Dealing with Crosscutting Concerns in Existing Software

Leon Moonen
Simula Research Laboratory, Norway
Leon.Moonen@computer.org

Abstract

This paper provides a roadmap for dealing with crosscutting concerns while trying to understand, maintain, and evolve existing software systems. We describe an integrated, systematic, approach that helps a software engineer with identifying, documenting and migrating crosscutting concerns in the source code of a software system, and discuss the integration considerations. We conclude with a number of lessons learned and directions for future research.

1. Introduction

It is well-known that the majority of software engineers work on the evolution of existing systems instead of the creation of new systems. As such, they are regularly confronted with the daunting task of needing to understand (part of) a complex system for which they have little to no a priori knowledge. Effective comprehension is essential for maintenance and evolution, and can take up as much as 50%–90% of the software's total costs [12, 39, 45].

A software engineering technique that helps to manage system complexity is *separation of concerns*: a modularization approach that divides a system in distinct pieces that have as little overlap in functionality as possible, thereby improving comprehensibility [2, 10, 38]. However, the majority of applications that are used and maintained today remain hard to understand, evolve, or reuse, because a complete separation of concerns is difficult or even impossible to achieve using the modularization mechanisms available in most popular programming paradigms [47].

Concerns that cannot be cleanly decomposed and isolated in a system's *dominant decomposition* into modules are called *crosscutting concerns*. Examples include the authorization of users, keeping a history of changes to a patient record, persistence of data, transaction management, and exception handling. Forcing the implementation of a crosscutting concern into an existing decomposition results in symptoms like *scattering*, where the implementation of a single concern is spread over several modules, and *tangling*, where a single module implements multiple concerns. As a side effect, it often also results in *code duplication*.

These symptoms have an obvious negative impact on the system's comprehensibility and evolvability: first of all, it becomes difficult to recognize and consistently change a crosscutting concern as a whole when it is scattered over the system. Second, the system's ordinary concerns become harder to understand because they get tangled with the crosscutting ones: classes and methods no longer only deal with their primary responsibility, but also need to take care of secondary, crosscutting, concerns. Also, because the concerns are not explicit, they may be overlooked, resulting in modifications or extensions that break the system's conceptual integrity. Figure 1 illustrates the scattering and tangling of crosscutting concerns throughout the modules of a system. The vertical bars represent modules, horizontal lines correspond to code fragments implementing a concern, and grayscales indicate various concerns.

Aspect-oriented software development (AOSD) has emerged as development paradigm for advanced separation of concerns. It seeks to separate even those crosscutting concerns that are difficult to decompose and isolate with earlier programming methodologies. *Aspect-oriented programming (AOP)* captures crosscutting concerns in a new modularization unit, the *aspect*, enabling developers to organize code, design and other artifacts in a more logical way, according to the concern they address. It offers composition facilities to create complete applications by *weaving* aspects into a core system at the appropriate places, keeping the core *oblivious* to the crosscutting concerns.

AOSD promises significant benefits in the areas of software comprehension, maintenance and evolution and could therefore play an important role in the revitalization of ex-

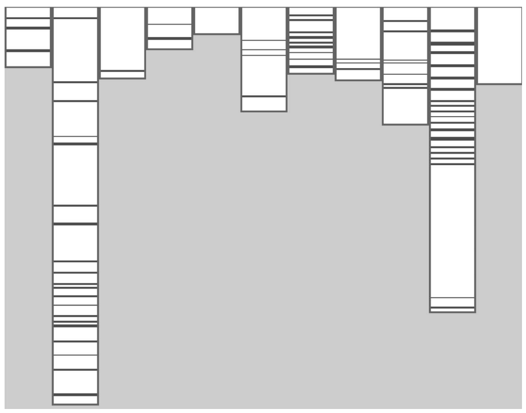

Figure 1. Scattering and tangling of crosscutting concerns throughout a system (src: Bruntink [6])

isting (legacy) software systems. However, most of the existing AOSD approaches seem to focus primarily on dealing with crosscutting concerns in new systems that are developed from scratch. Fully exploiting AOSD in the context of existing software systems imposes different requirements and constraints and has its own set of challenges and open issues that need to be solved.

This paper focuses on dealing with crosscutting concerns while trying to understand, maintain, and evolve existing (legacy) software systems. We take a code-centric point of view, looking at techniques that help a developer to (a) identify crosscutting concerns in the source code of an existing system, (b) document how these code fragments relate to each other and aggregate into higher level structures, and (c) manage evolution of the system by increasing the separation of concerns via views or migrations.

Our goal is to provide a roadmap for potential users of aspect-oriented technology in the context of existing systems. For more information about aspect-oriented programming, we refer to [1, 15, 26]. For a recent survey of automated aspect mining techniques, we refer to [36].

2. Background and Terminology

Over the recent years, various researchers have started to look at identifying, documenting and transforming crosscutting concerns in existing software systems, a practice that is commonly referred to as *aspect mining and refactoring*.[1]

Crosscutting concerns can be characterized by their *intent* and *extent*. A concern's *intent* is the objective of the concern and the *extent* is the concrete representation of that concern in the system's source code. For example, the intent of a tracing concern is that all relevant input and output parameters of public methods are appropriately traced, and the extent consists of the collection of all statements actually generating traces for a given method parameter.

In aspect mining, we search for source code elements that belong to the *extent* of crosscutting concerns. We will refer to such code elements as *(concern) seeds*. Once a seed for a concern is identified, *seed expansion* can be used to expand it to the full extent of the concern, for example by following data and control flow dependencies. Automated aspect mining techniques yield *candidate seeds*, which can become *confirmed seeds* (or simply "seeds") when they are validated by a human expert, or *non-seeds* if rejected. Non-seeds are also referred to as *false positives*; a *false negative* is a parts of a known crosscutting concern that is missed due to inherent limitations of the mining approach.

[1] Note that some people consider this a misnomer as aspects in AOP are already isolated modules representing a concern, so "mining" aspects is trivial. Likewise, aspect refactoring implies restructuring of existing aspects, and not software migration aimed at isolating crosscutting concerns. The terms *crosscutting concern identification* and *(crosscutting) concern migration* have been proposed as alternatives.

3. Identifying Crosscutting Concerns

Experience shows that manual adoption of aspect-oriented techniques in existing software systems is a difficult and error-prone process [8, 27]. Challenges include the, at times complicated and irregular, entanglement of concerns, the sheer size and complexity of the software, and the lack of up-to-date documentation and knowledge about the system.

Clearly, there is a need for tools and techniques that help developers to identify and document cross-cutting concerns in existing systems, and (potentially) help to migrate the discovered cross-cutting concerns into aspects.

Aspect mining aims at finding crosscutting concerns in existing code. Once these concerns have been identified, they can be used for program understanding and evolution purposes. We distinguish query-based and generative mining approaches; they are discussed in more detail below.

3.1. Query-based approaches

Query-Based techniques start exploration from search patterns provided by the user. Code fragments that match the pattern are used as concern seeds and can interactively be expanded to more complete concerns using the tool.

Aspect Browser, one of the first query-based tools, uses lexical pattern matching for querying the code and a map metaphor for visualizing the results [16]. The Aspect Mining Tool (AMT) extends the lexical search from the Aspect Browser with structural search for usage of types within a given piece of code [19]. Both tools display the query results in a Seesoft-type view [11], as also shown in figure 1.

AMT has evolved into PRISM, a tool supporting identification activities by means of lexical and type-based patterns called *fingerprints* [53]. A fingerprint can be defined, for example, as any method in a given class of which the name starts with a given word. A software engineer defining fingerprints is assisted by so-called *advisors*.

The Feature Exploration and Analysis Tool FEAT is an Eclipse plug-in aimed at locating, describing, and analyzing concerns in source code [40]. It is based on *concern graphs* which represent the elements of a concern and their relationships. A FEAT session starts with an element known to be a concern seed, and FEAT allows the user to query relations, such as direct call relations, between the seed and other elements in the program. Query results that the user considers relevant to the (crosscutting) concern under investigation can be added to the concern-graph of that concern.

The Concern Manipulation Environment CME aims at supporting the whole lifecycle of an aspect-oriented development project [20]. This includes aspect identification facilities through an integrated search component (Puma) that uses an extensible query language (Panther) [48]. CME also allows for concern management similar to FEAT. Most importantly, CME provides a infrastructure that is potentially usable for the integration of different approaches to aspect mining, including seed identification and concern exploration and management.

Various query-based tools have been compared in a recent study [37] which shows that the queries and patterns are mostly derived from application knowledge, code reading, words from task descriptions, or names of files. This clearly indicates that prior knowledge of the system or known starting points strongly affect the usefulness of the outcomes of an explorative approach.

3.2. Generative approaches

Generative aspect mining approaches aim at automatically generating crosscutting concern seeds, usually based on some form of program analysis. These techniques look for symptoms of crosscutting functionality (such as code scattering and tangling) and identify program elements exhibiting these symptoms as candidate aspect seeds.

Our own *fan-in analysis* [29, 32] aims at recognizing code scattering: It uses the idea that common functionality in scattered code is likely to be factored out into helper methods. Since these methods are called from many places, they have a high fan-in value, which makes high fan-in a good indicator of the scattered implementation of a concern.

Aspect mining using fan-in analysis consists of three steps: First, we identify the methods with the highest fan-in values. Second, we filter out methods that may have a high fan-in but for which it is unlikely that there is a systematic pattern in their usage that could be exploited in an aspect solution (e.g., getters, setters, certain utility methods). Third, we inspect the call sites of the high fan-in methods, in order to determine if the method in question does indeed implement crosscutting functionality. The last step is the most labor intensive, and it is based on an analysis of recurring patterns in, for example, the call sites of the high fan-in method. All steps are supported by a publicly available Eclipse plug-in called FINT.[2]

Note that there are multiple ways in which a fan-in metric can be defined, for example depending on the way that polymorphic callers and callees are taken into account [5, 21, 22]. For calls *from* polymorphic methods, we count the number of unique calls from different bodies; calls *to* polymorphic methods are counted for each occurrence in the inheritance hierarchy. We refer to [32] for a detailed discussion of the various options and their implications.

Additional assessments of fan-in analysis have been performed in [17], where the metric is used to measuring scattering, and in the Timna framework [43], which uses machine learning techniques to combine the results of several aspect mining techniques. In addition, software repository mining has been used to search for concerns by analyzing changes in fan-in values between different versions of the system under investigation [4].

[2] `http://swerl.tudelft.nl/view/AMR/FINT`

Several authors have looked at code cloning as a side effect of code scattering: Shepherd et al. [42] describe the use of clone detection based on program dependence graphs and the comparison of individual statement's abstract syntax trees for mining aspects in Java source code. Bruntink et al. [7] evaluate the use of three clone detection techniques for concern identification on an industrial C component. The authors start from four dedicated crosscutting concerns that were manually identified and annotated in the code and evaluate the suitability of various clone detection techniques for identifying these concerns automatically by measuring the coverage of the annotated concerns by detected clones.

Dynamic analysis approaches to aspect mining include examining execution traces for recurring patterns [3] and associating method executions to traces that are specific to certain use-cases using formal concept analysis [50]. The typical dynamic analysis challenge is having to exercise all functionality in the system that could lead to aspect candidates. This issues is avoided in a variation on the first approach which searches for recurring execution patterns in control flow graphs using static analysis [25].

Identifier analysis uses formal concept analysis to group program elements based on their names [52]. This is based on the idea that developers use naming conventions that hint at the relation between scattered elements of a concern.

The relation between interfaces and crosscutting concerns has been investigated through a number of indicators like the naming pattern used by the interface definition, the coupling between the methods of the implementing class and the methods declared by the interface, or the package location of the interface and its implementing class [49].

4. Crosscutting Concern *Sorts*

Despite significant research efforts on the design and development of aspect-oriented languages, and on aspect mining techniques, there is still little consensus on what exactly constitutes a (crosscutting) concern. Sutton and Rouvellou [46] define a concern to be "any matter of interest in a software system.", Filman et al. [13] refer crosscutting concerns as "systematic behavior" whose implementation is "scattered throughout the rest of an implementation", and Kiczales et al. [24] define such concerns as "properties" that "cannot be cleanly encapsulated in a generalized procedure".

As a result, it is not clear how such concerns can be systematically recognized, understood, and clearly documented in source code and aspect mining publications rely on non-uniform and ad-hoc descriptions of the crosscutting concerns they aim to identify and of the steps to be taken to map their results onto potentially associated concerns.

The lack of a sound definition of crosscutting concerns

Sort	Short description	Examples
Consistent Behavior	A set of method-elements consistently invoke a specific action as a step in their execution.	Log events; Wrap or translate business service exceptions [32]; Notify and register listeners; Authorization
Redirection Layer	A type-element acts as a front-end interface having its methods responsible for receiving calls and redirecting them to dedicated methods of a specific reference, optionally executing additional functionality.	Decorator (pattern), Adapter (pattern) [18]; Forward local calls to remote instances (RMI) [44]
Role Superimposition	Type-elements extend their core functionality through the implementation of a secondary role.	Many roles specific to design patterns: Observer, Command, Visitor, etc.; Persistence [32]
Expose Context (Pass Context)	Methods in a call chain consistently use parameter(s) to propagate context information along the chain.	Transaction management, Authorization [26]
Support Classes for Role Superimposition	Types implement secondary roles by enclosing support classes. The nesting defines and enforces the relation between the enclosing and the support class.	Undo in JHOTDRAW; Iterators for Collection types; Event dispatcher for managing notifications
Exception Propagation	Methods in a call chain consistently (re-)throw exceptions from their callees when no appropriate response is available.	Checked Exceptions, such as *IOException* for failed disk access, or SQLExceptions thrown in JDBC API

Table 1. Selection of Crosscutting Concern Sorts

prevents effective comparison and combination of mining techniques, as is shown by the following (hypothetical but likely) evaluation scenario: One approach presents results to an instance of the Observer pattern through elements that are crosscut by the super-imposed roles of Subject and Observer [14]. A second approach reports results related to the same instance, but identified through elements that consistently implement the notification of the observer. Human analyzers interpret the results and agree on ad-hoc convergence rules: the Observer instance is easily accepted as common finding based on the valid results from both techniques, especially since the Observer pattern is a well known example of crosscutting behavior. In addition, each technique can explain how the implementation of the Observer is related to its own identification mechanism.

The problems with the scenario sketched above are apparent: convergence is ad-hoc and relies on inconsistent interpretation of the reported findings (since the Observer actually comprises two distinct crosscutting concerns and each technique finds only one). The results require a tedious manual correlation effort as they do not (always) overlap directly, but are related by design decisions. Moreover, such a scenario can only be successful when both techniques provide detailed descriptions of results and associated concerns, a situation that is unlikely in practice (especially for other concerns than a well-known design pattern). A first step towards overcoming these issues is developing a consistent system for addressing and describing crosscutting concerns.

Over the last three to four years, we have analyzed crosscutting concerns in a range of Java systems, including JBoss, TOMCAT, JHOTDRAW, and the J2EE PETSTORE, totaling over 500,000 lines of code (a detailed description of the crosscutting concerns in the latter three of these systems is provided in [32] and [9]). The experience gained from these studies allows us to recognize and categorize a number of typical *atomic* "building blocks" for crosscutting functionality, i.e., concerns that cannot be decomposed into smaller, yet meaningful, concerns. Examples include the *superimposition of a new role* on an existing class or the implementation of *consistent behavior*, for example for precondition checking, refreshing the display after updates to a drawing or tracing certain events.

We refer to these "building blocks" as *crosscutting concern sorts* [30, 33]. They are characterized by a number of properties common to all the instances of the sort, such as a generic description of the sort (the sort's *intent*), and their specific underlying relations and implementation idioms in non-aspect-oriented languages (i.e., the sort's *extent* or its characterizing implementation *symptoms*). Table 1 shows a selection of the identified crosscutting concern sorts.

Observe that the list of sorts is open-ended, i.e. new sorts can be added if their underlying relations cannot be covered by the existing sorts. It is also important to note that the concerns described by sorts are meaningful on their own, and can occur in more complex compositions, like a transaction management mechanism or an Observer design. In fact, published aspect refactorings are often at the same granularity as concern sorts, like introduction of roles, or advice for consistent behavior, although they are often presented in a larger context of a specific feature or design [18, 26].

The classification of crosscutting concerns based on sorts ensures a number of important properties for systematic aspect mining: first, the atomicity of the sorts ensures a consistent granularity level for the mining results; second, sorts describe the relation between concrete instances and the associated crosscutting functionality; third, sorts provide a common language for referring to typical crosscutting behavior.

Crosscutting concern sorts can be formalized using relational calculus as queries over source models that are extracted from a system's source code. The elements and relations relevant to these queries are shown in Figure 2. An in-depth treatment of the concrete queries would go beyond the scope of this paper but we refer the interested reader to [33].

5. Documenting Crosscutting Concerns

A concern models documents crosscutting concerns in a system and identifies the program elements that play a role in the implementation of these concerns. An empirical study by Robillard and Murphy [40] indicates that concern models are helpful when performing software change tasks. Concern modeling tools help software engineers to build concern models for a system. Examples of such tools include the Feature Exploration and Analysis Tool FEAT [40] and the Concern Manipulation Environment CME [20].

Existing concern modeling tools create rather low level models which are based on concrete source elements from the system to be documented. Some tools allow for user-defined queries to be attached to these models, although these are rather simple and unstructured queries. We propose to raise the level of abstraction in concern modeling using *crosscutting concern sorts* [31]. Sorts can be integrated into concern models by allowing sort queries as elements in the concern hierarchy.

5.1. Sort-based concern modeling in SoQueT

To support such sort-based concern modeling, we have built an Eclipse plug-in called SoQueT (SOrts QUEry Tool), which is freely available for download.[3] The tool allows one to describe crosscutting relations in a system based on querying its source code for instances of concern sorts [34]. These queries can be composed and stored to create persistent, sort-based documentation of concerns in existing code.

SoQueT assists the user in documenting and understanding crosscutting concerns in a system in the following way: First, the user defines a query for a specific sort based on its predefined template. The template guides the user in querying for elements that pertain to concrete sort instances and the user can restrict the query context, for example, by limiting it to a certain inheritance hierarchy.

[3] http://swerl.tudelft.nl/view/AMR/SoQueT

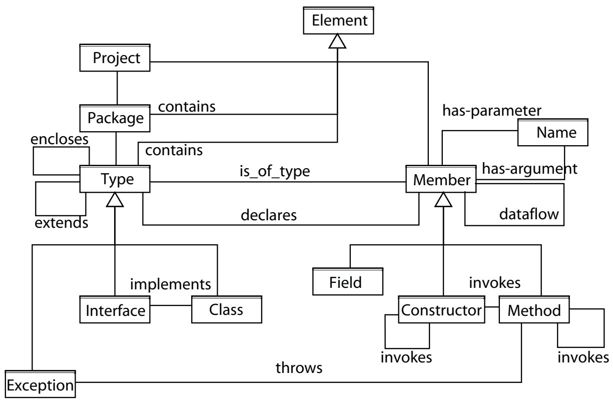

Figure 2. Meta-model relevant to sort queries

Next, the results of the query are displayed in the *Sort-search results* view. This view provides a number of options for navigating and investigating the results, like display and organization layouts, sorting and filtering options, links from the query results for source code inspection, etc.

Finally, a *Concern model* view allows one to organize sort instances in composite concerns and describe them by user defined names. The concern model is a tree that defines a view over the system that is complementary to Eclipse's standard *Package Explorer*. The system's sort instances are leaves in this tree and intermediate nodes describe composite concerns. Note that queries can be associated only with sort instances and not to a composite concern. A model can exist at various levels of abstraction and describe complex concerns, system features, or whole projects.

SoQueT introduces the concept of a *virtual interface* to define and describe a role whose definition is tangled within another type and cannot be identified by means of a standard (Java) interface. This mechanism allows the user to create a virtual interface by selecting in a graphical interface those members of the multi-role type, such as methods or fields, that are part of the role of interest.

5.2. Using SoQueT for software evolution

SoQueT can typically be used from two perspectives: (1) as a tool for consistently *creating* crosscutting concern documentation for a system, and (2) as a tool for *exploring* query-based crosscutting concern documentation that was defined earlier for the system under investigation. In the first scenario, the user has to be acquainted with the concerns to be documented. An example is a developer that wants to explicitly document some relations that are otherwise "hidden" by the object-based decomposition of a given system.

In the second scenario, aimed at supporting software evolution, the user explores a given system by loading (pre-existing) sort-based documentation of application into SoQueT in order to locate and better understand certain crosscutting concerns in the implementation. The tool allows for searching a concern model and displaying only those queries that are associated with a specific element. Such searches can be used to highlight relations and policies in the code that are relevant for a particular concern of interest.

The main challenge with documenting crosscutting concerns stems from the flexibility for defining query contexts. SoQueT could be improved by supporting set theoretic operations, such as the union of type hierarchies. In addition, defining contexts using pattern matching on names (e.g., all set* methods) is not implemented at the moment.

Adding new sort queries in the current version of the tool is fairly complex, as it relies on the "extension points" mechanism in Eclipse. We are exploring how we can prototype our queries using tools that support more direct source code queries. However, this support is still limited at the moment.

6. Systematic Aspect Mining

The increasing number of aspect mining techniques proposed in literature over the recent years draw inspiration from various domains and show a surprising variation in approaches. Structured research into aspect mining calls for a methodological approach for identifying potential mining opportunities and for comparing and combining existing approaches in order to evaluate, and improve their quality.

To addresses this challenge, we propose a common framework based on crosscutting concern sorts which allows to systematically define and consistently assess, compare and combine aspect mining techniques.

6.1. A framework for aspect mining

Our focus is on *generative* techniques: (semi-)automatic approaches that identify the program elements participating in crosscutting concerns by inspecting source code characteristics symptoms of crosscutting functionality.

Our framework defines the following requirements that ensure homogeneity in formulating the mining goals, presenting the results and evaluating their quality:

R1: Search-goal An aspect mining technique has to define the types crosscutting concerns that the analysis searches for. We call this the *search-goal* of the technique, and propose to define search-goals based on the classification of crosscutting concerns in sorts described earlier in section 4. Establishing a search-goal prevents that any fortuitous findings of the mining technique cloud its evaluation.

R2: Representation of candidate seeds An aspect mining technique has to define and describe the format for presenting mining results, i.e., the source code elements and relations that are generated as candidate seeds.

R3: Map candidate seeds on search-goal An aspect mining technique has to define how the candidate seeds map onto the targeted crosscutting concerns, i.e., the implementation idioms of the targeted concern sort(s). This mapping forms the relation between mining results and potentially associated concerns. Furthermore, it defines how one should understand and reason about the candidate seeds, and how they can be expanded into complete crosscutting concerns.

R4: Evaluation metrics An aspect mining technique has to define how the quality of identified candidate seeds are assessed (and possibly improved). We propose to use the following three metrics: (1) *Precision* is the percentage of correctly identified seeds in the whole set of candidate seeds reported by the technique. (2) *Absolute recall* counts the absolute number of identified concern seeds. We use this metric instead of standard recall because it is impossible to *objectively* determine the total number of concerns of a certain sort in a reasonably large system (as it requires interpretation of design decisions). (3) *Seed quality* characterizes *each* candidate seed by showing what percentage of (program) elements covered by the total mining result belong to the concern associated with the given candidate. This provides a measure for the effort required for reasoning about that candidate [9, 28]. *Average seed quality* can be used to bring this metric to the level of the complete technique. It indicates the level of confidence in the concern seeds identified by a particular technique.

6.2. Retrofitting existing techniques

To demonstrate the use and general applicability of the framework, we have retrofitted a number of known aspect mining techniques. The results are shown in Table 2.

Note that the descriptions presented here are, in general, more limiting than the original presentation by the authors since the table only reports the most representative sort as search goal and omit other findings. For example, fan-in analysis as described in [32], can identify a number of crosscutting concern sorts. The typical one is *consistent behavior* but fan-in analysis can also be used to find *role superimposition*: persistence across a set of classes can be implemented by methods that all invoke a particular (I/O) helper method, resulting in a high fan-in value for that helper method.

This "restriction" can be addressed by distinguishing a number of variants for the technique (or subtechniques) that each have their own search goal. This allows us to investigate and describe more clearly what part of a given technique exactly contributes to certain results or improvements.

6.3. Combining Mining Techniques

In addition to structuring the definition of aspect mining techniques, the framework presented in the previous section also supports more systematic investigation of *combinations* of mining techniques and their potential advantages. Below we will discuss for each of the evaluation metrics how combining mining techniques affects their value.

Improving precision Precision is measured by the percentage of correctly identified crosscutting concern seeds in the complete set of candidates reported by the mining technique. A straightforward combination of two aspect mining techniques that increases precision is achieved by intersecting their results (i.e., the set of candidates). This is basically pooling of evidence: if the same results reported by two or more different techniques, they are more likely to be valid. However, this can be done only when the techniques target the same crosscutting concern sorts, with compatible representations of the results. Two techniques that satisfy this condition are, for example, Fan-in and Grouped calls analysis. They can be combined by select those results of Fan-in analysis whose callees occur as callee in at least one of the Grouped calls candidates.

Improving absolute recall Absolute recall can be improved by considering the union of the results from different mining techniques. For techniques that target different

Technique	Search goal	Representation of candidate seeds	Mapping
Fan-in analysis [17, 32]	*Consistent behavior*	Call relation described by a callee and a set of callers.	The method with high fan-in (the callee) maps onto the crosscutting element; the callers of that method correspond to the elements being crosscut.
Grouped calls [31]	*Consistent behavior*	Set of (object,attribute) tuples, where the objects are the callers and the attributes are the grouped callees.	The attributes (i.e., the callees) map onto methods implementing crosscutting functionality, and the objects (i.e., the callers) match the crosscut elements.
Redirection finder [31]	*Redirection layer*	Redirection relation described by method pairs from two different classes, related by a one-to-one call relation.	The callers in the reported set match the methods executing the redirection, while their pair callees receive the redirection.
Aspectizable interfaces [49]	*Role superimposition*	Relation between (groups of) methods that belong (or can be abstracted) to interfaces, and the types that implement them.	The reported interface and its members map onto elements that crosscut the types that implement them.
Concepts in traces [9, 50]	*Role superimposition*	Set of methods in a type hierarchy defining the superimposed role, and the classes that implementing them.	The methods map onto the members of the superimposed type and cut across the classes that implement them.
Clone detection [7, 43]	*Consistent behavior*	Set of code fragments that are duplicated in multiple method bodies (and can be refactored by method extraction).	The methods containing the cloned code map onto the elements being crosscut; the (clone) method to be extracted maps onto the crosscutting element.
Execution patterns (dynamic [3] and static [25])	*Consistent behavior*	Relation between recurrent sequences of method invocations that match according to a given criterion	The recurrent sequence of method invocations maps onto the elements crosscutting the callers in the relation.
History-based mining [4]	*Consistent behavior*	Call relation between two sets of methods, where each method in the callers set calls all methods in the callees set.	The invoked methods map onto the elements crosscutting their reported callers.
Context flow mining [41]	*Context passing*	Call chain sequence annotated with the position of the parameter passed by each caller to its callee in the chain.	The caller in each invocation in the chain maps onto the method passing the context through the mapping parameter.

Table 2. Retrofitting existing aspect mining techniques to the framework.

concern sorts, the results will be complementary, and the number of seeds for the combination is the sum of the seeds identified by each technique.

Another way of improving the absolute recall is by being less selective, e.g. by lowering thresholds. However, this is likely to reduce precision. However, for Fan-in and Grouped calls analyses, precision can be restored by combining these two techniques with the same search-goal, and taking the intersection of their results. The lower thresholds allow for new candidates to be reported and the intersection filters the results so the precision does not drop significantly.

Improving seed quality Like precision, seed quality can be improved by combining techniques that target the same sort. For example, consider the intersection of Fan-in and Grouped calls analysis, selecting the common callees and the common callers of these callees. Since Grouped calls analysis is the most restrictive of both techniques, the number of callers for a callee is typically lower than for Fan-in analysis. Thus the combined result will have higher quality than the techniques alone (since false positives are filtered).

7. Migrating Crosscutting Concerns

The tangling and scattering that results from implementing crosscutting concerns in a software system using traditional object-oriented programming is a known challenge to program comprehension and software evolution.

One option to deal with the negative impact from crosscutting concerns on a system's comprehensibility and evolvability is to migrate the system to aspect-oriented programming (AOP) and transform the crosscutting concerns into aspects, a process known as *aspect refactoring*.

Despite significant efforts on various parts of the refactoring of crosscutting concerns from existing systems, to date there exists no compelling show-case for such a complete migration. One of the main causes for this void is the fact that there is no clearly defined, coherent migration strategy detailing the steps to be taken to perform this process.

7.1. Migration strategy

Successful migration requires a strategy comprising steps like identification of the concerns (i.e., aspect mining), description of the concerns to be refactored, and consistent refactoring solutions to be applied. Moreover, such a strategy requires *integrated* steps, so that aspect mining results, for example, can be consistently mapped onto concerns in code, and further refactored by general aspect solutions.

We propose an *integrated strategy for migrating crosscutting concerns to aspects* that builds upon the *crosscutting concern sorts* discussed earlier. The strategy integrates the following four steps [35]:

1. **Systematic aspect mining** This step consists of the idiom-driven identification of crosscutting concerns as described in the previous section. The mining targets a

Sort	Template aspect solution

Consistent Behavior	Refactor using pointcut and advice:
	`around(..) : callersContext(..){`
	`    invokeCB(..); //before`
	`    proceed();`
	`    // or after: invokeCB(..);`
	`}`

Table 3. Example refactoring template

specific crosscutting concern sorts by searching for its implementation idiom and is supported by our aspect mining tool FINT.[2]

2. **Sort-based concern exploration** The second step of our strategy, concern exploration, aims at expanding mining results (i.e., concern seeds) to the complete implementation of the associated concerns. In this step, we start from the discovered seeds and use the specific relation of the sort for the seed's concern to identify all the participants in the concern implementation. FINT integrates support for seed exploration and expansion to full concerns, such as detection of structural relations or similar call positions for the callers of a method.
3. **Sort-based concern modeling** The third step consists of sort-based concern modeling to document the concerns in the system. To ensure generally applicable solutions during migration, we need a systematic and coherent way of describing the concerns in a system with respect to its source code. This is achieved via the formalized queries for each of the concern sorts, which capture occurrences in terms of the sort's idiomatic relation between source code elements. This step is supported by our concern modeling tool SoQueT.[3]
4. **Sort-based migration** The final step consists of sort-based, idiom-driven approach to aspect refactoring which allows for consistent integration with the previous steps of our strategy. It is based on template aspect solutions for each of the concern sorts that can be instantiated to refactor sort occurrences. Like the previous steps, the refactoring addresses crosscutting behavior at the level of atomic concern sorts, providing an optimal trade-off between complexity of the refactoring and comprehensibility of the refactored element. An example refactoring template is shown in Table 3. Refactoring a sort instance starts from its query-based documentation in SoQueT. The query points to the elements participating in the concern, which can be used to configure the refactoring template. For example, the query for a *Consistent behavior* instance indicates the callers to be captured by a pointcut definition (the source context) and the action to be introduced by the advice (the target context). Other configurable elements, such as the type of advice to introduce the crosscutting call (e.g., before, after, after throwing, etc.), are decided at the refactoring time.

A detailed discussion of the refactoring templates for the remaining crosscutting concern sorts goes beyond the scope of this roadmap. We refer to [35] for more information.

7.2. Alternatives to migration

Especially in the context of large legacy systems, the risks involved with making changes and the uncertainty about desired future changes (e.g. towards yet another programming paradigm) make it worthwhile to explore alternatives to refactoring which increase separation of concerns without explicitly modifying the code.

One option in this context are so called *concern views*: representations of a part of the system that capture a concern like an aspect, but which are views on the underlying (unmodified) code that are generated by an advanced IDE. The disadvantage of such views is that they are static and cannot be used to make changes to the code, nor is it possible to reuse concerns in other applications.

However, aspects and concern views are only two extremes of a spectrum: somewhere in the middle of this spectrum we would find a approach like Fluid AOP [23] that supports *updatable concern views* where changes are propagated back to the appropriate locations in the underlying code via linked editing [51]. The use of domain-specific languages and generative techniques to modularize some concerns can also be viewed as points in this spectrum.

8. Concluding Remarks

8.1. Lessons Learned

The lessons learned from the work described in this paper can be summarized as follows:

- **Common language** We have proposed a fine-grained model for addressing crosscutting functionality in source code based on atomic crosscutting concern sorts. Such a model provides a consistent and coherent way of describing and referring to crosscutting concerns. As a result, these concern sorts become useful in program comprehension and concern documentation since they provide a common language.
- **Code based** Crosscutting concern sorts can be described as relations between sets of program elements which can be formalized as queries over source representations. This formalization enables the use of concern sorts (via their queries) as building blocks in semi-automated tools that support concern modeling based on program querying and analysis. This allows for systematic documentation of crosscutting concerns in source code.
- **Integration** Besides supporting comprehension and documentation, the systematic use of crosscutting concern sorts in defining aspect mining techniques ensures

homogeneity and compatibility in formulating the mining search-goals, presenting the results, and evaluating their quality. This enables detailed comparison and combination of such techniques which increases the quality of the results.

- **Structured investigation** Retrofitting aspect mining literature in terms of a common framework and language enables a structured investigation of the existing techniques and identifies potential opportunities for new research by locating areas that have been insufficiently addressed in the existing work.

8.2. Opportunities for Future Research

We distinguish a number of directions that have interesting opportunities for future research:

Aspect Mining As already hinted at in the previous section, there are a number of crosscutting concern sorts that are not yet targeted by aspect mining techniques. Table 2 indicates which sorts are already covered, and Table 1 indicates the specific idioms that should be targeted by a technique to address the gap.

Elaborate Evaluation Metrics Another direction to explore is elaboration on the metrics that are used as quality indicators in the framework. For example, it would be interesting to have an indication of *seed coverage*, a measure of how much of a concern's extent is covered by a particular seed. This metric complements the seed quality metric that relates the concern associated with the given seed to the total mining result. In addition to defining new metrics, the validity of the existing metrics could be empirically investigated on a larger collection of benchmark applications.

Support for Migration Besides aspect refactoring using template as discussed earlier, there are a number of additional questions that could be investigated to increase support for migration. One of these is pointcut generation: Given a list of affected source code locations or fragments, how can one find a good, or optimal, or generalized pointcut from this information? And what are the trade-offs between these variants for aspect migration and for further evolution of the system?

Alternatives to Migration As discussed in this paper, due to the inherent risks that are associated with changing legacy systems, it is worthwhile to pursue alternatives to refactoring that increase separation of concerns without explicitly modifying the code. Starting points for investigation include questions like: How can one automatically derive a view from source code that represents a cross-cutting concern? How can such views be made updatable? Given the spectrum of approaches discussed earlier, when is it better to use one approach over another?

Multi-Language Aspects A challenging direction of future research is the identification and migration of concerns that not only crosscut the dominant decomposition but also crosscut programming language boundaries. This is especially relevant in the context of dealing with legacy systems, many of which are know to be developed in a combination of (embedded) programming languages.

Acknowledgments The work described in this paper was carried out at Delft University of Technology in collaboration with Marius Marin and Arie van Deursen. The author wishes to thank Marius and Arie for the successful collaboration and fruitful discussions that helped to shape the ideas expressed here.

References

[1] The AspectJ Team. *The AspectJ Programming Guide*. Palo Alto Research Center, 2003. Version 1.2.

[2] C. Y. Baldwin and K. B. Clark. *Design Rules: The Power of Modularity Volume 1*. MIT Press, 1999. ISBN 0262024667.

[3] S. Breu and J. Krinke. Aspect mining using event traces. In *Proc. 19th IEEE Int'l Conf. on Automated Softw. Eng. (ASE)*, pages 310–315. IEEE, 2004.

[4] S. Breu and T. Zimmermann. Mining aspects from version history. In *Proc. 21st IEEE Int'l Conf. on Automated Softw. Eng. (ASE)*, pages 221–230. IEEE, 2006.

[5] L. C. Briand, J. W. Daly, and J. K. Wüst. A unified framework for coupling measurement in object-oriented systems. *IEEE Trans. Softw. Eng.*, 25(1):91–121, 1999.

[6] M. Bruntink. *Renovation of Idiomatic Crosscutting Concerns in Embedded Systems*. PhD thesis, Delft University of Technology, the Netherlands, March 2008.

[7] M. Bruntink, A. van Deursen, R. van Engelen, and T. Tourwé. On the use of clone detection for identifying crosscutting concern code. *IEEE Trans. Softw. Eng.*, 31(10):804–818, 2005.

[8] M. Bruntink, A. van Deursen, M. d'Hondt, and T. Tourwé. Simple crosscutting concerns are not so simple: analysing variability in large-scale idioms-based implementations. In *Proc. 6th Int'l Conf. on Aspect-Oriented Softw. Development (AOSD)*, pages 199–211. ACM, 2007.

[9] M. Ceccato, M. Marin, K. Mens, L. Moonen, P. Tonella, and T. Tourwé. Applying and combining three different aspect mining techniques. *Softw. Qual. J.*, 14(3):209–231, 2006.

[10] E. W. Dijkstra. On the role of scientific thought. In *Selected writings on Computing: A Personal Perspective*, pages 60–66. Springer, 1982.

[11] S. G. Eick, J. L. Steffen, and J. Eric E. Sumner. Seesoft-A Tool for Visualizing Line Oriented Software Statistics. *IEEE Trans. Softw. Eng.*, 18(11):957–968, 1992.

[12] L. Erlikh. Leveraging legacy system dollars for e-business. *IT Professional*, 2(3):17–23, 2000.

[13] R. E. Filman, T. Elrad, S. Clarke, and M. Akşit. *Aspect-Oriented Software Development*. Addison-Wesley, 2005.

[14] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1994.

[15] J. D. Gradecki and N. Lesiecki. *Mastering AspectJ - Aspect Oriented Programming in Java*. Wiley, 2003.

[16] W. G. Griswold, J. J. Yuan, and Y. Kato. Exploiting the map metaphor in a tool for software evolution. In *Proc. 23rd Int'l Conf. on Softw. Eng. (ICSE)*, pages 265–274. IEEE, 2001.

[17] K. Gybels and A. Kellens. Experiences with identifying aspects in smalltalk using unique methods. In *Proc. 1st Ws. on Linking Aspect Technology and Evolution (LATE)*, 2005.

[18] J. Hannemann and G. Kiczales. Design pattern implementation in Java and AspectJ. In *Proc. 17th Conf. on OO Progr., Syst., Lang., & Appl. (OOPSLA)*, pages 161–173. ACM, 2002.

[19] J. Hannemann and G. Kiczales. Overcoming the prevalent decomposition of legacy code. In *Ws. on Advanced Separation of Concerns*, 2001.

[20] W. Harrison, H. Ossher, S. M. S. Jr., and P. Tarr. Concern modeling in the Concern Manipulation Environment. Tech. Rep. RC23344, IBM T.J. Watson Research Center, 2004.

[21] B. Henderson-Sellers, L. L. Constantine, and I. M. Graham. Coupling and cohesion (towards a valid metrics suite for object-oriented analysis and design). *Object Oriented Systems*, 3:143–158, 1996.

[22] S. Henry and K. Kafura. Software structure metrics based on information flow. *IEEE Trans. Softw. Eng.*, 7(5):510–518, 1981.

[23] T. Hon and G. Kiczales. Fluid AOP join point models. In *Comp. 21st Conf. on OO Progr., Syst., Lang., & Appl. (OOPSLA)*, pages 712–713. ACM, 2006.

[24] G. Kiczales, J. Lamping, A. Menhdhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-oriented programming. In *Proc. 11th European Conf. on OO Prog. (ECOOP)*, pages 220–242. Springer-Verlag, 1997.

[25] J. Krinke. Mining control flow graphs for crosscutting concerns. In *Proc. of 13th Working Conf. on Reverse Engineering (WCRE)*, pages 334–342. IEEE, 2006.

[26] R. Laddad. *AspectJ in Action*. Manning, 2003.

[27] M. Marin. *An Integrated System to Manage Crosscutting Concerns in Source Code*. PhD thesis, Delft University of Technology, the Netherlands, Jan 2008.

[28] M. Marin. Reasoning about assessing and improving the seed quality of a generative aspect mining technique. In *Proc. 2nd Ws. on Linking Aspect Technology and Evolution (LATE)*, pages 23–27. CWI, Amsterdam, the Netherlands, 2006.

[29] M. Marin, A. van Deursen, and L. Moonen. Identifying aspects using fan-in analysis. In *Proc. 11th Working Conf. on Reverse Engineering (WCRE)*, pages 132–141. IEEE, 2004.

[30] M. Marin, L. Moonen, and A. van Deursen. A classification of crosscutting concerns. In *Proc. 21st IEEE Int'l Conf. on Softw. Maintenance (ICSM)*, pages 673–677. IEEE, 2005.

[31] M. Marin, L. Moonen, and A. van Deursen. A common framework for aspect mining based on crosscutting concern sorts. In *Proc. 13th Working Conf. on Reverse Engineering (WCRE)*, pages 29–38. IEEE, 2006.

[32] M. Marin, A. van Deursen, and L. Moonen. Identifying crosscutting concerns using fan-in analysis. *ACM Trans. Softw. Eng. Meth.*, 17(1):1–37, 2007.

[33] M. Marin, L. Moonen, and A. van Deursen. Documenting typical crosscutting concerns. In *Proc. 14th Working Conf. on Reverse Engineering (WCRE)*. IEEE, 2007.

[34] M. Marin, L. Moonen, and A. van Deursen. SoQueT: Query-based documentation of crosscutting concerns. In *Proc. 29th Int'l Conf. on Softw. Eng. (ICSE)*. IEEE, 2007.

[35] M. Marin, L. Moonen, and A. van Deursen. An integrated strategy to crosscutting concern migration and its aplication to JHotDraw. In *Proc. 7th IEEE Int'l Working Conf. on Source Code Analysis and Manip. (SCAM)*. IEEE, 2007.

[36] K. Mens, A. Kellens, and P. Tonella. A survey of automated code-level aspect mining techniques. *Trans. Aspect-Oriented Softw. Development*, 4:145–164, 2007.

[37] G. C. Murphy, W. G. Griswold, M. P. Robillard, J. Hannemann, and W. Leong. Design recommendations for concern elaboration tools. In [13], pages 507–530.

[38] D. L. Parnas. On the criteria to be used in decomposing systems into modules. *Comm. ACM*, 15(12):1053–1058, 1972.

[39] T. M. Pigoski. *Practical Software Maintenance: Best Practices for Managing Your Software Investment*. Wiley, 1996.

[40] M. P. Robillard and G. C. Murphy. Representing concerns in source code. *ACM Trans. Softw. Eng. Meth.*, 16(1):3, 2007.

[41] L. Seiter. Automatic mining of context passing in java programs. In *Proc. Ws. Towards Evolution of Aspect Mining (TEAM)*, pages 9–13. Delft University of Technlogy, 2006.

[42] D. Shepherd, E. Gibson, and L. Pollock. Design and evaluation of an automated aspect mining tool. In *Softw. Eng. Research and Practice*, pages 601–607. CSREA Press, 2004.

[43] D. Shepherd, J. Palm, L. Pollock, and M. Chu-Carroll. Timna: a framework for automatically combining aspect mining analyses. In *Proc. 20th Int'l Conf. on Automated Softw. Eng. (ASE)*, pages 184–193. ACM, 2005.

[44] S. Soares, E. Laureano, and P. Borba. Implementing distribution and persistence aspects with AspectJ. In *Proc. 17th Conf. on OO Progr., Syst., Lang., & Appl. (OOPSLA)*, pages 174–190. ACM, 2002.

[45] I. Sommerville. *Software Engineering*. 7th ed. Pearson, 2004.

[46] S. M. Sutton and I. Rouvellou. Concern modeling for aspect-oriented software development. pages 479–505.

[47] P. Tarr, H. Ossher, W. Harrison, and J. Stanley M. Sutton. N degrees of separation: multi-dimensional separation of concerns. In *Proc. 21st Int'l Conf. on Softw. Eng. (ICSE)*, pages 107–119. IEEE, 1999.

[48] P. Tarr, W. Harrison, and H. Ossher. Pervasive query support in the Concern Manipulation Environment. Technical Report RC23343, IBM TJ Watson Research Center, 2004.

[49] P. Tonella and M. Ceccato. Migrating interface implementation to aspect-oriented programming. In *Proc. 20th Int'l Conf. on Softw. Maintenance (ICSM)*, pages 220–229. IEEE, 2004.

[50] P. Tonella and M. Ceccato. Aspect mining through the formal concept analysis of execution traces. In *Proc. 11th Working Conf. on Reverse Engineering (WCRE)*. IEEE, 2004.

[51] M. Toomim, A. Begel, and S. L. Graham. Managing duplicated code with linked editing. In *Proc. Symp. on Visual Languages - Human Centric Computing (VLHCC)*, pages 173–180. IEEE, 2004.

[52] T. Tourwé and K. Mens. Mining aspectual views using formal concept analysis. In *Proc. 4th Int'l Ws. on Source Code Analysis and Manip. (SCAM)*. IEEE, September 2004.

[53] C. Zhang and H.-A. Jacobsen. PRISM is research in aspect mining. In *Comp. 19th Conf. on OO Progr., Syst., Lang., & Appl. (OOPSLA)*, pages 20–21. ACM, 2004.