix derived from dependency matrix of Table 11

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 (for our example in Table 11, these matrices are shown in Table 13 and Table 14 respectively):

- 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).

		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

Table 13. Scattering matrix for dependency matrix in Table 11

		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

Table 14. Tangling matrix for dependency matrix in Table 11

Now, the *crosscutting product matrix* is defined, 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*: $ccpm = sm \times tm$, where $ccpm[i][k] = sm[i][j] \cdot tm[j][k]$.

As it has been aforementioned, in the *crosscutting product matrix*, the cells denote the frequency of crosscutting. This can be used for quantification of crosscutting (crosscutting metrics). In Section 5.4, we use all these matrices to establish a whole set of concern-oriented metrics which allows the developer to assess modularity of software systems [51]. These metrics may be also used for assisting the developer to anticipate important decisions about the selected decomposition. As an example, in Section 5.5 we show how maintainability attributes are directly related to modularity.

In the *crosscutting matrix*, a matrix cell denotes the occurrence of crosscutting; it abstracts from the frequency 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) /\ (i ≠ j) then 1 else 0.*

The *crosscutting product matrix* and the *crosscutting matrix* for the example are given in Table 15 and Table 16 respectively. 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.

		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

Table 15. Crosscutting product matrix for dependency matrix in Table 11

		source				
		s[1]	s[2]	s[3]	s[4]	s[5]
Soure	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

Table 16. Crosscutting matrix obtained for the example

As it may be observed in the crosscutting matrix of Table 15, 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). The process to obtain the final crosscutting matrix is summarised in Figure 77.

The crosscutting pattern presented in previous chapter, has different application areas, such as the identification of crosscutting or the definition of aspect-oriented metrics (presented in Chapter 5). However, its generic property is one of its main contributions. Since the crosscutting pattern is not defined in terms of any specific development artefact, it is not tied to any abstraction level. That implies that it may be used at any development phase just selecting the corresponding source and target domains. As an example, a first analysis of crosscutting was presented in [23] and [24] at the design and requirements levels, respectively. Note that modularity is not restricted to any abstraction levels, e.g. class diagrams at design. Any modelling language introduces constructs for grouping entities as a way of modularity. For instances, use case models groups functionalities into use cases. This is why we generalise the concept of crosscutting using source and target domain. Thus,

considering the different models that may compose a system (concern models, use case models, structural models, behavioural models, ...) trace relations between them are analysed to discover concerns that crosscut to any of these models.

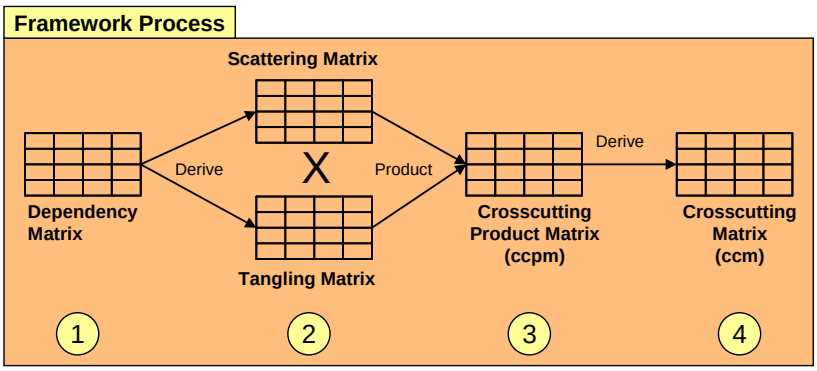


Figure 77. Overview of the steps to obtain the crosscutting matrix

4.2.2. Case Analysis of Crosscutting

Once scattering, tangling and crosscutting have been defined, we may discuss now a case analysis of possible combinations according to our definition. Assuming that the properties tangling, scattering, and crosscutting may be true or false, there are 8 combinations (see Table 17) [24]. Each case addresses a certain mapping from source to target. However, crosscutting requires tangling and scattering, which eliminates 3 of these combinations (Cases 6, 7 and 8: not feasible). There are five feasible cases listed in the table. In Case 4, we have scattering and tangling in which no common elements are involved. With our definition of crosscutting, we disentangle the cases with just tangling, just scattering and on the other hand crosscutting. Our proposition is that tangling and scattering are necessary but not sufficient conditions for crosscutting.

	tangling	scattering	crosscutting	feasibility
Case 1	no	no	no	feasible
Case 2	yes	no	no	feasible
Case 3	no	yes	no	feasible
Case 4	yes	yes	no	feasible
Case 5	yes	yes	yes	feasible
Case 6	no	no	yes	not feasible
Case 7	no	yes	yes	not feasible
Case 8	yes	no	yes	not feasible

Table 17. Feasibility of combinations of tangling, scattering and crosscutting

In order to illustrate the different possibilities, we discuss now how to apply the framework to some simple examples. The first examples aim at showing the combination of tangling or scattering and not crosscutting. The next examples show different cases where crosscutting occurs as the combination of scattering and tangling. All the examples shown in this section are focused on the design or programming level [23]. However, in Chapter 5 the application of our conceptual framework to other abstraction levels is illustrated, namely at the requirements level.

4.2.2.1. Crosscutting without scattering or tangling: is it feasible?

This section shows two examples where the Crosscutting Pattern presented helps in disentangling the situations of having scattering, tangling and crosscutting.

The Binary Search Tree

The first example is extracted from [177], where Tonella and Ceccato use the definition summarised in Section 2.1.3 to identify crosscutting concerns at the programming level. The example application consists of several classes that implement a simple Binary Search Tree. The main functionalities of the application are the *insertion* of elements in the data structure and the *search* of a particular element. The class diagram is shown in Figure 78.

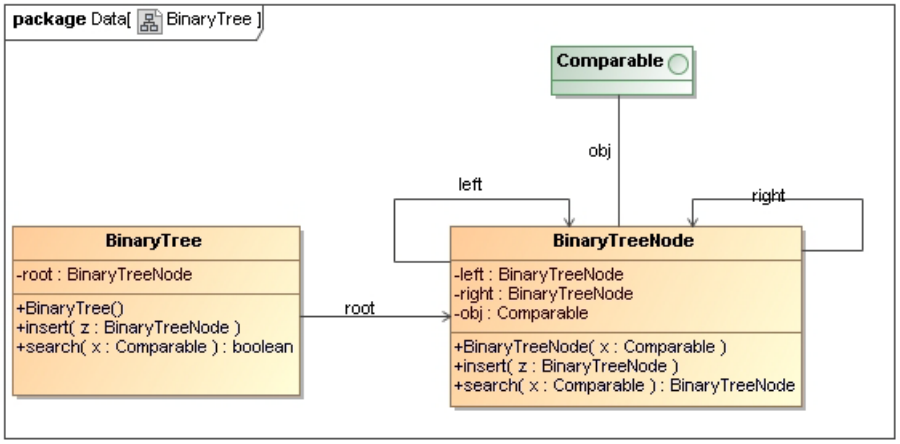


Figure 78. Binary Search Tree class diagram

In [177], the authors present a table where the two main concerns of the system, *insertion* and *search*, are related to the methods that contribute to such functionalities. Assuming that the *search* is performed in a pre-loaded binary tree, these methods are presented in Table 18.

method	**Insertion**
m1	BinaryTree.BinaryTree()
m2	BinaryTree.Insert(BinaryTreeNode)
m3	BinaryTreeNode.insert(BinaryTreeNode)
m4	BineryTreeNode.BinaryTreeNode(Comparable)
	Search
m1	BinaryTree.BinaryTree()
m5	BinaryTree.search(Comparable)
m6	BinaryTreeNode.search(Comparable)

Table 18. Relation between the main concerns and the executed methods for these concerns

Having the concerns and the methods that contribute to them as source and target domains respectively, we build the dependency matrix shown in Table 19. We have selected methods as the granularity level for the target elements. Note that the mappings between source and target elements represented in this matrix are those originally suggested by the authors in [177]. As we can see in this matrix, the BinaryTree.BinaryTree() method is executed for both the *insertion* and the *search* concerns. The existence of this method implies that our framework identifies both concerns as crosscutting (see crosscutting matrix in Table 20).

		methods						
		m1	m2	m3	m4	m5	m6	
concerns	insertion	1	1	1	1	0	0	S
	search	1	0	0	0	1	1	S
		T	NT	NT	NT	NT	NT	

Table 19. Dependency matrix for the BST application

		concerns	
		insertion	search
concerns	insertion	0	1
	search	1	0

Table 20. Crosscutting matrix for the BST application

The example explained above belongs to the fifth category of the eight possible combinations presented in Table 17 (i.e. scattering, tangling and crosscutting). However, we may find different situations with just scattering or just tangling and not crosscutting. For instance, since in [177] the authors consider the *search* concern having a pre-loaded tree, we do not consider that the constructor of BinaryTree class contributes to such functionality. In that case, we remove the mapping from *search* concern to method m1. The new dependency and crosscutting matrices are shown in Table 21 and Table 22 respectively.

		methods						
		m1	m2	m3	m4	m5	m6	
concerns	insertion	1	1	1	1	0	0	S
	search	0	0	0	0	1	1	S
		NT	NT	NT	NT	NT	NT	

Table 21. New dependency matrix for the BST

		concerns	
		insertion	search
concerns	insertion	0	0
	search	0	0

Table 22. New crosscutting matrix for the BST

As we can see in the dependency matrix of Table 21, we may have source elements scattered over different target elements without having crosscutting. Although usually we encounter scattering and tangling together, the utilization of a formal definition allows the differentiation of these concepts identifying such exceptional situations (with only one of the needed conditions to have crosscutting). This last situation belongs to the third case or category of Table 17.

Even if we consider that the constructor of the BinaryTree class contributes to the searching functionality (*search* concern), we could find a case where a source element is scattered over different target elements and there is not crosscutting. For instance, consider the same BST system explained above without the searching functionality. In that case, the *insertion* concern would be scattered over some methods and classes, we do not consider such a concern as being crosscutting. Obviously, if there is just one concern, it could not crosscut any other concern. However, note that our formal definition of crosscutting works properly in that case (that is what we are proofing).

The Remote Calculator

In order to show a different case with tangling and not crosscutting, a new simple example is shown now: a calculator with remote access. We apply the framework at concern level with respect to the design level (represented in a UML class diagram). The case study consists of a distributed Java application which allows a user to calculate the sum of integer numbers. The distribution is accomplished by means of sockets. The MVC pattern [34] is applied in order to perform a separation of representation and control concerns from the functional concerns of an application. In order to study the crosscutting in this case, we consider three main concerns in the system: *Client side distribution, Server side distribution* and *Calculation*. We take these concerns as source elements in our dependency matrix and the UML design classes are considered to be the target elements.

In Figure 79 a UML class diagram representing the design may be observed. We have developed the main functionality regarding to the socket concerns in a class called SocketConnection. This class just performs the remote connection and sends and receives integer values. We may say that this class has a low cohesion. Depending on the operation (sending or receiving), this class will invoke methods of the other classes. The Model, View and Control classes perform the actions to sum the integer, read user's selections and show the

results on screen respectively. In order to clarify the example, we have called these classes with the name of the role in the MVC pattern that they play. Therefore, the application has a good separation between model (a class with a vector of numbers and which performs the sum), view (a class which shows the result on the screen) and control (a class which reads the user's inputs). Although such classes are coupled by means of method calls, their level of cohesion is high because each class is only addressing its main functionality (concern).

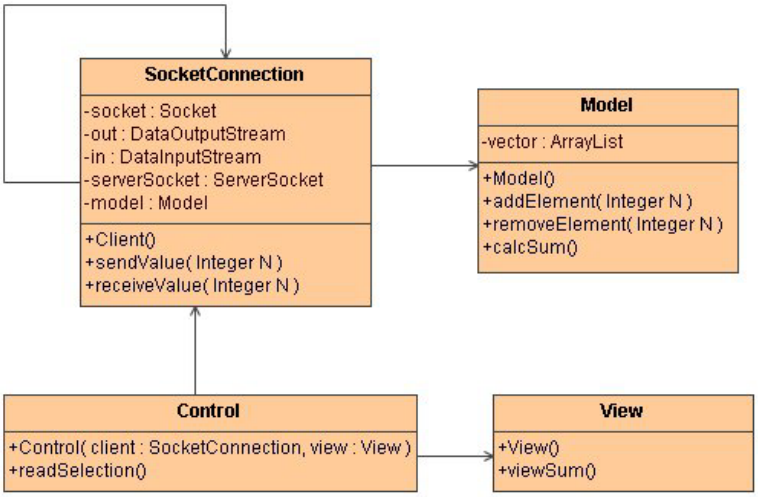


Figure 79. UML class diagram of Remote Calculator

So, taking such a decomposition (in classes) and applying the framework, we obtain the *dependency matrix* shown in Table 23. As we can see in the matrix, concerns *Client side distribution* and *Server side distribution* are tangled in the same class SocketConnection, whereas *Calculation* concern is scattered over the other classes. However, as can be seen in the table, the matrix has no *crosscutpoints*. The resulting *crosscutting matrix* is shown in Table 24: there are no crosscutting concerns in the system. Of course, other decompositions in this example are feasible. Even, different concerns could be selected and the results could vary. Obviously, as a different example, the socket functionality could have been developed in two separated classes (one for the client part and one for the server) or a unique concern could have been used to represent the Distribution functionality. However, the purpose of this example is just to show a situation where the crosscutting pattern could identify just tangling and not crosscutting (although being a strange situation since scattering usually implies tangling and vice verse).

dependency matrix

	Classes				
Concerns	SocketConnection	Model	View	Control	
Client side distribution	1	0	0	0	NS
Server side distribution	1	0	0	0	NS
Calculation	0	1	1	1	S
	T	NT	NT	NT	

Table 23. Dependency matrix for the Remote Calculator

crosscutting matrix with respect to classes

	Concerns		
Concerns	Client side distribution	Server side distribution	Calculation
Client side distribution	0	0	0
Server side distribution	0	0	0
Calculation	0	0	0

Table 24. Crosscutting matrix for the Remote Calculator application

4.2.2.2. When does crosscutting arise?

This section shows different examples where crosscutting exists. The cases selected are well-known examples previously published in the literature: on the one hand a DVD store system shown in [85] and on the other hand some GoF's (Gang of Four) design patterns [80], namely the Mediator and Adapter patterns.

The DVD Store

The DVD store example consists of 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 deeply described in Chaper 1 of the book *Mastering AspectJ, Aspect-Oriented Progrmming in Java*, written by Gradecki and Lesiecki [85]. In this example four concerns are considered: 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 [85] through concern modelling techniques. The object-oriented design of the system is shown in the UML class diagram in Figure 80. 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).

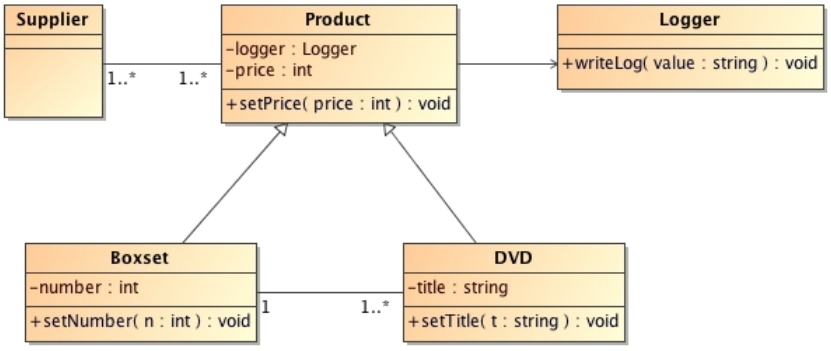


Figure 80. Class diagram of the example system for selling DVD products with logging (based on [85])

Based on an implicit and intuitive notion of crosscutting, Gradecki & Lesiecki [85] state that the logging concern is a crosscutting concern. However, since we are interested in the formal identification of crosscutting concerns, such as this logging concern, our definition of crosscutting has been applied in the example.

As it has been aforementioned, the selected concerns that conforms the decomposition of the source elements are: 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. Then, the dependency matrix established for this example is shown in Table 25. Note that 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.

	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	

Table 25. Dependency matrix for the DVD products system

Then, by applying our definition of crosscutting, the crosscutting matrix for this case is derived. This matrix is shown in Table 26. It may be observed how the logging concern is crosscutting the rest of concerns, but not the other way around (crosscutting is not symmetric)

	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

Table 26. Crosscutting matrix for the dependency matrix in Table 25 (DVD system)

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 now some other case studies where similar crosscutting concerns may be identified. In particular, the application of the crosscutting pattern to two different design patterns is illustrated.

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 81. A more detailed explanation of this example and the Mediator pattern can be found in [80].

As it is stated by Gamma et al. in [80], 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).

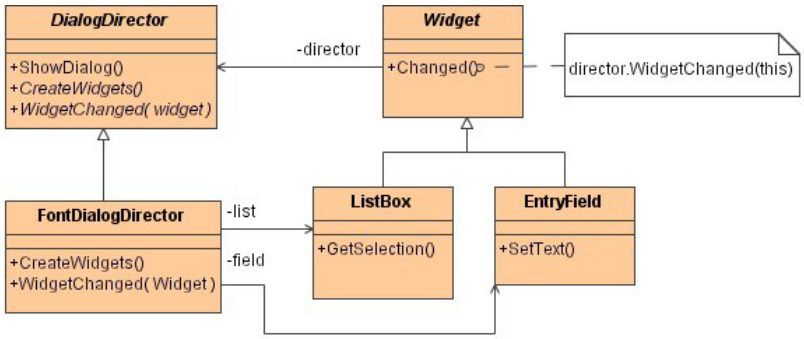


Figure 81. Mediator pattern applied to GUI design [80]

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 the introduced in [167] by Sutton and Rouvellau could be also used to discover the concerns. Having these concerns (as source elements) and the classes of the UML class diagram shown above (as target elements), the dependency and crosscutting matrices are obtained (shown in Table 27 and Table 28, respectively).

	Design Class					
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
Text Field	0	0	0	0	1	NS
Window	1	0	0	0	0	NS
	T	NT	NT	T	T	

Table 27. Dependency matrix for the Mediator Pattern applied to GUI design

	Concern			
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

Table 28. Crosscutting matrix for the Mediator Pattern example

As it may be observed in Table 27, the *DialogDirector* class addresses both the Communication and the Window concerns, because it has methods for showing the dialog and for allowing widgets communication. This is represented by mappings in cells [1,1] and [1,4] of the dependency matrix. The *FontDialogDirector* class only addresses the Communication concern since it is its own 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).

In the crosscutting matrix (Table 28), 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 and suggests the utilization of a different decomposition. In particular, an AspectJ implementation of this design pattern can be found in [93], where Hannemann and Kizales implemented all the GOF's patterns using aspects. This implementation removes the crosscutting from Mediator pattern.

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 [80]. The adaptation of a previously implemented interface to a new required one is known as the Adapter pattern or also a Wrapper. In [80] 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 82.

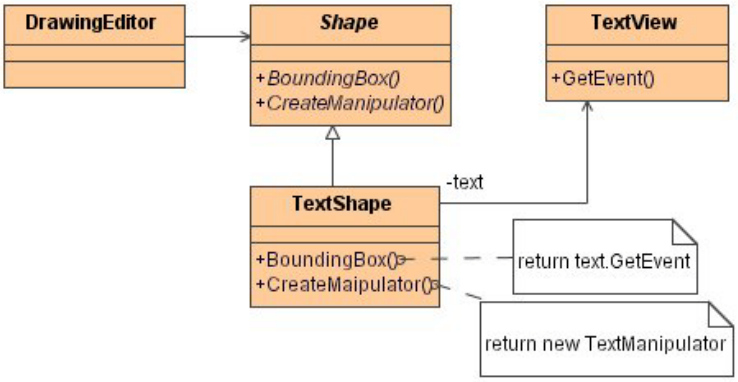


Figure 82. Adapter pattern applied to Drawing Editor [80]

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

	Design Class				
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	

Table 29. Dependency matrix for the Adapter Pattern example

	Concern			
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

Table 30. Crosscutting matrix for the Adapter Pattern example

Taking these decompositions as input, the dependency and crosscutting matrices can be determined as shown in Table 28 and Table 29 respectively. The crosscutting matrix shows that there is no crosscutting in this case.

As it was aforementioned, Hannemann and Kiczales used AspectJ as implementation language to develop these patterns in [93]. However, 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.

As a summary, in many situations, we have tangling, scattering and at the same time crosscutting. With our definitions, we clearly distinguish scattering and tangling from crosscutting and, as we stated in Section 3.1, scattering and tangling are necessary but not sufficient conditions for crosscutting. The analysis depends on the chosen decomposition of source and target, and sometimes other decompositions are feasible.

4.3. CROSSCUTTING AND TRANSITIVITY OF DEPENDENCIES

In this section we consider the transitivity of dependencies between levels and within the same level respectively. Such dependencies are based on different transitive relations that can be observed between source and target elements.

4.3.1. Transitivity of inter-level dependencies

Usually we encounter a number of consecutive levels or phases in software development. As an example, in MDA [129], we have transformations from Platform Independent Models, Platform Specific Models to Implementation Models. From the perspective of software life cycle phases, we could distinguish Domain Analysis, Concern Modelling, Requirement Analysis, Architectural Design, Detailed Design, and Implementation.

We consider here the cascading of two crosscutting patterns: the target of the first pattern serves as source for the second one. For convenience, we call the first target our intermediate level, and our second target just target (see Figure 83) [24].

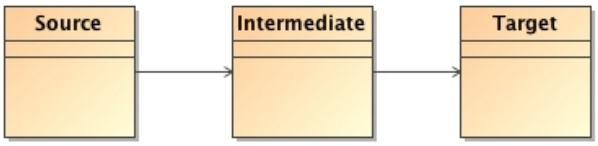


Figure 83. Two Cascaded Crosscutting Patterns

Each of these refinements can be described with a dependency matrix. We describe how to combine two consecutive dependency matrices, in an operation we call cascading. Cascading is an operation on two dependency matrices resulting in a new dependency matrix, which represents the dependency relation between source elements of the first matrix and target elements of the second matrix.

For cascading, it is essential to define the transitivity of dependency relations. Transitivity is defined as follows. Assume we have a source, an intermediate level, and a target. There is a dependency relation between an element in the source and an element in the target if there is some element at the intermediate level that has a dependence relation with this source element and a dependency relation with this target element. In other words, the transitivity dependency relation *R* for source *s*, intermediate level *u* and target *t*, and *card(u)* is the number of elements in *u*:

$$\exists k \in (1..card(u)) : (s[i] \; R \; u[k]) \wedge (u[k] \; R \; t[m]) \Rightarrow (s[i] \; R \; t[m])$$

We can also formalise this relation in terms of the dependency matrices. Assume we have three dependency matrices *m1 :: s ✕ u and m2 :: u ✕ t and m3 :: s ✕ t*, where s is the source, u is some intermediate level, *card(u)* is the cardinality of *u*, and *t* is the target. The cascaded dependency matrix *m3 = m1 cascade m2*.

Then, transitivity of the dependency relation is defined as follows:

$$\exists j \in (1..card(u)): m1[i,j] \wedge m2[j,k] \Rightarrow m3[i,k]$$

In terms of linear algebra, the dependency matrix is a relationship between two given domains, source and target. Accordingly, the cascading operation can be generalised as a composition of relationships as follows. Let Dom_K, *k = 1..n*, be *n* domains, and let f_i be the relationship between domains Dom_i and Dom_{i+1}, *1≤i<n*, denoted as $Dom_i \xrightarrow{f_i} Dom_{i+1}$. Let Source and Target be the domains Dom_1 and Dom_n, respectively. Consequently, we have the following relationship between the domains: $Source \xrightarrow{f_1} Dom_2 \xrightarrow{f_2} Dom_3 \xrightarrow{f_3} \dots Dom_{n-1} \xrightarrow{f_{n-1}} Target$.

As a result, the dependency relationship between the Source and the Target is defined as $DM \equiv f_{n-1} \circ f_{n-2} \circ \dots \circ f_2 \circ f_1$. In this way, the dependency matrix between a source and target is obtained through matrix multiplication of the dependency matrices that represents each f_i, *1≤i<n* .

dependency matrix 1

	use cases			
concern	uc[1]	uc[2]	uc[3]	uc[4]
c[1]	1	0	0	1
c[2]	0	1	0	0
c[3]	0	0	1	1

dependency matrix 2

	architectural component				
use cases	ac[1]	ac[2]	ac[3]	ac[4]	ac[5]
uc[1]	1	0	0	0	1
uc[2]	0	1	0	0	0
uc[3]	0	1	1	0	0
uc[4]	0	0	0	1	1

Table 31. Two dependency matrices that will be cascaded

resulting dependency matrix

	architectural component				
concern	ac[1]	ac[2]	ac[3]	ac[4]	ac[5]
c[1]	1	0	0	1	2
c[2]	0	1	0	0	0
c[3]	0	1	1	1	1

crosscutting matrix

	concern		
concern	c[1]	c[2]	c[3]
c[1]	0	0	1
c[2]	0	0	0
c[3]	1	1	0

Table 32. The resulting dependency matrix and crosscutting matrix based on cascading of the matrices in Table 31

As an example, we explain the cascading of two dependency matrices. The two dependency matrices are shown in Table 31. The first dependency matrix relates *concerns* with *use cases*. The second dependency matrix relates *use cases* with *architectural components*. The

resulting dependency matrix relates concerns with architectural components (see Table 32). This matrix can be used to derive the crosscutting matrix for *concern ✗ concern* with respect to *architectural components*. The crosscutting matrix in Table 32 is not symmetric. Based on this matrix we conclude, for the given dependency relations between *concerns* and *architectural components*, that: concern c[1] is crosscutting concern c[3]; concern c[2] does not crosscut any other concern; concern c[3] is crosscutting concerns c[1] and c[2].

We summarise the cascading operation in Figure 84. From this description it is clear that cascading can be used for traceability analysis across multiple levels, e.g. from concerns to implementation elements, via requirements, architecture and design (c.f. [167]).

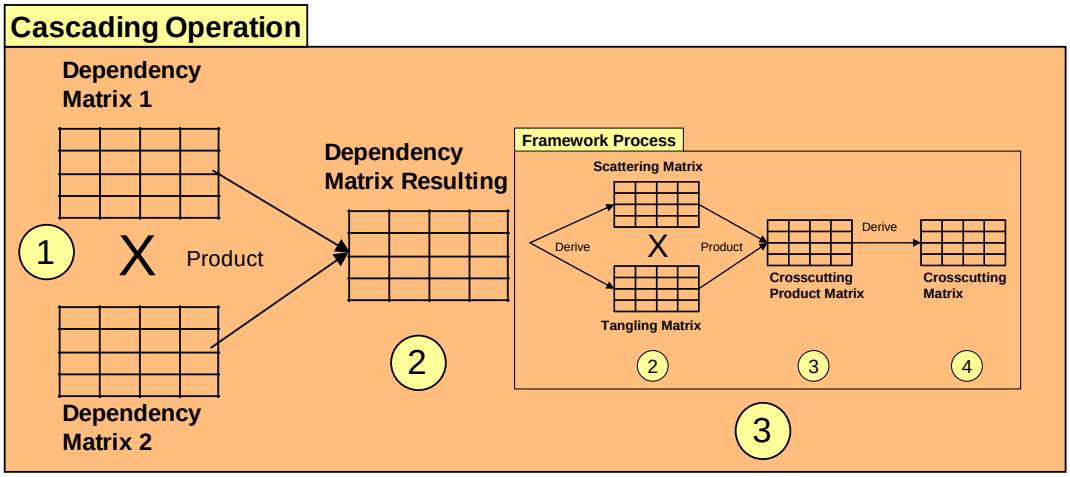


Figure 84. Overview of cascading operation

4.3.2. Transitivity of intra-level dependencies

Elements at a certain level usually have some relationship with other elements at the same level (intra-level relationships): they are coupled. There are many coupling types: generalisation/specialisation, aggregation, data coupling, control coupling, message coupling, and so on. In case of a dependency relation of a source element and a target element, which itself is coupled to a second target element, one could conceive also a dependency relation between the source element and the second target element.

Intra-level trace dependencies combined with inter-level trace dependencies may cause dependencies, which we call an *indirect trace dependency* based on a pseudo-transitivity. Assume source element s[i] has a coupling relation R' with source element s[j]. Moreover source element s[j] has a dependency relation R with target element t[k]. Then the indirect dependency relation is $(s[i]\ R'\ s[j]) \wedge (s[j]\ R\ t[k]) \Rightarrow (s[i]\ R' \circ R\ t[k])$. Analogously, assume source element s[i] has a dependency relation R with target element t[j] and target element t[j] is coupled with target element t[k] by means of R'. In that case the indirect dependency relation is $(s[i]\ R\ t[j]) \wedge (t[j]\ R'\ t[k]) \Rightarrow (s[i]\ R \circ R'\ t[k])$.

One should clearly distinguish the direct (inter-level) dependency relation from this indirect dependency relation. Our dependency matrix only represents direct dependency relations, however as we describe later on in Chapter 5 (Early aspect mining process), we may use a special kind of intra-level dependency relations to automatically derive new direct relations. In that case we use dependency relations in UML use case diagrams [180] (as coupling or intra-level relations) to infer new indirect trace dependencies between source and target elements (inter-level relations).

4.4. CONCLUSIONS

In this chapter, different representations of the crosscutting pattern have been illustrated. The utilization of these representations allows the identification of crosscutting situations in software systems. In particular, a process to obtain crosscutting occurrences using traceability matrices has been described. As we describe in Section 5.1, aspect mining is one of the main application areas of the crosscutting pattern. The identification of crosscutting allows crosscutting concerns to be refactored using different techniques such as aspect-oriented ones.

Observe that sometimes the presence of scattering, tangling and crosscutting may be solved using other techniques or selecting a different decomposition. This possibility is determined by the expressive power of the languages used to model source and target elements [20]. This assumption is also done in [135], where Mezini and Ostermann state:

> *"Crosscutting models are themselves not the problem. The problem is that our languages and decomposition techniques do not properly support crosscutting modularity."*

As an example, the utilization of design patterns [80] could solve some modularity problems. However, there are cases where the utilization of design pattern is not enough to solve modularity problems (as it is demonstrated in [93] and [83] by Hannemann and Kiczales and Garcia el al. respectively). In cases where limitations in the expressive power of the languages are the cause of tangling, scattering, and/or crosscutting, we can use the terms *intrinsic tangling, intrinsic scattering* and *intrinsic crosscutting* (as it is assumed in [20]). These situations are where aspect-oriented techniques come into play.

Thus, the utilization of a particular language influences the results obtained by the crosscutting matrix and the analysis performed. This is why the crosscutting pattern uses two domains to identify crosscutting situations. Crosscutting is identified in terms of a particular mapping from source to target elements. We could not state that concerns crosscut each other without observing the target domain (or language) where these concerns are implemented or represented. In that sense, this assumption implies that we do not consider crosscutting as being an intrinsic property of any concern. As an example, although logging concern is usually considered as a crosscutting concern, it really depends on the selected language to express the decomposition selected (that implements the system).

Another factor that usually determines the presence of crosscutting concerns is the selected decomposition. There are usually alternative decompositions possible in the same system. In fact, the decomposition selected for source and target are affected by different factors, such as the granularity level, type of dependencies relating source and target, composition operators, etc. As an example, depending on the goal of the analysis performed, different granularity levels should be considered. One could consider a class with its attributes and operations as a single element (coarse granularity), or one could consider each operation and each attribute as separate elements (fine granularity) [20]. By applying the process to different abstraction levels or domains, we identified the need for using fine granularity levels (e.g. classes or methods) at programming (and detailed design) abstraction levels and coarser granularity levels (e.g. use cases or components) at early stages of development (like requirements or architecture).

One has to compare combinations of alternative compositions on quality attributes such as adaptability, reusability and maintainability. However, in order to detect the cases where aspect-oriented techniques should be used, we need the identification of crosscutting concerns in the system (c.f. the process described in Section 5.1). Moreover the empirical study of the values obtained by the different matrices helps to measure what concerns should

be considered to be refactored or how modularity affects to other software quality attributes, such as maintainability (e.g. using the concern metrics presented in Section 5.4).

5

Applicability and evaluation of the framework

As it is mentioned in the introduction, the utilization of a formal definition of crosscutting becomes mandatory for certain application areas such as the identification of crosscutting concerns or the definition of a crosscutting measurement framework. In this chapter we show the main application areas of the crosscutting pattern, namely aspect mining at different abstraction levels [24] [21], detection of crosscutting dependences between features in software product lines [52], the definition of new concern-oriented metrics [50][51] and prediction of software quality attributes such as stability or changeability [51] in software systems. There are other application areas of the crosscutting pattern described as future research lines in Section 6.2 (e.g. building traceability tools or incorporating the aspect mining process into a Model-Driven Development).

5.1. ASPECT MINING

One of the main research areas where the framework may be applied is the identification of early aspects. In the last few years, aspect mining has emerged as an important technique for improving software modularity in legacy systems. Aspect mining refers to the process of identifying non localized crosscutting concerns throughout an existing software system which can be then refactored using aspect-oriented techniques [32]. Most of the research on aspect mining has traditionally focused on the implementation level, where different techniques such as fan-in, identifier or dynamic analyses have been used to the semi-automated discovery of potential aspects in the source code [36]. However, postponing the identification of crosscutting concerns until the latest phases of development avoids taking advantage of aspect-orientation at early stages of the software life cycle such as requirements or analysis. Furthermore, the refactoring of a legacy system using aspects leads to inconsistencies between the final code and the original design, making the later maintenance and evolution of the system difficult. How are the aspect-related changes introduced at the implementation level reflected in the design? [44]. And, how are these changes also reflected at early stages? The earlier we identify crosscutting concerns in a system, the sooner we may model it using aspect oriented techniques.

The early aspects community has just focused on dealing with crosscutting properties at early phases, which are known as early-aspects [65]. However, as it is stated in [16], it isn't likely to be clear what aspects are present in an existing set of requirements. In the same work, the authors reveals that software architect uses manual approaches for finding crosscutting architectural concerns. Few works have addressed the automatic identification of crosscutting concerns at the requirements level: EA-Miner [155] and Theme/DOC [14] use natural language processing techniques for identifying crosscutting concerns at the requirements level. Although these proposals contribute to aspect mining at the requirements level, they cannot be applied to requirements artifacts other than text. Nevertheless, an important application area of mining early aspect is the refactoring of legacy systems, which are usually described using other requirements artefacts, such as UML use cases or viewpoints.

Based on the framework presented in previous sections, we propose an aspect mining process at early phases which may be also applied to other phases of the software life cycle. Unlike other aspect mining approaches, our process is founded on the formal definition of crosscutting presented in Section 3.1. Trace relations between source and target domains are represented in the dependency matrix. Using syntactical and dependencies-based analyses, the dependency matrix is automatically obtained, thus allowing the automation of the aspect mining process. These analyses allow us to automatically relate elements of source domain to elements of target domains. The framework extended with these analyses may be also applied in any phase of the development process.

As it is aforementioned, the crosscutting pattern is not defined in terms of specific abstraction levels so that it may be applied in any development stage. However, in this section we only focus on early phases so that crosscutting concerns are identified at soon as possible. In particular, an specific instantiation of the process to be applied at the requirements level has been provided. In that sense, the process has been applied using use case diagrams as the target domain. Note that use cases usually specify functionality belonging to different concerns, so that these concerns are tangled in them. Then, we consider that this is a clear situation where aspect mining comes into play. In this setting, the aspect mining process presented below aims at identifying crosscutting situations based on concerns scattered over different use cases and where other concerns are tangled.

The main steps of the aspect mining process at the requirements level are outlined in Figure 85 and summarised as follows:

(A) **Identifying source elements**. Requirements are usually represented in several documents and they provided from different interviews with stakeholders. We analyse these requirements in order to identify the main concerns: functional (Figure 85-(1)) and non-functional (Figure 85-(2)). In order to identify the non-functional concerns (NFC), we use a catalogue where common non-functional concerns are defined. Both functional and non-functional concerns are represented in XML format.

(B) **Identifying target elements.** In this phase we model the requirements (Figure 85-(3)) using use cases [180]. As we do with concerns, we represent the requirements in a XML format, exporting the use case diagrams to XMI [185].

(C) **Build the dependency matrix**. Taking concerns and requirements as source and target respectively we establish the trace relations between them; this is the function *f* defined in Section 3.1.2. These trace relations are automatically established by means of some syntactical (Figure 85-(4)) and dependencies based (Figure 85-(5)) analyses so that the dependency matrix is automatically obtained.

(D) **Identification of crosscutting by matrix operations**. The next step consists in the application of several simple matrix operations (Figure 85-(6) and Figure 85-(7)), shown in Figure 77, to obtain the crosscutting concerns at the requirements level. Both, the crosscutting product and crosscutting matrices may be used for assessing the degree of crosscutting in the system (we use them to establish several concern-oriented metrics, shown in Section 5.4).

(E) **Aspect-oriented refactoring**. Finally, the identified crosscutting concerns are modelled using aspect-oriented techniques (Figure 85-(8 and 9)). By means of this refactoring, these crosscutting concerns are isolated and encapsulated in separated entities, improving modularity and reusability of the system. Using simple composition rules, the system may be composed later weaving the crosscutting concerns identified with the base system (Figure 85-(10)).

We explain these steps throughout this section showing their application to a simplification of the case study presented in [174], a Concurrent File Versions System.

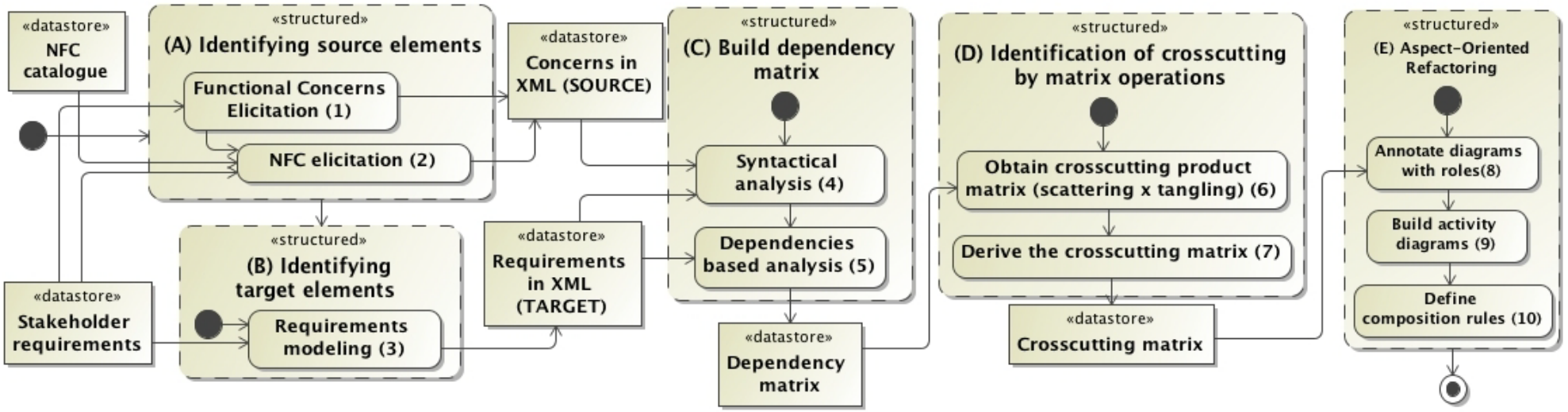


Figure 85. Main phases of the aspect mining process

5.1.1. The Concurrent File Versioning System

This application is a modification of the Concurrent File Versioning System (CFVS for the rest of this section) presented in [174] by Tekinerdoğan et al. The CFVS allows different versions of files of a project to be maintained. It also allows a group of developers to work on the same project and modify the same files at the same time. As the authors explain in [174], a developer may download a version of a software project. This operation is called **check-out** a project. After changing the current version of the project, the developer may upload a new version of the project to the CFVS system. This operation of committing a file is called **chek-in** a project.

Table 33 shows the main requirements that the system must fulfil. These requirements are extracted from [174]. Based on these requirements we perform the whole analysis explained in following sections.

	Description
r1.	Users should be able to insert files into a project. A message is stored with the inserted files.
r2.	Users can retrieve files from a project, these files are a copy and are saved on the computer of the user.
r3.	Users are able to change a file. This is an action that occurs outside the version system but is necessary because a versioning system is useless without change.
r4.	Users can commit files to a project. This means that the file in the project is updated with the working copy of the user. A message is stored with the committed files and the version number is updated when a file is committed.
r5.	User can update a working file, it updates the working copy with the latest version available in the project.
r6.	The commit and update actions both have to perform some form of conflict management, in the case that a file has been changed in the central system while the user made some changes to its working copy.
r7.	A user can ask the system to show the message that is stored with a certain version of a file.
r8.	Users can ask the system to show differences between two versions of a file in a certain project.
r9.	Users can remove files from projects.
r10.	Users can undo a file, which means that a file is restored.
r11.	A user can label (tag) a specific set of files (file versions), thus taking a static snapshot of those files. Tagging can be used to snapshot the development line when the software is ready to be released to the customer or to mark a stable set of files. After the release and continued development, a tagged set of files can be retrieved as if they were never changed.
r12.	A User can branch a set of files off the main trunk or other branch. A branch can only be constructed when the set of files has been tagged. A branch does not disturb the main trunk or any other branch, thus making them useful for testing bug fixes or new features.
r13.	A User can merge a branch back into the branch where it originated from. The merge action is similar to the update action; it only merges two branches and has to do also some conflict management in the same manner as the action merge and update.
r14.	The Administrator is responsible for project management (place holder for the different branches) and for assigning permissions to different users.
r15.	The Administrator has also the possibility to monitor different activities that occur in the system.
r16.	The system should also be able to store the different users and their permissions, restricting the different users in their possibilities to access parts of the projects.
r17.	The system should support concurrent usage of the versioning system.

Table 33. Requirements of the CFVS system

5.1.2. Identifying source elements

The first step of the process is to decide the main concerns that the system must address. There are some works about finding concerns on a system (e.g. [134] [153]), however they are usually focused on programming phases. In our process, we use the requirements documents to obtain the concerns of the system. We distinguish two different kinds of concerns, functional and non-functional concerns, related to functional and non-functional

requirements, respectively. The second category usually has an important impact on the decomposition selected to address the concerns of the first category.

Functional concerns elicitation

Requirements are analysed in order to identify the functional concerns (Figure 85-(1)). Each requirement may be addressing one or more concerns. The ideal situation would be to have a one to one relationship between concerns and requirements. However, this situation is not always possible in real systems, and concerns are usually scattered over the requirements and tangled with other concerns, so that crosscutting concerns emerge. Concern scoping is one of the major issues in aspect-orientation. Sometimes the task of discovering concerns is really difficult and it is left to the developers' expertise the decision of what a concern is and what not. In our running example we have used the same concerns that were identified by the original authors in [174]. These concerns are described in Table 34. The identification of functional concerns in the system could be also performed by extracting information from the results of other requirements elicitation techniques, e.g. stakeholders' interviews transcripts.

Concern	Description
c1	Insert File
c2	Retrieve File
c3	Commit File
c4	Update Working Files
c5	Remove Files
c6	Store Message
c7	Retrieve Message
c8	Difference
c9	Tag a Set of Files
c10	Branch a Set of Files
c11	Merge Set of Files

Table 34. Functional concerns in the CFVS system

Once the functional concerns are identified, they are represented in a XML file. This file will be automatically processed in later steps of the process. We use a XML file with simple <concern> tags. Each <concern> tag may have three sub-elements: <description>, <stakeholder> and <keyword>. The tag <stakeholder> is used to identify who is interested in the concern. This tag may contain three different children: <user>, <administrator> or <developer>. The <keyword> tag represents the word that we use to relate this concern with elements of the target domain (use cases). We explain later how to use this tag. In Figure 86 and Figure 87 we show a representation of the XML-Schema defined to validate this format and an example of several concerns represented in XML, respectively.

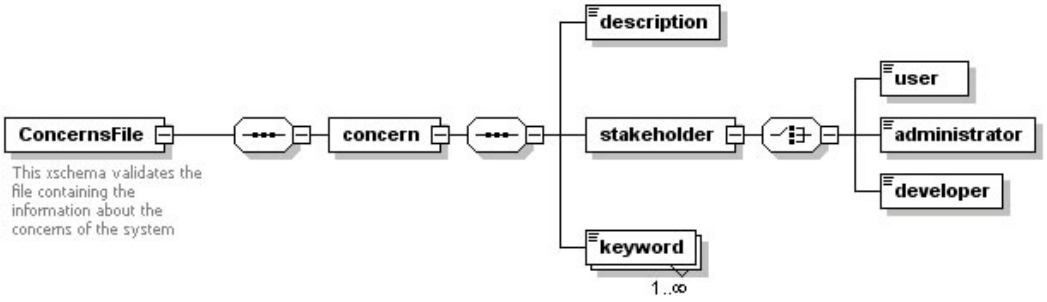


Figure 86. XML-Schema to validate the concerns file

```
<?xml version="1.0" encoding="UTF-8"?>
 <ConcernsFile>
    <concern id="c1" name="Insert file">
       <description>Concern related to the insertion of
files in the CFVS</description>
       <stakeholder>
              <user>Final User</user>
       </stakeholder>
       <keyword>Insert</keyword>
    </concern>
    ...
    <concern id="c2" name="Retrieve file">
       <description>Concern</description>
       <stakeholder>
              <user>Final User</user>
       </stakeholder>
       <keyword>Retrieve</keyword>
    </concern>
    ...
</ConcernsFile>
```

Figure 87. Example of functional and non-functional concerns in XML format

In [42], Cleland-Huang et al. introduced a set of best practices to automate traceability in requirements stages. One of the best practices presented by the authors consists of the building of a project glossary where the keywords of the project and specific domain are defined. In that sense, the definition of the concerns file presented in this section is very related to this technique since the main functionalities (concerns) and the keywords related to them are also defined. Moreover, the authors demonstrated the benefits obtained in projects where this glossary was defined at the very beginning of the development process, in contrast to the projects where it was created at the end of the development.

Non-functional concerns elicitation

We apply a syntactical analysis based on identifiers or keywords to elicit the non-functional concerns (Figure 85-(2)). In order to apply this analysis, we use a non-functional concerns catalogue similar to the one presented in the EA-Miner tool [155]. The catalogue consists in a XML file where common non-functional concerns are presented and related to different words that usually appears in requirements documents. The catalogue is an extension of the one used by the EA-Miner tool since it was completed with new words related to non-functional concerns. Non-functional concerns decomposition is considered since each concern is related to several different words (which may represent different granularity levels of the non-functional concern). We use these words to analyse the stakeholder requirements so that non-functional concerns are identified when one of these words appears in the text. In Figure 88 we may observe an example which relates the words “authorise” and “permission” to the Security concern defined in the catalogue.

```
<?xml version='1.0' encoding='utf-8'?>
<lexicon>
    <word content="authorise" nfr="security"/>
    <word content="permission" nfr="security"/>
</lexicon>
```

Figure 88. Part of the catalogue of non-functional concerns

Table 35 shows an example of some occurrences of words from the catalogue in the requirements. As an example, the requirements r1, r4 and r7 contain the word *“stored”*. This word appears in the catalogue related to the Persistence concern. Then we relate these requirements to the Persistence concern. The first column of Table 35 also shows other concerns of the CFVS.

Persistence	<word content="**stored**" nfr="**Persistence**"> </word>
r1.	A message is **stored** with the inserted files.
r4.	A message is **stored** with the committed files ...
r7.	… show the message that is **stored** with a certain version ....
Persistence	<word content="**saved**" nfr="Persistence"> </word>
r2.	… these files are a copy and are **saved** on the computer ...
Visual representation	<word content="**show**" nfr="Data representa tion" > </word>
r7.	… ask the system to **show** the message that …
r8.	… ask the system to **show** differences …
Security	<word content="**permission**" nfr="**Security**"> </word>
r14.	… and for assigning **permissions** to different users.
r16.	… users and their **permissions**, restricting ...
Logging	<word content="**monitor**" nfr="**Logging**"> </word>
r15.	… the possibility to **monitor** different ...
Persistence	<word content="**store**" nfr="**Persistence**"> </word>
r16.	… also be able to **store** the different ...
Security	<word content="**access**" nfr="**Security**"> </word>
r16.	… possibilities to **access** parts of the projects.
Concurrency	<word content="**concurrent**" nfr="concurrency" > </word>
r17.	… should support **concurrent** usage ….
Concurrency	<word content="**conflict**" nfr= "Concurrency"> </word>
r6.	… some form of **conflict** management, ...
r13.	… to do also some **conflict** management in ...

Table 35. Finding NF-Concerns in the CFVS requirements

After the analysis of the requirements, we have completed Table 34 with the non-functional concerns identified: Persistence, Visual Representation, Concurrency, Logging, Security. These non-funcional concerns are shown in the lower part of Table 36. In order to automatically process these concerns in later phases, we also represent them using XML. In particular, we use the same concerns file used for the functional concerns completing it with the non-functional concerns identified. The keyword tags for the non-functional concerns are completed with the words presented in the catalogue for the concerns identified. In Figure 89 we show the Persistence concern represented in the lower part of the XML file (extended).

Concern	**Description**	
c1	Insert File	Functional concerns derived from the analysis of the stakeholders' requirements (Figure 85-(1)).
c2	Retrieve File	
c3	Commit File	
c4	Update Working File	
c5	Remove File	
c6	Store Message	
c7	Retrieve Message	
c8	Difference	
c9	Tag a Set of Files	
c10	Branch a Set of Files	
c11	Merge Set of Files	
c12	Persistence	Non-functional concerns automatically derived by means of keywords analysis using a catalogue (Figure 85-(2)).
c13	Visual Representation	
c14	Concurrency	
c15	Logging	
c16	Security	

Table 36. Functional and non-functional concerns of the CFVS

```
<?xml version="1.0" encoding="UTF-8"?>
 <ConcernsFile>
    <concern id="c1" name="Insert file">
       <description>Concern related to the insertion of
files in the CFVS</description>
       <stakeholder>
              <user>Final User</user>
       </stakeholder>
       <keyword>Insert</keyword>
    </concern>
    ...
    <concern id="c2" name="Retrieve file">
       <description>Concern</description>
       <stakeholder>
              <user>Final User</user>
       </stakeholder>
       <keyword>Retrieve</keyword>
    </concern>
    ...
    <concern id="c13" name="Persistence">
        <description>Concern related to… </description>
        <stakeholder>
            <developer>Analyst</developer>
        </stakeholder>
        <keyword>store</keyword>
        <keyword>storage</keyword>
        <keyword>data</keyword>
    </concern>
</ConcernsFile>
```

Figure 89. Concerns file extended with non-functional concerns

5.1.3. Identifying target elements

In this activity we select the target elements that we use to build the dependency matrix later on. The previous activity (Identifying source elements) and this one could be concurrently done. However, it seems natural to perform this activity after identifying the concerns of the system.

Requirements modelling

In this activity the requirements engineer must build the first representation of the system using some requirements language or notation (Figure 85-(3)). In our example, we have selected UML [180] as the modelling notation to represent the requirements. In particular, we utilize use case diagrams. Figure 90 depicts the use case diagram that represents the requirements described in Table 33.

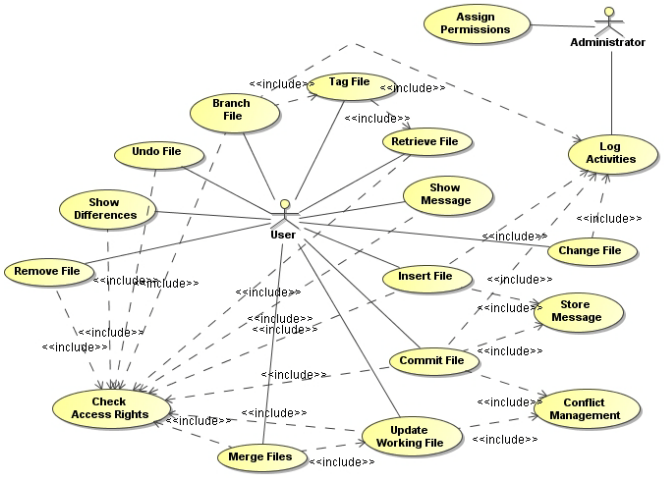


Figure 90. Use case diagram of the CFVS system

As we have done with concerns in previous activity Figure 85-(3), we need to represent the elements of target in a XML format. In this case, we use XMI [185] to describe the use case diagram. The use of XMI assures that we may perform the same analysis in other phases of the

development life-cycle. For instance, we may identify crosscutting concerns at design, taking concerns as source and UML class diagrams as target respectively (see [21]). In Figure 91 we show part of the XMI file which represents the use case diagram of Figure 90.

```
<packagedElement xmi:type="uml:Actor" xmi:id="_11_0_55e01df_1189682768937_411313_23"
    name="Administrator"/>
<packagedElement xmi:type="uml:UseCase"
   xmi:id="_11_0_55e01df_1189682802703_797484_41" name="Insert File">
   <include xmi:id="_11_0_55e01df_1189682942328_645326_149"
     addition="_11_0_55e01df_1189682927984_574791_137"/>
     …
</packagedElement>
<packagedElement xmi:type="uml:UseCase"
   xmi:id="_11_0_55e01df_1189683228359_463673_281" name="Check Access Rights"/>
</packagedElement>
```

Figure 91. Part of the XMI file for use case diagram of Figure 90

5.1.4. Building the dependency matrix

The trace relations between elements of source and target domains are represented by means of our dependency matrix (introduced in Section 4.2) which represents the starting point for the analysis of crosscutting. In our running example, the matrix shows the relations between concerns (source elements) and artefacts of the use case diagram (target elements). In this case, a cell with digit one denotes that a use case is addressing the concern of the corresponding row.

We explain now how to automatically obtain the dependency matrix. This phase is divided into two main activities: Syntactical analysis based on keywords (Figure 85-(4)); Dependencies based analysis (Figure 85-(5)). The inputs of this phase are the XML files generated in previous phases: the concerns file and the XMI file which represents the use case diagram. The output of the activity is the dependency matrix built. In Figure 92 we show a graphical representation of the different analyses performed for building the dependency matrix.

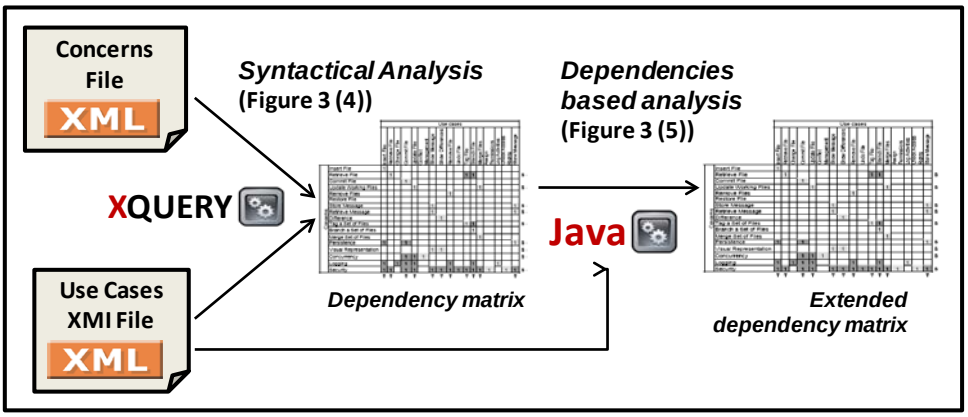


Figure 92. Building the dependency matrix

Syntactical analysis based on keywords

In this activity we derive the trace relations between concerns (source elements) and artifacts of the use case diagram (target elements). Since both functional and non-functional concerns are represented in the same XML file, we take into account both type of concerns in the analysis. In order to relate these two set of elements, we perform an analysis based on the similarities between the identifiers of both concerns and artifacts of the use case diagram. We use the <keyword> and <packagedElement> tags of concerns and XMI file respectively to

establish the matching between the identifiers. In particular, we use the attribute name of the <packagedElement> tag. We perform a syntactical analysis where the values of <keyword> tag and name attribute of <packagedElement> are totally or partially compared. That means that we can use either the whole word of the identifier to perform the analysis or we may use the morpheme of the word to compare the identifiers. For instance, we can relate a concern with the <keyword> "Insert" to a <packagedElement> with the name "Insert file" but also "Insertion of files".

In order to compare the keywords of each concern and the identifiers of the elements defined in the XMI file, we use the XQuery language [186] (a recommendation of the W3C) to search all the matching. In this language, we can write functions to obtain all the <packagedElement> tags in the XMI file which contain a name with a specific word. In Figure 93 we show an example of a query which provides all the use cases whose names match with the keyword of a particular concern.

```
for $b in doc("useCasesXMI.xml")//packagedElement
 where some $p in doc("concerns.xml")//concern
   satisfies (contains ($b/@name,$p/keyword))
   return $b
```

Figure 93. Example of query in XQuery

Applying the same analysis with all the use cases, we obtain the mappings shown in dependency matrix presented in Table 37. As we can see in this matrix, we have related both functional and non-functional concerns with the use cases that contribute to them (0's are not shown for making the table clearer). The mappings corresponding to non-functional concerns are shown in grey background. In case some mappings were missed or wrong added, the user could use the use case template descriptions to correct the results obtained by the analysis. However, as we show in Section 5.4, the process may be also extended by the addition of a metrics suite that allows the developer to identify possible errors in the process and to avoid these situations.

		Use cases															
		Insert File	Retrieve File	Change File	Commit File	Update File	Conflict Management	Show Message	Show Differences	Remove File	Undo File	Tag File	Branch File	Merge Files	Assign Permissions	Log Activities	Check Access Rights	Store Message
Functional Concerns	Insert File	1																
	Retrieve File		1															
	Commit File				1													
	Update Working File					1								1				
	Remove File									1								
	Store Message							1										1
	Retrieve Message							1										1
	Difference								1									
	Tag a Set of Files											1						
	Branch a Set of Files												1					
	Merge Set of Files													1				
NF Concerns	Persistence																	1
	Visual Representation							1	1									
	Concurrency						1											
	Logging															1		
	Security														1		1	

Table 37. Dependency matrix after the syntactical analysis

Dependencies based analysis

As it was explained in Section 4.4, coupling relations could be also used to infer new mappings between source and target elements, using a transitivity property. In this activity we search for relations between elements of the target domain (intra-level relations) to complete the dependency matrix (Figure 85-(5)). The relations that we take into account are dependencies. We use these dependencies to derive a new mapping or indirect relation between source and target. In particular, we use the *<<include>>* relationships of the use case diagrams to relate an element of source domain with an element of target domain. Observe that, although include relationships allow to separate the included functionality into separated entities, they imply a dependency between the two use cases. This dependency is translated into coupling relations in later stages of development. Then, they are an indication of crosscutting relations between the functionality coupled. In Figure 94 we show an example of the derivation of an indirect relation. We establish a special kind of transitivity relation between elements of source and target so that *if s1 is related to t1 and t2 depends on t1 then s1 is related to t2*. So, in the use case diagrams, we consider that *if use case t2 includes use case t1, then t2 depends on t1*. We do not use *<<extend>>* relations because they represent a specialization and not a dependency (the extended use case does not really depend on the use case which extends it).

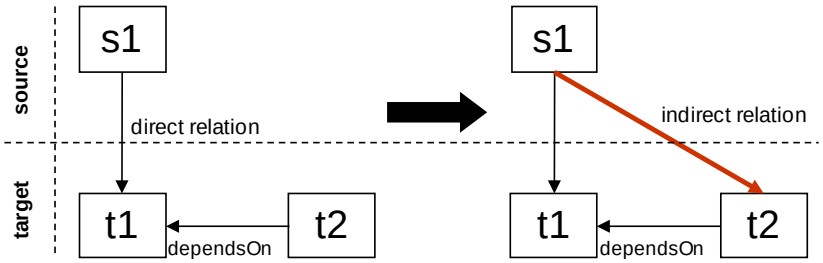


Figure 94. Indirect relation derived between s1 and t2

We can see in the use case diagram of Figure 90 that there are different *<<include>>* relationships. In particular, there are several use cases that include the functionality of the *Check Access Rights* use case. Since the *Check Access Rights* use case is contributing to the Security concern (see Figure 90), we relate all the use cases which include the *Check Access Rights* use case to the Security concern. The extended dependency matrix with the new relations (in dark grey) is shown in Table 38. We added a column and a row showing the concerns scattered (marked as S) and the use cases where concerns are tangled (marked as T) respectively.

The application of the dependencies based analysis is also automatically done just analysing the XMI file that represents the use case diagram of Figure 90. As we can see in the XMI file of the example (see Figure 95), the *<<include>>* relations are represented as sub-elements (*<include>* tag) of the corresponding *<packagedElement>* tags. The *<include>* tag appears in the use case source of the relationships (where the arrow starts). It has an attribute called addition that indicates the use case target of the *<<include>>* relationship, this is the use case where the arrow (of the include relation) ends. This use case is indicated by means of an alphanumeric identifier. As we can see in Figure 95 we just need to search the identifier in the rest of *<packagedElement>* tags. For instance, the Insert file use case has an include relation with the addition attribute pointing out the *Check Access Rights* use case. We use a simple Java program to localize the corresponding elements in the XMI file (see Figure 92).

		Use cases																	
		Insert File	Retrieve File	Change File	Commit File	Update File	Conflict Management	Show Message	Show Differences	Remove File	Undo File	Tag File	Branch File	Merge Files	Assign Permissions	Log Activities	Check Access Rights	Store Message	
Functional concerns	Insert File	1																	
	Retrieve File		1									**1**	**1**						**S**
	Commit File				1														
	Update Working File					1								1					**S**
	Remove File									1									
	Store Message							1										1	**S**
	Retrieve Message							1										1	**S**
	Difference								1										
	Tag a Set of Files											1	**1**						**S**
	Branch a Set of Files												1						
	Merge Set of Files													1					
NF Concerns	Persistence	**1**			**1**													1	**S**
	Visual Representation							1	1										**S**
	Concurrency				**1**	**1**	1												**S**
	Logging	**1**		**1**	**1**	**1**							**1**			1			
	Security	**1**	**1**		**1**	**1**		**1**	**1**	**1**	**1**	**1**	**1**	**1**	1		1	**1**	**S**
		T	**T**		**T**	**T**		**T**	**T**	**T**		**T**	**T**	**T**				**T**	

Table 38. Extended matrix after dependencies analysis

```
<packagedElement xmi:type="uml:UseCase" xmi:id="_11_0_55e01df_1189682802703_797484_41"
     name="Insert File">
    ...
    <include xmi:id="_11_0_55e01df_1189683262343_262295_293" name=""
          visibility="public" addition="_11_0_55e01df_1189683228359_463673_281"/>
</packagedElement>
<packagedElement xmi:type="uml:UseCase" xmi:id="_11_0_55e01df_1189683228359_463673_281"
     name="Check Access Rights"/>
</packagedElement>
```

Figure 95. Include relations in the XMI file

5.1.5. Identification of crosscutting by matrix operations

In this last phase, some simple matrix operations are performed in order to identify crosscutting concerns. From the dependency matrix, and once the scattering and tangling matrices have been derived, the crosscutting product matrix is obtained (Figure 85-(6)), where crosscutting concerns may be identified and quantified. Using this matrix we finally obtain the crosscutting matrix (Figure 85-(7)). In the crosscutting matrix, a matrix cell denotes the occurrence of crosscutting; it abstracts from the quantity of crosscutting (since it is a binary matrix). Nevertheless, all these matrices may be used to quantify crosscutting and to establish different metrics about aspect-orientation. In particular, in Section 5.4 we present a set of concern-oriented metrics which are automatically calculated using the matrices [51]. These metrics allow the developer to avoid the presence of false crosscutting concerns and to have a more realistic measurement of the values obtained by the matrix operations. In addition, they may be used to anticipate important decisions about software quality (such as stability) at early phases of development [51]. In Section 5.4.5, we discuss about the utilization of these metrics for the validation of the process.

		Concerns											NF concerns				
		Insert File	Retrieve File	Commit File	Update Working Files	Remove File	Store Message	Retrieve Message	Difference	Tag a Set of Files	Branch a Set of Files	Merge Set of Files	Persistence	Visual Representation	Concurrency	Logging	Security
Functional concerns	Insert File																
	Retrieve File		3							2	1					1	3
	Commit File																
	Update Working File				2							1			1	1	2
	Remove File																
	Store Message						2	2					1	1			2
	Retrieve Message						2	2					1	1			2
	Difference																
	Tag a Set of Files		2							2	1					1	2
	Branch a Set of Files																
	Merge Set of Files																
NF concerns	Persistence	1		1			1	1					3		1	2	3
	Visual Representation						1	1	1					2			2
	Concurrency			1	1								1		2	2	2
	Logging	1	1	1	1	1				1	1		2		2	5	5
	Security	1	3	1	2	1	2	2	1	2	1	1	3	2	2	5	11

Table 39. Crosscutting product matrix for the CFVS example

		Functional concerns											NF Concerns				
		Insert File	Retrieve File	Commit File	Update Working Files	Remove File	Store Message	Retrieve Message	Difference	Tag a Set of Files	Branch a Set of Files	Merge Set of Files	Persistence	Visual Representation	Concurrency	Logging	Security
Functional concerns	Insert File																
	Retrieve File									1	1					1	1
	Commit File																
	Update Working File											1			1	1	1
	Remove File																
	Store Message							1					1	1			1
	Retrieve Message						1						1	1			1
	Difference																
	Tag a Set of Files		1								1					1	1
	Branch a Set of Files																
	Merge Set of Files																
NF Concerns	Persistence	1		1			1	1							1	1	1
	Data Representation						1	1	1								1
	Concurrency			1	1								1			1	1
	Logging	1	1	1	1	1				1	1		1		1		1
	Security	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	

Table 40. Crosscutting matrix for the CVFS example

In Table 39 and Table 40 we can see the crosscutting product matrix and crosscutting matrix for the CFVS example, respectively. As we can see in the crosscutting matrix shown in Table 40, there are several concerns which can be considered as candidate crosscutting concerns. We can also observe how the candidate crosscutting concerns may be both functional and non-functional concerns. In particular, Table 40 firstly confirms what intuition perceives: the non-functional concerns are the elements which crosscut more concerns. In particular, Logging and Security concerns crosscut 10 and 15 concerns respectively. Observe in Table 39 that the values are in general higher in the rows for these concerns than in the rest ones. We can also see in Table 40 how there are functional concerns crosscutting other concerns (both functional and non-functional). This situation suggests the use of aspect-

oriented techniques to isolate and refactor the crosscutting concerns so that modularity of the system is highly improved. Sometimes refactoring a certain crosscutting concern removes the crosscutting dependencies between both crosscutting and crosscut concern. However, if two given concerns A and B are crosscutting each other, what concern should be refactored, A or B? In Section 5.4, we show how to take such decisions by an empirical analysis driven by the set of concern metrics defined [51].

5.1.6. Aspect-Oriented refactoring

As we mentioned in previous section, once the crosscutting concerns are identified, we may refactor them so that they are encapsulated into separated entities. Actually the refactoring is performed by isolating the use cases that implement crosscutting concerns. In other words, the refactoring is performed for target elements that address crosscutting source elements. In particular, we adapt the technique used in [137], where the authors present a method to modularize volatile concerns at the requirements level by aspect-oriented techniques. They utilize a Use Case Pattern Specification [79] and some templates to "mark" the use cases which address volatile concerns. We use here the same technique to refactor the crosscutting concerns in our CFVS example.

Figure 96. Use case diagram before refactoring

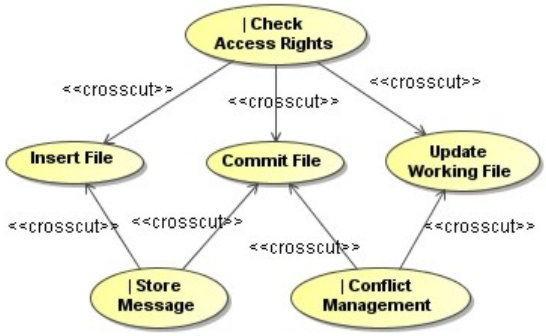
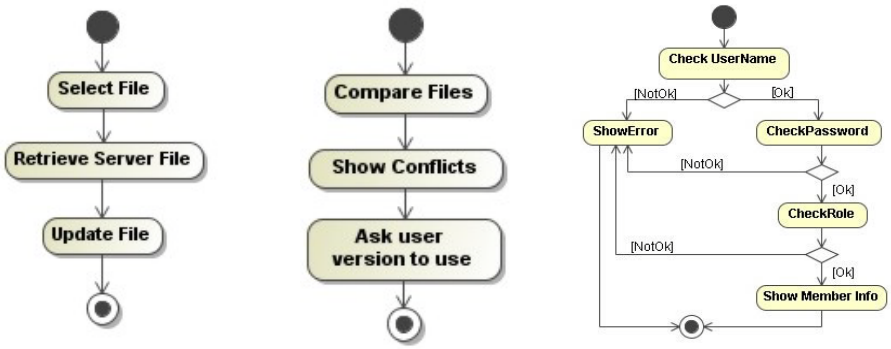


Figure 97. Use case diagram marked

The concerns which are classified as crosscutting are marked as roles. Then, we mark the use cases implementing these concerns (using the special symbol "|") and they are modelled using a pattern specification model (Figure 85-(8)). A new <<crosscut>> relation is added to the use case diagram which is used to relate use cases implementing crosscutting concerns to those which are considered as the base system. The information needed to decide where to add these new relations (<<crosscut>>) is derived from the results obtained by the matrices (and also supported by the metrics introduced in Section 5.4). On one hand, the results obtained by the crosscutting matrix indicate the concerns that should be refactored (basically those that crosscut more concerns). On the other hand, the dependency matrix is used to trace the use cases which address these crosscutting concerns. Figure 97 depicts a part of the use case diagram of the system with the marked use cases and <<crosscut>> relations. Figure 96 shows the same part of the use case diagram in the original system (without refactoring). As we can see in these figures, by refactoring, the direction of the relations is changed (<<crosscut>> instead of <<include>>) so that the base use cases (those implementing base concerns) are independent of the crosscutting concerns, fulfilling the AOSD obliviousness principle introduced by Filman and Friedman [75].



a) Update working file use case **b) Manage conflicts use case** **c) Check access rights use case**

Figure 98. Activity diagrams for several use cases of the CFVS system

Then, once we have isolated the implementation of the crosscutting concerns, we may evolve the system and change the crosscutting concerns using composition rules. By these composition rules we may compose different activity diagrams (in particular, we use Activity Pattern Specifications [79]). As an example, in Figure 98 we show three activity diagrams (Figure 85-(9)) which represent the main flows of three use cases: Update Working Files, Manage Conflicts and Check Access Rights. These activity diagrams are composed (Figure 85-(10)) using the composition rules shown in Figure 99 and Figure 100. Then, we can easily change the concurrency feature or the security policy just composing the base activity diagram (for Update working file use case) with other activity diagrams, thus using different composition rules. Analogously, we can also reuse these activity diagrams in other systems just using new composition rules.

Compose Update working file **with** Check access rights
Insert Check user name **before** Select file

Figure 99. Composition rule to add Check access rights functionality to Update working file

Compose Update working file **with** Manage conflict
Insert Compare files **after** Retrieve server files

Figure 100. Composition rule to add Manage conflicts functionality to Update working file

Of course, as it was explained in Section 4.4, there are other techniques (different from aspect-oriented ones) that we may use to get a high level of flexibility or configurability. Sometimes, we may use design patterns [80] to improve flexibility and reusability of particular designs. However, as it has been demonstrated in several publications [83] [93], the utilization of aspect-oriented techniques considerably improves the benefits obtained by the utilization of some design patterns.

5.1.7. Comparison with other Approaches

In this section we show the application of our process to other case studies used by other approaches (which also deal with crosscutting concerns at early stages). The goal of the section is twofold: on one hand, to validate the aspect mining process presented in this section; on the other hand, the identification of some open issues that should be considered about mining aspects at early stages.

For the validation of the process, we have applied our process to 5 different examples: a Conference Review System (used in our previous work [24] without syntactic and dependencies based analyses), the Portuguese Highways Toll System (used in [151]), a Siemens Toll Gate System (used by the EA-Miner tool in [157]), a Course Management System (used by Theme/Doc [15]) and the well-know PetStore example (used by Theme/Doc and its extension with Latent Semantic Analysis both in [113]). Then we compare the results obtained by the different approaches with regard to the results obtained by our aspect mining process. Note

that we have also compared the process with the results obtained in our previous work [24], where the framework did not use the syntactical and dependency based analyses.

A summary of the results obtained is shown in Table 41. As we can see in the table, we have represented the number of concerns (both functional and non-functional) and crosscutting concerns identified by the different approaches in the case studies. We have also represented the false positive and negative rates for each example. False positives refer to crosscutting concerns wrongly identified by the tool whilst the false negatives are related to crosscutting concerns that the different tools do not identify. The actual crosscutting concerns were also calculated by an analysis of the results obtained by the original approach and the application of our aspect mining process. In particular, the actual crosscutting concerns are calculated by the union of the crosscutting concerns and the false negatives identified by both approaches (the original and ours) minus the false positives (obtained also by both approaches). As an example, in the STS the union of crosscutting concerns and false negatives identified is formed by 19 concerns but the union of false positives is 2. Thus, the actual number of crosscutting concerns is 17.

We established that the best technique is the one with lowest sum of false positives and negatives. In that sense, our framework presents worse values that other manual approaches (e.g. for the PHTS). However, the approach presents better rates than similar automatic approaches. Of course, in the manual approaches, the engineer may have better knowledge of the domain. However, in real systems, the use of manual approaches may be unfeasible and the utilization of automatic tools helps to the developer in important tasks reducing effort and development time.

	Conference Review System (CRS)		Portuguese Highways Toll System (PHTS)		Siemens Toll System (STS)		Course Management System (CMS)		PetStore		
	Original	Crosscutting Pattern	Original	Crosscutting Pattern	EA-Miner	Crosscutting Pattern	Theme/ Doc	Crosscutting Pattern	Theme/ Doc	LSA	Crosscutting Pattern
Concerns identified	8	10	7	4	21	16	5	6	4	10	11
Functional Concerns	4	6	-	-	13	6	4	4	-	-	7
Non-Functional Concerns	4	4	7	4	8	10	1	2	-	-	4
Candidate Early Aspects	4	8	7	4	14	13	1	4	15#	19#	7
False Positives	0	3	0	0	2	1	0	2	12	16	1
False positive rate*	0	3/5 = 0,6	0	0	2/4 = 0,5	1/4 = 0,25	0	2/4 = 0,5	12/5 = 2,4	16/5 = 3,2	1/5 = 0,2
False Negatives	1	0	0	3	5	5	1	0	3	3	0
False negative rate*	1/5 = 0,2	0	0	3/7 = 0,42	5/17 = 0,29	5/17 = 0,29	1/2 = 0,5	0	3/6 = 0,5	3/6 = 0,5	0
Actual Crossc. Concerns	5		7		17		2		6		

* For more information about how to calculate False positive and negative rates see [139].
# The approaches based on Theme/Doc identify aspectual requirements. This is the reason why the number of early aspects is higher than the number of concerns.

Table 41. Comparative table with the different case studies and tools

5.1.7.1. Analysing false positives

The crosscutting matrix could lead to consider some false crosscutting concerns (false positives). Sometimes, these false positives may be caused by the decomposition selected for the elements in source or target. In these cases, it is possible to avoid crosscutting by choosing another decomposition of source and target, a possibility determined by the expressive power of the languages in which the source and target are represented.

As we can see in Table 41, we may find false positives in the application of almost all the different approaches analysed. As an example of the explained above, in the CRS presented in [24], our framework identified three functional crosscutting concerns (Submission, Review and Conference Management). However, some of these concerns are considered as crosscutting

because of the granularity of target elements selected. For instance, the submission process takes place in several use cases: submit paper and change submission (both in the same package). Obviously, these use cases are related to the same functionality. Since the implementation of this concern is tangled with other concerns (e.g. non-functional), it is identified as a candidate crosscutting concern. However, the isolation of some crosscutting concerns could cause that crosscutting disappears from submission concern as well (since tangling would be removed). However, the decision of what concerns should be refactored is not trivial. This decision should be supported by empirical data. In that sense, the concern metrics presented in Section 5.4 provide the empirical data and allow the identification of the concerns with a higher degree of crosscutting. In fact, by the addition of these metrics to the process presented here, the results obtained may be more realistic and the developer has a statistical model to take such decisions.

The same situation occurs with Review and Conference Management concerns. We consider these concerns false positives. Moreover, the application of the process to more complex systems, such as the the MobileMedia system (shown in Section 5.4) illustrates how the problem of false positives is reduced in such complex systems and it may emerge in small systems with just a few concerns.

5.1.7.2. Analysing false negatives

In the False negatives column of Table 41 we can see how our framework did not identified some crosscutting concerns in some case studies. In most of cases, the reason to get these false negatives is that we do not identify as crosscutting concerns those that are not somehow represented in the use case models. For instance, there are some candidate crosscutting concerns like scalability, reliability or compatibility (identified in STS) that usually are mapped to design or hardware decisions later on and they are not represented in diagrams or code. Although we use the non-functional concerns catalogue to identify such concerns, the crosscutting matrix does not identify them as crosscutting concerns since they are not mentioned in the use cases based representation. This is the reason why the framework did not identify some crosscutting concerns in the PHTS as well (e.g. Availability or Correctness).

In the STS case study, we did not identify a crosscutting concern that EA-Miner did. This concern is what they called Charge calculation. However, the authors in [157] claim that they focused on the requirements related to communication. We took the same requirements for our analysis and we did not find any presence of the Charge calculation concern in those requirements. We think this concern could be considered as a false positive by their approach.

5.1.7.3. Accuracy of requirements

Since we are applying our framework at early phases, the first source of information for the process is the set of requirements (usually in text). Sometimes, there are some non-functional concerns which may not be inferred by means of an automatic analysis of the requirements. For instance, in the PHTS presented in [151] the authors identified some non-functional concerns by manually analysing the requirements documents and extracting conclusions about them. As an example, in the text “If an unauthorized vehicle passes through it, a yellow light is turned on and a camera takes a photo of the plate” the photo must be quickly taken, otherwise the plate will not appear in it. Then, response time is present in such a requirement. In the application of our framework to the PHTS we obtained less crosscutting concerns that the authors did in [151]. We did not identify the non-functional crosscutting concerns that were implicitly (but not explicitly in the text) present in the requirements. The problem of accuracy of requirements is also present in the rest of tools analysed. As an example, in the STS, the EA-Miner identified 21 concerns. However, as the authors say in [157], these concerns are the result of editing and sorting the original set of concerns identified by the tool. Then,

the engineer must spend quite time selecting the concerns of the system after applying the tool to the original requirements.

A lesson learned from the results obtained in Table 41 is that, in some cases, the results obtained by the manual approaches may be better than those obtained by automatic approaches. However, in real systems with higher sets of requirements, the utilization of manual approaches is unfeasible and the automatic tools may save much time for the developer. Anyway, like in the other approaches, the requirements engineer may assist our process changing the dependency matrix as he/she considers necessary. The results obtained may be improved based on his/her experience in the current system or systems previously developed. Moreover, the developer could assist the process by the utilization of techniques for documenting the process, e.g. shadowing or marking the use cases artefacts, as it is suggested in [71] (the authors use shadowing techniques at source code level).

5.1.7.4. The non-functional concerns catalogue

The utilization of a non-functional concerns catalogue or repository improves the identification of such concerns (the represented in the catalogue) in software systems. We noticed such improvements in the different case studies we have analysed. For instance, in the CRS case study we identified a new non-functional crosscutting concern, Data Presentation. In the second case study, the PHTS, we also identified some non-functional crosscutting concerns (such as Persistence and Data Presentation) that the authors didn't identify in [151].

In the STS example, the two approaches compared use a similar catalogue so that in that sense the results obtained are similar. For the PetStore example, the Theme/Doc also uses a set of keywords. However, this catalogue just contains words related to the application domain, ignoring non-functional concerns common to different domains. As we can see in Table 41, the results obtained by applying the LSA approach to the PetStore are similar to those obtained by our framework.

This fact was also observed by Cleland-Huang et al. in [43]. In this work, the authors presented an automated method to classify non-functional requirements in software requirement specifications. They claimed that a firstly detection of the relevant keywords related to each non-functional requirements allows an important improvement of the results obtained in next projects (using these keywords identified). Moreover, the authors indicated the lack of standardized non-functional requirements catalogues so that the introduction of such kind of glossaries aims at bringing this gap.

5.1.7.5. Granularity of source and target elements

As it has been aforementioned, concern scoping is one of the major issues in the aspect-oriented area. Sometimes it is not trivial to decide what a concern is in a particular system and the concern decomposition is lead to developers' expertise. In that sense, the granularity selected to decompose source and target elements may affect the results obtained by any aspect mining process. One has to decide the granularity level to be selected for analysing the mappings between source and target. Even, alternative decompositions are possible to avoid the problem of crosscutting concerns, sometimes using design decisions, such as the utilization of design patterns [80]. However, as it has been demonstrated in several publications [83] [93], sometimes design patterns are not enough to solve these modularity problems. The problem in these cases is related to the limited expressivity power of the languages used and new constructs are needed, provided by aspect-oriented languages.

The decision of selecting the interplay between source and target decompositions is not trivial. Observe that, in [42], the authors presented the selection of the suitable trace granularity as one of the best practices to be considered for obtaining automated traceability

methods. As an example, Egyed et al. evaluated the advantages of tracing at lower levels of granularity versus the effort needed to create the links between the elements at this finer granularity level [67]. They concluded that the benefits obtained by improving the granularity of trace links beyond a certain level were really limited. In that sense, by applying our aspect mining process to several case studies and comparing with similar approaches, we checked that the results obtained by the process were consistent with those obtained by the other approaches. This fact validates the process presented even when the source and target decompositions have not been selected by the authors of this process (we used the original systems presented by other authors). Anyway, by using the expertise obtained by applying the framework to different abstraction levels [23] [24] and case studies (shown throughout this Chapter), we obtained some indications (as an oracle) of the granularity level that should be used at different abstraction levels. We identified the need for using fine granularity levels (e.g. classes or methods) at programming (and detailed design) phases and coarser granularity levels (e.g. use cases or components) at early stages of development (like requirements or architecture). The use of the generic crosscutting pattern allows utilizing the wished granularity level depending on the abstraction level and the purpose of the analysis.

5.1.8. Conclusions and discussion

As it has been widely mentioned in this document, aspect mining is one of the main challenges in aspect-orientation. Moreover, the identification of crosscutting concern has been traditionally tackled at programming level, relegating the benefits of AOSD to the latest phases of the development. In this setting, the use of an aspect mining process at early stages of development allows the incorporation of these benefits at the very beginning of the development. The aspect mining process presented in this section allows the identification of crosscutting concerns at the requirements level. Moreover, the utilization of different XML representation (e.g. XMI files) ensures that the process may be easily instantiated to be used in other software phases or abstraction levels.

The process has been validated showing its application to different case studies. The comparison showed that the results are, in general, consistent with the obtained by the other approaches although there are cases where the results are different. As an example, the manual approaches analysed obtain, in general, better results that the semi-automatically ones. This is due to the fact that the former are based on developers' expertise and it may be identify some crosscutting concerns implicitly present in the requirements. However, the utilization of manual approaches is not feasible in large and complex systems so that other approaches (semi-automatic) must be considered. The different results obtained in some cases suggest the existence of false positives and negatives. All the approaches analysed present this problem. However, the utilization of a concern-driven metrics suite (like the introduced in Section 5.4) aims at avoiding these situations.

Now, the criteria considered to compare the different aspect mining approaches in Section 2.2 are also used to compare our aspect mining process. Note that some criteria are highly coupled to the programming level, however, they are considered at the requirements level as well. For instance, the static or dynamic analysis factor considers whether the approach is applied at compile or run time. Then, at the requirements level, this criterion could not have sense at least an executable model of the requirements is built and the approach identifies crosscutting using this simulation. However, most of the approaches analysed lack of this executable model. Then, they apply all an static analysis. The criteria used at the programming level are the following:

- **Static or dynamic analysis**: as it has been mentioned above, most of the early aspects approaches use static data to perform their analysis. Our approach is not an exception and it utilizes static data to perform the aspect mining process.

- **Token or Structural analysis**: Our approach could be considered as a mixin of token and structural techniques. While the textual analysis is a token-based technique, we combine it with the dependency analysis which considers structural information of the use case diagrams. Anyway, since the dependency analysis is also performed using the XMI file and searching for textual concordances, it could be classified as textual.
- **Granularity level:** as it has been widely mentioned, the crosscutting pattern is not tied to any specific deployment artifact or abstraction level. Thus, the aspect mining process could be instantiated to be used at any development phase. In this Section a concrete instantiation (use cases) has been shown, however others are possible. Then, the granularity level may be selected in terms of the purpose of the analysis. As an example, at early stages, coarse granularity levels are used (e.g. use cases) whilst at latest stages, closer to the implementation, finer granularity levels may be used (e.g. class methods and fields).
- **Scattering and Tangling considered:** as it has been explained in Section 3.1.2, our definition of crosscutting considers scattering and tangling as needed conditions to have crosscutting. Then, the aspect mining process identifies crosscutting situations as special combinations of scattering and tangling.
- **User involvement:** our approach for aspect mining automates almost all the tasks to allow the user to identify the mappings between source and target elements. Of course, there is still a need for user involvement since the user must select the initial functional concerns that the system must implement. This fact makes the approach semi-automatic, however, once the concerns have been selected, the rest of tasks are automatically performed. Nevertheless, some techniques to identify the functional concerns could be used in order to provide a higher degree of automation in the process. As an example, concerns are usually extracted by analysing the results of other requirements elicitation techniques, e.g. stakeholders' interviews transcripts. Other techniques tackle the semi-automatic identification of these concerns using different heuristics, usually based on the semantic analysis of the text in the requirement documents. Examples of concern modelling techniques at an early abstraction level are COSMOS [167] or EA-Miner [155]. Nevertheless, most of the techniques to automatically identify concerns are based on the analysis of source code at the programming level, e.g. FEAT [153] (a deeper comparison of these techniques may be found in [182]). Moreover, as it was shown in Section 2.2.2, all the approaches needed some involvement by the user either to provide the input data to the approach or to filter the results obtained.
- **Larger system:** our aspect mining approach has been applied to several case studies. We have applied it mainly to the requirements of the system. However, since some of the case studies where we applied the process were externally implemented, we only had for the analysis the same part of the requirements used by the original authors (e.g. the Siemens Toll System used in Section 5.1.7). In that sense, the larger system (with the whole set of requirements) where we applied the process was the MobileMedia system (shown in Sections 5.4 and 5.5). This system has about 3 KLOC and it has been used in several analysis by other authors (e.g. [71][188]).
- **Empirical validation:** the aspect mining process presented here has been applied to several case studies in order to compare the results obtained with those obtained by other similar approaches. However, unlike the rest of approaches, our aspect mining process has been complemented by the definition of a set of concern-oriented metrics that allow empirically analysing modularity in software systems. These concern metrics are presented in next section, Section 5.4. Moreover, these metrics have been both theoretical and empirically validated.
- **Preconditions:** there is no precondition in our approach besides relating the source and target domains. That implies that the process must be applied once a first

decomposition or representation of the concerns is available. However, this representation could be done using any abstraction level or artefacts: e.g. natural language statements, use cases, architectural components, design classes, programming classes, …

Regarding to the criteria used to compare the early aspects approaches, they are summarised as follows:

- **Traceability through the software lifecycle**: traceability of crosscutting concerns has been illustrated in our approach by the utilization of the cascading of crosscutting pattern and the transitivity of traceability matrices shown in Section 4.3.
- **Composability**: composability in the process is achieved by using simple composition rules that allow the composition of the crosscutting concerns identified with the base concerns (see Section 5.1.6). In order to use these composition rules, the use cases that implement crosscutting concerns are marked as roles and new relations are added to UML use case diagrams in order to relate these artefacts with the base ones.
- **Evolvability:** the aspect mining process improves evolvability by two different ways. On one hand, the utilization of traceability matrices provides information of the target elements involved in a change in any source element. On the other hand, the refactoring of the crosscutting concerns improves evolvability since a change in these concerns only implies to change the artefacts where these concerns have been isolated.
- **Scalability:** scability of the process is ensured by the utilization of simple tools that are responsible for the automation of the process, e.g. for the comparison of the NFR catalogue with requirements or the analysis of the XMI files.

5.2. TRACEABILITY OF CROSSCUTTING CONCERNS

In Section 4.3 we explained how our work may be used to trace crosscutting concerns between different phases of the software development. This traceability is achieved by means of transitivity of dependency matrices. In this section we illustrate this process using the CFVS presented in previous section.

5.2.1. Transitivity of dependency matrices in the Concurrent File Versioning System

In terms of our dependency matrices, transitivity of trace relations is achieved by a simple multiplication between two matrices. Assume we have the dependency matrix for concerns with respect to requirements' artifacts (e.g. use cases) and a new dependency matrix which relates those use cases with the architecture classes which implement each use case. Both matrices are obtained using the analyses explained in the previous section. By means of the product of these two matrices, we obtain a new one where trace relations between concerns and architecture classes are represented. After obtaining this new dependency matrix, we can apply the analysis of crosscutting again just performing the matrix operations. We illustrate the transitivity of dependency matrices by the CFVS example. We consider three refinement levels, concern modelling, requirements modelling and conceptual architecture modelling as source, intermediate and target domains respectively. In Figure 101 we show the architectural design of the CFVS system. As we did with the requirements, this architectural model is basically the same that the one shown in [174].

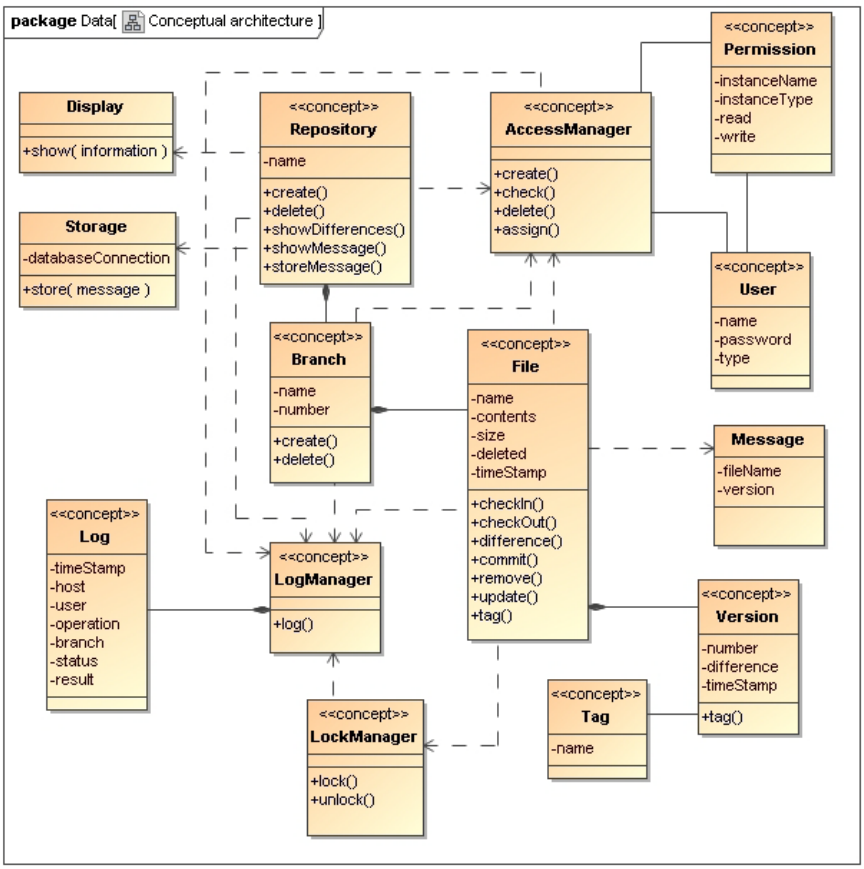


Figure 101. Conceptual architecture diagram of the CFVS

The process to apply the transitivity of dependency matrices in the CFVS example is very simple. Firstly we build the dependency matrix relating requirements' artifacts to architectural units. This matrix contains the direct mappings between use cases onto conceptual classes. These mappings or relations are extracted using the same analyses that we used at requirements stages. Basically these analyses take as input the names of the use cases and compare them with the name of each conceptual class and its methods (also taking into account dependencies). As an example, the *Branch file* use case is related to the Branch class. Analogously, since the Display class contains a method named *show*, it is related to the *Show Message* and *Show Difference* use cases. We also use XMI as the standard format to represent the architectural class diagram. The queries to compare the identifiers are written and executed by XQuery. The dependency matrix relating use cases and architectural elements is shown in Table 42.

Secondly, we perform the matrix product of the two dependency matrices which relates the three refinements levels (source, intermediate and target). These two matrices are shown in Table 38 (concerns x use cases) and in Table 42 (use cases x architectural classes) respectively. The result of the product is a new matrix relating concerns to architectural classes. This matrix is shown in Table 43.

		Conceptual classes													
		Repository	Branch	File	Version	Tag	User	Permission	Access Manager	Log	LogManager	LockManager	Message	Display	Storage
Use cases	Insert File			1											
	Retrieve File			1											
	Change File			1											
	Commit File			1											
	Update File			1											
	Conflict Management														
	Show Message	1											1	1	
	Show Differences	1		1										1	
	Remove File			1											
	Undo File			1											
	Tag File			1	1	1									
	Branch File		1	1											
	Merge Files			1											
	Assign Permissions							1							
	Log Activities									1	1				
	Check Access Rights								1						
	Store Message												1		1

Table 42. Dependency matrix for use case with respect to the conceptual classes

		conceptual clases													
		Repository	Branch	File	Version	Tag	User	Permission	Access Manager	Log	LogManager	LockManager	Message	Display	Storage
Functional concerns	Insert File			1											
	Retrieve File		1	3	1	1									
	Commit File			1											
	Update Working File			2											
	Remove File			1											
	Store Message	1											2	1	1
	Retrieve Message	1											2	1	1
	Difference	1		1										1	
	Tag a Set of Files		1	2	1	1									
	Branch a Set of Files		1	1											
	Merge Set of Files			1											
NF Concerns	Persistence			2									1		1
	Visual Representation	2		1									1	2	
	Concurrency			2											
	Logging		1	5						1	1				
	Security	2	1	10	1	1		1	1				2	2	1

Table 43. Dependency matrix derived from the transitivity property

In order to avoid possible missing mappings, we apply again our syntactical and dependencies-based analyses taking into account concerns and architectural classes. Then, the dependency matrix obtained by the cascading operation (shown in Table 43) is automatically completed with the mappings obtained by these analyses (shown in dark grey in Table 44). As an example of the new mappings identified, we can see how the Lock Manager class was not related to any concern in Table 43. The use of traceability matrices also allows to identify such situations that usually indicate some missing mappings. After applying the syntactical analysis,

the Lock Manager class is related to the concurrency concern (see the mapping in Table 44 in dark grey background). As a different example, we can see how Repository class is related to Persistence concern. This is due to the fact that this class has a dependency with Storage class. Then, this mapping is added. Observe that Repository class has also a dependency with Display class which indicates that Repository class should be related to the concern being addressed by Display (Visual Representation). However, since this mapping was already considered in Table 43 there is no need for adding a new mapping

		conceptual classes													
		Repository	Branch	File	Version	Tag	User	Permission	Access Manager	Log	LogManager	LockManager	Message	Display	Storage
Functional concerns	Insert File			1											
	Retrieve File		1	3	1	1									
	Commit File			1											
	Update Working File			2											
	Remove File			1											
	Store Message	1		**1**									2	1	1
	Retrieve Message	1		**1**									2	1	1
	Difference	1		1										1	
	Tag a Set of Files		1	2	1	1									
	Branch a Set of Files		1	1											
	Merge Set of Files			1											
NF Concerns	Persistence	**1**		2									1		1
	Visual Representation	2		1									1	2	
	Concurrency			2								**1**			
	Logging	**1**	1	5					**1**	1	1	**1**			
	Security	2	1	10	1	1	**1**	1	1				2	2	1

Table 44. Extended dependency matrix for concerns with respect to conceptual classes

		Functional Concerns											NF Concerns				
		Insert File	Retrieve File	Commit File	Update Working File	Remove File	Store Message	Retrieve Message	Difference	Tag a Set of Files	Branch a Set of Files	Merge Set of Files	Persistence	Visual Representation	Concurrency	Logging	Security
Functional concerns	Insert File																
	Retrieve File	1		1	1	1	1	1	1	1	1	1	1	1	1	1	1
	Commit File																
	Update Working File																
	Remove File																
	Store Message	1	1	1	1	1		1	1	1	1	1	1	1	1	1	1
	Retrieve Message	1	1	1	1	1	1		1	1	1	1	1	1	1	1	1
	Difference	1	1	1	1	1	1	1		1	1	1	1	1	1	1	1
	Tag a Set of Files	1	1	1	1	1	1	1	1		1	1	1	1	1	1	1
	Branch a Set of Files	1	1	1	1	1	1	1	1	1		1	1	1	1	1	1
	Merge Set of Files																
NF Concerns	Persistence	1	1	1	1	1	1	1	1	1	1	1		1	1	1	1
	Visual Representation	1	1	1	1	1	1	1	1	1	1	1	1		1	1	1
	Concurrency	1	1	1	1	1	1	1	1	1	1	1	1	1		1	1
	Logging	1	1	1	1	1	1	1	1	1	1	1	1	1	1		1
	Security	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	

Table 45. Crosscutting matrix for concerns with respect to the conceptual classes

In order to perform the syntactical and dependencies based analyses, explained in the previous section, we export the architectural diagram to XMI and use the dependency relations (usage relations). These relations appear in the diagram relating two different classes, e.g. *Repository* and *Display*. The dependency matrix with the relations added by both analyses can be observed in Table 44. Taking the dependency matrix shown in Table 44, we apply the analysis of crosscutting again for obtaining the final crosscutting matrix (Table 45).

The crosscutting matrix of Table 45 shows that there are several concerns which may be considered as crosscutting concerns with respect to the architectural design.In this table we can see how there are several crosscutting concerns that already existed at the requirements level. There are other concerns that were well modularized at the requirements level and they do not present crosscutting concerns at the architectural level. Note that we have designed the architecture of the system using the non aspect-oriented version of the requirements of CFVS, shown in Section 5.1. That implies that the refactored use cases have not been considered. The reason for using that version is that we were interested in analysing the traceability of crosscutting concerns and how they evolve throughout different development stages. If we would have used the aspect oriented version of the requirements, we could have avoided some of the crosscutting concerns identified at the architecture. As a summary, we can observe that:

- The concerns Insert File, Commit File and Remove File that were well-modularized at the requirements level do not present crosscutting at the the architecture. Note that these concerns are mainly implemented by using different methods in the File conceptual class.
- There are some concerns that were considered as crosscutting concerns at the requirements level and they keep crosscutting other concerns at the architectural level. Examples of these concerns are Retrieve File, Store Message, Retrieve Message, Branch a Set of Files and all the non-functional concerns. In these cases, these concerns are crosscutting more concerns so that we can say that they have increased their degree of crosscutting (this concept will be empirically measured using metrics in Section 5.4). This is the more usual situation when the crosscutting concerns are not refactored at the requirements level.
- There is also a situation of a crosscutting concern that has been removed at the architectural level. This concern is Update Working File. It may be observed in Table 38 how this concern was crosscutting other concerns at the requirements level. However, Table 44 shows how it is not identified as a crosscutting concern at the architectural level. The reason for having this situation is mainly due to the fact that most of the operations related to managing files are encapsulated into the File class. Then, some use cases related to these operations are implemented by this class. Then, some concerns that could be addressed by several use cases at the requirements level may be addressed now by this class.

5.2.2. Conclusions and discussion

As it has been illustrated in this section, the crosscutting pattern provides a means to trace different elements of the development through the different abstraction levels or stages. In particular, the transitivity of dependency matrices illustrated in this section helps in the traceability of crosscutting concerns. In that sense, the CFVS example has was used to show how crosscutting concerns emerge, disappear or are maintained through two different stages. There were examples of these three situations. We observed that usually crosscutting concerns are maintained throughout the development process. Even, the problem is emphasised in late stages: the degree of crosscutting of early aspects increases as we focus on

tasks closer to the programming level. This is why early aspects modularization is needed at the very beginning of the development.

However, aspect orientation is not always the only solution. Sometimes, the utilization of certain design patterns or decompositions may solve the problem. In other cases, some crosscutting concerns may be addressed using a particular hardware platform (e.g. for obtaining high performance).

Also, concerns classified as non-crosscutting at the requirements level keep localized at the architectural level so that they do not need to be refactored. However, again, this situation is not a rule and concerns not identified as crosscutting at early phases could be considered as crosscutting concerns at later stages.

5.3. Reducing Dependencies between Features in Software Product Lines

Software product lines (SPL) have become an emerging trend in software development where products related to a particular domain are created from the combination of a shared set of common and variable software assets [45]. SPL approaches [45] [147] aim at reducing development costs and efforts, while improving the productivity, adaptability and reliability of software systems.

In this setting, feature-oriented modelling techniques analyse commonalities and variabilities among products of a family [55] [104], whereas feature dependency analysis identifies the dependencies among features of a SPL [120]. The effectiveness of a software product line approach highly depends on how well features are managed throughout the development lifecycle [181]: the more independent the assets are, the easier the products may be built [46]. However, features may crosscut each other, making them dependant and reducing thus the flexibility, reusability and adaptability of the SPL assets [181] [46] [79] [121].

In that sense, several works have introduced the benefits of using aspect-oriented techniques to deal with crosscutting features, reducing dependencies between them [46] [87] [120] [126] [138] [181] (see a deeper analysis in Section 2.3). However, these proposals only focus on the modelling of variable features in SPL using aspect-orientation. But, as it is stated in [120], common features could be also modelled using aspect-oriented techniques (e.g. aspectual components) if they crosscut other features. Analogously, variable features need not to be defined always as crosscutting concerns. They may be effectively implemented in modular components if they do not crosscut other features. Accordingly, although the need of identifying crosscutting features in SPL has been demonstrated in previous works, all the aforementioned approaches just analyse the benefits of incorporating aspect-oriented techniques in SPL and they do not deal with the identification of the crosscutting features (common or variable).

Moreover, most of the aforementioned approaches (e.g. [87] [120] [126]) focus on programming or design stages, where architectural decisions have already been made, relegating the benefits of aspect-orientation to the latest phases of the development. However, as it has been shown above, crosscutting features manifest in early development artifacts, such as requirements descriptions [24] and architectural models [82][160], due to their widely-scoped influence in software decompositions. They can be observed in every kind of requirements and design representations, such as use cases and component models [16][24][82][160]. Thus, the incorporation of aspect-oriented techniques at early phases of development improves flexibility and reutilization of the product assets from beginning of the development. However, to the best of our knowledge, to date there is no empirical study

demonstrating the benefits of using aspect-oriented techniques in product families at these early stages of development.

This section shows the application of the crosscutting pattern to the SPL domain in order to identify the crosscutting features of a product family [52]. Once the crosscutting features have been identified, they may be modelled using aspect-oriented techniques so that dependencies between these features are highly reduced. In order to apply the crosscutting pattern to the SPL domain, the aspect mining process presented in Section 5.1 is used. We just need to adapt the process to the SPL domain changing some of the steps by other tasks specific of the SPL domain. As an example, at the beginning of the process a Feature Oriented Domain Analysis is performed to identify the main features of the product family, both mandatory and variable.

5.3.1. The process to identify crosscutting features.

As we have mentioned in previous section, 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 102, 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.

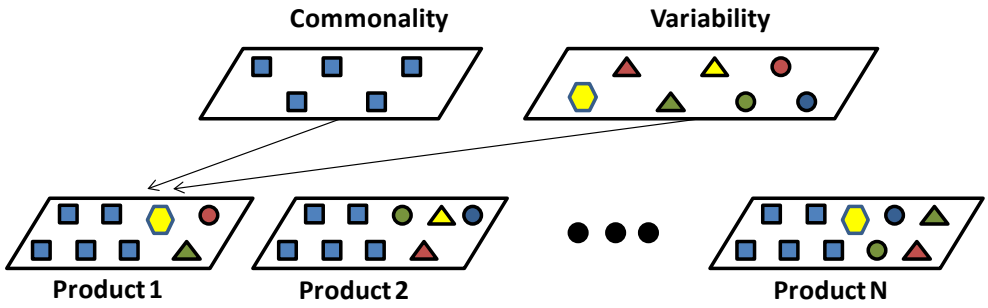


Figure 102. Combination of commonality and variability in product lines

In order to identify crosscutting features (both common and variable), we have adapted the aspect mining process presented in Section 5.1 to the SPL domain [52]. Since the crosscutting pattern is not tied to any specific deployment artefact, it may be used at any abstraction or domain level. Then, in this case, again we just need to decide what the source and target elements are. Taking into account the SPL domain, we considered as Source elements the different features of the product line, and as target elements the use cases implementing the product line (as we also did in Section 5.1). Then, the main changes to the aspect mining process are focused on the first tasks, where we identify the source elements for the analysis. In that sense, we add to the process a Feature Oriented Analysis (FOA) where the main features of the product line are identified. Then, these features will be completed also with the analysis of non-functional concerns presented in Section 5.1.2. Finally, both features and non-functional concerns are represented in a XML format. In Figure 103 we can see the outline of the process with the only change in task *A)* (Identifying source elements).

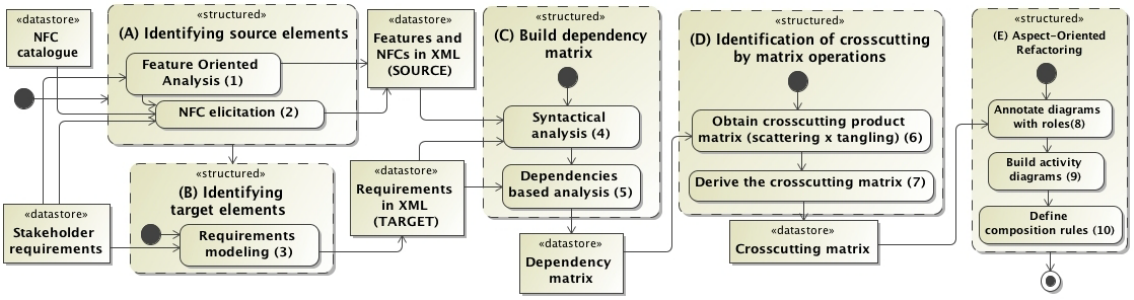


Figure 103. Main phases of the aspect mining process adapted to the SPL domain

Then, taking into account the FOA introduced in the process, next subsections illustrate the application of the process to two different case studies. The first example used is a well-know case study, the Arcade Game Maker product line company [9]. The second product line used is the MobileMedia system [71], a product line built in 8 different releases that allows the execution of different operations of a mobile device. The whole process will not be described here since it was already described in Section 5.1. We will focus on the first steps of the process (including the FOA) and on the results obtained in the rest of activities.

5.3.2. Crosscutting features in the Arcade Game Maker

The Arcade Game Maker (AGM) product line company [9] has been (fictional) created by the Software Engineering Institute at the Carnegie Mellon University [166] 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 section 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 [9]. In order to identify the crosscutting features in this system, we apply the steps shown in Figure 103.

Feature Oriented Analysis

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) [104]. 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 [9]. In this section 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 104, where we have represented the features that the product line has (Figure 103-(1)). As we can see in the figure, variability between products is mainly concentrated in the different rules that the games must fulfill. In [9], the authors of the case study performed a commonality and variability analysis. Based on this analysis, we summarise 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.
- Movement algorithm. The sprites which participate in the different games follow different rules to move throughout the game board.

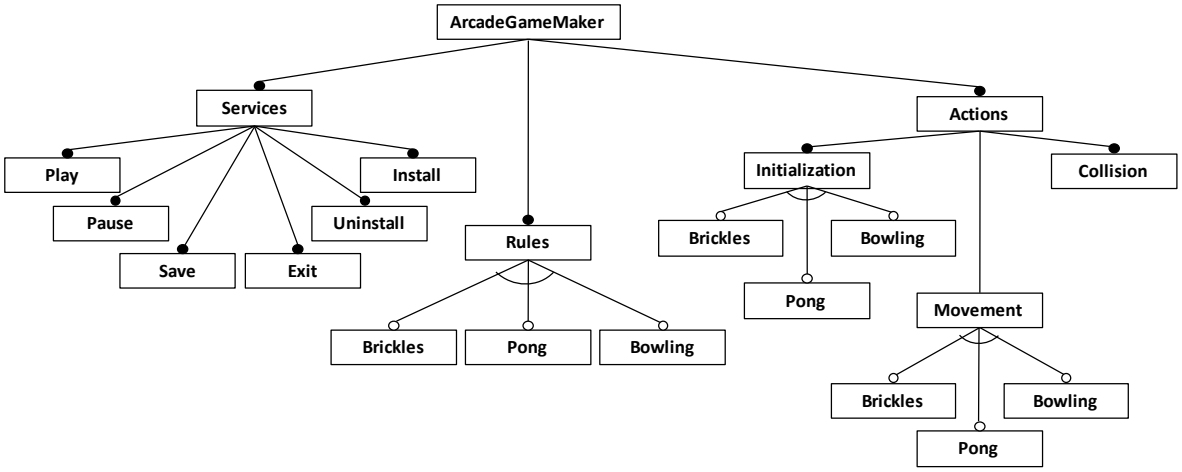


Figure 104. Features model for the Arcade Game Maker product line

As we did with concerns in the Aspect Mining Process, we represent the features in a XML file (Figure 105) with *<feature>* tags. The subelement *<keyword>* represents the words that we use to relate the source element with elements of the target domain.

```
<?xml version="1.0" encoding="UTF-8"?>
<FeaturesFile>
  <feature id="f2" name="Rules_Brickles">
   <description>Feature related to … </description>
   <stakeholder>
        <user>developer</user>
   </stakeholder>
   <keyword>Rule</keyword>
   <keyword>Brickles</keyword>
 </feature>
 ...
 <nfc id="c1" name="Persistence"
   <description>Way of storing … </description>
   <stakeholder>
        <user>developer</user>
   </stakeholder>
   <keyword>store</keyword>
   <keyword>retrieve</keyword>
 </nfc>
</FeaturesFile>
```

Figure 105. Feature and NFC in XML format

The rest of the steps in the process are the same that those explained in Section 5.1. Then, we will show just the results for each activity.

Non-functional concerns elicitation

In [9], the authors identified some non-functional concerns such as Performance, Display Quality, Evolvability and Maintainability. In addition to these non-functional concerns we analysed the requirements (using the NFC catalogue described in Section 5.1.2) of the AGM product line and identified other ones like Persistence (the game must be loaded or saved from file) and Data Presentation (the game board and the sprites must be represented) (Figure 103-(2)). Some of these non-functional concerns may be just related to design or hardware decisions, e.g. Evolvability and Maintainability may be addressed by means of using different developing approaches 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. The non-functional concerns are also represented in the XML features file, using <nfc> tags instead of <feature> (see Figure 105).

After the Feature-Oriented Analysis and the syntactical analysis performed using the NFRs catalogue, we have identified the elements of the source domain to apply the framework in the case study. These source elements are shown in Table 46.

Features				NFC	
f1.	Play	f7.	Rules_Brickles	nfc1.	Performance
f2.	Pause	f8.	Rules_Bowling	nfc2.	Data_Presentation
f3.	Save	f9.	Rules_Pong	nfc3.	Persistence
f4.	Exit	f10.	Initialization		
f5.	Install	f11.	Movement		
f6.	Uninstall	f12.	Collision		

Table 46. Features and NFRs of the AGM product line

Requirements modelling

In this activity the requirements engineer must build the first representation of the system (at the solution space) using some requirements language or notation (Figure 103-(3)). Again, we used UML use case diagrams to represent the requirements of the AGM product line (see Figure 106).

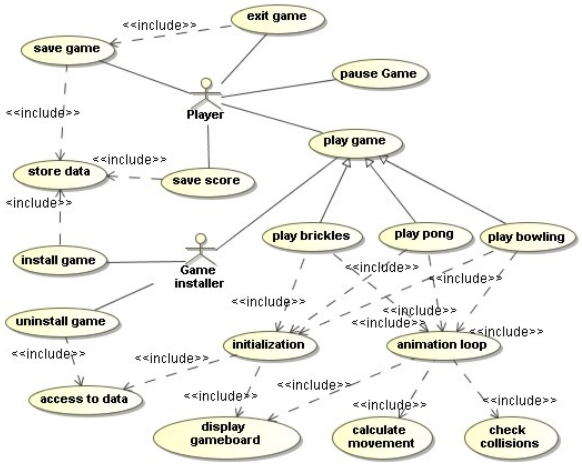


Figure 106. Use case diagram for the AGM product line (adapted from [9])

As we have done with features in previous activities, the elements of target are represented in a XML format, using XMI [185]. Figure 107 shows part of this XMI file.

```
<packagedElement xmi:type="uml:UseCase"
  xmi:id="_12_5_1_55e01df_1201781675343_652740_364" name="uninstall game">
  <include xmi:id="_12_5_1_55e01df_1204455631229_
    83791_865" name="" visibility="public"
    addition="_12_5_1_55e01df_1204455618761_136339_837"/>
</packagedElement>

<packagedElement xmi:type="uml:UseCase"
  xmi:id="_12_5_1_55e01df_1204455618761_136339_837" name="access to data"/>
```

Figure 107. XMI file generated from the use case diagram of Figure 106

Once the features of the products (including the non-functional requirements) have been identified and the requirements have been modelled, we use all this information to build the dependency matrix. In this case, the features and non-functional concerns are the source elements and the use case artefacts are the target elements. Then, what we do is to

automatically relate elements of the problem space to elements of the solution space (see Figure 108).

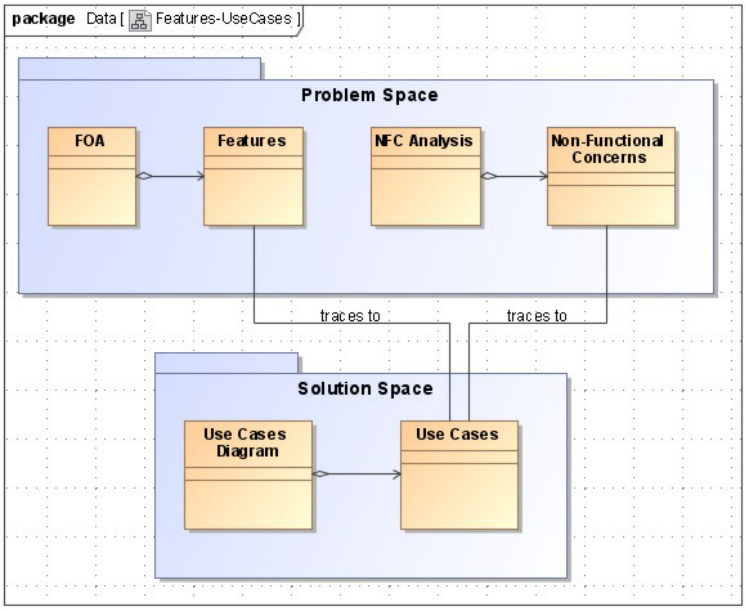


Figure 108. Relation between elements of the problem and solution spaces

Syntactical and dependency based analyses

As we explained in Section 5.1.4, the dependency matrix is automatically built by means of two different techniques: a syntactical analysis (Figure 103-(4)) and a dependencies-based analysis (Figure 103-(5)). The syntactical analysis uses the identifiers defined in the features and XMI files to relate these features (and NFCs) to the use case artefacts which contribute to them. Then, in this analysis we establish the relations between elements of source and target domains (inter-level relations). In our running example, the Arcade Game Maker, the mappings obtained by this analysis are shown in light grey background in Table 47.

		Use cases																
		Save Game	Exit Game	Save Score	Pause Game	Play Brickles	Play Bowling	Play Pong	Initialization	Animation Loop	Install	Uninstall	Store Data	Access to Data	Check Collisions	Display GameBoard	Calculate movement	
Features	Play					1	1	1										S
	Pause				1													
	Save	1	**1**	1														S
	Exit		1															
	Install										1							
	Uninstall											1						
	Rules_Brickles					1												
	Rules_Bowling						1											
	Rules_Pong							1										
	Initialization					**1**	**1**	**1**	1									S
	Movement					**1**	**1**	**1**		1							1	S
	Collision									**1**					1			S
NFR	Performance																	
	Data_Presentation								**1**	**1**						1		S
	Persistence	**1**		**1**							**1**	**1**	1	1				S
		T	**T**	**T**		**T**	**T**	**T**	**T**	**T**	**T**	**T**						

Table 47. Extended dependency matrix for the AGM after dependency analyses

The dependency-based analysis searches for dependency relations between elements of the target domain (intra-level relations). In this case, since the target elements are the use cases artefacts, we consider the <<include>> relations in the use case diagram (as shown in Section 5.1.4). After the dependency analysis, the mappings shown in dark grey background of Table 47 were identified.

Identification of crosscutting by matrix operations

Once the dependency matrix has been built, the crosscutting product (Figure 103-(6)) and crosscutting matrix (Figure 103-(7)) are obtained using the matrix operations of the conceptual framework. Remember that the crosscutting product matrix is obtained by the product of scattering and tangling matrices whilst the crosscutting matrix is derived from the crosscutting product one (just converting it into a binary matrix). In Table 48 and Table 49 depict the crosscutting product matrix and crosscutting matrix for the AGM product line respectively.

		Features												NFR's		
		Play	Pause	Save	Exit	Install	Uninstall	Rules_Brickles	Rules_Bowling	Rules_Pong	Initialization	Movement	Collision	Performance	Data_Presentation	Persistence
Features	Play	3						1	1	1	3	3				
	Pause															
	Save			3	1											2
	Exit															
	Install															
	Uninstall															
	Rules_Brickles															
	Rules_Bowling															
	Rules_Pong															
	Initialization	3						1	1	1	4	3			1	
	Movement	3						1	1	1	3	4	1		1	
	Collision											1	1		1	
NFC	Performance															
	Data_Present.										1	1	1		2	
	Persistence			2		1	1									4

Table 48. Crosscutting product matrix for the AGM product line

In Table 49 we can see how there are several features and non-functional concerns that may be candidates to be modularized by aspects. In some cases, the crosscutting features are part of the variability of the product line such as the different initialization or movement (they are different for each game). The addition of aspects to model such crosscutting features allows considerably reducing the number of dependencies between features. However since the framework may also identify as crosscutting features some common parts of the product line, we do not only focus on variability.

		Features												NFR's		
		Play	Pause	Save	Exit	Install	Uninstall	Rules_Brickles	Rules_Bowling	Rules_Pong	Initialization	Movement	Collision	Performance	Data_Presentation	Persistence
Features	Play							1	1	1	1	1				
	Pause															
	Save				1											1
	Exit															
	Install															
	Uninstall															
	Rules_Brickles															
	Rules_Bowling															
	Rules_Pong															
	Initialization	1						1	1	1		1			1	
	Movement	1						1	1	1	1		1		1	
	Collision											1			1	
NFC	Performance															
	Data_Present.										1	1	1			
	Persistence			1		1	1									

Table 49. Crosscutting matrix for the AGM product line

Aspect-oriented refactoring of crosscutting features

Once the crosscutting features have been identified, the method described in Section 5.1.6 is used to refactor these features and to modularize them in separated use cases. In particular, the use cases where these features are implemented are marked as roles with the special symbol "|"(Figure 103-(8)). A new relation <<crosscut>> is also added to the diagram to relate the use cases marked to those which are crosscut by the former. In Figure 109 we can see a part of the use case diagram for the AGM marked and with the new <<crosscut>> relations. In this diagram, the use cases implementing Initialization feature and Persistence NFC have been marked.

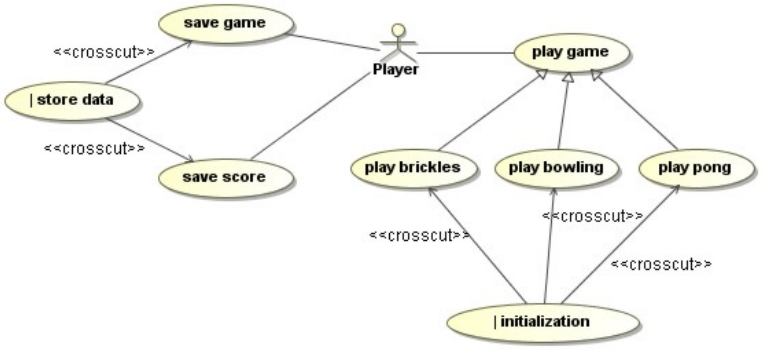


Figure 109. Marked use case diagram for the AGM product line

Then, the different products of the line may be built just composing the different features. In particular, we use composition rules to compose the activity diagrams which represent the behaviour of the use case diagrams (Figure 103-(9)). In Figure 110 and Figure 111 we can observe the activity diagrams for the save score and store data use cases respectively.

Figure 110. Activity diagram for save score use case

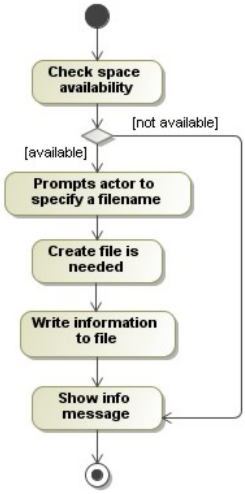


Figure 111. Activity diagram for store data use case

Finally, using the composition rule shown in Figure 112 we can merge the activity diagrams to add the Persistence feature to the product. We could also change the persistence policy (e.g. using a database instead of files) just changing the activity diagram for store data use case and the composition rule used to build the final product (Figure 103-(10)).

Compose Save score **with** Store data
Insert Check space availability **before** Return to pre-SAVE status

Figure 112. Composition rule for the activity diagrams in the AGM product line

5.3.3. Crosscutting features in the MobileMedia system

The MobileMedia [71] is a product line system built to allow the user of a mobile device to perform different options, such as visualizing photos, playing music or videos, and sending photos by SMS (among other concerns). It has about 3 KLOC. The MobileMedia application has been used for performing different analyses in software product lines mainly at architectural and programming level [71][72]. Our work complements those previous analyses since we focus on modularity at the requirements level. The system has been built as a product line in 8 different releases. In Table 50 the different releases of the system are described. Table 51 shows the concerns added in each release (see [71] for more details).

Release	Description
r0	MobilePhoto core
r1	Error handling added
r2	Sort Photos by frequency and Edit Label concerns added
r3	Set Favourites photos added
r4	Added a concern to copy photo to an album
r5	Added a concern for sending photos by SMS
r6	Added the concern for playing music
r7	Added the concern for playing videos and capture media

Table 50. Different releases of MobileMedia

Concern	Releases	Concern	Releases
Album	r0 - r7,	Copy	r4 - r7
Photo	r0 - r7,	SMS	r5 - r7
Label	r0 - r7,	Music	r6, r7
Persistence	r0 - r7,	Media	r6, r7
Error Handling	r1 - r7	Video	r7
Sorting	r2 - r7	Capture	r7
Favourites	r3 - r7		

Table 51. Concerns and releases where are included

This section shows the application of the aspect mining process to one of the 8 different releases of this product line. In this section we focus on release 3 to show the identification of crosscutting features in this product line. This release includes the functionality to manage albums and photos and some other optional features like sort photos by frequency, edit labels

and set favourite photos. This release was selected because it presents some variable features and includes the presence of non-functional concerns. Nevertheless, the release is simple enough to not complicate the explanation of the process. Anyway, in [64] the results of the application of the process to the whole system (including all the releases) are shown.

Feature Oriented Analysis

Since this system has been used in previous analyses, we utilize the same feature model (Figure 103-(1)) used by the original authors [71] (shown in Figure 113). Note that variability between products is mainly concentrated in the possibility of sorting photos, setting the favourites and editing labels.

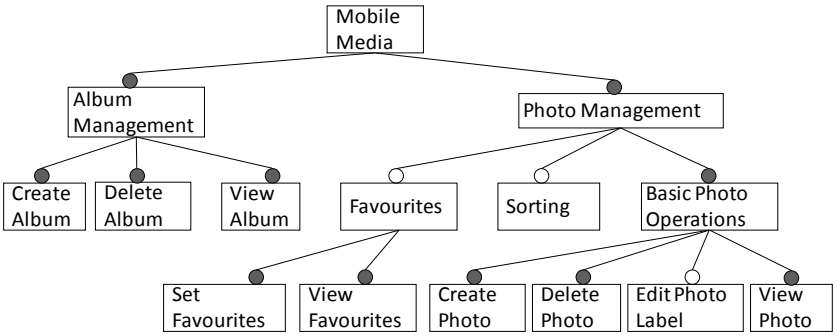


Figure 113. Features model for release 3 in MobileMedia

Non-functional concerns elicitation

Using the requirements of the system, we have identified some non-functional concerns: Persistence and Error Handling (Figure 103-(2)). Persistence is present in the system since the photos or any media file must be stored in the mobile memory. Error Handling is added in release 1 and it is included in the rest of releases (from 2 to 7). Then, after the analysis of the requirements, the source elements have been identified. These source elements are shown in Table 52. This table shows the features and non-functional concerns identified in the higher and lower parts, respectively.

Feature or NFC	Description	
f1.	Album	Features derived from the FOA analysis (Figure 103-(1)).
f2.	Photo	
f3.	Label	
f4.	Sorting	
f5.	Favourites	
nfc1.	Persistence	Non-functional concerns automatically derived by means of keywords analysis using a catalogue (Figure 103-(2)).
nfc2.	Visual Representation	

Table 52. Features and non-functional concerns in MobileMedia (release 3)

Requirements modelling

The use case diagram (Figure 103-(3)) that represents the requirements for release 3 in the MobileMedia can be observed in Figure 114.

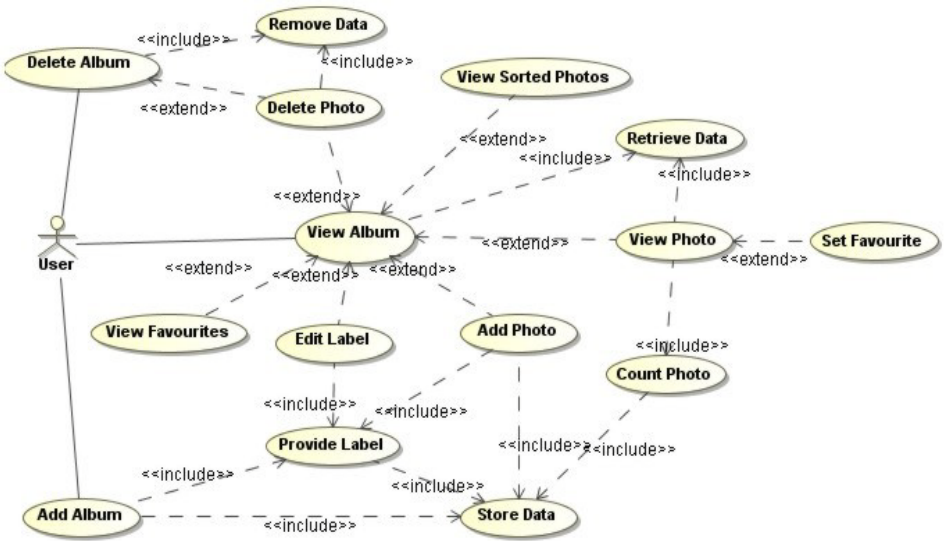


Figure 114. Use case diagram of the MobileMedia system

Build the dependency matrix

Again, the syntactical and dependency analyses are performed to obtain the dependency matrix. The dependency matrix after the syntactical analysis (Figure 103-(4)) is shown in Table 53 whilst the extended dependency matrix using the dependency analysis (Figure 103-(5)) is shown in Table 54 (new mappings are shown in grey background).

		Use cases														
		Add Album	Delete Album	Add Photo	Delete Photo	View Photo	View Album	Provide Label	Store Data	Remove Data	Retrieve Data	Edit Label	Count Photo	View Sorted Photos	Set Favourite	View Favourites
Features	Album	1	1				1									
	Photo			1	1	1										
	Label							1				1				
	Sorting												1	1		
	Favourites														1	1
NFC	Persistence								1	1	1					
	Error Handling															

Table 53. Dependency matrix after the analyses

		Use cases														
		Add Album	Delete Album	Add Photo	Delete Photo	View Photo	View Album	Provide Label	Store Data	Remove Data	Retrieve Data	Edit Label	Count Photo	View Sorted Photos	Set Favourite	View Favourites
Features	Album	1	1				1									
	Photo			1	1	1										
	Label	1		1				1				1				
	Sorting					1							1	1		
	Favourites														1	1
NFC	Persistence	1	1	1	1	1	1	1	1	1	1		1		1	
	Error Handling	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1

Table 54.Extended matrix after dependency analysis

In some cases, there are some NFC that do not explicitly appear in the use case diagram. The process helps to identify these situations since the dependency matrix would have a null row (without any mapping). For instance, in MobileMedia, Error Handling is not explicitly present in the use cases so that it would not be related to any use case. These situations may be solved by manually reviewing the use case descriptions or templates. We identified that all the use cases were constrained by the Error Handling NFC (then we added the mappings shown in black cells in last row of Table 54). In other cases, the NFC could be related to an architectural or hardware decision (e.g. using a particular hardware platform to deal with performance or an architectural pattern to improve reusability) and thus it would not constrain the software modularity.

Identification of crosscutting by matrix operations

Using the dependency matrix obtained by the previous analyses and the operations introduced by the conceptual framework, the crosscutting product matrix (Figure 103-(6)) and the crosscutting matrix (Figure 103-(7)) obtained for the MobileMedia release 3 are shown in Table 55 and Table 56 respectively.

		Features					NFC		
		Album	Photo	Label	Sorting	Favourites	Persistence	Error Handling	
Features	Album	3		1			3	3	M
	Photo		3	1	1		3	3	M
	Label	1	1	4			3	4	V
	Sorting		1		3		2	3	V
	Favourites					2	1	2	V
NFC	Persistence	3	3	3	2	1	12	12	
	Error Handling	3	3	4	3	2	12	15	

Table 55. Crosscutting product matrix for MobileMedia

		Features					NFC		
		Album	Photo	Label	Sorting	Favourites	Persistence	Error Handling	
Features	Album			1			1	1	M
	Photo			1	1		1	1	M
	Label	1	1				1	1	V
	Sorting		1				1	1	V
	Favourites						1	1	V
NFC	Persistence	1	1	1	1	1		1	
	Error Handling	1	1	1	1	1	1		

Table 56. Crosscutting matrix for MobileMedia

These tables firstly confirm what intuition perceives: NFC Persistence and Error Handling are the elements which crosscut more features. These tables also show how Mandatory features (Album and Photo) may crosscut Variable features (e.g. Label or Sorting) and vice verse. This situation suggests the use of aspect-oriented techniques to isolate and refactor NFCs and crosscutting features. Isolating a certain crosscutting feature removes the crosscutting dependencies between features. However, if two given features A and B are crosscutting each other, what feature should be refactored, A or B? In Section 5.4, we show how to take such decisions by an empirical analysis driven by the set of concern metrics introduced.

Of course, as it is discussed in Section 5.1.7, the aspect mining process could obtain false positives or negatives, which is a very common situation in aspect mining approaches. As an example, in [71] the authors used the MVC architectural pattern for implementing the MobileMedia system. They considered this issue as an important concern for the system (at architectural level). However, since this is a developer's design decision related to adaptability and maintainability of the system, this concern is not explicitly present in the requirements and it was not identified by our process.

Aspect-oriented refactoring of crosscutting features

Again, in this step we refactor the crosscutting concerns by isolating their implementation using Pattern Specification [79]. The marked use case diagram for the MobileMedia example is illustrated in Figure 115. Note that the use cases implementing the crosscutting concerns Label, Storing and Persistence have been marked. Note that Album and Photo concerns have

not been considered for refactoring since they are affected by Label, Storing and Persistence. Then, the refactoring of these three concerns makes that crosscutting is also removed from Album and Photo.

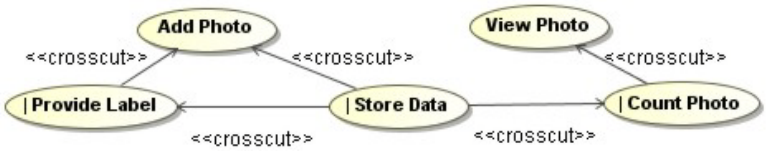


Figure 115. Use case diagram marked for the MobileMedia system

Figure 116 and Figure 117 show two activity diagrams which represent the main flows of the View Photo and Count Photo use cases (Figure 103-(9)) respectively. These activity diagrams are composed (Figure 103-(10)) using the composition rule shown in Figure 118. The addition or removal of the Sorting feature is as easy as applying or not the composition rule in the system, respectively. We could also use a different way of sorting photos just using a different composition rule and thus composing different activity diagrams.

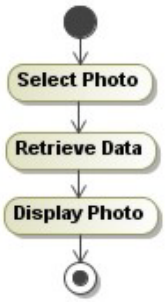


Figure 116. View Photo activity diagram

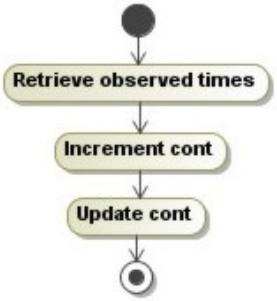


Figure 117. Count Photo activity diagram

Compose ViewPhoto **with** CountPhoto
Insert Retrieve observed times **after** Display Photo

Figure 118. Composition rule for the diagrams

5.3.4. Conclusions and discussion

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 process presented allows identifying crosscutting features in SPL in order to be modularized by aspect-oriented techniques. This process consists of several activities that assist the developer in the identification of the crosscutting features. These activities include a feature-oriented analysis, a non-functional concerns analysis, and a requirements modelling (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 concerns and the use cases which contributes to them. By simple matrix operations, we derive the crosscutting matrix where crosscutting features are identified. Finally, crosscutting features are refactored using aspect-oriented techniques so that they are isolated and encapsulated into separated entities reducing, thus, dependencies between features.

Once the process for identifying crosscutting features in SPL has been presented, the criteria used to compare other similar approaches are also used to discuss the process. These criteria were shown in Section 2.3.7 and its application to the different approaches for aspect-

oriented product lines is summarised in Table 6. The criteria and their relation with our approach are described below:

- **Evolvability:** evolvability is highly improved by using our approach. Note that crosscutting features introduce a special kind of relation or dependency between features. By isolating crosscutting features, these dependencies are reduced and their implementations are encapsulated into separated entities. Then, a change in these features does not imply to change all the crosscut features. Moreover, the utilization of traceability matrices allows having a visual representation of the impact of a change in a particular feature.
- **Verification:** since the main purpose of our process is the identification of crosscutting features, we have not dealt with the validation of the resulting product lines in this thesis. Note that the refactoring process just models the crosscutting features using isolated entities but it does not modify the final behavior of the product family. However, it could be easily tackled by using the features diagrams and adding configuration files to the process.
- **Scalability:** scalability is ensured by using different tools to support the process, e.g. for automatically discovering dependencies. Of course, there are manual parts in the process (e.g. identification of the functional concerns) that could compromise the process. However, as it has mentioned in Section 5.1.8, concerns are usually extracted from requirements documents (e.g. stakegolders' interviews) and different heuristics or techniques may be also used to automatically identify them. We are planning to complete the process in that sense (as a future work).
- **Traceability:** traceability is one of the main areas where the crosscutting pattern presents contributions. Note that traceability is highly improved by maintaining the links in traceability matrices. In addition, these links are automatically derived by using different kinds of analyses. Finally, the impact of a change in a feature may be also evaluated by using the matrices but also using several metrics, as it is illustrated in Section 5.5.3.
- **Abstraction level**: although our approach has been instantiated at the requirements level, the crosscutting pattern is not tied to any abstraction level so that it may be applied at different development stages.
- **Tool support**: the approach counts with tool support for dealing with the discovering of dependencies between source and target domains, e.g. syntactical analysis of use cases diagrams and identification of dependencies.
- **Crosscutting features modelled**: the approach considers as crosscutting features those that are identified by the process. That implies that any feature, mandatory or variable, could be modelled using aspect-oriented techniques.
- **Crosscutting identification process:** since the main puporse of this section is the identification of crosscutting features, of course, the approach counts with this process.

5.4. CONCERN-ORIENTED METRICS

It is claimed that crosscutting concerns often lead to harmful software instabilities, such as increased modularity anomalies [71] [86] and higher number of introduced faults [62]. However aspect-oriented community has not dedicated too much effort to empirically demonstrate how crosscutting concerns really affect to the software quality. Most of the systematic studies of crosscutting concerns (e.g. [63] [71] [83] [86]) concentrate on the analysis of source code, when architectural decisions have already been made [51]. Even worse, a survey of existing crosscutting metrics has pointed out that they are defined in terms

of specific object-oriented and aspect-oriented programming languages [72]. However, most of the crosscutting concerns manifest in early development artifacts, such as requirements descriptions [24] and architectural models [160] [82], due to their widely-scoped influence in software decompositions. They can be observed in every kind of requirements and design representations, such as usecases and component models [24] [160] [82] [16].

There are few works which have defined crosscutting metrics for early design representations, nevertheless these metrics are very specific to certain models, such as component-and-connector models [160]. These metrics are overly limited as many crosscutting concerns are visible in certain system representations, but not in others [72]. Thus, the use of aspect-oriented decompositions cannot be straightforwardly applied without proper assessment mechanisms for early software development stages. In that sense, in this section we complement our conceptual framework with a set of concern-oriented metrics. These metrics allow the developer to assess different modularity attributes such as the degree of scattering or crosscutting. These metrics are generic and they are not tied to any specific deployment artefact so that they may be used at any abstraction level (not only source code). Moreover, since these metrics may be applied at early stages of development, they may be used to anticipate important decisions related to other quality attributes (such as maintainability characteristics, as we show in Section 5.5) [51].

Our concern-oriented metric suite is based on the relation between source and target domains represented by the crosscutting pattern. In order to illustrate the metrics, we rely on requirement descriptions of the running example shown in the previous section, the MobileMedia.

Although the concern metrics introduced here have been presented in a separated section, actually, they are included into the aspect mining process presented before. In that sense, the metrics are used to decide whether a concern should be refactored (in a step before the aspect-oriented refactorizing). In particular, this section shows how the concerns with higher values for the *Degree of Crosscutting* metric presented should be firstly considered for being refactored. Moreover, the refactoring of the implementation of these concerns usually involves that crosscutting is also removed from the implementation of other crosscutting concerns. The same could be said considering tangling or scattering metrics. It is obvious that those use cases where more concerns are tangled should be considered for performing a refactoring (externalizing the implementation of the crosscutting concerns). Even, the metrics allow considering the definition of threshold values as a limit to identify crosscutting concerns (those with values for the metrics higher than the threshold).

5.4.1. The MobileMedia System

As it has been aforementioned, the MobileMedia [71] is a product line system built to allow the user of a mobile device to perform different options, such as visualizing photos, playing music or videos, and sending photos by SMS (among other concerns). In this section we show a simple usecase diagram (Figure 119) which corresponds to a part of the usecase diagram used for release 0 in the MobileMedia system. In this part of the diagram, the actions for adding albums and photos to the system are implemented. These actions include the option for providing a label. Some actions for recording the data into a persistent storage are also included. Then, we consider that four main concerns are involved in this part of the system: album, photo, label, and persistence.

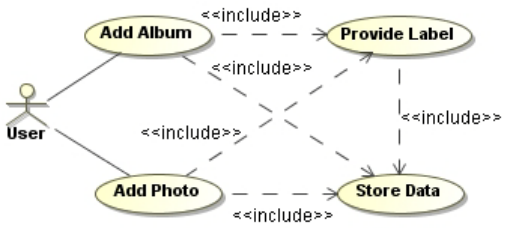


Figure 119. Simplification of the usecase diagram for release 0 in MobileMedia

In Table 57, we show a simplified description of the usecases shown in Figure 119. We have shadowed in light and dark grey colors the flows and relations in these usecases corresponding to Label and Persistence concerns, respectively. Although there are other two concerns involved in the example (album and photo), we have not shadowed them to keep the example clear.

Usecase: Add Album	**Usecase:** Add Photo	**Usecase:** Provide Label	**Usecase:** Store Data
Actor: Mobile Phone (system) and User	**Actor:** Mobile Phone (system) and User	**Actor:** Mobile Phone (system) and User	**Actor:** Mobile Phone (system)
Description: The user can store (add) an album to the mobile phone	**Description:** User can store (add) a photo in an album available	**Description:** The user provides label for the photo and album	**Description:** The data of a photo or an album must be stored into the device storage
Pre/Posconditions: (…)	**Pre/Posconditions: (…)**	**Pre/Posconditions: (…)**	**Pre/Posconditions: (…)**
Basic flows:	**Basic flows:**	**Basic flows:**	**Basic flow:**
1. (add)The user selects the option to add an album.	1. (select) The user selects an album to store the photo.	1. (label) The users provides a name for a photo or an album	1. (space) The device select a space in the storage
2. (label) User provides label to the new created album	2. (add)The user selects the option to add photo.	2. (save) The edited label is saved	2. (save) The data are saved in the storage
3. (saved) A new album is available	3. (path) User provides the path for uploading the photo.	**Includes:**	
4. (listing) The list of photos is displayed	4. (label) User assigns a label to the photo.	1. Store Data usecase	
Includes:	5. (saved) The new photo is stored in the album.		
1.Provide Label usecase	**Includes:**		
2. Store Data usecase	1. Provide Label usecase		
	2. Store Data usecase		

Table 57. Usecase descriptions for Add Album, Add Photo and Provide Label usecases

In order to clarify the metrics presented in next sections, each metric is illustrated using the described MobileMedia example. In particular, we use release 3 and the partial use case diagram shown in Figure 119. In that sense, the values for each metric are calculated using the use case descriptions presented in Table 57. The whole system with the 8 releases is used later on to validate the metrics so that their values are calculated for all the releases.

5.4.2. Metrics for Scattering

According to Definition 1 in Section 3.1.2, *Nscattering* metric is proposed. *Nscattering* of a source element s_k is defined as the number of 1's in the corresponding row (k) of the dependency matrix:

$$NScattering\ (s_k) = \sum_{j=1}^{|T|} dm_{kj} \qquad (1)$$

where $|T|$ is the number of target elements and dm_{kj} is the value of the cell [k,j] of the dependency matrix. We may also express this metric according to the functions defined in Section 3.1.2 as *NScattering* (s_k) = *card* $\{t \in Target : f'(s_k)=t\}$, i.e. *card*$(f(s_k))$.

This metric measures how scattered a concern is. In the MobileMedia example (shown in Section 5.4.1), the *NScattering* for Label and Persistence concerns is 3 and 4, respectively. As we can see in Table 57, all the usecases descriptions have some flows or relations shadowed with dark grey (related to Persistence), however there are only 3 usecases with light grey (related to Label). Then we may assure that the Persistence concern is more scattered than Label.

This *NScattering* metric can be normalized in order to obtain a value between 0 and 1. Then, we define *Degree of scattering* of the source element s_k as:

$$Degree\ of\ scattering\ (s_k) = \begin{cases} \dfrac{\sum_{j=1}^{|T|} dm_{kj}}{|T|} & if \ \sum_{j=1}^{|T|} dm_{kj} > 1 \\ 0 & if \ \sum_{j=1}^{|T|} dm_{kj} = 1 \end{cases} \quad (2)$$

The closer to zero this metric for a source element (i.e., a concern), the better encapsulated the source element. Conversely, when the metric has a value closer to 1, the source element is highly spread over the target elements and it is worse encapsulated. Using the same aforementioned example, *Degree of Scattering* for Label and Persistence is 0,75 and 1 respectively. This metric could have been also defined in terms of the scattering matrix (instead of dependency matrix). The results would be the same.

In order to have a global metric for how much scattering the system's concerns are, we define the concept of *Global scattering (Gscattering)* which is obtained just by calculating the average of the *Degree of scattering* values for each source elements:

$$Gscattering = \frac{\sum_{i=1}^{|S|} Degree\ of\ scattering(s_i)}{|S|} \quad (3)$$

where $|S|$ is the number of analysed source elements. In our particular case, it represents the number of concerns of interest in the system.

5.4.3. Metrics for Tangling

Similarly to *Nscattering* for scattering, we also defined the *Ntangling* metric for the target element t_k, where $|S|$ is the number of source elements and dm_{ik} is the value of the cell [i,k] of the dependency matrix:

$$Ntangling\ (t_k) = \sum_{i=1}^{|S|} dm_{ik} \quad (4)$$

Again, according to the functions introduced in Section 3.1.2, *Ntangling* (t_k) = *card* $\{s \in Source : g'(t_k)= s\}$, *i.e., card*$(g(t_k))$. Then, this metric measures the number of source elements addressed by a particular target element. In the MobileMedia example, the *NTangling* for the Add Album and Store Data usecases are 2 and 1, respectively. As we can see in Table 57, Store Data usecase is only shadowed in dark grey color so that it just addresses the Persistence concern (whilst Add Album addresses Persistence and Label).

The same steps performed for the scattering metrics are followed now to define two tangling metrics: *Degree of tangling* and *Gtangling*. These metrics represent the normalized tangling for the target element t_k and the global tangling, respectively:

$$Degree\ of\ tangling\ (t_k) = \begin{cases} \frac{\sum_{i=1}^{|S|} dm_{ik}}{|S|} & if \quad \sum_{i=1}^{|S|} dm_{ik} > 1 \\ 0 & if \quad \sum_{i=1}^{|S|} dm_{ik} = 1 \end{cases} \qquad (5)$$

$$Gtangling = \frac{\sum_{j=1}^{|T|} Degree\ of\ tangling(t_j)}{|T|} \qquad (6)$$

Like *Degree of scattering*, the *Degree of tangling* metric may take values between 0 and 1, where the value 0 represents a target element addressing only one source element. The number of source elements addressed by the target element increases as the metric is closer to 1.

5.4.4. Metrics for Crosscutting

Finally, this section defines three metrics for crosscutting: *Crosscutpoints*, *NCrosscut* and *Degree of crosscutting*. These metrics are extracted from the crosscutting product matrix and the crosscutting matrix of the framework presented in Section 3.1.2.

The *Crosscutpoints* metric is defined for a source element s_k as the number of target elements where s_k is crosscutting other source elements. This metric is calculated from the crosscutting product matrix (remember that this matrix is calculated by the product of scattering and tangling matrices). The *Crosscutpoints* metric for s_k corresponds to the value of the cell in the diagonal of the row k (cell [k,k] or $ccpm_{kk}$).

$$Crosscutpoints(s_k) = ccpm_{kk} \qquad (7)$$

According to our running example, we can see that *Crosscutpoints* metric for Persistence has a value of 3. Note that there are three usecases descriptions (Table 57) which are shadowed with both light and dark color (Add Album, Add Photo and Provide Label). Then, the Persistence and Label concerns cut across each other in these usecases.

The *NCrosscut* metric is defined for the source element s_k as the number of source elements crosscut by s_k. The *NCrosscut* metric for s_k is calculated by the addition of all the cells of the row k in the crosscutting matrix:

$$NCrosscut(s_k) = \sum_{i=1}^{|S|} ccm_{ki} \qquad (8)$$

In our example, *NCrosscut* for Persistence is 1 since it is crosscutting just to the Label concern. Finally, the two crosscutting metrics above allow us to define the *Degree of crosscutting* metric of a source element s_k. Note that, *Degree of crosscutting* is normalized between 0 and 1, so that those source elements with lower values for this metric are the best modularized.

$$Degree\ of\ crosscutting(s_k) = \frac{Crosscutpoints(s_k) + Concerns\ crosscut(s_k)}{|S| + |T|} \qquad (9)$$

We summarise all our metrics in Table 58. In this table we show the definition of each metric and the relation with the matrices used by the crosscutting pattern.

Metric	Definition	Relation with matrices	Calculation
NScattering (s_k)	Number of target elements addressing source element s_k	Addition of the values of cells in row *k* in dependency matrix (dm)	$= \sum_{j=1}^{\lvert T\rvert} dm_{kj}$
Degree of scattering (s_k)	Normalization of NScattering (s_k) between 0 and 1		$= \begin{cases} \frac{\sum_{j=1}^{\lvert T\rvert} dm_{kj}}{\lvert T\rvert} & if \ \sum_{j=1}^{\lvert T\rvert} dm_{kj} > 1 \\ 0 & if \ \sum_{j=1}^{\lvert T\rvert} dm_{kj} = 1 \end{cases}$
Gscattering (s_k)	Average of Degree of scattering of the source elments		$= \frac{\sum_{i=1}^{\lvert S\rvert} Degree\ of\ scattering(s_i)}{\lvert S\rvert}$
NTangling (t_k)	Number of source elements addressed by target element t_k	Addition of the values of cells in column *k* in dependency matrix (dm)	$= \sum_{i=1}^{\lvert S\rvert} dm_{ik}$
Degree of tangling (t_k)	Normalization of NTangling (t_k) between 0 and 1		$= \begin{cases} \frac{\sum_{i=1}^{\lvert S\rvert} dm_{ik}}{\lvert S\rvert} & if \ \sum_{i=1}^{\lvert S\rvert} dm_{ik} > 1 \\ 0 & if \ \sum_{i=1}^{\lvert S\rvert} dm_{ik} = 1 \end{cases}$
Gtangling (t_k)	Average of Degree of tangling of the target elments		$= \frac{\sum_{j=1}^{\lvert T\rvert} Degree\ of\ tangling(t_j)}{\lvert T\rvert}$
Crosscutpoints (s_k)	Number of target elements where the source element s_k crosscuts other source elements	Diagonal cell of row *k* in the crosscutting product matrix (ccpm)	$= ccpm_{kk}$
NCrosscut (s_k)	Number of source elements crosscut by the source element s_k	Addition of the values of cells in row *k* in the crosscutting matrix (ccm)	$= \sum_{i=1}^{\lvert S\rvert} ccm_{ki}$
Degree of crosscutting (s_k)	Addition of the two last metrics normalized between 0 and 1		$= \frac{ccpm_{kk} + \sum_{i=1}^{\lvert S\rvert} ccm_{ki}}{\lvert S\rvert + \lvert T\rvert}$

Table 58. Summary of the concern-oriented metrics based on the Crosscutting Pattern

Once the metrics have been presented, next section shows the process followed to validate them. In particular next section shows how the metrics are theoretically validated.An empirical validation of the metrics has been also conducted and shown in Section 5.5.

5.4.5. Metrics evaluation

Any software measurement process must be properly validated. Software metrics validation has been traditionally performed by two different approaches [31] [96] [103]: theoretical and empirical validations. By the theoretical validation, the accuracy of the metrics for measuring its original purpose is proven. In other words, this validation demonstrates that the metrics measure what it is expected. This first validation is also usually called an internal validation. Secondly, the empirical validation evaluates the utility of the metrics related to other quality attributes. In other words, it demonstrates that the metrics are useful so that they help in some goals related to these other quality attributes (assessment, prediction, ...)[31].

In that sense, this section shows a theoretical validation of the metrics where the accuracy of the metrics for measuring crosscutting properties is evaluated. For this validation, we compare the application of our framework (including the calculation of the metrics) to the 8 releases of the MobileMedia product line system. We compare the results obtained by our metrics with those obtained by other authors' metrics. These metrics defined by other authors are firstly summarised in next section.

On the other hand, in Section 5.5 the metrics are empirically validated demonstrating the utility of the metrics for maintainability analyses [51]. In particular, this validation is carried out

by correlating the degree of scattering and crosscutting of the different concerns with stability and changeability measures.

5.4.5.1. Compiling existing concern-driven metrics

In this section several concern-oriented metrics [58] [62] [159] [160] [183] which have been used in our theoretical validation are presented. These metrics are summarised in Table 59 and they were all deeply described in Section 2.4. Unlike the metrics presented in this thesis, the metrics summarised in Table 59 are mainly defined in terms of specific design or implementation artefacts. Accordingly, these metrics have been adapted to the requirements level in order to compare the results obtained by our metrics with those obtained by the rest of metrics (in Section 5.4.5.4). The adaptation has mainly consisted of a change in the target element used for the different measures. For example, if a metric was defined for measuring concepts using components or classes, we have adapted the metric to the requirements domain by using usecases as the target entity. We have also taken into account the different granularity levels used by the metrics. For instance, there are some metrics which use operations or lines of code (instead of components or classes) as the target entity. In order to adapt these metrics, operations are changed by usecase flows or steps, since flows represent a finer granularity level (similar to operations or lines of code) than usecases. Then, the granularity level used at requirements keeps consistent with the used by the original metrics.

Authors		Metric	Definition
Sant'Anna et al.	[159]	Concern Diffusion over Components (CDC)	It counts the number of components addressing a concern.
		Concern Diffusion over Operations (CDO)	It counts the number of methods and advices addressing a concern.
		Concern Diffusion over Lines of Code (CDLOC)	It counts the number of lines of code related to a particular concern.
	[160]	Lack of Concern Cohesion (LOCC)	It counts the number of concerns addressed by the assessed component.
		Component-level Interlacing Between Concerns (CIBC)	It counts the number of other concerns with which the assessed concerns share at least a component.
Ducasse et al. [58]		Size	It counts the number of internal members of classes (methods or attributes) associated to a concern.
		Spread	It counts the number of modules (classes or components) related to a particular concern
		Focus	It measures the closeness between a module and a property or concern
		Touch	It assesses the relative size of a concern or a property (Size of property divided into total size of system)
Wong et al. [183]		Concentration (CONC)	It measures how much a concern is concentrated in a component
		Dedication (DEDI)	It quantifies how much a component is dedicated to a concern.
		Disparity (DISP)	It measures how many blocks related to a particular property (or concern) are localised in a particular component
Eaddy et al. [62]		Degree of scattering (DOS)	It is defined as the variance of the Concentration of a concern over all program elements with respect to the worst case
		Degree of tangling (DOT)	It is defined as the variance of the Dedication of a component for all the concern with respect to the worst case

Table 59. Survey of metrics defined by other authors

5.4.5.2. The case study

In this section the results obtained by calculating our metrics to the MobileMedia system [188] are shown. As we have already mentioned, our work complements previous analyses [71][72] performed at architectural and programming level using the MobileMedia application since we focus on modularity at the requirements level. The reason for calculating the metrics at this level is to identify the concerns with a higher *Degree of Crosscutting* (a poor modularity) as

soon as possible. Then, the developer may anticipate important decisions regarding quality attributes at early stages of development (illustrated also in Section 5.5).

As it is aforementioned, MobileMedia has evolved to 8 successive releases by adding different concerns to the product line. For instance, release 0 implements the original system with just the functionality of viewing photos and organizing them by albums. In order to remember the system, Table 60 shows again the different releases whilst Table 61 shows the concerns added in each release. These tables were previously presented in Section 5.3.3. The reasons for choosing this application for our first analysis are several: (1) The MobileMedia application is a product line, where software instability is of upmost importance (instability an changeability are maintainability characteristics used in our analysis in Section 5.5); instabilities affect negatively not only the Software Product Line (SPL) architecture, but also all the instantiated products. (2) The software architecture and the requirements had all-encompassing documentation; e.g., the description of all the usecases were made available as well as a complete specification of all the component interfaces. (3) The architectural components were independently defined and provided by the real developers, rather than ourselves. The architectural part could be used by a second study to analyse traceability of crosscutting concerns (observed using our metrics). (4) The author of the studies in [71][72] had implemented an aspect-oriented version of the system, which may be also used in a different analysis for comparing the metrics applied in different paradigms.

Release	Description
r8	MobilePhoto core
r9	Error handling added
r10	Sort Photos by frequency and Edit Label concerns added
r11	Set Favourites photos added
r12	Added a concern to copy photo to an album
r13	Added a concern for sending photos by SMS
r14	Added the concern for playing music
r15	Added the concern for playing videos and capture media

Table 60. Different releases of MobileMedia

Concern	Releases	Concern	Releases
Album	r0 - r7,	Copy	r4 - r7
Photo	r0 - r7,	SMS	r5 - r7
Label	r0 - r7,	Music	r6, r7
Persistence	r0 - r7,	Media	r6, r7
Error Handling	r1 - r7	Video	r7
Sorting	r2 - r7	Capture	r7
Favourites	r3 - r7		

Table 61. Concerns and releases where are included

We have calculated not only our metrics but also those introduced by other authors for the requirements of each release. We have considered the different concerns of each release and the usecases implementing the system as the source and target domains, respectively. Table 61 shows the concerns used for the analysis and the releases in which these concerns were included. The whole analysis of the experiment may be found in [64].

5.4.5.3. Calculating the Metrics

Based on the two domains (concerns and usecases as source and target, respectively) the dependency matrix for each release is built showing the usecases contributing to the different concerns. Our metrics and those summarised in Table 59 are automatically calculated using as input the dependency matrix. Based on this dependency matrix, we derive the rest of matrices: Scattering, Tangling, Crosscutting Product, and Crosscutting Matrices. In this section we just show the global results and those obtained for release 7 (which includes all the concerns of the system). However, the whole analysis performed is available in [64]. The dependency matrix obtained for the MobileMedia in release 7 is shown in Table 62.

Although our original dependency matrix is a binary matrix, in this case we have used a not-binary matrix in order to allow the calculation of metrics which utilize a granularity level different from usecase (e.g. Eaddy's Degree of Scattering or Degree of Tangling). That means that a cell represents the number of control flows or steps of the usecase addressing a

particular concern. For instance, in Table 62 we can see how the View Album usecase has 3 and 1 control flows addressing the Album and Label concerns, respectively.

		Usecases																					
		Add Album	Delete Album	Add Media	Delete Media	View Photo	View Album	Provide Label	Store Data	Remove Data	Retrieve Data	Edit Label	Count Media	View Sorted Media	Set Favourite	View Favourites	Copy Media	Send Media	Receive Media	Music Control	Access Media	Play Video	Capture Media
Concerns	Album	2	2	1			3																
	Photo					1																	
	Label	2		2			1	1				1					2		1				
	Persistence	2	2	2	2	2	2	2	2	2	2		2		2		3	1			2		
	Error Handling	2	2	3	2	2	1	2	2	2	2	1	1	1	1	1	2	2	2		2		1
	Sorting					1	1						1	2							1		
	Favourites						2								1	1							
	Copy					1											1				1		
	SMS					1												1	1		1		
	Music																			3			
	Media		1	2	1		3												2		3		1
	Video																					1	
	Capture																						1

Table 62. Dependency matrix for the MobileMedia system in release 7.

Using this dependency matrix, we automatically calculate all the metrics for this release. In Table 63 the results obtained for metrics calculated for concerns (source elements in the Crosscutting Pattern) are illustrated. Table 64 shows the results obtained for the metrics calculated for use cases (target elements).

Release	7											
Authors	Conejero					Sant'Anna			Ducasse			Eaddy
Metrics / Concerns	Nscattering	Degree of scattering	Crosscutpoints	NCrosscut	Degree of crosscutting	Concern Difussuion over Usecases	Concern Difussion over Flows	Usecase level Interlacing between Concerns	Size	Spread	Focus	Degree of Scattering
Album	3	0,136	3	6	0,25	3	8	6	8	3	0,232	0,752
Photo	1	0	0	0	0	1	1	5	1	1	0,125	0
Label	7	0,318	7	8	0,416	7	10	8	10	7	0,234	0,88
Persistence	16	0,727	16	9	0,694	16	30	9	30	16	0,355	0,975
Error Handling	20	0,909	20	10	0,833	20	34	10	34	20	0,349	0,989
Sorting	5	0,227	5	9	0,388	5	6	9	6	5	0,314	0,814
Favourites	3	0,136	3	6	0,25	3	4	6	4	3	0,264	0,654
Copy	3	0,136	3	7	0,277	3	3	7	3	3	0,116	0,698
SMS	4	0,181	4	7	0,305	4	4	7	4	4	0,160	0,785
Music	1	0	0	0	0	1	3	0	3	1	1	0
Media	7	0,318	7	9	0,444	7	13	9	13	7	0,256	0,867
Video	1	0	0	0	0	1	1	0	1	1	1	0
Capture	1	0	0	0	0	1	1	2	1	1	0,333	0
GScattering		0,27			0,34							

Table 63. Metrics calculated for concerns of MobileMedia in release 7

Release	7			
Authors	Conejero		Sant'Anna	Eaddy
Metrics / Use cases	Ntangling	Degree of Tangling	LOCC	Degree of Tangling
Add Album	4	0,286	4	0,81
Delete Album	4	0,286	4	0,79
Add Photo	4	0,286	4	0,84
Delete Photo	3	0,214	3	0,69
View Photo	6	0,429	6	0,88
View Content	7	0,5	7	0,89
Provide Label	3	0,214	3	0,69
Store Data	2	0,143	2	0,54
Remove Data	2	0,143	2	0,54
Retrieve Data	2	0,143	2	0,54
Edit Label	2	0,143	2	0,54
Count Photo	3	0,214	3	0,67
View Sorted Photos	2	0,143	2	0,48
Set Favourite	3	0,214	3	0,67
View Favourite	2	0,143	2	0,54
Copy Photo	4	0,286	4	0,77
Send Photo	3	0,214	3	0,67
Receive Photo	5	0,357	5	0,78
Music Control	1	0	1	0
Access Media	6	0,429	6	0,86
Play video	1	0	1	0
Capture Media	3	0,214	3	0,72
GTangling		0,24		

Table 64. Metrics calculated for use cases of MobileMedia in release 7

In order to have a global view of the results obtained, the averages of the metrics for all the releases have been also calculated. These results are shown in Table 65 (metrics calculated for source elements) and Table 66 (metrics calculated for target elements).

Releases	Average of all releases											
Authors	Ours					Sant'Anna			Ducasse			Eaddy
Metrics / Concerns	Nscattering	Degree of scattering	Crosscutpoints	NCrosscut	Degree of crosscutting	Concern Difussuion over Usecases	Concern Difussion over Flows	Usecase level Interlacing between Concerns	Size	Spread	Focus	Degree of Scattering
Album	3,63	0,26	3,63	5,25	0,39	3,63	8	5,25	8	3,63	0,25	0,77
Photo	4,13	0,3	3,88	4,13	0,38	4,13	7,38	5,38	7,38	4,13	0,24	0,62
Label	5,38	0,34	5,38	6	0,46	5,38	7,88	6,13	7,88	5,38	0,25	0,82
Persistence	12,8	0,85	12,4	6,38	0,77	12,8	25,1	6,38	25,1	12,8	0,39	0,98
Error Handling	15,9	0,98	15,9	7	0,89	15,9	27,6	7	27,6	15,9	0,36	0,99
Sorting	4,33	0,25	4,33	7,33	0,43	4,33	5,33	7,33	5,33	4,33	0,34	0,78
Favourites	3	0,17	3	6	0,32	3	4	6	4	3	0,26	0,66
Copy	2,5	0,13	2,5	6,25	0,29	2,5	2,5	6,25	2,5	2,5	0,09	0,61
SMS	3,67	0,18	3,67	6,67	0,32	3,67	3,67	6,67	3,67	3,67	0,16	0,76
Music	1	0	0	0	0	1	3	0	3	1	1	0
Media	6,5	0,31	6,5	8,5	0,44	6,5	12,5	8,5	12,5	6,5	0,25	0,85
Video	1	0	0	0	0	1	1	0	1	1	1	0
Capture	1	0	0	0	0	1	1	2	1	1	0,33	0
GScattering		0,29			0,362							

Table 65. Average of source metrics for all the releases

Release	Average of all releases			
Authors	Conejero		Sant'Anna	Eaddy
Metrics / Use cases	Ntangling	Degree of Tangling	LOCC	Degree of Tangling
Add Album	3,875	0,475	3,875	0,846
Delete Album	3,875	0,475	3,875	0,826
Add Photo	4,5	0,568	4,5	0,886
Delete Photo	2,875	0,350	2,875	0,702
View Photo	4,5	0,506	4,5	0,833
View Content	6	0,694	6	0,900
Provide Label	2,875	0,350	2,875	0,702
Store Data	1,875	0,200	1,875	0,494
Remove Data	1,875	0,200	1,875	0,494
Retrieve Data	1,875	0,200	1,875	0,494
Edit Label	2	0,211	2	0,559
Count Photo	3	0,316	3	0,699
View Sorted Photos	2	0,211	2	0,497
Set Favourite	3	0,294	3	0,693
View Favourite	2	0,196	2	0,554
Copy Photo	4	0,365	4	0,791
Send Photo	3	0,254	3	0,683
Receive Photo	4,666	0,391	4,666	0,789
Music Control	1	0	1	0
Access Media	6	0,464	6	0,867
Play video	1	0	1	0
Capture Media	3	0,214	3	0,717
GTangling		0,45		

Table 66. Average of target metrics for all the releases

5.4.5.4. Conclusions and discussion

The main goal of the measures shown in previous section is to analyse the consistency of the crosscutting metrics with respect to other authors' metrics. However, by means of this validation we have also extracted important conclusions about the used metrics:

- First of all, we have observed through an analytical comparison (which was confirmed by an analysis of the MobileMedia data) that our proposed metrics are generic enough to embrace existing code-level metrics currently used in studies based on source-code analysis [62] [83] [86] [159]. Examples of these metrics are Sant`Anna's *Concern Diffusion over Components* or Eaddy's *Degree of Scattering*. Observe that these metrics are defined in terms of specific development artefacts. On the other hand, our metrics are defined in terms of the development level-agnostic crosscutting pattern (presented in Chapter 3). In that sense, the metrics presented allow anticipating the modularity analysis to the early stages of development, such as requirements.

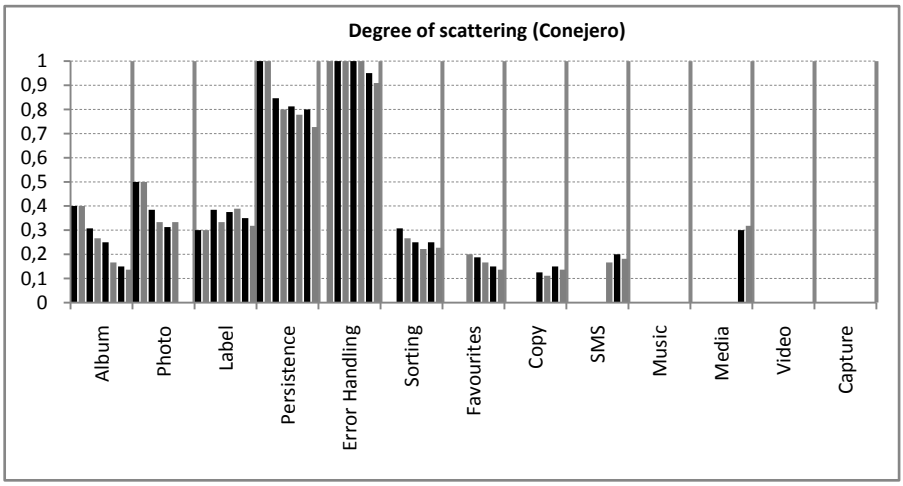
- As we can see in the above tables, some of the metrics are equivalent or proportional to other ones. For instance, our *NScattering* metric, *Concern Diffusion over Components* and *Spread* measure the same concept and they obtain the same results. If we focus on a finer granularity level, we may see how the metrics *Concern Diffusion over Flows* and *Size* also obtain the same values so that we can conclude that they are equivalent (they measure similar concepts). We also concluded that our *Degree of Scattering* metric is, in general, proportional to Eaddy's *Degree of Scattering* and inversely proportional to *CONC*. Obviously, the *Degree of Scattering* metric is also proportional to *NScattering* (note that it is just a normalization of this metric). We can see in the tables that there are two features which have values for our *Degree of Scattering* and Eaddy's *Degree of Scattering* which do not follow the general tendency. These features are Photo and Media. We explain in Section 5.5 why these features behave in a special way. Altough we have not showed

CONC and *DEDI* metrics, we also concluded and empirically demonstrated from the data that these metrics were proportional to Focus whereas Focus was inversely proportional to *DISP.* For target elements we also found similarities between different metrics. As an example, we can see how the *NTangling* and *LOCC* metrics assess the same concept (they obtain the same results) and also that our *Degree of Scattering* and Eaddy's *Degree of Scattering* metrics tend towards the same values. These two metrics are also inversely proportional to *DEDI*.

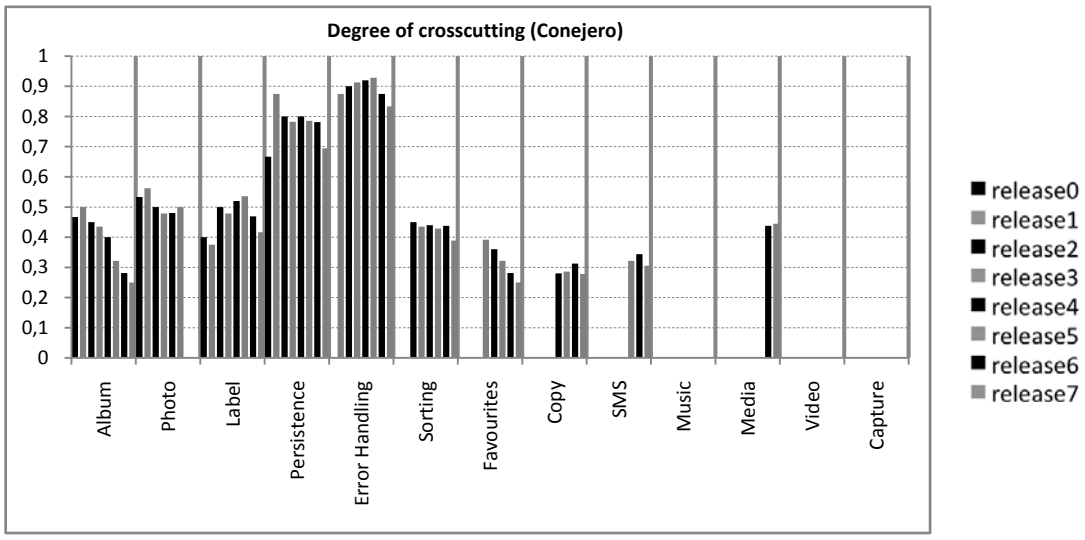
- From Table 63 to Table 66 we have shown the metrics which are more interesting for extracting conclusions on source and target elements. Focusing on source elements, we can see how there are several features or concerns with high values for Degree of Scattering metrics (ours and Eaddy's) and for Degree of Crosscutting. Some examples are Persistence, Error Handling or Media. As it is aforementioned, note that the higher the values for our metrics (Degree of Scattering or Degree of Crosscutting) are, the higher Eaddy's or Sant'Anna's metrics as well. Then, we can assure that the metrics are behaving as it was expected. In Figure 120 we show three graphics where Degree of Scattering and Degree of Crosscutting metrics are represented in order to have a more general view of the metrics. Using these graphics we can intuitively see how the metrics tend towards the same values for the same features. We also show tangling metrics in Figure 121. We can also see in these charts how the metrics calculated for target elements are also consistent.

- We may also see that the values for Eaddy's Degree of Scattering are in general considerably higher than the values for our Degree of Scattering metric. This is due to the fact that the granularity levels used for assessing each metric are different. Whilst Eaddy's metric has been adapted for counting control flows in the use cases, our metric uses a coarser granularity level (use cases). Our conclusion was that these metrics may complement each other in different situations where different granularity levels are needed. For instance, Eaddy's metric have been defined for applying them at programming level (Lines of Code). However, in earlier phases of the development (like requirements), we encourage to utilize metrics with a coarser granularity level.

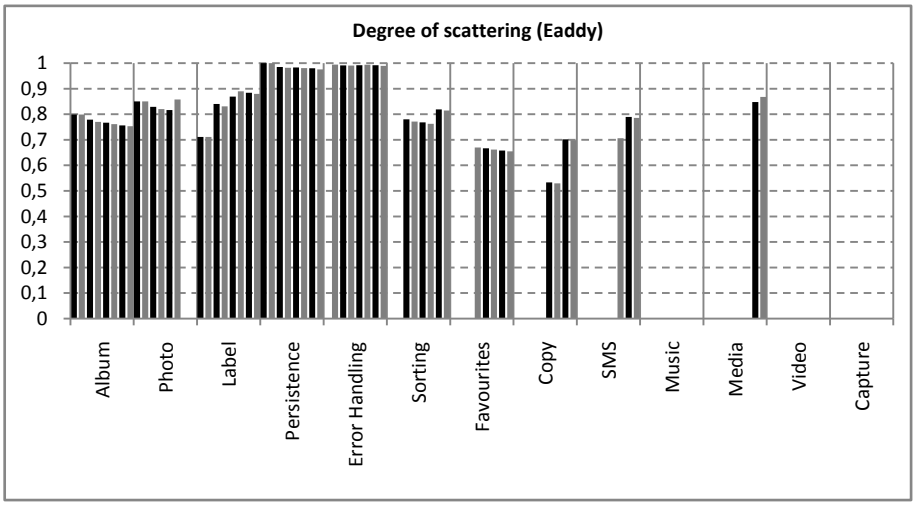
- One important conclusion that we have extracted from the analysis is the need for using the Degree of Crosscutting metric (exclusive of our metrics suite). This metric is calculated using Crosscutpoints and NCrosscut metrics (see Section 5.4.4), and it is a special combination of scattering and tangling metrics. Figure 122 shows the Degree of Scattering and Degree of Tangling metrics for releases 0 and 1. Note that in these releases, the Album concern presents the same value for Degree of Scattering. However, the Degree of Crosscutting metric for this concern is higher in release 1 than in release 0 (see Figure 123). This is due to the tangling of the use cases where the Album concern is addressed (see in Figure 122 b). Accordingly, we observed that the Album concern is worse modularized in release 1 than in release 0 (there are other examples, such as Persistence or Photo). Note that this situation could not be discovered using only the Degree of Scattering metric. Although the combination of the Degree of Scattering and Degree of Tangling metrics could help to disclose the problem, it would be a tedious task since the metrics do not provide information about which target elements are addressing each source element. Thus, the utilization of Degree of Crosscutting allows the detection of this problem just observing the values for this metric. The same analysis could be done for Eaddy's metrics since they do not have a specific metric for crosscutting (see the Album concern in releases 0 and 1, in Figure 121 c).



a)



b)



c)

Figure 120. Charts showing *Degree of Scattering* (ours and Eaddy's) and *Degree of Crosscutting*

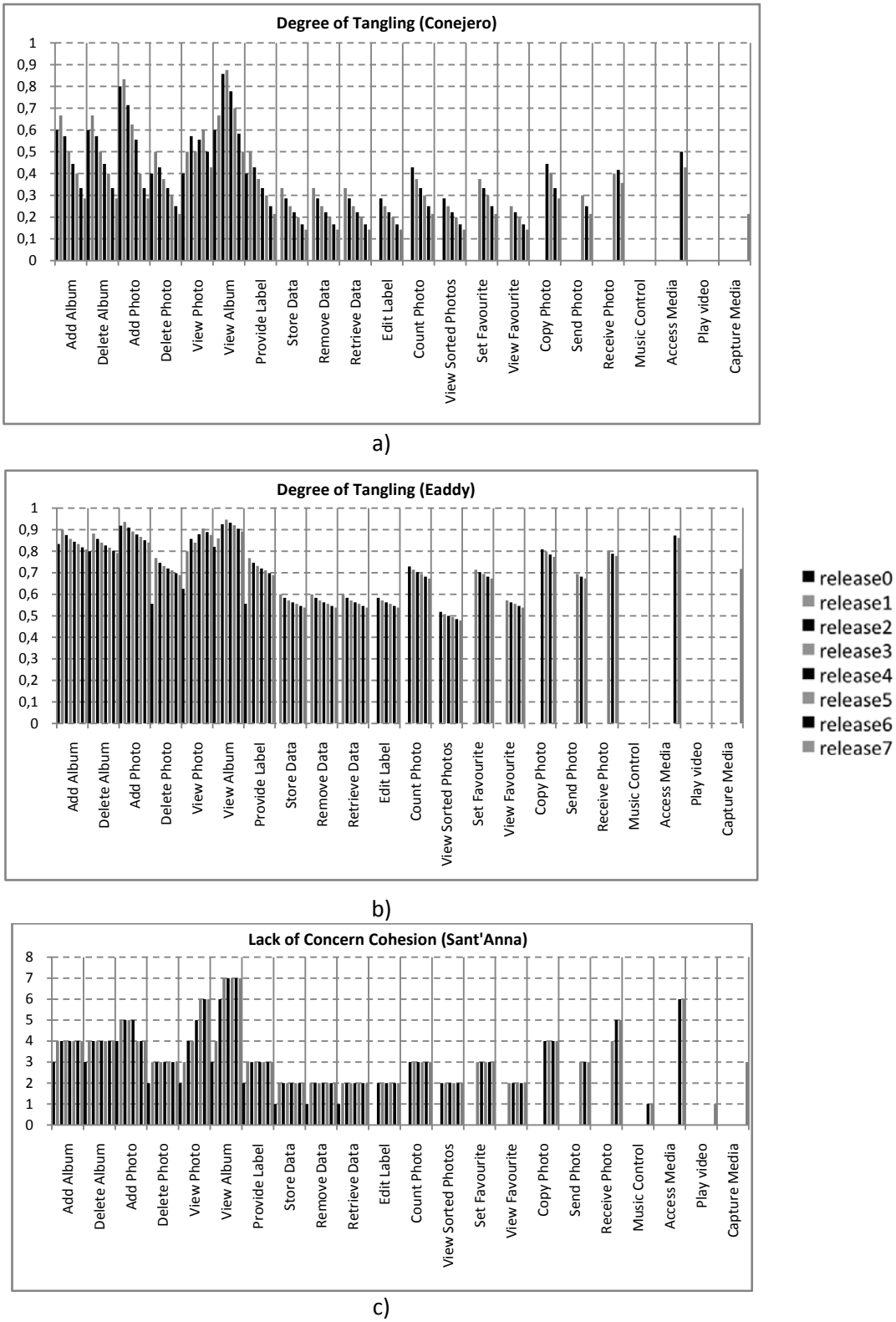


Figure 121. Charts showing the *Degree of Tangling* (ours and Eaddy's) and *Lack of Concern Cohesion* metrics

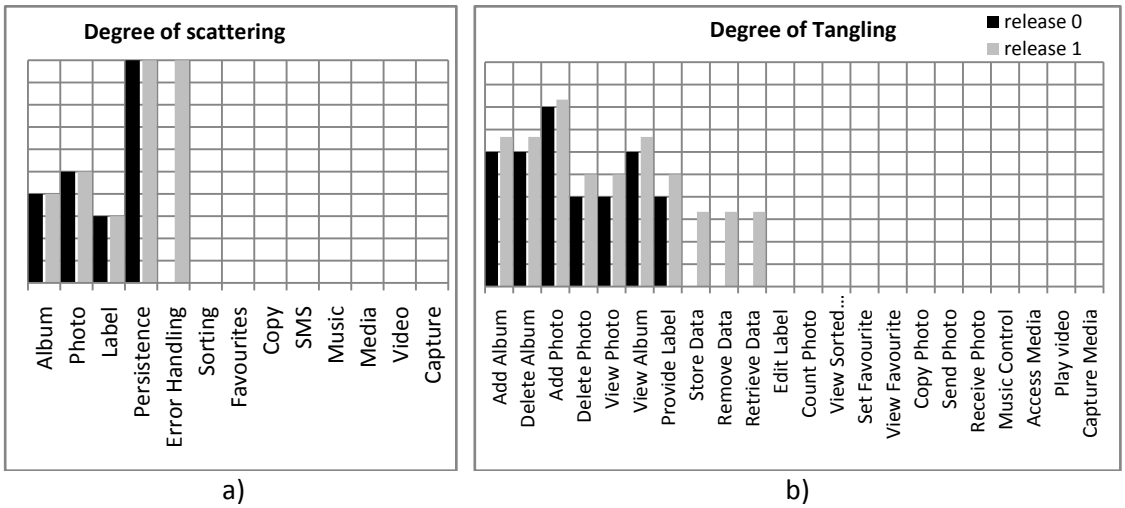


Figure 122. *Degree of Scattering* and *Degree of Tangling* for releases 0 and 1

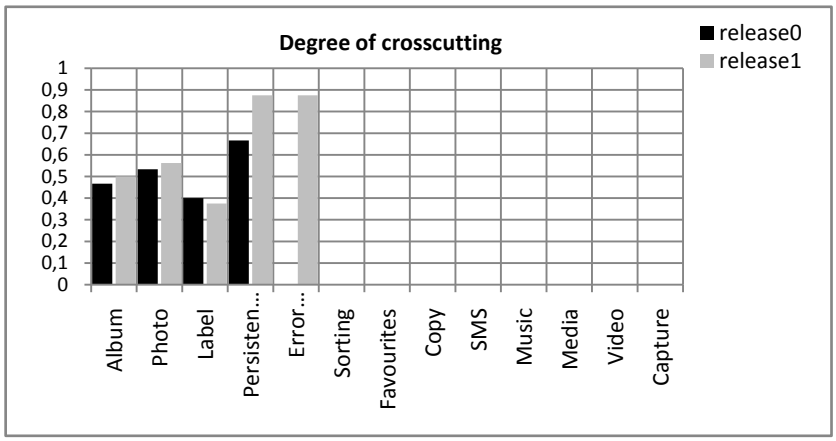


Figure 123. *Degree of Crosscutting* for releases 0 and 1

5.5. SUPPORTING MAINTAINABILITY ANALYSES BY USING CONCERN-DRIVEN METRICS

The international standard ISO/IEC 9126 [96] categorises software quality attributes into a set of characteristics such as functionality, reliability, usability or maintainability, among others. In particular, maintainability is defined as *the capability of the software product to be modified*. These modifications include corrections, improvements or adaptation of software to changes in environment, requirements or functional specifications. Moreover, maintainability is divided into a number of sub-characteristics which may be measured by software metrics, more notably stability and changeability. Stability is defined *as the capability of software product to avoid unexpected effects from modifications of software*; changeability is defined *as the capability of the software product to enable a specified modification to be implemented*.

In this setting, this section reports an empirical study on the usefulness of the concern-driven metrics presented in Section 5.4. The goal is to evaluate to what extent they support the anticipation of further problems associated with the quality attributes, namely stability and changeability. The main goal of the analysis is to observe how our concern-oriented metrics may help in predicting maintainability attributes based on the analysis of early SPL artefacts (in this case we focus on the SPL domain since the case study is the MobileMedia system, widely introduced in previous sections). We observed that these metrics provide indications that the crosscutting features negatively affect to software quality. This hypothesis

is tested by using a three-dimensional evaluation: (1) an analysis of the crosscutting metrics in terms of their predictability power of software stability [108]; (2) an evaluation showing the ability of the metrics for predicting change impact (i.e. supporting changeability improvements) [22]; (3) an evaluation of the metrics capability to assess feature dependencies [71] and how these dependencies affect to stability and changeability.

The first analysis is performed by empirically observing the actual changes produced in a system and relating these changes to the degree of scattering and crosscutting (the analysis is performed once the changes have produced, "a posteriori").

The second evaluation is carried out by observing the relations between source and target domains and it provides a new metric for anticipating the set of impacted elements for a change in a particular source element ("a priori", without the need of performing the change). Moreover, this last analysis provides important traceability information since it analyses the elements affected by a change in a source element. This is even more important in the SPL domain where family products may be built just changing features. Moreover, the results obtained by the analyses were consistent showing that the features with a higher impact set were also those implemented by more unstable use cases. In addition, those features had, as expected, a higher degree of crosscutting.

Finally, a third empirical analysis is performed where feature dependencies are compared with our *Degree of Crosscutting* metric. The goal of this analysis is to probe whether crosscutting introduces dependencies between features, making reusability of the features difficult. In particular, the analysis shows evidences of crosscutting is harmful for concern interlacing [71], an important characteristic of software product lines. Moreover, the consequences of adding feature dependencies is also empirically compared with the two aforementioned maintainability attributes, indicating that the reduction of dependencies between features (including crosscutting dependencies) enhances stability and changeability. This analysis comes to empirically confirm the assumptions made by several authors who claim that crosscutting adds dependencies between features, e.g. [46][121]. However, the analysis also shows that both mandatory and variable features may be considered as crosscutting features, in contract to the works that propose the utilization of aspect-oriented techniques to model only variable features [46][87][121][126][138][181].

5.5.1. Stability analysis

To date, there is no empirical study that investigates whether scattering and crosscutting negatively affect to software stability in software product lines at requirements level. In this section, our analysis shows that the concerns with a higher degree of scattering and crosscutting are addressed by more unstable use cases than concerns with lower degree of scattering and crosscutting. Stability is highly related to change management so that the more unstable a system is, the more complicated the change management becomes [22]. Unstable models gradually lead to the degeneration of the design maintainability and its quality in general [22]. Then, we can infer that crosscutting has also a negative effect on software quality. In this analysis we mainly focus on changes in the functionality of the system. It is important to highlight that our focus is not on corrective changes performed to remove bugs or on perfective modifications. However, we do not rule out this kind of changes in future analyses and they should be the target of further studies.

In order to perform our empirical study, we have shown in Table 67 the use cases (rows) which change in the different releases (columns). A change in a use case is mainly due to: either (i) the concerns, which it addresses, have evolved, or (ii) it has been affected by the addition, modification or removal of a concern to the system. In this table, a '1' in a cell represents that the corresponding use case has changed in that release. As an example, in

Release 1 (r1) all the cells in the column present the value 1. This is due to the fact that Error Handling is added in this release, and this concern affects all use cases. An "a" in a cell represents that the use case is added in that release. There are also some use cases which change their names in a release. These use cases are marked in the "Renaming" column, where the release which introduces the change in the name is shown (e.g., Add Photo use cases changes its name to Add Media in Release 6). Finally, use cases with a number of changes higher than a threshold value (e.g., 2 in our analysis) are marked as unstable.

		Releases									
Renaming	Requirements Element	r0	r1	r2	r3	r4	r5	r6	r7	#Changes	Unstable?
	Add Album	a	1	0	0	0	0	0	0	1	no
	Delete Album	a	1	0	0	0	0	0	0	1	no
r6	Add Photo [Media]	a	1	0	0	0	0	1	0	2	yes
r6	Delete Photo [Media]	a	1	0	0	0	0	1	0	2	yes
	View Photo	a	1	1	0	1	1	1	0	5	yes
	View Album	a	1	1	1	0	0	1	0	4	yes
	Provide Label	a	1	0	0	0	0	0	0	1	no
	Store Data	a	1	0	0	0	0	0	0	1	no
	Remove Data	a	1	0	0	0	0	0	0	1	no
	Retrieve Data	a	1	0	0	0	0	0	0	1	no
	Edit Label			a	0	0	0	0	0	0	no
r6	Count Photo [Media]			a	0	0	0	1	0	1	no
r6	View Sorted Photo			a	0	0	0	1	0	1	no
	Set Favourites				a	0	0	0	0	0	no
	View Favourites				a	0	0	0	0	0	no
r6	Copy Photo [Media]					A	0	1	0	1	no
r6	Send Photo [Media]						a	1	0	1	no
r6	Receive Photo [Media]						a	1	0	1	no
	Play Music							a	0	0	no
	Access Media							a	0	0	no
	Play Video								a	0	no
	Capture Media								a	0	no

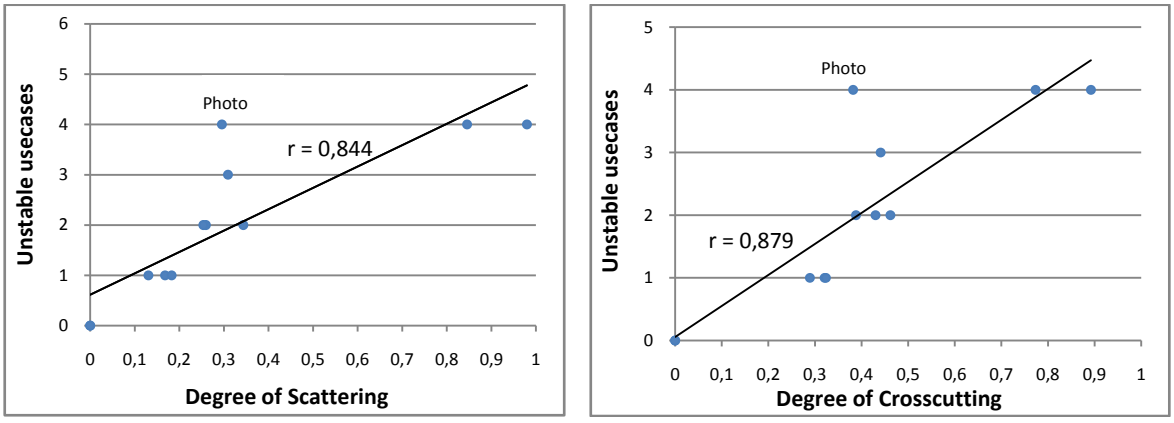
Table 67. Changes in use cases in the different releases

Once the changes affecting each use case are known, the number of unstable use cases which realise each concern is calculated. The columns of Table 68 show the unstable use cases, in particular, those with two changes or more. The concerns are shown in the rows. A cell with 1 represents that the use case addresses the corresponding concern. The last column of the table shows the total number of unstable use cases contributing to each concern.

		Use cases				
		Add Media	Delete Media	View Photo	View Album	Unstable use cases
Concerns	Album	1			1	2
	Photo	1	1	1	1	4
	Label	1			1	2
	Persistence	1	1	1	1	4
	Error Handling	1	1	1	1	4
	Sorting			1	1	2
	Favourites				1	1
	Copy			1		1
	SMS			1		1
	Music					0
	Media	1	1		1	3
	Video					0
	Capture					0

Table 68. Number of unstable use cases addressing each concern

We relate the number of unstable use cases for each concern with the degree of scattering and crosscutting for such concerns. In particular, Figure 124 shows the linear regression between the number of unstable use cases and the *Degree of scattering* and *Degree of crosscutting* metrics, respectively. We have used the least squares criteria to estimate the linear regression between the variables assessed so that the higher the degree of scattering or crosscutting for a concern is, the more unstable use cases addressing such a concern are. We can anticipate that use cases addressing scattered or crosscutting concerns are more prone to be unstable.



a) b)

Figure 124. Correlation between *Degree of scattering* and *Degree of crosscutting* and stability

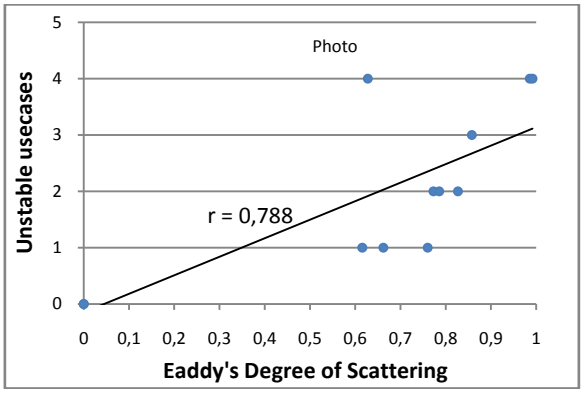


Figure 125. Correlation between Eaddy's Degree of Scattering and stability

The Eaddy's *Degree of Scattering* metric (introduced in Section 2.4.4) has been also related with stability (Figure 125). This figure complements the previously presented validation of the metrics (Section 5.4.5) by showing consistency in the correlations of Figure 124-(a) and Figure 125 (they follow the same tendency).

5.5.2. Discussion on stability analysis

In this section, we present some conclusions extracted from the analysis performed in the previous section.

- As we can see in Figure 124 and Figure 125, correlations follow a linear tendency so that the higher the degree of scattering or crosscutting for a concern is, the more unstable use cases addressing this concern are. This analysis allows the developer to decide which parts of the system are more unstable just observing the degree of scattering or crosscutting. Also, since the analysis is performed at the requirements level, the developer may anticipate important decisions about stability at this early

stage of development, improving the later architecture or detailed design of the system.

- Figure 124 and Figure 125 also show the value for Pearson's *r* (a common measure of the linear dependence between two variables) [172]. The values of *r* shown in Figure 124-(a) and Figure 124-(b) are 0.844 and 0.879 respectively. These values indicate that *Degree of scattering* and *Degree of crosscutting* are highly correlated with the number of unstable use cases. Using the critical values table for *r* [172], we calculated the probability after N measurements (in our case 13) that the two variables are not correlated. For Figure 124-a) and Figure 124-b), the value obtained for this probability is 0.1%. Accordingly, the probability that these variables are correlated is 99.9%. For Figure 125, we obtained that *r* is 0.788. Analogously, Eaddy's *Degree of Scattering* is also linearly correlated with the number of unstable components because the probability that the variables assessed in Figure 125 are correlated is 99.2%.
- It has been observed that, in general, it is obtained a higher correlation for the *Degree of crosscutting* with stability than for *Degree of scattering* with stability. After analysing the data, we observed that the correlations between *Degree of scattering* metrics (both ours and Eaddy's) and stability were much influenced by those concerns either without scattering or completely scattered. As an example, we can see in Figure 124-a) that there is a point with a *Degree of scattering* of almost 1 while most of the points present a *Degree of scattering* lower than 0.4. This situation is even more evident in Figure 125 where the correlation coefficient obtained is lower than for the other correlations. The reason is the difference between the values obtained for this metric in cases without scattering and the rest of cases. This metric obtained high values (greater than 0.5) for almost all the concerns assessed. However, when a concern does not present scattering the result of the metric is 0. This fact highly influences the correlation. Finally, we concluded that *Degree of crosscutting* presents a better correlation with stability since this metric somehow takes into account not only scattering but also tangling. This conclusion supports the need for having a specific metric for assessing crosscutting (introduced in Section 5.4.5).
- All the correlations in Figure 124 and Figure 125 have been also annotated with a point called *Photo*. These points are the most digressed from the linear regression in all the figures. We observed that although this concern (Photo) presents values for the scattering and crosscutting metrics not very high, the number of unstable use cases was high. After analysing this situation, we realized that this concern presents a high degree of scattering and crosscutting in the six first releases. After Release 5, a new concern is added (Media) which is responsible for addressing the actions common to photo, music and video, and carrying out many actions previously assigned to the Photo concern. This is why *Degree of scattering* and *Degree of crosscutting* for Photo drastically decrease in releases 6 and 7. It highly influences the average of the metrics and this is the reason why, although having non-relatively high values for the metrics, the number of unstable use cases remains high.

5.5.3. Changeability analysis

As it is aforementioned, change management is a prerequisite for high-quality software development. Software modificaitons may be caused by altering user requirements and business goals or be induced by changes in implementation technologies. In the SPL domain, changes affect negatively not only the product line architecture but also all the instantiated products. In fact, the quality of any product line depends largely on the importance attached to the development process and its evolution [1]. Then, the changeability analysis – i.e. an analysis of the impact of candidate changes - is necessary for cost effective software development [22].

In that sense, the use of traceability tools may help to anticipate the impact of a change in a feature. As an example, in [1] the authors annotate use cases with the features they are contributing to. In this setting, the dependency matrix used by our conceptual framework (Section 4.2) provides the different use cases where each feature (or non-functional requirement) is addressed. Then, the impact of a change in a feature is quantified by this matrix since it provides the set of use cases related to the feature. In this section, a changeability analysis is shown. In particular, the section shows how the features with higher impact of changes (no matter whether variable or mandatory) are those with higher *Degree of Crosscutting*. Therefore, these results support the need for identifying crosscutting features in product families improving, thus, the change management.

In order to have a way to assess the change impact of a feature, we use the idea presented in [22]. This work is also based on the crosscutting pattern introduced in Section 3.1.1 and it defines change impact in terms of the elements involved in the change of a source element. Assuming there are two related domains (source and target), the change impact of a source element is defined as the addition of the number of trace links that must be changed and the number of trace links that must be preserved. In Figure 126 an example of several source and target elements related is shown. A change in the source element s1 implies to change target elements t1, t3 and t4 (there are three trace links from s1 to these target elements). However, since s3 is also addressed by two of the target elements changed, the trace links from s3 to t3 and from s3 to t4 must be also preserved, so that s3 is also affected by the change of s1: *impact (s1) = change ((s1,t1), (s1,t3), (s1,t4)) + preserve ((s3,t3), (s3,t4)).*

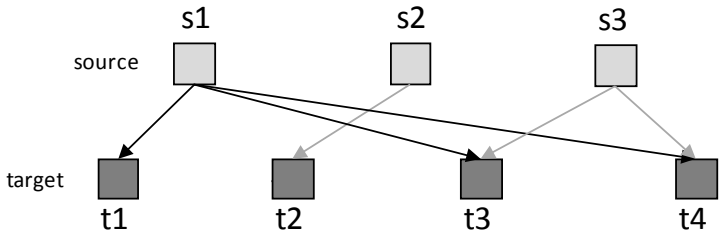


Figure 126. A dependency graph with three and four source and target elements respectively

Using this assumption, the change impact of a source element may be quantified using our dependency matrix. In particular, the number of trace links affected by a change of a source element s_k is calculated counting the number of 1 values of row k and the number of 1 values in the columns where there are 1's in row k. This may be formally defined as:

$$Impact\ Set\ (s_k) = \sum_{i=1}^{|T|}\left(\mathrm{dm}_{\mathrm{ki}}\left(\sum_{\mathrm{j}=1}^{|S|}\mathrm{dm}_{\mathrm{ij}}\right)\right) \qquad (10)$$

Applying formula 10, the impact of each feature in the different releases of MobileMedia has been calculated. In Table 69, we show these results. In this table we show, for each feature, the number of trace links affected by a change in the feature. These trace links include both the links to be changed and the links to be preserved. In this table, an M, V, or O has been shown with each feature indicating whether the feature is mandatory, variable or optional respectively. Note also that the *Degree of Crosscutting* for each feature is also shown in the table in order to be compared with the *Impact Set* metric.

Finally, taking the results obtained by the change impact analysis, both *Impact Set* and *Degree of Crosscutting* measurements are correlated. The results are shown in Figure 127 where we can see the correlations for the 8 different releases.

	Release1		Release2		Release3		Release4		Release5		Release6		Release7		Release8	
Features	Impact Set	Degree of Crosscutting	Impact Set	Degree of Crosscutting	Impact Set	Degree of Crosscutting	Impact Set	Degree of Crosscutting	Impact Set	Degree of Crosscutting	Impact Set	Degree of Crosscutting	Impact Set	Degree of Crosscutting	Impact Set	Degree of Crosscutting
Album (M)	13	0,46	17	0,50	19	0,45	20	0,43	20	0,40	15	0,32	15	0,28	15	0,25
Photo (M)	14	0,53	19	0,56	22	0,50	23	0,47	25	0,48	28	0,50	1	0	1	0
Label (V)	9	0,40	12	0,37	20	0,50	21	0,47	25	0,52	28	0,53	29	0,46	29	0,41
Persistence	22	0,66	32	0,87	38	0,80	42	0,78	47	0,80	50	0,78	61	0,78	61	0,69
Error Handling			32	0,87	42	0,90	48	0,91	53	0,92	60	0,92	67	0,87	70	0,83
Sorting (V)					15	0,45	16	0,43	17	0,44	18	0,42	24	0,43	24	0,38
Favourites (V)							12	0,39	12	0,36	12	0,32	12	0,28	12	0,25
Copy (V)									9	0,28	10	0,28	16	0,31	16	0,27
SMS (V)											13	0,32	20	0,34	20	0,30
Music (O)													1	0	1	0
Media (M)													29	0,43	32	0,44
Video (O)															1	0
Capture (V)															1	0

Table 69. Size of the impact set for features and NFRs and their *Degree of Crosscutting*

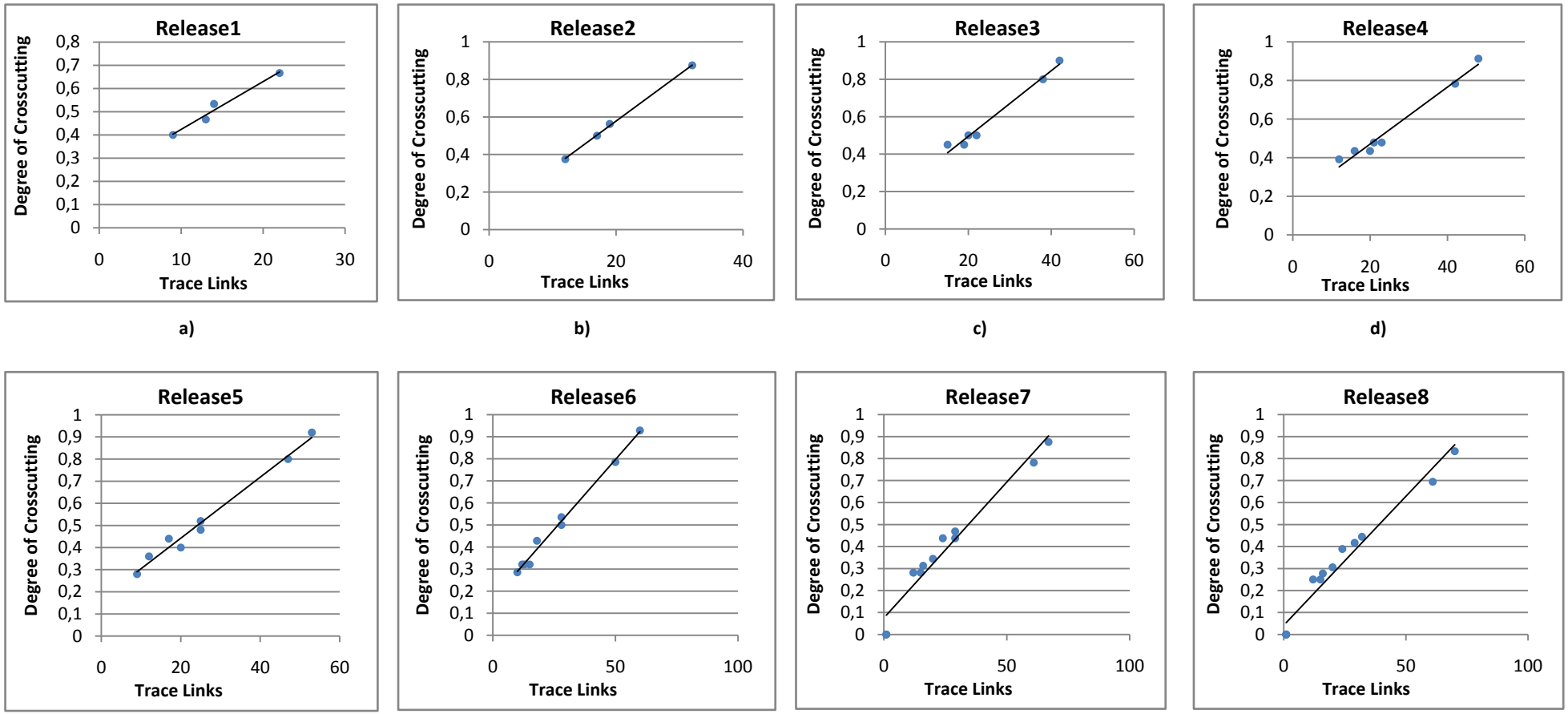


Figure 127. Correlations between impact set and Degree of Crosscutting for the 8 releases in MobileMedia

5.5.4. Discussion on changeability analysis

The main conclusions extracted from the changeability analysis are explained next:

- The first conclusion that may be extracted from the results shown in Figure 127 is that *Impact Set* of the features and *Degree of Crosscutting* are linearly correlated. That indicates that those features with higher *Degree of Crosscutting* have a bigger set of elements impacted than those with lower *Degree of Crosscutting*. This conclusion supports the need for having a framework for identifying crosscutting features in product lines. Refactoring these crosscutting features helps to isolate these features removing dependencies between them and improving, thus, the change management of the products (see Sections 5.3.3 and 0 to obtain more details on how to refactor crosscutting concerns). Moreover, the results obtained by this analysis were consistent with those shown in Section 5.5.1. In particular, the features with a higher *Degree of Crosscutting* are those implemented by more unstable use cases and also those with a bigger *Impact Set* of elements. As an example, Persistence and Error Handling have the higher *Impact Set* values and also are the concerns more unstable. Media or Photo features also present high values for *Impact Set* metric (in particular, Photo in the first six releases) and we can see in Table 68 that they are also addressed by more unstable use cases than other features.
- Secondly, we can observe that the fact of being mandatory or variable does not influence on the impact set of elements of a feature. As an example, it can be observed in Table 69 how, in release 7, Album (a mandatory feature) has a bigger impact set than Favourites (which belongs to variability). However, we can also observe how Label (variability) has a bigger impact set than Album. In that sense, as it has been showed in Section 5.3, there are several works which have introduced the benefits of using aspect-oriented techniques to deal with crosscutting features (e.g. [46][87][126][138][181]). However, these proposals only focus on the modelling of variable features. The conclusion observed above empirically demonstrates that mandatory features may be crosscutting features as well, supporting the need for performing an analysis to identify these crosscutting features. This conclusion is also consistent with the work presented in [121] where it is stated that common features could be also modelled using aspect-oriented techniques (e.g., aspectual components) if they crosscut other features. Analogously, variable features do not need to be always defined as crosscutting concerns. They may be effectively implemented in modular components if they do not crosscut other features.
- Finally, although the *Degree of Crosscutting* metric is perfectly correlated with the *Impact Set*, it has been also observed some cases where features with the same *Degree of Crosscutting* have different impact sets. This is due to the fact that *Degree of Crosscutting* metric is calculated based on the number of source and target elements involved in the crosscutting relation. Note that *Degree of Crosscutting* metric is based on *Crosscutpoints* and *NCrosscut*. Remember that *Crosscutpoints* measures the number of target elements where a source element crosscuts other source elements whilst *NCrosscut* counts the source elements crosscut. However, the *Impact Set* is quantified in terms of trace links instead of source or target elements. This is the reason why some features with the same *Degree of Crosscutting* may have different *Impact Set* values. As an example, in Figure 126 we can see how the *NCrosscut* and *Crosscutpoints* metrics for s1 are 2 and 1 respectively (s1 is crosscutting s3 source element in t3 and t4 target elements). However, the impact set of s1 includes 5 different trace links: those from s1 to t1, t3, t4 and those from s3 to t3 and t4. Observe that while *NCrosscut* metric counts the source element s3 once, the *Impact Set* counts the two links from s3 that must be preserved.

5.5.5. Feature dependency analysis

This section shows a third empirical analysis illustrating how the metrics presented in Section 5.4 may be used to infer information about the feature dependencies existing in a system, specially a product line. It has been already demonstrated that feature dependencies are harmful for software assets reusability [46][52][71]. However, to date there is no empirical studies demonstrating whether the presence of crosscutting features at requirements level increments dependencies between features, making thus, their reusability difficult. Moreover, since crosscutting has been compared with stability and change impact in previous sections, we may also compare these two maintainability attributes with feature dependencies showing their relation.

In [71], Figueiredo et al. introduce the concepts of interlacing and overlapping in order to assess feature dependencies in a product line. Interlacing denotes the situation where *the implementations of two features, F1 and F2, have one or more components (or operations) in common* [71]. Thus, the authors define a dependency as component-level interlacing when two features shares at least one component and as operation-level interlacing when the features share, at least, an operation. Based on these dependencies, they introduce the metrics *Component-level Interlacing Between Concerns*[8] (CIBC) and *Operation-level Interlacing Between Concerns (OIBC)*, which counts the number of components and operations shared by two different features, respectively. On the other hand, overlapping occurs when two different features share one or more statements, attributes, entire methods or entire components [71]. In that sense, unlike interlacing, a dependency is defined as component-level overlapping when the shared elements entirely contribute to the implementation of the two features, instead of being disjoint. Therefore, the authors define the metrics *Component-level, Operation-Level and Lines of Code-level Overlapping Between Concerns (COBC, OOBC, LOCOBC)* to measure overlapping between features at different granularity levels.

All the aforementioned metrics have been previously used to assess dependencies at the programming, detailed design or architectural levels [71][83][74][160]. However, to date, they have not been applied to measure dependencies at an earlier stage, i.e. requirement level. In this setting, this section shows the application of the metrics to the requirements of the MobileMedia system. In particular, we have calculated a new *Use Case-level Interlacing Between Concerns* (UCIBC) metric, which measures the number of use cases shared by the implementation of two features. Moreover, we have defined a new metric, called *Concern Interlacing* (CI), as the total number of feature dependencies of a particular feature. We have not used overlapping here since we did not detect use cases entirely shared by two features or concerns. All the relations between features and use cases identified were were disjoint so that only interlacing is considered.

Concern Interlacing metric is calculated by the addition of the UCIBC values for a feature with respect to the rest of features. As an example, if feature A shares 2 use cases with feature B and 4 uses cases with feature C, the CI metric for A is 6. Both metrics, UCIBC and CI may be calculated using our dependency matrix. In particular, the calculation of UCIBC metric is obtained by counting the number of columns of the dependency matrix where two particular features have a 1. In other words, this metric may be calculated for all the features by performing the product of the dependency matrix and its transpose. The matrix obtained by this product is called the Concern Interlacing Matrix. CI metric is calculated by counting the values obtained for each row of this matrix (discarding the values obtained in the diagonal). The formalisation of Concern Interlacing Matrix and CI metric is as follows:

[8] Note that the term of concern is used as a synonym of feature.

$$Concern\ Interlacing\ Matrix\ =\ \mathrm{DM}\ \cdot \mathrm{DM}^{\mathrm{T}} \qquad (11)$$

$$Concern\ Interlacing\ (s_k) = \left(\sum_{i=1}^{|T|} \mathrm{cim_{ki}}\right) - \mathrm{cim_{kk}} \qquad (12)$$

where DM and cim_{ki} represent the dependency matrix and the cell[k,i] of the concern interlacing matrix, respectively.

		Use cases														
		Add Album	Delete Album	Add Photo	Delete Photo	View Photo	View Album	Provide Label	Store Data	Remove Data	Retrieve Data	Edit Label	Count Photo	View Sorted Photos	Set Favourite	View Favourites
Concerns	Album	1	1				1									
	Photo			1	1	1										
	Label	1		1				1				1				
	Sorting					1							1	1		
	Favourites														1	1
	Persistence	1	1	1	1	1	1	1	1	1	1		1		1	
	Error Handling	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1

Table 70. Dependency matrix for the MobileMedia

		Features					NFRs		
		Album	Photo	Label	Sorting	Favourites	Persistence	Error Handling	CI
Concerns	Album	3		1			3	3	7
	Photo		3	1	1		3	3	8
	Label	1	1	4			3	4	9
	Sorting		1		3		2	3	6
	Favourites					2	1	2	3
	Persistence	3	3	3	2	1	12	12	24
	Error Handling	3	3	4	3	2	12	15	27

Table 71. Concern interlacing matrix for the MobileMedia

In order to illustrate the calculation of the UCIBC metric, Table 70 shows the dependency matrix obtained for release 3 of the MobileMedia system. By applying the product of this matrix and its transpose, the Concern Interlacing Matrix shown in Table 71 is obtained. In this matrix, a cell denotes the value of the UCIBC metric for the feature represented in the row with respect to the feature of the corresponding column. The values of the CI metric for each feature are also shown in the last column of this matrix. As an example, observe that the UCIBC metric for Photo and Label features is 1 since these features share the implementation of the *AddPhoto* use case. However, Photo shares the implementation of *AddPhoto, DeletePhoto* and *ViewPhoto* use cases with Persistence, thus the metric is 3 for these two features. Finally, the resulting CI metric for photo is 8 since the feature shares 8 different use cases with other features or concerns.

	Release0		Release1		Release2		Release3		Release4		Release5		Release6		Release7		Average	
Features	Concern Interlaicng	Degree of Crosscutting	Concern Interlaicng	Degree of Crosscutting	Concern Interlaicng	Degree of Crosscutting	Concern Interlaicng	Degree of Crosscutting	Concern Interlaicng	Degree of Crosscutting	Concern Interlaicng	Degree of Crosscutting	Concern Interlaicng	Degree of Crosscutting	Concern Interlaicng	Degree of Crosscutting	Concern Interlaicng	Degree of Crosscutting
Album (M)	9	0,46	13	0,50	15	0,45	16	0,43	16	0,40	12	0,32	12	0,28	12	0,25	13,12	0,38
Photo (M)	9	0,53	14	0,56	17	0,50	18	0,47	19	0,48	22	0,50	5	0	5	0	13,62	0,38
Label (V)	6	0,40	9	0,37	15	0,50	16	0,47	19	0,52	21	0,53	22	0,46	22	0,41	16,25	0,46
Persistence	12	0,66	22	0,87	27	0,80	30	0,78	34	0,80	36	0,78	45	0,78	45	0,69	31,37	0,77
Error Handling			22	0,87	29	0,90	33	0,91	37	0,92	42	0,92	48	0,87	50	0,83	37,28	0,89
Sorting (V)					11	0,45	12	0,43	13	0,44	14	0,42	19	0,43	19	0,38	14,66	0,42
Favourites (V)							9	0,39	9	0,36	9	0,32	9	0,28	9	0,25	9	0,32
Copy (V)									7	0,28	8	0,28	13	0,31	13	0,27	10,25	0,28
SMS (V)											10	0,32	16	0,34	16	0,30	14	0,32
Music (O)													0	0	0	0	0	0
Media (M)													23	0,43	25	0,44	24	0,44
Video (O)															0	0	0	0
Capture (V)															2	0	2	0

Table 72. *Concern Interlacing* and *Degree of Crosscutting* for the MobileMedia system

Using the dependency matrix for each release, we have calculated the CI metric for the 8 releases of the MobileMedia. The gathered data are shown in Table 72. For each release the degree of crosscutting for each feature has been also shown in order to have a way of comparing both metrics. The last column of the table shows the average of both metrics taking into account the 8 releases.

Based on the data presented in Table 72, *Concern Interlacing* and *Degree of Crosscutting* metrics have been compared in order to observe the correlation that exists between both metrics. The graphical results of the correlation for each release are shown in Figure 128 from a) to h).

Since the *Degree of Crosscutting* metric has been compared with the *Concern Interlacing* and *Change Impact Set* metrics, we also compare these two metrics. Intuitively, since *Degree of Crosscutting* is correlated with *Concern Interlacing* and *Change Impact*, these two last metrics should be also correlated. However, in order to check this conclusion, we have empirically compared the two metrics. The results are used to check whether the features with higher concern interlacing (more dependencies) are those with a bigger change impact. The graphics which show the correlation between *Change Impact* and *Concern Interlacing* metrics for each release are shown in Figure 129.

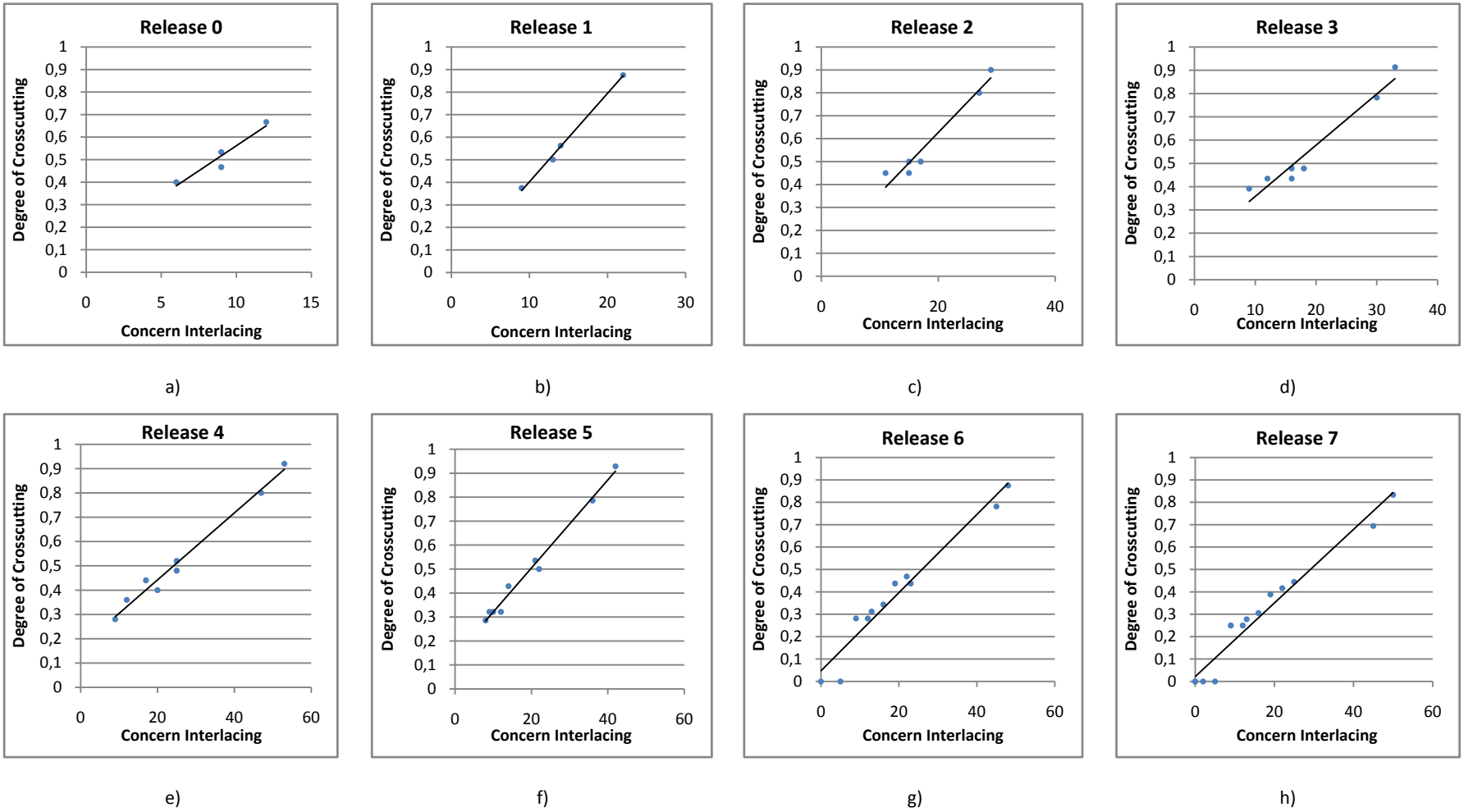


Figure 128. Correlation between *Concern Interlacing* and *Degree of Crosscutting* metrics

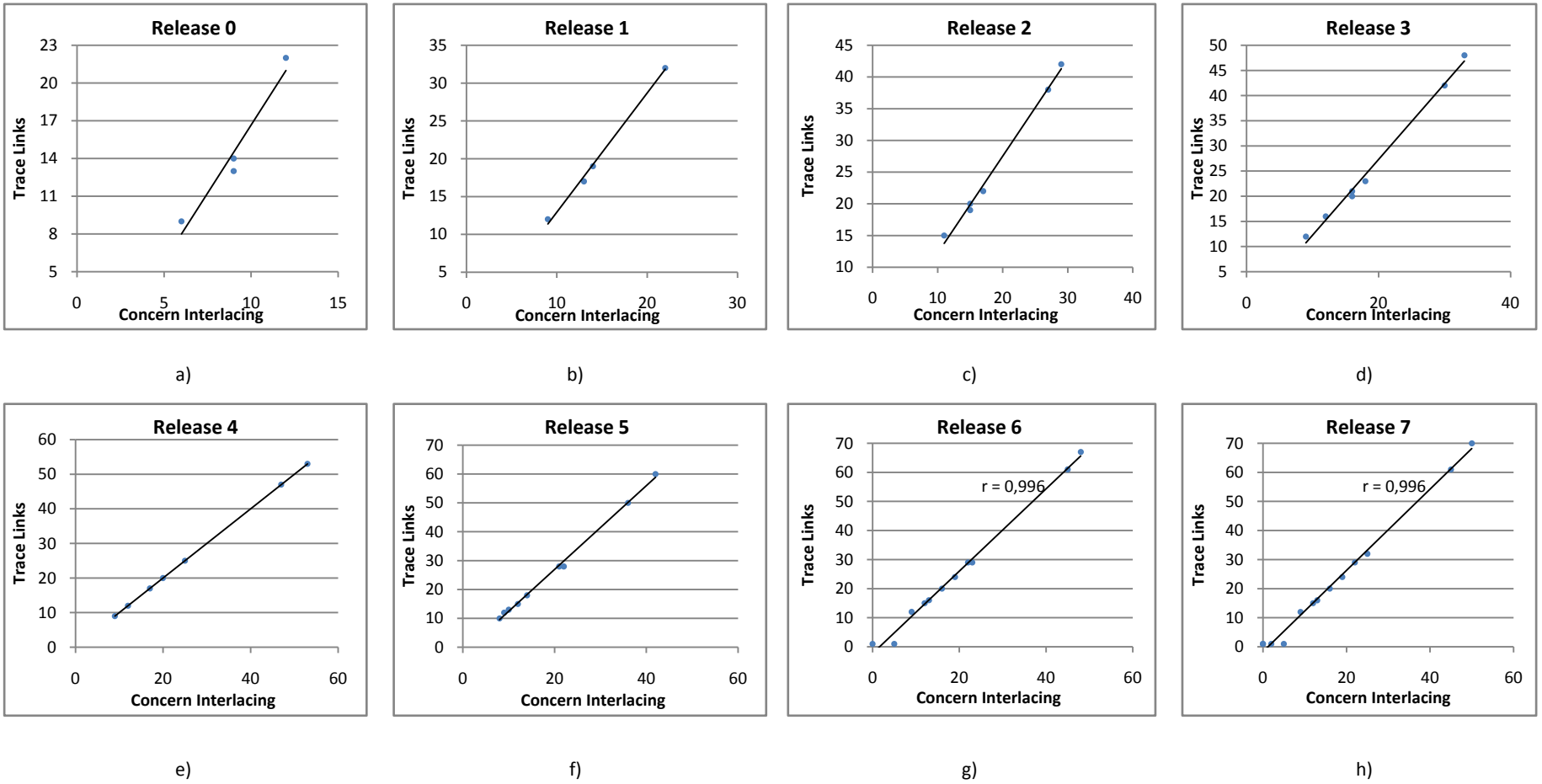


Figure 129. Correlation between *Concern Interlacing* and *Change Impact Set* metrics

Finally, once *Concern Interlacing* has been compared with *Degree of Crosscutting* and *Change Impact Set* metrics, we may also compare it with the stability of the different features. The results allow us to check whether the features with more dependencies are those implemented by more unstable use cases. Figure 130 shows the correlation existing between these two measurements. The data used to obtain this graphic are extracted from Table 72 (concern interlacing average) and Table 68 (unstable use cases).

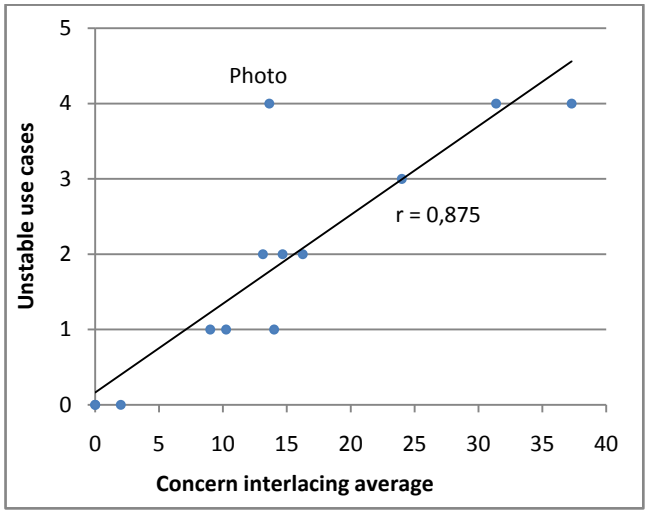


Figure 130. Correlation between concern interlacing and instability

5.5.6. Discussion on feature dependency analysis

The main conclusions extracted while observing the results obtained in previous section are:

- Firstly, observe that the results presented in Figure 128 shows that *Degree of Crosscutting* and *Concern Interlacing* metrics are linearly correlated so that the more dependencies between features, the higher the *Degree of Crosscutting* of these features. Of course, the inverse conclusion could be also stated, since a higher *Degree of Crosscutting* introduces feature dependencies, making, thus, reusability of the features difficult. This conclusion supports the need for introducing a modularity analysis so that the identification of crosscutting features at early development stages enhances the identification of feature dependencies.
- Secondly, the nature of the features, in terms of being mandatory or variable, has not been considered a determinant characteristic for feature dependencies (similarly to the conclusion mentioned for changeability). As it may be observed in Table 72, there are variable features with values for interlacing similar to the obtained for mandatory features. Even, there are variable features with lower concern interlacing results. Therefore, the conclusion extracted was that feature dependencies are highly determined by the crosscutting property of the features but not by their commonality or variability nature.
- Regarding to the *Concern Interlacing* metric, observe that a new metric to measure this concept for each feature has been introduced. This metric is based on those previously introduced in [71][74][83][160]. However, previous section has demonstrated how the calculation of this metric may be automated using our matrices, like with the rest of metrics presented here. This fact, together with the process to automatically discover source-to-target mappings (presented in Section 5.1), allows achieving a high degree of automation in the modularity analysis performed.

- Considering the relation between *Concern Interlacing* and *Change Impact Set* metric, the correlations of these two metrics with respect to *Degree of Crosscutting* one has been empirically demonstrated. Observe that the latter is correlated with the formers and, thus, *Concern Interlacing* and *Change Impact Set* metrics are also linearly correlated. Moreover, as it may be observed in Figure 129, the correlation between these two metrics is almost linearly perfect. Note that the value of *r* in the last two releases is 0,996. That means that the probability that these two metrics are not correlated is less than 0,05% (data obtained by using a critical values table for *r* [172]). Obviously, this conclusion was not surprising at all since the more dependencies between features, the more impact a change in the feature will have. However, as it has been mentioned in Section 5.5.4, this conclusion is opposite to the works which propose the utilization of aspect-oriented techniques to reduce dependencies only between variable and mandatory features [46][87][126][138][181]. The data obtained here, again, demonstrate that feature dependencies must be reduced as much as possible independently of the feature being variable or not. As it has been mentioned above, the number of dependencies and, thus, the change impact, are directly proportional to the degree of crosscutting.
- The results obtained in Figure 128 and Figure 129 also support the conclusions previously extracted in [71], where the authors analysed modularity in SPL at architectural level. Indeed, the results shown here provide insights of the importance of feature dependency analysis also at earlier development levels, i.e. requirements.
- According to the data presented in Figure 130, it may be also observed that concern interlacing and instability are correlated. In other words, the more feature dependencies, the more unstable a feature is (actually the software artefacts implementing the feature). This conclusion is even more important in product line domains, where the products are built by combining different features. Note that products may be built by adding variable functionality to core architecture (in an additive or positive variability approach, according to [89]). Then, the addition of new features implies too many changes in those core features with a higher interlacing (more dependencies). Since it has been demonstrated that the features with more dependencies are those with higher crosscutting, the need for identifying and assessing crosscutting properties is highlighted again.
- Observe, finally, that a point of the correlation shown in Figure 130 has been labelled as *"Photo"*. This point corresponds to the data obtained for the Photo feature. As it has been already mentioned in Section 5.5.2, this feature is the one more digressed from the correlation. After observing the results obtained, the reason here is the same explained in Section 5.5.2. In release 6, part of the functionality implemented in the use cases related to this feature is moved to the use cases related to the Media feature. This is why the degree of crosscutting for this release drastically decreases after release 5.

5.5.7. Conclusions

The empirical studies presented in this section have shown how crosscutting negatively affects to maintainability. In particular our metrics have been used to relate crosscutting with stability and changeability showing that they are directly proportional and follow a linear correlation. On one hand, the analysis of stability was driven by observing changes in target elements. This implies that the analysis was carried out after the changes have occurred (*"a posteriori"*). On the other hand, the analysis of changeability showed how the impact of a change could be anticipated before the change occurs (*"a priori"*). Both analyses complement each other and allow the developer to anticipate maintainability decisions. In fact the results obtained by the analyses were consistent showing that the features with a higher impact set were also those

implemented by more unstable use cases (and, of course, with a higher degree of crosscutting). Moreover, the analysis of changeability introduces a new metric used to empirically quantify the impact of a change in the form of traceability links affected. This is even more useful with the transition to model-driven software engineering [1] gaining momentum, since this traceability information may be used to build model transformations. A third analysis has also demonstrated that crosscutting introduces dependencies between features which are also harmful for stability and changeability. In order to assess these dependencies, a concern interlacing metric was used and adapted to the requirement level, providing a way to automate its calculation by simple matrix operations. The results obtained by this analysis were also consistent with the previous ones showing that concern interlacing is also correlated to instability and change impact.

The three analyses support the need for identifying crosscutting features independently of their nature (mandatory or variable). This conclusion is contrary to the works that only propose the modelling of variable features using aspect-oriented techniques. The results showed that mandatory crosscutting features also negatively affect to the quality of the system being also candidate to be modularized using aspect-oriented techniques. On the contrary, variable features may be also implemented in modular components and they do not need to be always defined as crosscutting concerns.

6

Conclusions and future work

In this chapter a summary of the main conclusions extracted from the work presented in this thesis is presented. The chapter is divided into two sections where the conclusions and some future research lines are described, respectively.

6.1. CONCLUSIONS

As it was mentioned in Chapter 1, nowadays the software market is very competitive and the increasing complexity of software systems challenges most of the well-know software techniques used. In this setting, new techniques, methodologies, methods and tools are needed to adapt the software development process to this competitive environment. In the last years, several new techniques have emerged in order to tackle desing and development of complex systems, reducing not only the time-to-market but also maintainability costs and efforts. Examples of these technologies are Component-based Software Engineering (CBSE) or, more recently, Model-Driven Architecture (MDA), Software Product Line Engineering (SPLE) or Aspect-Oriented Software Development (AOSD). However, sometimes these new technologies still have limitations that should be solved in order to gain a wider acceptance by the community. Examples of these limitations in AOSD are the lack of formal definitions or the existence of just few works (mostly focused at the programming level) empirically demonstrating the benefits obtained by aspect-orientation. This thesis has faced up these limitations, among others, and the main conclusions extracted from the work developed are described in this section.

In order to conclude the work presented in this thesis, the goals proposed in Section 1.2 (in the Introduction) are shown here again. These goals are used to drive this section since the main conclusions extracted are mainly related to each goal proposed.

Goal 1) Provide a formal definition of crosscutting. This definition should be independent of any abstraction level or deployment artefact. A general definition ensures that it may be applied at different application domains. This general definition is the base for other application areas, such as crosscutting identification or the definition of crosscutting metrics.

In this thesis, a formal definition for aspect-oriented terminology has been presented. In particular, the terms of scattering, tangling and crosscutting have been defined. In order to define these concepts, a conceptual framework was presented. This conceptual framework is based on the concept of the crosscutting pattern. The crosscutting pattern denotes the situation where two different domains, source and target are related by means of mappings or traceability relations. The terms of scattering, tangling and crosscutting have been defined based on the cardinality of the mapping. The utilization of formal definitions is mandatory for certain research areas such as the formal identification of crosscutting concerns in software systems. It was shown that there are other similar definitions of crosscutting in the literature. However, most of these definitions are focused on the programming level and it was demonstrated that the definition presented here generalises most of the other definitions described. In particular, our definition generalises the definition introduced by Mashuara and Kiczales and the one proposed by Mezini and Ostermman since both definitions are more restrictive than ours and they do not identify cases of crosscutting identied by our definition (e.g. some tracing situations). These two definitions consider crosscutting as a symmetric property whilst our definition does not. Our definition was also compared to the definition presented by Ceccato and Tonella demonstrating that they are equivalent since identify the same crosscutting situations. Both definitions consider crosscutting as an asymmetric

property. Nevertheless, our definition generalises the definition by Tonella and Ceccato since it may be applied to any development level (not only programming).

The definition of crosscutting presented has been formalised and represented in terms of linear algebra. A particular extension of traceability matrices was used to represent the mappings between source and target elements. This matrix, called the dependency matrix, easily allows to visualize scattering and tangling in the system. From this matrix, and using simple matrix operations, we derive two matrices called: crosscutting product matrix and crosscutting matrix which identify crosscutting situations using any source and target domains. This can be applied to any phase or abstraction level in a software development process, including early phases. This application demonstrated the utility of the crosscutting pattern for the aspect mining area. In fact, the matrix operations were used to define an early aspect mining process, presented as one of the application areas of the conceptual framework.

In addition, the use of traceability matrices aims at improving traceability information between different development phases. This traceability allows the analysis of crosscutting across several refinement levels in software development, for example from concern modelling to requirements and architectural desing or from architectural design to detailed design and implementation. This analysis has been formalised through what we called the cascading of the crosscutting pattern. By this operation a special kind of transitivity relation is established between three different domains: source, intermediate and target. Then, using this transitivity relation, elements of source domain are related to element of the final target domain. Of course, the number of intermediate levels could be extended in order to apply the cascading operation to N levels. The cascading operation was also formalised in terms of linear algebra and operationalized by means of simple matrix operations.

Goal 2) Definition of an aspect mining process to identify crosscutting concerns at early stages of development. The process should consider different requirements artefacts in order to be applied in legacy systems. However, it should not be tied to a particular level so that the identification of crosscutting concerns in different domains or abstraction levels is possible. The process should be based on the formal definition of crosscutting described by Goal 1.

As it has been also mentioned in Chapter 1, aspect mining is still one of the main challenges in aspect-orientation. AOSD is meaningless unless crosscutting concerns are properly identified in software systems. Moreover, the identification of these crosscutting concerns at early stages of development allows the incorporation of the benefits of AOSD in the very beginning of the development. In this setting, this thesis has presented an aspect mining process to identify crosscutting situations in software systems. This aspect mining process is based on the definition of crosscutting also presented in this work. The process may be used to identify well-known crosscutting concerns but also domain-specific crosscutting concerns that have not been previously identified in the literature. Since the aspect mining process is based on the crosscutting pattern, it is independent of any abstraction level being its application possible to any software development phase. Note that it is not tied to any specific deployment artefact. However, since we were interested in applying aspect-oriented techniques at early stages of development, this thesis presented an instantiation of the process at the requirements level using use case models.

The process presented extends the crosscutting pattern by the addition of several analyses that allows the automatic generation of the traceability matrices. These analyses are a syntactical analysis based on identifiers matching and a dependency-based anlaysis which inspects intra-level relations in requirement models. In order to perform these analyses, XML was selected as the standard language to represent source and target domains. In particular,

an XML file is used to model concerns in the system (source) whilst requirements, in case of use case models, are modelled using XMI files (target). The utilization of XMI ensures that the process may be used at other abstraction levels such as detailed design using UML class diagrams. The syntactical analysis also uses a non-functional concerns catalogue where common words related to these concerns are shown. Both analyses have tool support.

The process has been also validated by applying it to different systems. The results obtained by our aspect mining process have been compared with those obtained by similar tools used with the same purpose: the identification of crosscutting concerns at early stages of development. In particular, the approaches used to compare were EA-Miner and Theme/DOC. The results showed that manual approaches behave slightly better than automatic approaches, however, the application of manual approaches in large systems is unfeasible so that automatic approaches should be considered. The automatic approaches obtained similar results, however in some cases, the user involvement in the process was really high in order to filter the first results obtained (like in EA-Miner). The results also showed that all the approches identified false positives (crosscutting concerns wrongly identified). In some cases, the concerns identified by them were different being an evidence, sometimes, of the presence of false negatives (actual crosscutting concerns not identified by an approach). The results also show that the accuracy of the requirements may produce these false positives and negatives. This is why manual approaches provide, in general, better results. By manually exploring the requirements of a system, the developer may infer information explicitly not mentioned in the requirements. However, as it has been aforementioned, the utilization of manual approaches is not feasible in large and complex systems. Finally, the comparison showed that the results obtained by the approaches may be improved when catalogues are used. The utilization of these catalogues improves the identification of well-know crosscutting concerns, such as non-functional concerns. However, in order to identify domain-specific crosscutting concerns, other techniques must be used (like the used in our aspect mining process).

Although the aspect mining process has been instantiated to be used at the requirements levels, a key contribution of the approach is that it may be applied across several refinement levels allowing traceability analysis. In particular, the application of the cascading operation for the architectural level has been also illustrated.

Goal 3) Provide a process to identify crosscutting features at early stages of product line developments. This process should identify crosscutting features independently of their nature, mandatory or variable. The process should be based on the formal definition of crosscutting described by Goal 1.

It has been claimed by aspect-oriented community that crosscutting is harmful to software modularity. In the SPL domain, this has been also observed. Features in SPL are negatively affected by crosscutting dependencies, making these features dependant and reducing thus the maintainability, reusability and adaptability of the SPL assets. In this thesis, the aspect mining process presented has been also adapted to be used at the SPL domain aiming at identifying crosscutting features. The identification of these features allows to refactor their implementations into separated entities improving the reusability and flexibility of product line assets. The application of the process was illustrated by identifying crosscutting features in several product lines. The results showed that no matter whether the features are variable or mandatory to being considered as crosscutting.

The concern-metrics presented were also used to extend the aspect mining process in this analysis and the validations applied to the metrics were carried out using the same product line system. The results obtained by applying the metrics to the product line confirmed our previous statement about the need for identifying crosscutting features independently of their

nature (mandatory or variable). Even, the changeability analysis was more important in the SPL domain where products are built by composing features. The capability for anticipating a change in a feature helps to easily measure the cost of building a new product in the product line.

Goal 4) Provide a measurements framework driven by generic metrics for early quantification of crosscutting. Although the framework must be applied at early stages of development, it should be generic enough to be applied at different abstraction levels. The metrics should be validated both theoretically and empirically. The internal validation must show that the metrics are measuring what it is supposed. The external validation must check out whether the metrics are useful for inferring information about other internal or external quality attributes (e.g. maintainability characteristics such as instability or changeability [96]).

It has been already mentioned that crosscutting leads to software quality defects. However, as it was commented in the introduction, there has not been too much effort dedicated to empirically demonstrate that crosscutting negatively affects to software quality. Most of the analyses performed were focused on the programming level and there is a lack of crosscutting analyses at the requirements level. In this setting, this thesis has shown how to use the crosscutting pattern for the assessment of software modularity in order to measure the degree of crosscutting. The traceability matrices used by the crosscutting pattern aims at establishing a set of concern metrics to drive this assessment. Again, since these metrics are based on the conceptual framework, they are not tied to any specific deployment artefact and they are language-agnostic. Thus, the metrics may be used at any abstraction level. In this thesis, they have been instantiated to be used at the requirements level as a complement to the aspect mining process presented. Then, an early analsysis of crosscutting has been performed showing that it leads to software instabilities and it difficults software changeability.

The set of concern metrics includes metrics for scattering and tangling and also specific metrics for assessing the degree of crosscutting. They have been validated by using a double validation: a theoretical and an empirical validation. The theoretical validation has demonstrated the accuracy of the concern metrics to measure crosscutting properties. In other words, this validation showed that the metrics measure what it was expected. This validation was done by comparing the results obtained by the metrics with those obtained by similar metrics introduced by other authors. Unlike our metrics, most of the metrics compared were defined at the programming level or they were tied to an specific abstraction level. The first conclusion extracted was that our metrics were generic enough to embrace these existing code-level metrics. The comparison also showed that, in general, the results obtained by our metrics were consistent with the obtained by the other metrics. There were also some equivalent metrics; they obtained the same results since they measure the same concepts. One important conclusion extracted from the analysis was the need for using a metric specific for crosscutting (exclusive of our concern-metrics). It was empirically demonstrated how this metric provides information, sometimes, not considered by other metrics related to scattering or tangling.

The empirical validation of the metrics also showed their utility in terms of their relation with other software quality attributes, such as maintainability. In the analysis, it was empirically demonstrated that the degree of scattering and crosscutting are related to two ISO/IEC 9126 software maintainability attributes, namely stability and changeability. In particular, the results showed that the crosscutting is correlated to stability and changeability so that the higher the degree of crosscutting, the more unstable and less changeable a software system is. This analysis was conducted by two different experiments. In a first

experiment, the stability of a software system was measured by observing the changes occurred in the requirements of the system (once the changes have occurred, *"a posteriori"*). Then, these changes were measured and compared with the degree of scattering and degree of crosscutting metrics. The results showed that both metrics were correlated to instability with a high correlation coeficient. An interesting result obtained was the higher correlation between crosscutting and stability than scattering and stability, which supports the need for having a metric specific for crosscutting. In the second experiment, the impact of a change was measured by means of a new metric introduced. This metric was also calculated in terms of the values of the traceability matrices. Unlike in the previous experiment, this metric was used to anticipate the impact of a change in the system without the need of performing this change (the impact was measured *"a priori"*). The change impact of the source elements were, thus, compared with their degree of crosscutting. The results showed that the source elements with a higher degree of crosscutting were also the elements with a higher change impact. Both experiments demonstrated the ability of the metrics for inferring and anticipating information related to other software quality attributes different from modularity.

A third empirical analysis also showed evidences of the correlation between feature dependencies and the degree of crosscutting. In particular, this correlation illustrated how the higher the degree of crosscutting for a feature is, the more dependencies with other features it has. In order to establish this correlation, a *Concern Interlacing* metric was used and adapted to the requirement level. This metric may be automatically calculated by using our dependency matrix. Finally, the analysis showed that this metric is also correlated with instability and changeability. In that sense, the analysis empirically demonstrated that feature depencies are harmful for the two aforementioned ISO/IEC 9126 quality attributes.

6.2. FUTURE WORK

A PhD is a complex and large process that usually takes several years of research and work. It has been even defined by Easterbrook in [66] as *"probably the largest (most self-indulgent) piece of work you'll ever do"* (from an academic point of view). However, I consider it as a first research work, an important beginning for a research career of many years. In that sense, while doing the work presented in this thesis, other research tasks and topics that I would like to face up have been identified. Some of these future works are extensions of the work presented in this thesis where the applicability of the crosscutting pattern may be demonstrated. An example of these application areas is the utilization of the dependency matrix to build traceability tools. Even, this traceability topic is more interesting taking into account the peak achieved by the Model-Driven Development. In that sense, the analysis of crosscutting concerns in different models (PIM or PSM) would allow the developer to build transformations considering the crosscutting concerns (e.g. generating weaved code, or using aspect-oriented languages). On the other hand, other future activities are focused on the exploration of new research lines where the lessons learnt by these years of work could be applied.

The future research activities that we are planning to do are the next:

- Extend the syntactical analysis to improve the results obtained by adding to the tool support for considering semantic matchings. The utilization of the WordNet database [184] is considered to achieve this semantical information.
- Apply different techniques to automatically discover concerns in requirements descriptions. As it has been previously mentioned, there are some techniques that tackle the semi-automatic identification of these concerns by applying different

heuristics that usually use semantic information to localize them. We do not rule out to analyse this research line in the near future.

- Explore other application areas of the crosscutting pattern. As an example, a first exploratory study on the use of the crosscutting pattern to identify volatile requirements has been showed in [49] and [53]. We would like to keep on working on this topic and to analyse the automation of the identification of these volatile requirements in order to improve software evolution and adaptability.
- Perform new empirical analysis in different domains or contexts. As an example, we plan to compare the results obtained by our metrics at the requirements level with those obtained at different abstraction levels, e.g. architectural design, detailed design or implementation. The application of the metrics at these other levels of abstractions also allows the comparison of the results with the obtained by similar empirical studies performed by other authors at these levels (mainly at the programming level). These experiments may allow to extract conclusions about crosscutting properties and other quality attributes (similar to what we did with stability and changeability). They may be also used to check whether the early aspects identified at the requirements level really lead to crosscutting concerns at the programming level or, sometimes, they are false positives and are removed at the programming level. This analysis could be also completed by using aspect-oriented verions of the systems analysed.
- Incorporate the crosscutting pattern and the aspect mining process into a Model Driven Development process. Note that model transformations provide the links between source and target domains. These links could be also used to obtain the dependency matrix and to apply the crosscutting analysis. Then, the results obtained by the process could be used to refactor the model transformations in order to consider crosscutting concerns (e.g. using an aspect-oriented modelling approach as target of the model transformation). In fact, a first analysis of crosscutting (using our crosscutting pattern) in model transformations have been already done in [25].
- Analyse the impact of crosscutting concerns in all the stages of the product line developments. In that sense, the empirical analyses performed at the different abstraction levels may help to this task. However, my exploration in this area is not limited to the study of the addition of aspect-oriented techniques to it. In the last months, my attention has been strongly attracted by the Software Product Lines area. I consider SPLE as a really interesting topic and I would like to spend my future research time in this area. I would like to deeply explore the current SPL development techniques and also the automation of the development processes. In that sense, with the transition to model driven software engineering gaining momentum, the application of MDE techniques to develop product families could be a good way to automatize the software process. In addition, the improvement achieved in this area could be even more important for the application domain where I started to work in my first steps as a researcher: the Ambient Intelligence (AI) area (more precisely in Home Automation systems [47]). In the future I would like to combine these two research areas (SPLE and Home Automation) by applying the research results obtained in the SPLE topic to the Home Automation domain.

7

Publications

This chapter details the main publications, related to the topics presented in this thesis, that have been obtained in this period of time:

- Conejero, J., Hernández, J. and Rodríguez, R. UML Profile Definition for Dealing with the Notification Aspect in Distributed Environments. In proceedings of **6th International Workshop on Aspect-Oriented Modelling at 4th International Conference on Aspect-Oriented Software Development (AOSD)**, Chicago, USA, 2005
- Berg, K. van den, Conejero, J. and Chitchyan, R. (2005). AOSD Ontology 1.0 - Public Ontology of Aspect-Orientation. **AOSD Europe Network of Excellence**, retrieved May 2005, from http://www.aosd-europe.net/documents/d9Ont.pdf
- Berg, K. van den and Conejero, J. (2005). Disentangling Crosscutting in AOSD: A Conceptual Framework. In proceedings of **European Interactive Workshop on Aspects in Software**, Brussels, Belgium
- Berg, K. van den and Conejero, J. (2005). A Conceptual Formalisation of Crosscutting in AOSD. In proceedings of Aspect-Oriented Software Development workshop at **10th Conference on Software Engineering and Databases (JISBD)**, Granada, Spain
- Berg, K. van den, Conejero, J. and Hernández, J. (2006). Identification of crosscutting in software design. In proceedings of **7th International Workshop on Aspect Oriented Modelling Workshop at 5th International Conference on Aspect-Oriented Software Development (AOSD)**, Bonn, Germany
- Berg, K. van den, Conejero, J. and Hernández, J. (2006). Analysis of Crosscutting across Software Development phases based on Traceability. In proceedings of **International Early Aspects Workshop at 28th International Conference on Software Engineering (ICSE)**, Shanghai, China
- Berg, K. van den, Conejero, J. and Hernández, J. (2007) Analysis of Crosscutting in Early Software Development Phases based on Traceability. **In Journal LNCS Transactions on Aspect-Oriented Software Development III**, LNCS 4620, pp. 73–104, Springer-Verlag.
- Conejero, J., Hernández, 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 TR3/07, University of Extremadura**, March 2007. http://quercusseg.unex.es/chemacm/research/TR3_07.pdf
- Conejero, J., Hernández, J., Moreira, A. and Araujo, J. (2007). Discovering volatile and aspectual requirements using a crosscutting pattern. In proceedings of the **15th IEEE International Requirements Engineering Conference**, Posters, October 2007
- Conejero, J., Hernández, J and Berg, K. van den (2007). Disentangling Crosscutting in AOSD: Formalisation based on a Crosscutting Pattern. In proceedings of **11th Conference on Software Engineering and Databases (JISBD)**, ISBN: 84-95999-99-4, pp.325-334, Sitges, Spain
- Conejero, J., Hernández, J and Berg, K. van den (2007). Disentangling Crosscutting in AOSD: Formalisation based on a Crosscutting Pattern. In **Journal IEEE America Latina**, vol. 5, ISSN: 1548-0992
- Conejero, J. and Hernández, J. (2008). Analysis of Crosscutting Features in Software Product Lines. In proceedings of **International Workshop on Early Aspects at 30th International Conference on Software Engineering (ICSE).** Leipzig, Germany
- Conejero, J., Hernández, J., Jurado, E. and Berg, K. van den (2008). Analysis of modularity by an aspect-oriented measurement process. In proceedings of **13th Conference on Software Engineering and Databases (JISBD)**, Gijón, Spain

- Conejero, J., Figueiredo, E., Garcia, A., Hernández, J. and Jurado, E. (2009). Early Crosscutting Metrics as Predictors of Software Instability. In proceedings of **TOOLS-EUROPE 2009, 47th International Conference Objects, Models, Components, Patterns**. Zurich, Switzerland
- Conejero, J., Hernandez, J., Jurado, E., Clemente P.J. and Rodríguez, R. (2009). Early Analysis of Modularity in Software Product Lines. In proceedings of the **21st International Conference on Software Engineering and Knowledge Engineering (SEKE)**. Boston, USA
- Conejero, J., Hernández, J., Moreira, A. and Araujo, J. (2009). Adapting Software by Identifying Volatile and Aspectual Requirements. In proceedings of the **14th Software Engineering and Database Conference (JISBD)**, San Sebastián, September 2009
- Conejero, J., Hernández, J., Jurado, E. and Berg, K. van den (2010). Mining Early Aspects based on Syntactical and Dependency Analyses. **Science of Computer Programming Journal**, **Elsevier**. JCR position: 34/86. **5-Year Impact Factor: 1.379** (to be published).

8

References

[1] Aférez, M., Kulesza, U., Sousa, A., Santos, J., Moreira, A., Araújo, J. and Amaral, A., A Model-Driven Approach for Software Product Lines Requirements Engineering. *20th International Conference on Software Engineering and Knowledge Engineering (SEKE)*, San Francisco Bay, USA (2006).

[2] Bergmans, L. and Aksit, M. Composing crosscutting concerns using composition filters. *Communications of the ACM 44*, 10 (2001).

[3] Aldekoa, G., Trujillo, S., Sagardui, G., and Diaz, O. Quantifying Maintainability in Feature Oriented Product Lines. *12th European Conference on Software Maintenance and Reengineering*, IEEE (2008), 243-247

[4] AMPLE (Aspect-Oriented, Model Driven Product Line Engineering) Project. URL: http://ample.holos.pt/

[5] AOSD Europe Network of Excellence, URL: http://www.aosd-europe.net/

[6] Apel, S., Leich, T., and Saake, G. Aspectual mixin layers: aspects and features in concert. *28th International Conference on Software Engineering (ICSE)*, Shanghai, China (2006), 122-131.

[7] Apel, S., Leich, T., and Saake, G. Aspectual Feature Modules. *IEEE Trans. Software Eng. 34*, (2008), 162-180.

[8] Araujo, J., Moreira, A., Brito, I., Rashid, A.: Aspect-Oriented Requirements with UML. *Workshop on Aspect-Oriented Modelling with UML at 5th International Conference on Unified Modelling Language.* Dresden, Germany (2002)

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

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

[11] Baker, B.S. On finding duplication and near-duplication in large software systems. *Second Working Conference on Reverse Engineering (WCRE),* Toronto, Canada (1995) 86-95

[12] Baldwin, C.Y., Clark, K.B.: *Design Rules, The Power of Modularity, vol. 1*. MIT Press, Cambridge (2000).

[13] Balmas, F. Displaying dependence graphs: a hierarchical approach. *Journal of Software Maintenance and Evolution: Research and Practice 16, 3* (2004) 151-185

[14] Baniassad, E. and Clarke, S. Theme: An Approach for Aspect-Oriented Analysis and Design. *26th International Conference on Software Engineering (ICSE)*, Edinburgh, Scotland (2004) 158-167

[15] Baniassad, E., Clarke, S.: *Aspect-Oriented Analysis and Design: The Theme Approach*. Addison-Wesley, Reading (2005)

[16] Baniassad, E., Clements, P., Araújo, P., Moreira, A., Rashid, A., Tekinerdogan, B.: Discovering early aspects. *IEEE Software 23(1)*, (2006) 61–70

[17] Bass, L., Klein, M. and Northrop, L., Identifying aspects using architectural reasoning. *Early Aspects: Aspect-Oriented Requirements Engineering and Architecture Design Workshop; at 3rd Aspect-Oriented Software Development Conference (AOSD)*, Lancaster, UK (2004)

[18] Batory, D., Sarvela, J., and Rauschmayer, A. Scaling step-wise refinement. *IEEE Transactions on Software Engineering 30, 6* (2004), 355-371.

[19] Baxter, I.D., Yahin, A., Moura, L., Anna, M.S., and Bier, L. Clone Detection Using Abstract Syntax Trees. *International Conference on Software Maintenance* (ICSM), Maryland, USA (1998), 368-377

[20] Berg, K. van den and Conejero, J.. Disentangling Crosscutting in AOSD: A Conceptual Framework. *European Interactive Workshop on Aspects in Software*, Brussels, Belgium (2005)

[21] Berg, K. van den, Conejero, J. and Chitchyan, R. *AOSD Ontology 1.0 - Public Ontology of Aspect-Orientation*. AOSD Europe Network of Excellence (2005), URL: http://www.aosd-europe.net/documents/d9Ont.pdf

[22] Berg, K. van den. Change Impact Analysis of Crosscutting, in Software Architectural Design. *Workshop on Architecture-Centric Evolution at 20th European Conference on Object-Oriented Programming (ECOOP)*, Nantes, France (2006)

[23] Berg, K. van den, Conejero, J. and Hernández, J. Identification of crosscutting in software design. *Aspect Oriented Modelling Workshop at 5th Aspect-Oriented Sofftware Development Conference (AOSD)*, Bonn, Germany (2006)

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

[25] Berg, K. van den, Tekinerdogan, B. and Nguyen H. Analysis of Crosscutting in Model Transformations. *Traceability Workshop at Second European Confernce on Model Driven Architecture (ECMDA)*, SINTEF Report A219 (2006), 51-64

[26] Bergmans, L. and Aksit, M. Composing Crosscutting Concerns Using Composition Filters, *Communications of the ACM, Vol. 44, No. 10* (2001), 51-57

[27] Bertolino, A. and Gnesi, S. Use case-based testing of product lines. *ACM SIGSOFT Software Engineering Notes 28, 5* (2003)

[28] Bonifácio, R. and Borba, P. Modelling scenario variability as crosscutting mechanisms. *8th Aspect-Oriented Software Development Conference (AOSD)*, Charlottesville, USA (2009), 125-136

[29] Breu, S. Extending Dynamic Aspect Mining with Static Information. *IEEE International Workshop on Source Code Analysis and Manipulation (SCAM)*, Budapest, Hungary (2005) pp. 57-65.

[30] Breu, S. and Krinke, J. Aspect Mining Using Event Traces. *19th International Conference on Automated Software Engineering (ASE)*, IEEE Computer Society, Linz, Austria (2004), 310-315

[31] Briand, L., El Eman, K., Morasca, S. T*heoretical and Empirical Validation of Software Product Measures.* Technical Report ISERN-95-03, Fraunhofer Inst. for Experimental Software Eng., Germany (1995)

[32] Bruntink, M., van Deursen, A., Engelen, R. v., and Tourwé, T. On the Use of Clone Detection for Identifying Crosscutting Concern Code, *IEEE Transactions of Software Engineering, 31(10),* (2005), 804-818.

[33] Budanitsky, A. and Hirst, G. Semantic distance in WordNet: An experimental, application-oriented evaluation of five measures. *Workshop on WordNet and Other Lexical Resources, Second meeting of the North American Chapter of the Association for Computational Linguistics*, Pittsburgh, USA (2001)

[34] Bushmann, F., Meunier, R., Rohnert, H., Sommerlad, P., and Stal, M. *Pattern-Oriented Software Architecture: A System of Patterns*. Wiley, West Sussex, England (1996).

[35] Ceccato, M. and Tonella, P. Measuring the Effects of Software Aspectization. *First Workshop on Aspect Reverse Engineering*, Delft University of Technology, the Netherlands (2004)

[36] Ceccato, M., Marin, M., Mens, K. Moonen, L., Tonella, P. and Tourwé, T. Applying and Combining Three Different Aspect Mining Techniques. *Software Quality Control 14, 3* (2006), 209-231.

[37] Centrum, T. and Tourwé, T. (2004). Mining Aspectual Views using Formal Concept Analysis. *4th IEEE International Workshop on Source Code Analysis and Manipulation (SCAM)*, IEEE Computer Society (2004), 97-106.

[38] Chidamber, S.R. and Kemerer, C.F. A Metrics Suite for Object Oriented Design. *IEEE Transactions on Software Engineering 20, 6* (1994).

[39] Chitchyan, R., Rashid, A., Sawyer, P., et al. *Survey of Analysis and Design Approaches.* AOSD Europe Network of Excellence (2005). URL: http://www.aosd-europe.net/deliverables/d11.pdf

[40] Chitchyan, R., Rashid, A., Rayson, P., and Waters, R. Semantics-based composition for aspect-oriented requirements engineering. *6th Aspect-Oriented Software Development Conference (AOSD); vol. 208*, Vancouver, Canada (2007), 36-48

[41] Cho, H., Lee, K., and Kang, K.C. Feature Relation and Dependency Management: An Aspect-Oriented Approach. *13th International Conference on Software Product Line (SPL)*, Limerick, Ireland (2008), 3-11.

[42] Cleland-Huang, J. Settimi, R., Romanova, E., Berenbach, B. and Clark, S. Best Practices for Automated Traceability, *Computer Journal, vol. 40* (2007), 27-35

[43] Cleland-Huang, J., Settimi, R., Zou, X. and Solc, P. Automated Classification of Non Functional Requirements, *Requirements Engineering Jorunal, vol. 12* (2007), 103-120

[44] Clemente, P., Hernández, J., Herrero, J., Murillo, J. and Sánchez, F. Aspect-Orientation in the Software Life Cycle: Fact and Fiction. *Aspect Oriented Software Development. In Mehmet Aksit, Siobhan Clarke, Tzilla Elrad and Robert Filman (eds), Chapter 18*, Addison-Wesley (2005), 407-424,

[45] Clements, P. and Northrop, L. *Software Product Lines: Practices and Patterns.* MA: Addison-Wesley, Boston (2002)

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

[47] Conejero, J., Pedrero, J., and Hernández, J. Una plataforma JAVA para el control y monitorizacion de instalaciones domoticas EIB. *1st JavaHispano Conference*, Madrid, Spain, (2003)

[48] Conejero, J., Hernández, J. and Rodríguez, R. UML Profile Definition for Dealing with the Notification Aspect in Distributed Environments. *6th International Workshop on Aspect-Oriented Modelling at 4th International Conference on Aspect-Oriented Software Development (AOSD)*, Chicago, USA (2005)

[49] Conejero, J., Hernández, J., Moreira, A. and Araujo, J. Discovering volatile and aspectual requirements using a crosscutting pattern. *15th. IEEE Internacional Requirements Engineering Conference (RE), Posters session*, Nueva Delhi, India (2007)

[50] Conejero, J., Hernández, J., Jurado, E. and Berg, K. van den. Analysis of modularity by an aspect-oriented measurement process. *13th Conference on Software Engineering and Databases (JISBD)*, Gijón, Spain (2008)

[51] Conejero, J., Figueiredo, E., Garcia, A., Hernández, J. and Jurado, E. Early Crosscutting Metrics as Predictors of Software Instability. *TOOLS-EUROPE 2009, 47th International Conference Objects, Models, Components, Patterns*. Zurich, Switzerland (2009)

[52] Conejero, J., Hernandez, J., Jurado, E., Clemente P.J. and Rodríguez, R. Early Analysis of Modularity in Software Product Lines. *21st International Conference on Software Engineering and Knowledge Engineering (SEKE)*. Boston, USA (2009)

[53] Conejero, J., Hernández, J., Moreira, A. and Araujo, J. Adapting Software by Identifying Volatile and Aspectual Requirements. *14th. Software engineering and Database Conference* (JISBD), San Sebastián, Spain (2009)

[54] Conejero, J., Hernández, J., Jurado, E. and Berg, K. van den. Mining Early Aspects based on Syntactical and Dependency Analyses. *Submitted to the Science of Computer Programming Journal, Elsevier. JCR position: 34/86. 5-Year Impact Factor: 1.312*. Current status: re-submitted with major revisions (2009)

[55] Czarnecki, K. and Eisenecker, U. (2000). *Generative Programming: Methods, Tools, and Applications*. Addison-Wesley, Reading, MA

[56] Davis, A. *Software Requirements: Objects, Functions and States*, *2nd edn*. Prentice-Hall, Englewood Cliffs (1993)

[57] Dijkstra, E. *A Discipline of Programming*. Prentice Hall (1976)

[58] Ducasse, S., Girba, T., Kuhn, A. Distribution Map. *22th International Conference on Software Maintenance (ICSM)*, Philadelphia, USA (2006)

[59] Ducasse, S., Rieger, M., and Demeyer, S. A Language Independent Approach for Detecting Duplicated Code. *15th International Conference on Software Maintenance (ICSM)*, IEEE, (1999) 109-118.

[60] Eaddy M., Aho A. Towards Assessing the Impact of Crosscutting Concerns on Modularity, *Workshop on Assessment of Aspect Techniques (ASAT) at 6th Aspect Oriented Software Development Conference (AOSD)*. Vancouver, BC, Canada, (2007)

[61] Eaddy M., Aho A., Murphy, G. Identifying, Assigning, and Quantifying Crosscutting Concerns, *First International Workshop on Assessment of Contemporary Modularization Techniques (ACoM)*, Minneapolis, USA, (2007)

[62] Eaddy, M., Zimmermann, T., Sherwood, K., Garg, V., Murphy, G., Nagappan, N., Aho, A. Do Crosscutting Concerns Cause Defects?, *IEEE Transactions on Software Engineering, Volume 34, Issue 4* (2008) 497-515

[63] Eaddy M. *An Empirical Assessment of the Crosscutting Concern Problem*. Ph.D. Dissertation, Columbia University (2008)

[64] *Early Aspect Mining based on Syntactical and Dependency analyses*. http://www.unex.es/eweb/earlyaspectmining

[65] *Early Aspects Workshop at Sixth International Conference on AOSD (2007).* http://aosd.di.fct.unl.pt/ea2007/.

[66] Easterbrook, E. *How Theses Get Written: Some Cool Tips*. University of Toronto (2004) http://www.cs.toronto.edu/~sme/presentations/thesiswriting.pdf

[67] Egyed, A., Biffl, S., Heindl, M. and Grünbacher, P. A value-based approach for understanding cost-benefit trade-offs during automated software traceability, *3rd International Workshop on Traceability in Emerging Forms of Software Engineering*, Long Beach, USA (2005)

[68] Elrad, T., Aksit, M., Kiczales, G., Lieberherr, K., Ossher, H. (2001) Discussing Aspects of Aspect-Oriented Programming AOP: Frequently-Asked Questions. *Communications of the ACM, 44 (10),* ISSN 0001-0782 (2001) 33-38.

[69] Eriksson, M., Börstler, J., and Borg, K. The pluss approach - domain modelling with features, use cases and use case realizations. *9th International Conference on Software Product Lines*, *LNCS, Vol. 3714,* Springer-Verlag (2005) 33-44

[70] Ferrante, J., Ottenstein, K.J. and Warren J.D. The Program Dependence Graph and its Use in Optimization. *ACM Transactions on Programming Language and System, Vol. 9, No. 3* (1987)

[71] Figueiredo, E., Cacho, N., Sant'Anna, C., Monteiro, M., Kulesza, U., Garcia, A., Soares, S., Ferrari, F. Khan, S., Filho, F. and Dantas, F. Evolving Software Product Lines with Aspects: An Empirical Study on Design Stability. *30th International Conference on Software Engineering (ICSE)*, Leipzig, Germany (2008)261-270.

[72] Figueiredo, E., Sant'Anna, C., Garcia, A., Bartolomei, T., Cazzola, W., Marchetto, A.: On the Maintainability of Aspect-Oriented Software: A Concern-Oriented Measurement Framework. *12th European Conference on Software Maintenance and Reengineering (CSMR),* Athens, Greece (2008) 183-192.

[73] Figueiredo, E., Silva, B., Sant'Anna, C., Garcia, A., Whittle, J. and Nunes, D. Crosscutting Patterns and Design Stability: An Exploratory Analysis. *17th IEEE International Conference on Program Comprehension (ICPC)*, May, Vancouver (2009)

[74] Figueiredo, E., Galvao, I., Khan, S., Garcia, A., Sant'Anna, C., Pimentel, A., Medeiros, A., Fernandes, L., Batista, T., Ribeiro, R., van den Broek, P., Aksit, M., Zschaler, S. and Moreira, A. Detecting Architecture Instabilities with Concern Traces: an Exploratory Study. Joint 8th Working IEEE/IFIP Conference on Software Architecture, Cambridge, UK (2009)

[75] Filman, R. and Friedman, D. *Aspect-Oriented Programming is Quantification and Obliviousness.* Technical Report 01.12, (2000) http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.28.287.

[76] Filman, R., Elrad, T., Clarke, S. and Aksit, M. *Aspect-Oriented Software Development,* Addison-Wesley (2004)

[77] Finkelstein, A. and Sommerville, I. The Viewpoints FAQ. *BCS/IEE Software Engineering Journal 11(1)*, (1996) 2–4

[78] Fox, J. (2005) A Formal Foundation for Aspect Oriented Software Development. *14th Congreso Internacional de Computación CIC*, ISBN 970-36-0266-5, Mexico (2005) 434-444

[79] France, R., Kim, D., Ghosh, S. and Song, E. A UML-Based Pattern Specification Technique. *IEEE Transactions on Software Engineering, Volume 30(3)* (2004)

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

[81] Ganter, B. and Wille, R. *Formal Concept Analysis*. Springer-Verlag, Berlin, Heidelberg, New York (1996)

[82] Garcia, A, Lucena, C. Taming Heterogeneous Agent Architectures. *Commun. ACM 51(5)* (2008) 75-81

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

[84] *Gears home page*. http://www.biglever.com/solution/product.html.

[85] Gradecki, J.D. and Lesiecki, N. *Mastering AspectJ: Aspect-Oriented Programming in Java*. Wiley, 2003

[86] Greenwood, P., Bartolomei, T., Figueiredo, E., Dosea, M., Garcia, A., Cacho, N., Sant'Anna, C., Soares, S., Borba, P., Kulesza, U., Rashid, A. On the Impact of Aspectual Decompositions on Design Stability: An Empirical Study. *21st European Conference on Object Oriented Programming,* Berling, Germany (2007) 176-200.

[87] Griss, M. Implementing product-line features by composing aspects. 1st International Software Product Line Conference Conference, Denver, USA (1996) 271—288,

[88] Groher, I. and Völter, M. Using Aspects to Model Product Line Variability. *Early Aspects Workshop at 12th International Conference on Software Product Line*, (2008), 89-95.

[89] Groher, I. and Völter, M. Aspect-Oriented Model-Driven Software Product Line Engineering. *Transactions on Aspect-Oriented Software Development VI 5560*, (2009) 111-152

[90] Grundy, J. Aspect-Oriented Requirements Engineering for Component-based Software Systems, *4th IEEE International Symposium on Requirements Engineering*, Limerick, Ireland (1999)

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

[92] Hannemann, J. and Kiczales, G. Overcoming the Prevalent Decomposition in Legacy Code. *Workshop on Advanced Separations of Concerns at 23rd International Conference on Software engineering (ICSE)*, Toronto, Canada (2001)

[93] Hannemann, J. and Kiczales, G. Design pattern implementation in Java and AspectJ. *17th ACM conference on Object Oriented Programming, Systems, Languages and Applications (OOPSLA),* Seattle, USA (2002) 161—173.

[94] Hannemann, J. *The Aspect Mining Tool website* (2007) http://hannemann.pbworks.com/Aspect+Mining+Tool

[95] *Hyper/J: A Multi-Dimensional Separation of Concerns for Java* http://www.research.ibm.com/hyperspace/HyperJ/HyperJ.htm

[96] ISO, *Software engineering - Product quality - Part 1: Quality model, ISO/IEC 9126-1*, International Organization of Standardization (2001)

[97] Jackson, M. *Problem frames: analyzing and structuring software development problems*. Addison-Wesley Longman Publishing Co., Inc., Boston, USA (2000)

[98] Jacobson, I. Use Cases and Aspects – Working Seamlessly Together, *Journal of Object Technology, vol. 2, no. 4* (2003) 7-28

[99] Jacobson, I. and Ng, P. *Aspect-Oriented Software Development with Use Cases*. Addison-Wesley Professional (2004)

[100] *JHotDraw version 6.0* (2007) http://jhotdraw.org

[101] Johnson, J. Identifying Redundancy in Source Code Using Fingerprints. IBM Centre for Advanced Studies Conference (1993) 171-183

[102] Kamiya, T., Kusumoto, S., and Inoue, K. CCFinder: a multilinguistic token-based code clone detection system for large scale source code. IEEE Transsactions of Software Engineering. vol. 28, no 7 (2002) 654 - 670

[103] Kan, S.H. Metrics and Models in Software Quality Engineering (2nd Edition). Addison-Wesley Professional (2002)

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

[105] Karanjkar, S. Development of graph clustering algorithms. Master's thesis, University of Minnesota (1998)

[106] Katz, S. and Rashid, A. From Aspectual Requirements to Proof Obligations for Aspect-Oriented Systems, *12th International Conference on Requirements Engineering (RE)*, Kyoto, Japan (2004)

[107] Kellens, A., Mens, K. and Tonella, P. A survey of automated code-level aspect mining techniques, *Transactions on Aspect-Oriented Software Development IV, LNCS Vol. 4640,* (2007) 143-162

[108] Kelly, D. A Study of Design Characteristics in Evolving Software Using Stability as a Criterion. *IEEE Transactions on Software Engineering. Vol. 32* (2006) 315-329

[109] Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C., Loingtier, J.M., Irwin, J.: Aspect-Oriented Programming. *11th European Conference on Object Oriented Programming (ECOOP), LNCS, vol. 1241*, Springer, Heidelberg (1997) 220–242.

[110] Kiczales, G., Hilsdale, E., Hugunin, J., Kersten, M., Palm, J. and Griswold, W.G. An overview of AspectJ. *15th European Conference on Object-Oriented Programming (ECOOP), LNCS 2072*, Springer (2001) 327-353.

[111] Kiczales, G. and Hilsdale, E. Aspect-oriented programming. *Foundations of Software Engineering (Tutorial)*, (2001)

[112] Kiczales, G. Crosscutting. *AOSD.NET Glossary* (2005) http://aosd.net/wiki/index.php?title=Crosscutting

[113] Kit, L., Man, C. and Baniassad, E. Isolating and Relating Concerns in Requirements using Latent Semantic Analysis. *ACM conference on Object Oriented Programming, Systems, Languages and Applications (OOPSLA)*, Portland, USA (2006)

[114] Komondoor, R. and Horwitz, S. Using Slicing to Identify Duplication in Source Code. *International Symposium Static Analysis (SAS)*, Lecture Notes In Computer Science; Vol. 2126 (2001), 40-56.

[115] Kovačević, J., Aférez, M., Kulesza, U., Moreira, A., Araújo, J., Amaral, V., Alves, V., Rashid, A. and Chitchyan, R. *Survey of the state-of-the-art in Requirements Engineering for Software Product Line and Model-Driven Requirements Engineering*, Deliverable D1.1 of the AMPLE Project (2007)

[116] Krinke, J. Identifying Similar Code with Program Dependence Graphs. *8th Working Conference on Reverse Engineering (WCRE'01)*, (2001) 301-309

[117] Laddad, R. AspectJ in Action: Practical Aspect-Oriented Programming. *Manning Publicacions Co.* (2002)

[118] Lamsweerde, A. van, Goal-oriented requirements engineering: a guided tour. *5th International Symposium on Requirements Engineering*, IEEE CS Press (2001) 249-261

[119] Landauer, T.K., Foltz, P.W., and Laham, D. An Introduction to Latent Semantic Analysis. *Discourse Processes*, (1998) 259-284.

[120] Lee, K., Kang, K. and Kim, M. Feature Dependency Analysis for Product Line Component Design. *8th International Conference on Software Reuse: Methods, Techniques and Tools (ICSR)*, LNCS 3107, Springer, Madrid, Spain (2004) 69-85

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

[122] Lieberherr, K. Aspect-Oriented Programming with Adaptative Methods. *Communications of the ACM. Vol. 44, 10* (2001)

[123] Linden, F.J., Schmid, K., and Rommes, E. *Software Product Lines in Action: The Best Industrial Practice in Product Line Engineering*. Springer (2007)

[124] Lopez-Herrejon, R., Apel, S. Measuring and Characterizing Crosscutting in Aspect-Based Programs: Basic Metrics and Case Studies. International Conference on Fundamental Approaches to Software Engineering (FASE), Braga, Portugal (2007)

[125] Loughran, N. and Rashid, A. Framed Aspects: Supporting Variability and Configurability for AOP. *8th International Conference on Software Reuse: Methods, Techniques and Tools (ICSR)*, LNCS 3107, Springer, Madrid, Spain (2004) 127 - 140

[126] Loughran N., Sampaio, A. and Rashid, A. From Requirements Documents to Feature Models for Aspect Oriented Product Line Implementation. *Workshop on Model-Driven Development for Product Lines at 8th MODELS*, Montego Bay, Jamaica (2005)

[127] Marcus, A. and Maletic, J.I. Identification of High-Level Concept Clones in Source Code. *16th IEEE International Conference Automated Software Engineering (ASE)*, San Diego, USA (2001) 107-114

[128] Marin, M., Deursen, A.V., and Moonen, L. Identifying Aspects Using Fan-In Analysis. *11th Working Conference on Reverse Engineering (WCRE)*, IEEE Computer Society Press (2004) 132-141

[129] Masuhara, H., Kiczales, G. Modelling Crosscutting in Aspect-Oriented Mechanisms. *17th European Conference on Object Oriented Programming (ECOOP)*, Darmstadt (2003) 2–28

[130] Mayrand, J., Leblanc, C., and Merlo, E. Experiment on the Automatic Detection of Function Clones in a Software System Using Metrics. *12th International Conference on Software Maintenance (ICSM)*, Monterey, USA (1996), 244-254

[131] McGregor, J., Northrop, L., Jarrad, S., and Pohl, K. *Initiating software product lines*. IEEE Software 19, 4 (2002) 24-27

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

[133] Meier, W. eXist XML Database Management System, (2002), http://exist.sourceforge.net/

[134] Mens, K. and Tourwé, T. Delving source code with formal concept analysis. *Elsevier Journal on Computer Languages, Systems & Structures, 31(3-4),* Special Issue: Smalltalk. Elsevier (2005) 183-198

[135] Mezini, M. and Ostermann, K. Modules for Crosscutting Models. *Ada-Europe 2003*, LNCS, vol. 2655, Springer, Heidelberg (2003) 24–44

[136] Mezini, M. and Ostermann, K. Conquering Aspects with Caesar. *2nd International Conference on Aspect-Oriented Software Development (AOSD)*, ACM Press, Boston, USA (2003) 90-100

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

[138] Morin, B., Barais, O. and Jézéquel, J.M. Weaving Aspect Configurations for Managing System Variability. *2nd International Workshop on Variability Modelling of Software-intensive Systems (VaMoS)*, Essen, Germany (2008)

[139] Neyman, J. & Pearson, E.S. On the Use and Interpretation of Certain Test Criteria for Purposes of Statistical Inference, *Joint Statistical Papers*, Cambridge University Press (1967)

[140] Niu, N. and Easterbrook, S. Analysis of Early Aspects in Requirements Goal Models: A Concept-Driven Approach. *Transactions on Aspect-Oriented Software Development III 4620* (2007) 40-72

[141] Nora, B., Said. G. and Fadila, A. A comparative classification of aspect mining approaches, *Journal of Computer Science 2 (4),* ISSN 1549-3636 (2006) 322-325

[142] Object Technology International, INC. *Eclipse platform technical overview*. *White paper*, (2003) http://www.eclipse.org/whitepapers/eclipse-overview.pdf

[143] *openArchitectureWare.* http://www.openarchitectureware.org/

[144] Parnas, D. On the Criteria to Be Used in Decomposing Systems into Modules. *Communications of the ACM, Vol. 15.* (1972)

[145] *PetStore 2.0* (2006) https://blueprints.dev.java.net/petstore/

[146] Pinto, M. *CAM/DAOP: Modelo y Plataforma Basados en Componentes y Aspectos*. PhD Thesis, University of Malaga (2004)

[147] Pohl, K., Böckle, G. and van der Linden, F. *Software Product Line Engineering: Foundations, Principles and Techniques*, Springer, Berlin, Germany (2005)

[148] Project Bauhaus (2005) http://www.bauhaus-stuttgart.de/

[149] Ramesh, B., Jarke, M. Toward reference models for requirements traceability. *IEEE Transactions on Software Engineering 27(4)*, (2001) 58–93

[150] Rashid, A. *Arcade Tool* (2006), http://www.comp.lancs.ac.uk/computing/aop/Software.htm.

[151] Rashid, A., Moreira, A., Araujo, J. Modularisation and Composition of Aspectual Requirements. *2nd Aspect Oriented Software Conference* (AOSD), Boston, USA (2003)

[152] Robillard, M. and Murphy, G. Concern Graphs: Finding and Describing Concerns Using Structural Program Dependencies. *24th International Conference on Software Engineering (ICSE)*, Orlando, USA (2002) 406-416

[153] Robillard, M. and Murphy, G. FEAT a tool for locating, describing, and analyzing concerns in source code. *25th International Conferenc on Software Engineering (ICSE)*, Portland, USA (2003) 822-823.

[154] Robillard, M. and Murphy, G. Representing concerns in source code. *ACM Transactions Software Engineering and Methodology, Vol. 16, Nº 1* (2007)

[155] Sampaio, A., Chitchyan, R., Rashid, A. and Rayson, P. EA-Miner: A Tool for Automating Aspect-Oriented Requirements Identification. *20th International Conference on Automated Software Engineering (ASE)*, California, USA (2005)

[156] Sampaio, A., Loughran, L., Rashid, A., Rayson, P. Mining Aspects in Requirements. *Early Aspects Workshop at 4th Aspect Oriented Software Development Conference (AOSD)*, Chicago, USA (2005)

[157] Sampaio, A. and Rashid, A. *Report on Evaluation of Aspect Identification Tool (EA-Miner) in Case Studies*. AOSD-Europe Network of Excellence. AOSD-Europe-ULANC-33 (2007)

[158] Sampaio, A., Rashid, A., Chitchyan, R., and Rayson, P. EA-Miner: Towards Automation in Aspect-Oriented Requirements Engineering. *Transactions on Aspect-Oriented Software Development 3* (2007) 4-39

[159] Sant'Anna, C., Garcia, A., Chavez, C., Lucena, C., von Staa, A. On the Reuse and Maintenance of Aspect-Oriented Software: an Assessment Framework. *Brazilian Symposium on Software Engineering (SBES)*, Manaus, Brazil (2003)

[160] Sant'Anna, C., Figueiredo, E., Garcia, A., Lucena, C. On the Modularity Assessment of Software Architectures: Do my architectural concerns count? *First European Conference on Software Architecture*, Madrid, Spain (2007)

[161] Sawyer, P., Rayson, P., and Garside, R. REVERE: Support for Requirements Synthesis from Documents. *Information Systems Frontiers 4, 3* (2002)

[162] Schmid, K. and Verlage, M. The Economic Impact of Product Line Adoption and Evolution. *IEEE Software 19,* (2002) 50-57

[163] Shepherd, D. and Pollock, L. Interfaces, Aspects, and Views. *Workshop on Linking Aspect Technology and Evolution (LATE) at 4th International Conference on Aspect Oriented Software Development (AOSD)*, Chicago, USA (2005)

[164] Shepherd, D., Pollock, L., and Tourwé, T. Using language clues to discover crosscutting concerns. *Workshop on Modelling and Analysis of Concerns in Software at 27th International Conference on Software Engineering (ICSE)*, vol. 30, issue 4 (2005)

[165] Shonle, M., Neddenriep, J., and Griswold, W. AspectBrowser for Eclipse: a case study in plug-in retargeting. *Workshop on Eclipse Technology eXchange at 19th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, Vancouver, Canada (2004) 78-82

[166] *Software Engineering Institute at the Carnegie Mellon University*, http://www.sei.cmu.edu/

[167] Sutton, S. and Rouvellou, I. Modelling of Software Concerns in Cosmos. *1st Aspect Oriented Software Development Conference* (AOSD), Enschede, The Netherlands (2002)

[168] Sutton, S. Concerns in a Requirements Model - A Small Case Study. *Early Aspects Workshop at 2nd Aspect Oriented Software Development Conference (AOSD)*, Boston, USA (2003)

[169] Sutton, S. and Rouvellou, I. *Concern Modelling for Aspect-Oriented Software Development*. Aspect-Oriented Software Development, R. E. Filman, T. Elrad, S. Clarke, and M. Aksit, Eds.: Addison-Wesley (2004) 479-505.

[170] Suvèe, D., Vanderperren, W. and Jonkers, V. JAsCo: An Aspect-Oriented Approach Tailored for Component Based Software Development. *2nd International Conference on Aspect-Oriented Software Development (AOSD)*, Boston, USA (2003)

[171] Tarr, P., Ossher, H., Harrison, W., and Sutton, S. N Degrees of Separation: Multi-Dimensional Separation of Concerns. *21st International Conference on Software Engineering* (1999) 107-119

[172] Taylor, J. *An Introduction to Error Analysis. The Study of Uncertainties in Physical Measurements*. University Science Books, 2nd Edition. ISBN: 0-935702-75-X (1997)

[173] Tekinerdogan, B.: ASAAM: Aspectual Software Architecture Analysis Method. *4th Working IEEE/IFIP Conference on Software Architecture (WICSA), Cambridge, UK* (2004)

[174] Tekinerdoğan, B., Akşit, M. and Henninger, F. Impact of Evolution of Concerns in the Model-Driven Architecture Design Approach, *Electronic Notes in Theoretical Computer Science 163(2)*, (2007) 45-64

[175] *Theme/UML website*, https://www.dsg.cs.tcd.ie/aspects/themeUML

[176] *Tomcat* (1999) http://tomcat.apache.org/, 1999

[177] Tonella, P. and Ceccato, M. Aspect Mining through the Formal Concept Analysis of Execution Traces. *11th Working Conference on Reverse Engineering (WCRE)*, Delft, the Netherlands (2004)

[178] Trujillo, S., Batory, D., and Diaz, O. Feature Oriented Model Driven Development: A Case Study for Portlets. *29th International Conference on Software Engineering (ICSE)*, Minneapolis, USA (2007)

[179] Tufis, D. and Mason, O. Tagging Romanian Texts: a Case Study for QTAG, a Language Independent Probabilistic Tagger. *1st International Conference on Language Resources and Evaluation (LREC)*, Granada, Spain (1998) 589-596

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

[181] Voelter, M. and Groher, I. Product Line Implementation using Aspect-Oriented and Model-Driven Software Development. *11th International Software Product Line Conference (SPLC)*, Kyoto, Japan (2007)

[182] Wilde, N., Buckellew, M., Page, H., Rajlich, V. and Pounds, L. A comparison of methods for locating features in legacy software, *Journal of Systems and Software, vol. 65* (2003) 105-114

[183] Wong, W., Gokhale, S., Horgan, J. Quantifying the Closeness between Program Components and Features. *Journal of Systems and Software,* (2000)

[184] *WordNet database*. Princeton University. http://wordnet.princeton.edu/

[185] *XMI Mapping Specification, v2.1* (2005) http://www.omg.org/technology/documents/formal/xmi.htm

[186] XQuery 1.0 *An XML Query Language*. W3C Recommendation, (2007) http://www.w3.org/TR/xquery/

[187] Yoshikiyo, W., Griswold, W., Y, Y. and Yuan, J. Aspect Browser: Tool Support for Managing Dispersed Aspects. *1st Workshop on Multi-Dimensional Separation of Concerns in Object-oriented Systems at 14th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, Denver, USA (1999)

[188] Young, T. *Using AspectJ to Build a Software Product Line for Mobile Devices*. MSc dissertation, Univ. of British Columbia (2005)

[189] Yourdon, E. and Constantine, L. *Structured Design: Fundamentals of a Discipline of Computer Program and Systems Design*. Prentice Hall (1979)

[190] Yu, Y., Leite, J. and Mylopoulos, J. From Goals to Aspects: Discovering Aspects from Requirements Goal Models. *12th International Requirements Engineering Conference*, IEEE Computer Society (2004) 38-47

[191] Zhang, C., Dapeng, G. and Jacobsen, H. *Extended Aspect Mining Tool*, (2002) http://www.eecg.utoronto.ca/~czhang/amtex/

[192] Zhang, C. and Jacobsen, H. *A Prism for Research in Software Modularization Through Aspect Mining*, Technical Communication, Middleware Systems Research Group, University of Toronto (2003)

[193] Zhao, J. and Rinard, M. System dependence graph construction for aspect-oriented programs. Technical Report MITLCS-TR-891, Laboratory for Computer Science, MIT (2003)

... in the coming years, we will be able to revolutionize the production of software and then bring my final dream to fruition...

We will then have other dreams to fulfill!

- Ivar Jacobson