Santibáñez *et al. Journal of the Brazilian Computer Society* (2015) 21:10
DOI 10.1186/s13173-015-0030-3

Journal of
the Brazilian Computer Society
a SpringerOpen Journal

RESEARCH **Open Access**

A combined approach for concern identification in KDM models

Daniel San Martín Santibáñez[1*], Rafael Serapilha Durelli[2] and Valter Vieira de Camargo[1]

Abstract

Background: Systems are considered legacy when their maintenance costs raise to unmanageable levels, but they still deliver valuable benefits for companies. One intrinsic problem of this kind of system is the presence of crosscutting concerns in their architecture, hindering its comprehension and evolution. Architecture-driven modernization (ADM) is the new generation of reengineering in which models are used as main artifacts during the whole process. Using ADM, it is possible to modernize legacy systems by remodularizing their concerns in a more modular shape. In this sense, the first step is the identification of source code elements that contribute to the implementation of those concerns, a process known as concern mining. Although there exist a number of concern mining approaches in the literature, none of them are devoted to ADM, leading individual groups to create their own ad hoc proprietary solutions. In this paper, we propose an approach called crosscutting-concern knowledge discovery meta-model (CCKDM) whose goal is to mine crosscutting concerns in ADM context. Our approach employs a combination of a concern library and a *K*-means clustering algorithm.

Methods: We have conducted an experimental study composed of two analyses. The first one aimed to identify the most suitable levenshtein values to apply the clustering algorithm. The second one aimed to check the recall and precision of our approach when compared to oracles and also to two other existing mining techniques (XScan and Timna) found in literature.

Results: The main result of this work is a combined mining approach for KDM that enables a concern-oriented modernization to be performed. As a secondary and more general result, this work shows that it is possible to adapt existing concern mining code-level approaches for being used in ADM processes and maintain the same level of precision and recall.

Conclusions: By using the approach herein presented, it was possible to conclude the following: (i) it is possible to automate the identification of crosscutting concerns in KDM models and (ii) the results are similar or equal to other approaches.

Keywords: ADM; KDM; Crosscutting concerns; Concern mining

Background

Software systems are considered legacy when their maintenance costs raise to undesirable levels, but they still provide valuable benefits for organizations. In general, these systems cannot be discarded because they encompass an extensive body of knowledge resulted from years of maintenance [1]. Since these systems still provide significant business value, a better alternative is to reengineer them, retaining the incorporated knowledge to keep their maintenance cost within acceptable levels. Reengineering is the examination and alteration of a subject system to reconstitute it in a new form and the subsequent implementation of the new form [2]. The first required activity to obtain a useful and high-level representations of a legacy systems is called reverse engineering (RE) [3], and it still remains a complex goal to achieve [4]. In 2003, the Object Management Group (OMG) created a group called Architecture-Driven Modernization Task Force (ADMTF). Its goal was to analyze and evolve conventional reengineering processes, formalizing them

*Correspondence: daniel.santibanez@dc.ufscar.br
[1] Departamento de Computação, Universidade Federal de São Carlos, Caixa Postal 676—13.565-905, São Carlos, Brazil
Full list of author information is available at the end of the article

and making them to be supported by models [5]. ADM advocates the conduction of reengineering processes following the model-driven architecture (MDA) principles [2, 5], i.e., all the main software artifacts considered along with the process are platform-independent model (PIM), platform-specific model (PSM) or computational-independent model (CIM).

According to OMG, the most important artifact provided by ADM is the knowledge discovery metamodel (KDM). By means of KDM, it is possible to represent all system's artifacts, such as configuration files, graphical user interface, architectural views, and source code. It is divided into four layers, representing physical and logical software assets at different abstraction levels. Each layer is further organized into packages that, by their turn, define a set of meta-model elements whose purpose is to represent a certain independent facet of knowledge related to legacy systems. One of the primary uses of KDM is in reverse engineering, in that a parser reads the source code of systems and generates KDM instances representing them. After that, minings, refactorings, and otimizations can be performed over this model, aiming to solve previously identified problems. The main idea behind KDM is that the community starts to create parsers and tools that work over KDM instances; thus, every tool that takes KDM as input can be considered platform and language-independent. For instance, a concern mining technique for KDM can be used for mining concerns in systems that are implemented in different languages. Aspect mining (or concern mining) is another important research field very related to reverse engineering [6, 7]. The main purpose is to identify source code elements that contribute to the implementation of crosscutting concerns (CCs) [8], such as persistence and cryptography. After the identification, the next step is the remodularization of the system using aspect-oriented programming [9]. As the presence of CCs is an inherent problem of legacy systems, we claim that aspect-oriented modernization (or modularity-oriented modernizations) is an import kind of modernization to be performed [10].

However, although ADM/KDM had been created to support modernization of legacy systems, to the best of our knowledge, there is no research that investigates concern mining in KDM models. Although there exist some modernization works in the literature that cover all modernization stages [4, 11, 12], none of them detail a mining technique for KDM. In order to overcome this limitation, in this paper, we present an approach, named CCKDM, for mining concerns in the KDM models. Our mining technique employs a combination of a concern library and a string clustering algorithm. The required input of the approach is a KDM instance representing the legacy system, and the output is the same KDM with the concerns clearly annotated. As our mining technique acts over the KDM instance, it is language-independent [13–15]. To support our approach, we have implemented an Eclipse plug-in that assists the modernization engineer along the process. To evaluate our approach, we conducted three analyses using two systems: Health Watcher v10 [16] and PetStore v1.3.2 [17]. The first analysis was focused on the identification of which *levenshtein* values (for the clustering algorithm) provide the best results for the mining process. The second one was intended to characterize and show the recall and precision values for our combined approach using oracles found in the literature [18, 19]. The third analysis concentrated on comparing the results of our approach with two other existing mining techniques found on literature: XScan [20] and Timna [21].

The most important contributions of this paper are (i) the proposal of a concern mining approach for KDM, what may encourage other groups to start researching on concern/aspect-oriented modernizations or modularity-oriented modernizations; (ii) the demonstration that existing mining techniques (library-based and clustering), primarily non-intended to work on models, can be adapted to KDM-based mining and the same level of precision and recall metrics can be maintained or even improved; and (iii) pointing out that KDM must be annotated in some way to clearly identify parts that contribute to concerns.

This article is structured as follows: The "Background" section presents the necessary background to understand this article. In the "Methods" section, the proposed approach for identification of concern by means of KDM and our tool are presented. In the "Results and discussion" section, the evaluation of our approach and some related works are presented. Finally, the conclusions and future works are drawn in the "Conclusions" section.

Architecture-driven modernization (ADM)

In 2003, OMG initiated efforts to standardize the process of modernization of legacy systems using models by means of the ADMTF [22]. The goal of ADM has been the revitalization of existing applications by adding or improving functionalities, using existing OMG modeling standards and also considering MDA principles. In other words, OMG took the initiative to standardize reengineering processes.According to OMG, ADM does not replace reengineering processes but improves them through the use of MDA.

The basic process flow of modernization has three phases: reverse engineering, restructuring, and forward engineering, as can be seen in Fig. 1. In the reverse reengineering, the knowledge is extracted and a platform-specific model (PSM) is generated. The PSM serves as the basis for the generation of a platform-independent model (PIM) called KDM. Then, this PIM can serve as

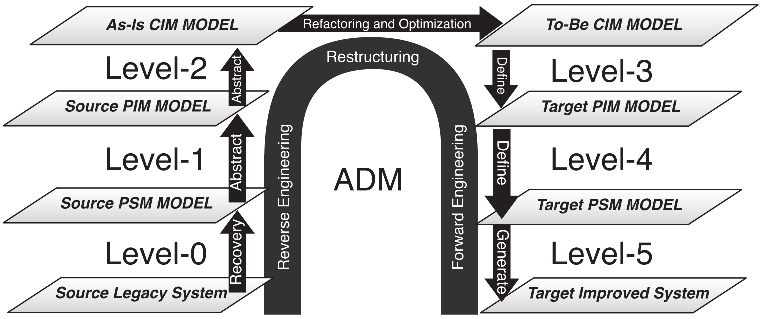


Fig. 1 Horseshoe modernization model (OMG Group [24])

basis for the creation of a computing-independent model (CIM) [23].

ADM solves the formalization problem by representing all the artifacts involved in the reengineering process as models [5]. So, ADM treats all models homogeneously, allowing the creation of model-to-model transformation between them. Although ADM follows all the principles of MDE, ADM does not preclude source-to-source migrations (where appropriate) but encourages user organizations to consider modernization from an analysis and design perspective.

Knowledge discovery meta-model (KDM)

KDM is a meta-model for representing existing software systems, its elements, associations, and operational environments. One of the most important characteristics of it is its completeness and broadness, since it is able to represent low levels details, like source regions, methods, and properties as well as higher level ones, such as architecture, business rules, and events. Besides, another important characteristic is that all of these low- and high-level characteristics are linked into the same meta-model, allowing traceability among them.

In 2011, KDM becomes an ISO/IEC standard under the number 19506. Moreover, KDM is defined via meta-object facility (MOF) and determines the interchange format via XML metadata interchange (XMI) by applying the standard MOF to XMI mapping to the KDM MOF model. The interchange format defined by KDM is called the KDM XMI schema. The KDM XMI schema is provided as the normative part of this specification. According to OMG and Perez-Castillo ([24, 25]), the goal of KDM is to facilitate the exchange of metadata across different applications, languages, platforms, and environments, more specifically

- represents artifacts of legacy software as entities, relationships, and attributes
- includes external artifacts with which software artifacts interact
- supports a variety of platforms and languages
- consists of a platform and language independent core, with extensions where needed
- defines a unified terminology for legacy software artifacts
- describes the physical and logical structure of legacy systems
- can aggregate or modify, i.e., refactor, the legacy system
- facilitates tracing artifacts from logical structure back to physical structure
- represents behavioral artifacts down to, but not below, the procedural level.

As stated before, the KDM is an OMG discovery meta-model specification, and at the present time, it is being adopted as ISO/IEC 19506 by the International Standards Organization for representing information related to existing software systems. Moreover, KDM is defined via meta-object facility (MOF). KDM determines the interchange format via the XML metadata interchange (XMI) by applying the standard MOF to XMI mapping to the KDM MOF model. The interchange format defined by KDM is called the KDM XMI schema. The KDM XMI schema is provided as the normative part of this specification.

The KDM meta-model consists of four abstraction layers: (*i*) infrastructure layer, (*ii*) program elements layer, (*iii*) runtime resource layer, and (*iv*) abstractions layer. Each layer is further organized into packages, as can be seen in Fig. 2. Each package defines a set of meta-model elements whose purpose is to represent a certain independent facet of knowledge related to existing software systems.

The infrastructure layer consists of the following three packages: core, "kdm", and source. Core package and the package named "kdm" do not describe separate KDM models. Instead, these packages define common meta-model elements that constitute the infrastructure for other packages. The source package defines the inventory model, which enumerates the artifacts of the existing software system and defines the mechanism of traceability

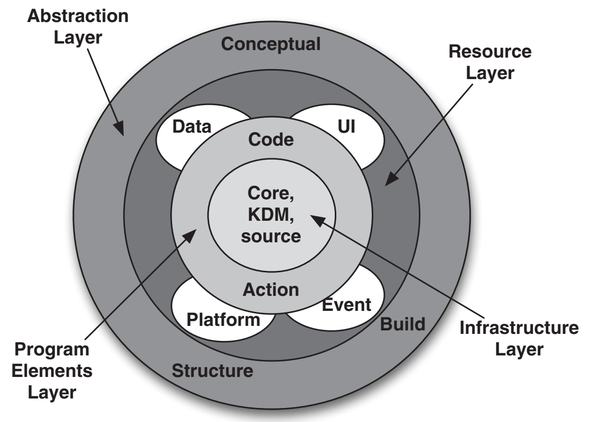


Fig. 2 Layers, packages, and separation of concerns in KDM (adapted from [24])

links between the KDM elements and their original representation in the "source code" of the existing software system.

The program elements layer consists of the code and action packages. These packages collectively define the code model that represents the implementation level assets of the existing software system, determined by the programming languages used in the developments of the existing software system. The code package focuses on the named items from the "source code" and several basic structural relationships between them. The action package focuses on behavior descriptions and control- and data-flow relationships determined by them. The action package is extended by other KDM packages to describe higher-level behavior abstractions that are key elements of knowledge about existing software systems. Note that our mining approach is strongly based on both code and action packages. Therefore, it is important to dig into these packages; the "KDM code package" section shows more details on these packages as well as some of the metaclasses that are used by our approach.

The runtime resources layer consists of the following four packages: platform, UI, event, and data. The abstractions layer consists of the following three packages: structure, conceptual, and build.

KDM code package

In a given KDM instance, each instance of the code meta-model element represents some programming language construct, determined by the programming language of the existing software system. Each instance of a code meta-model element corresponds to a certain region of the source code in one of the artifacts of the existing software system. In addition, the code package consists of 24 classes and contains all the abstract elements for modeling the static structure of the source code. However, we are particularly interested in *MethodUnit* and *StorableUnit* because they describe source code elements that usually contribute to the implementation of crosscutting concerns. In Fig. 3, a chunk of the code package is depicted. It worth to notice that the metaclasses used in this project are highlighted.

As can be seen, the root class is *ComputationalObject* which has two classes, i.e., *DataElement* and *ControlElement*. *DataElement* is a generic modeling element that defines the common properties of several concrete classes that represent the named data items of existing software systems, for example, global and local variables, record files, and formal parameters. *DataElement* has five classes—*StorableUnit*, *IndexUnit*, *ItemUnit*, *ParameterUnit*, and *MemberUnit*. *StorableUnit* is a concrete class that represents variables of the existing system. *IndexUnit* class is a concrete class that represents an index of an array datatype. Instances of *ItemUnit* class are endpoints of KDM data relations which describes access

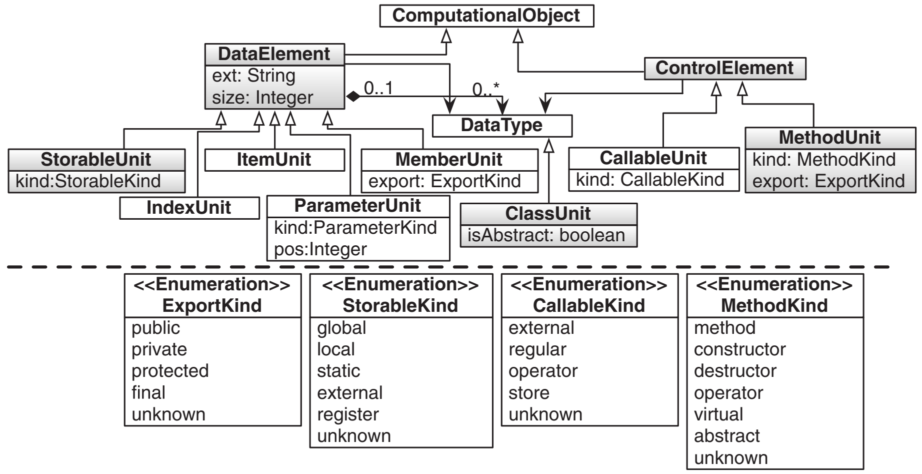


Fig. 3 Chunk of the code package (OMG group [24])

to complex datatypes. *ParameterUnit* class is a concrete class that represents a formal parameter, for example, a formal parameter of a procedure. *MemberUnit* class is a concrete class that represents a member of a class type. Finally, the latter, *ControlElement* is a class that contains two classes—*MethodUnit* and *CallableUnit*. *MethodUnit* element represents member functions owned by a *ClassUnit*, including user-defined operators, constructors, and destructors. The *CallableUnit* represents a basic standalone element that can be called, such as a procedure or a function.

Although KDM Code package can represent several code structures related to structured and object-oriented programming, there are no metaclasses for representing aspect-oriented concepts or crosscutting concerns. However, as crosscutting concerns is a reality in legacy systems and aspect-oriented programming is the most mature technique for solving this problem, it is necessary to represent those concepts into KDM somehow. Without this representation, it is very hard to conduct aspect-oriented modernizations.

OMG recognizes the importance of customizing KDM to specific needs by providing metaclasses for the creation of profiles, which are light-weight extensions that rely on stereotypes and tagged values. Another way of extending KDM is by means of a heavy weight extension, by creating/changing existing metaclasses. Both alternatives have pros and cons. The extension mechanism provided by KDM for creating profiles is too simplistic, making difficult for someone to use it, but existing tools will keep compatible with the KDM extension. The heavy weight alternative is better in terms of use, consistency and correctness; however, existing tools may require adjustments to work with the new KDM extension. In this paper, we have opted for a different strategy. We have added a tag called "concern" just in the model instances and not in the KDM meta-model. So, we have not modified KDM; therefore, our mining technique relies on the existence of this tag in model instances.

Mining of crosscutting concerns

Concern mining is a technique that can automatically suggest sets of related code fragments that contribute to the implementation of a concern. In the literature, we can find several types of techniques to find concerns into the source code [26]. Some of them are focused in the code structure while others in the code behavior of a system, so it is recommended the use of combined techniques in order to achieve better results in terms of precision and recall [27]. They generally use techniques from data mining and data analysis like formal concept analysis and clustering and other ones such as program slicing, clone detection, pattern matching, natural language processing, dynamic analysis, and so on [8].

Once the concerns have been found, software engineers could perform some kind of modularization into the source code to achieve a better understanding of the system. Concern mining techniques generate concern seeds, i.e., sets of related code entities that likely contribute to the implementation (set of instances) of a concern. These techniques focus especially on crosscutting concerns, as modular concerns can be easily identified manually. Depending on the intended usage of a technique, it can be applied as frequently as once per release (for documentation) up until once per feature request or even bug report [28].

In our work, we use the clustering-based technique which aims at identifying groups of methods or statements related to the crosscutting concerns guided by a distance measure [28–30]. Clustering is a division of data into groups of similar objects where each of these subsets (groups, clusters) consists of objects that are similar between themselves and dissimilar to objects of other groups. In particular, we use the K-means clustering which partitions a collection of n objects into k distinct and non-empty clusters where data being grouped in an exclusive way, that is, each object will belong to a single cluster. Firstly, the procedure classify a given data through a certain number of clusters K, fixed a-priori. Secondly, the algorithm starts with k initial objects called centroids, then iteratively recalculates the clusters and centroids where each object is assigned to the closest cluster—centroid until convergence is achieved.

Methods

This section presents CCKDM, which is our approach for mining concerns in KDM instances. The most fundamental goal of this work is to present a manner of mining concerns in this meta-model, therefore, highlighting two important points. The first one is how to associate model elements to specific concerns and the second is how to annotate these model elements so the relationship becomes explicit. Figure 4 depicts the overall process, which is divided into four steps denoted by its corresponding letters and titles at the left side. More details about each step is provided in the next sections.

The step *A*, recovery of application structure, recovers from the legacy KDM, source code information about the system under analysis and persists them in a repository. This was divided into three steps to simplify the queries, making them clearer and easier to maintain and evolve.

The step *B*, API-based identification, uses an API-based library to start the mining process. Here, we compare the concerns terms available in the library with the source code information previously recovered from the legacy system. Based on this comparison, it is possible to identify, match, and annotate the first set of model elements that contribute to the implementation of a given concern.

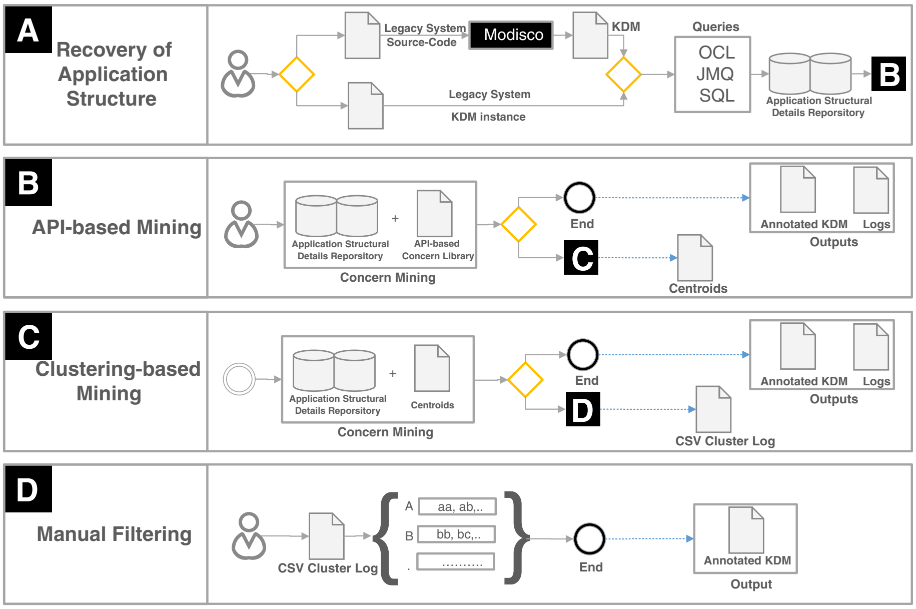


Fig. 4 The concern mining process

Notice that the API-based library is the strategy we are using to match keywords (concern terms) to the concerns.

The step *C*, clustering-based mining, aims to apply the clustering algorithm to identify and annotate other model element instances that contribute to the implementation of the same given concern. So, the previously identified set is expanded.

In step *D*, manual filtering, a manual filtering is performed to filter out erroneously identified elements.

Recovery of application structure

The step *A* starts when the user provides the source code of the legacy system or a KDM file that represents it. If the source code is the input, then it is firstly converted to a KDM instance. This is performed by means of *Modisco*™ [31], which is a parser which transforms the source code into a KDM instance.

After that, many queries are performed over the KDM model to recover the application structure, which is represented by some instances of the KDM metaclasses. In our case, we recover all instances of StorableUnits (variables) and MethodUnits (methods) metaclasses. We have chosen these two source code elements because they are most suitable to find crosscutting concerns [32].

In order to conduct this step, three activities must be done. The first one is recovering all packages (package metaclass), all classes and interfaces (ClassUnit and InterfaceUnit metaclasses), all methods (MethodUnit metaclass), and all properties (StorableUnit metaclass) of the system. We have decided to use object constraint language (OCL) [33] because it is well-known and well-documented. For example, if we apply the following instruction, *(object.allInstances())* we can get all the instances of an object belonging to a KDM model.

It is important to know how these elements are arranged into the meta-model so that we can properly identify and tag them during the annotation activity. Thus, the second activity aims at recuperating method calls (calls metaclass), method containers (BlockUnit metaclass), container classes (CodeModel metaclass), method signatures (signatures metaclass), method types, and property types (ParameterUnit metaclass). In this second step, we used Java model query (JMQ) which allows to navigate through models using Java. The third and last activity is to persist these elements into a relational database called application structural details repository (ASDR). The entity-relationship (ER) model can be seen in Fig. 5.

Module persists class and interface names. *Method* persists method names, signature, and return types. *Import* persists application imports. *Package* persists application packages. *ModuleProperty* persists class properties and its type. *MethodProperty* persists variables belonging to methods.

Notice that this entity-relationship model can be seen as a relational subset of the KDM. Considering our mining purposes, we have eliminated everything that was not of our interest, leaving only the information that really matter. It is important to highlight that although we have not detailed/documented the steps of creating ER models from KDM, we believe this is a canonical step in other processes like that. In general, the creation of these ER models will be guided by the intention of use. For example, each KDM metaclass that has some relation with crosscutting concerns must be turned into a relational database.

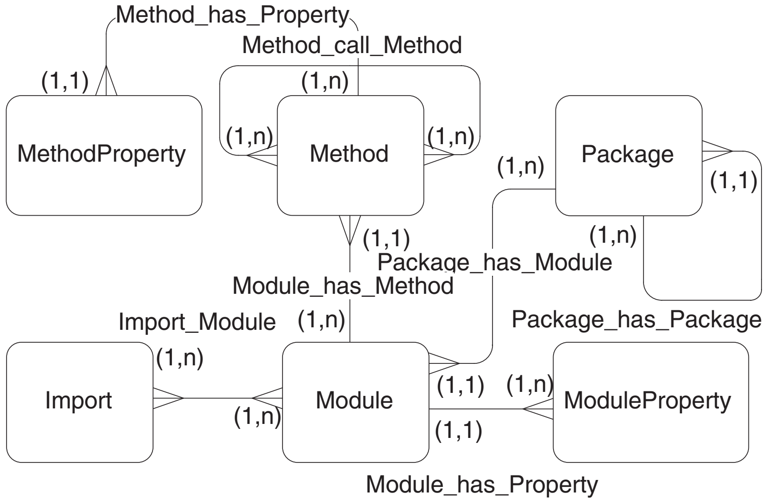


Fig. 5 ASDR entity-relationship model

Although OCL is a powerful language to perform queries over MOF models, it has two main problems: *(i)* it does not scale well when building complex queries [34] and *(ii)* changing complex queries may induce to developers in making mistakes. It is important to note that new queries must be created if we want to query other packages of the KDM instance to search for crosscutting concerns. Logically, this will add new entry points of maintenance.

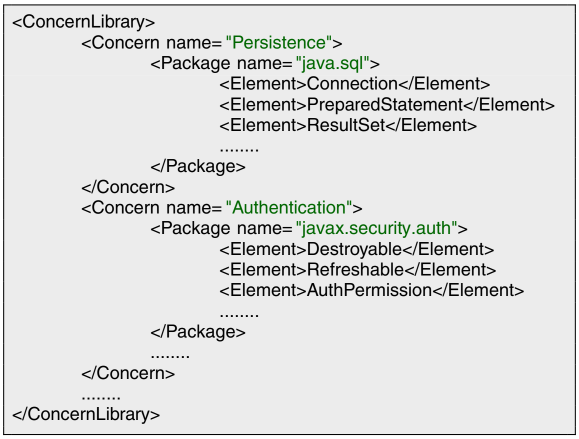
Whenever one wants to persist the code elements of a KDM instance, the use of a database seems to be the most natural approach. The entire data model in conjunction with the KDM file can be ported and reused without to generate the KDM file and search for code structures again if no modifications are performed. Also, the use of standard query language (SQL) queries also must facilitate future maintenance because this technology is widely used.

API-based mining

The step *B* is triggered by the user, and the process operates over an API-based library and the ASDR. In our approach, the concern library is devoted to store only concern-related APIs, that is, APIs which contribute to the implementation of specific concerns—mostly non-functional concerns. For example, the *java.sql* API provide support for “persistence” while *java.security.auth* supports the implementation of the “authentication” concern. APIs are extensively used by developers, and we believe that they can provide a good starting point for concern identification. In the future, we expect to modify our approach to attend functional concerns, such as features of a software product line [35].

In our library, a concern definition is composed of a concern name and one or more well-known class names which contribute to the implementation of a concern. Figure 6 shows part of a concern library, implemented by means of XML file. In this case, the file contains two concern definitions, persistence and authentication. The concern persistence is implemented by the Java package named “java.sql” with several elements which corresponds to Java classes. In the same way, the concern authentication is implemented by the Java package named “javax.security.auth” with its elements.

The concern library is used jointly with the ASDR by means predefined SQL queries. Listing 1 shows the predefined SQL query to identify methods which may implement a concern. To perform the identification, CCKDM queries *Module*, *Method*, *Import* and *Element* tables. These tables belong to our database model, and the table *Elements* is a temporary table populated with information of the concern library. This information depends on which



```
<ConcernLibrary>
        <Concern name="Persistence">
                <Package name="java.sql">
                        <Element>Connection</Element>
                        <Element>PreparedStatement</Element>
                        <Element>ResultSet</Element>
                        ........
                </Package>
        </Concern>
        <Concern name="Authentication">
                <Package name="javax.security.auth">
                        <Element>Destroyable</Element>
                        <Element>Refreshable</Element>
                        <Element>AuthPermission</Element>
                        ........
                </Package>
                ........
        </Concern>
        ........
</ConcernLibrary>
```

Fig. 6 Example of concern library

concern is being identified at that moment. The result is a set of elements (method and property names) related to some of the classes of the concern library. These elements will be the initial model seeds of our approach, and the initial information (centroids) for our clustering algorithm are detailed in the next section. If the user does not want to perform the step *C*, then the mining process ends and the outputs are log files and the annotated KDM with the identified concern names.

```
SELECT MO.NAME AS MODULE_NAME,
       M.NAME AS METHOD_NAME,
       M.TYPE AS METHOD_TYPE,
       M.RETURNTYPE AS METHOD_RETURN
FROM IMPORT I INNER JOIN
     IMPORT_MODULE MI ON
     I.IDIMPORT = MI.IMPORT_IDIMPORT
             INNER JOIN MODULE MO ON
     MI.MODULE_IDMODULE = MO.IDMODULE
             INNER JOIN METHOD M ON
     M.MODULE_IDMODULE = MO.IDMODULE
WHERE I.NAME LIKE ? AND
      M.RETURNTYPE IN
      (SELECT ELEMENT FROM ELEMENTS)
```

Listing 1 Methods identification implementing a concern

Clustering-based mining

The step *C* performs a string clustering by means of a *K*-means algorithm [36] which uses the Levenshtein distance to cluster the strings depending on their similarities [37]. This step complements the previous one because it aims at identifying code elements which were not identified in the previous step but have similar identifiers (names). Thus, we suppose they implement same concerns.

The strings identified in step *B*, which already belong to a certain concern, are the *K* centroids for our clustering algorithm (i.e., the strings where others strings will be clustered). Algorithm 1 presents the implemented clustering function in a simplified way. It receives as input parameters a concern name *c* and a threshold value *v* and returns a matrix($n \times m$).

Lines 2–3 of the algorithm creates two lists which are the initial centroids. In lines 4–5, the lists are joined and duplicates values are removed. Line 8 calculates the Levenshtein distance between all centroids and all the methods and properties of the system. In line 11, Levenshtein values are compared with a threshold *v* which is given by a user. If the threshold value is closer to 1.0, then the algorithm will cluster a small quantity of strings but with more similarities; on the other hand, if the threshold value is closer to 0.0, then the algorithm will cluster a big quantity of strings but with more dissimilarities. Levenshtein values are stored into a vector, and then this vector is added into a matrix with *K* rows which represents the number of centroids and *T* columns which represents the number of non-duplicate methods and property names of the system. In line 20, the matrix is returned and the most high value of a given column indicates to which centroid the string must be clustered.

Algorithm 1 Modified *K*-means cluster.

```
Require: Concern Name, Levenshtein Threshold
Ensure: Matrix
 1: function CLUSTERING($c$, $t$)
 2:     methodCentroid[] = $c$.getMethodByLibrary($c$);
 3:     propertyCentroid[] = $c$.getPropertyByLibrary($c$);
 4:     $P$ = join(methodCentroid[],propertyCentroid[]);
 5:     eliminateDuplicates($P$);
 6:     matrix($K$, $T$);
 7:     for each $p$ into $P$ do
 8:         values[] = levenshtein($p$ , propertyMethodList[]);
 9:         $i = 0$;
10:         for each $v$ in values[] do
11:             if $v \geq t$ then
12:                 vector[$i$] = $v$;
13:             else
14:                 vector[$i$] = 0;
15:             end if
16:             $i = i + 1$;
17:         end for
18:         matrix.addRow(vector[]);
19:     end for
20:     return matrix;
21: end function
```

Annotating concerns

As soon as the previous mining steps are over, our technique performs the annotation activity. Note that the term “annotation” used in this work is referring to the action of tagging the model element with a simple word and is not related with the term used to tag classes or models using an “@” symbol.

This activity is carried out by adding a new tag which is an attribute called “concern” in KDM instances (XMI files). This tag can be attached in methods and attributes, and its value is the name of a concern recuperated from the concern library.

It is important to take into account that KDM model is saved as an XMI file which follows the XML guidelines, so it is necessary to use the XML query language. In our case, we used XQuery, a powerful query language that provides the means to extract and manipulate data from XML documents or any data source that can be viewed as XML. Listing 2 shows an extract of the XQuery code in charge of inserting an annotation applied to a method with name *$method*. The name of the attribute is “concern”, and the value of the attribute is given by the variablel *${concern}*.

```
.....
    [@xsi:type="code:MethodUnit" and
           @name=$method and empty(@concern)]
return insert node attribute concern {$concern} as last
    into $a
```

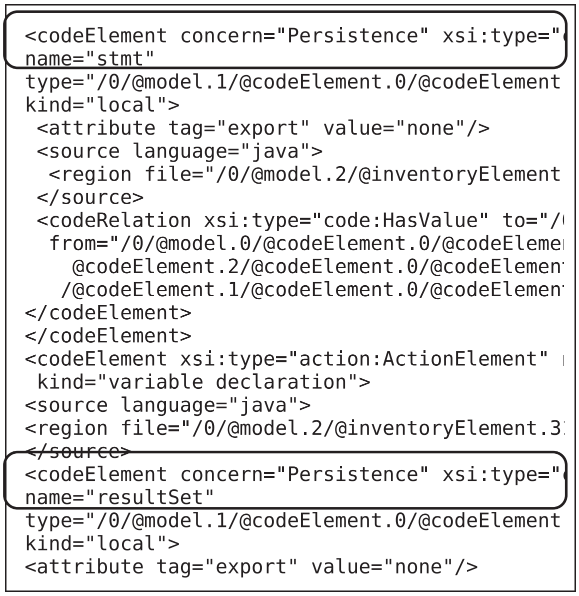
Listing 2 Insert an annotation into a KDM instance

Figure 7 shows part of a KDM instance of the HealthWatcher system. It has two annotations that make evident the presence of persistence concern affecting some model elements. The first annotation, on the top of the figure, shows that the *stmt* attribute contributes to the implementation of the persistence concern. The type of this code element is *PreparedStatement* which is a class belonging to the Java persistence API. In the bottom of the figure, there is another annotation for the code element *resultSet*. In this case, this code element is a class of type *ResultSet* which also belongs to the Java persistence API.

Manual filtering

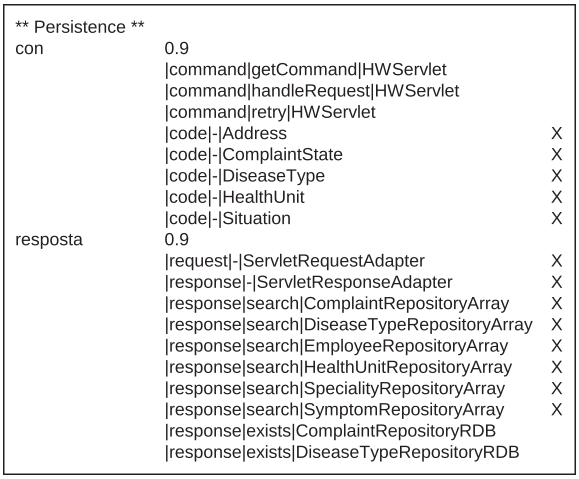
In step *D*, the goal is to show the preliminar result to the user and let somebody to perform a manual filtering. The visualization of the result is supported by our CCKDM tool, which is described in the next section. This filtering step, also supported by our tool, gives the user the possibility of excluding erroneously identified elements. The result is presented in a comma-separated value (CSV) file containing all the strings identified by our cluster algorithm. To exclude an annotated element, the user must tag with an "X" at the end of each line of the CSV file.

Figure 8 shows part of a CSV file, generated by CCKDM. The file was generated after executing the step *C* for identifying persistence into HealthWatcher. The first column exhibits two centroids, *con* and *Resposta*. Variable *con* clustered 9 elements and *Resposta* 10 elements. Each clustered element is composed of three or two strings, depending whether it is a method, an attribute of a class, or a variable. For example, the line |command|getCommand|HWServlet of the Fig. 8 indicates that the variable called command belonging to the method getCommand, and class HWServlet may implement the same concern that the *con* variable implements. The line |code|-|Address indicates that the attribute code of the class Address may implement a concern. Finally, another possibility of line which is not shown in the figure could be |-|method|class where the method "method" is the clustered element. For all the elements tagged with a "X", their respective annotations into the KDM file will be removed.



```
<codeElement concern="Persistence" xsi:type="
name="stmt"
type="/0/@model.1/@codeElement.0/@codeElement
kind="local">
 <attribute tag="export" value="none"/>
 <source language="java">
  <region file="/0/@model.2/@inventoryElement
 </source>
 <codeRelation xsi:type="code:HasValue" to="/
  from="/0/@model.0/@codeElement.0/@codeEleme
    @codeElement.2/@codeElement.0/@codeElemen
   /@codeElement.1/@codeElement.0/@codeElemen
</codeElement>
</codeElement>
<codeElement xsi:type="action:ActionElement"
 kind="variable declaration">
<source language="java">
<region file="/0/@model.2/@inventoryElement.3
</source>
<codeElement concern="Persistence" xsi:type="
name="resultSet"
type="/0/@model.1/@codeElement.0/@codeElement
kind="local">
<attribute tag="export" value="none"/>
```

Fig. 7 Annotated concerns into KDM instance



```
** Persistence **
con            0.9
               |command|getCommand|HWServlet
               |command|handleRequest|HWServlet
               |command|retry|HWServlet
               |code|-|Address                              X
               |code|-|ComplaintState                       X
               |code|-|DiseaseType                          X
               |code|-|HealthUnit                           X
               |code|-|Situation                            X
resposta       0.9
               |request|-|ServletRequestAdapter             X
               |response|-|ServletResponseAdapter           X
               |response|search|ComplaintRepositoryArray    X
               |response|search|DiseaseTypeRepositoryArray  X
               |response|search|EmployeeRepositoryArray     X
               |response|search|HealthUnitRepositoryArray   X
               |response|search|SpecialityRepositoryArray   X
               |response|search|SymptomRepositoryArray      X
               |response|exists|ComplaintRepositoryRDB
               |response|exists|DiseaseTypeRepositoryRDB
```

Fig. 8 CSV file

Listing 3 presents an extract of the XQuery code to delete an annotation. In this case, the XQuery code deletes the attribute "concern" of a code element where its type is a MethodUnit.

```
.....
    [@xsi:type="code:MethodUnit" and
          @name=$method and
          not(empty(@concern))]/@concern
return delete node $a
```

Listing 3 Delete an annotation into a KDM model instance

It is important to highlight that in our annotation strategy, we have not changed the KDM meta-model, neither creating a profile nor creating a new metaclass. We do not see any problem with that because we consider our model instances as an internal and intermediary representations, that is, it can be discarded or ignored as long as they had been used. That means whether one wants to proceed in the modernization process, someone needs to make transformations on this model, for example, generating a new KDM instance from other extended KDMs.

CCKDM tool

Overview

In this section, we present the developed Eclipse plug-in named CCKDM (sourceforge.net/projects/cckdm/) which implements our approach. Figure 9 shows the main window that is presented to users after the step *A* is executed, and Fig. 10 shows the API-based library management.

In Fig. 9, we can identify three sections denoted by the letters *A*, *B*, and *C*. Section *A* exhibits some information related with the size of the project in terms of number of classes, interfaces, methods, and properties. It also exhibits method fan-in metric providing evidence of possible crosscutting concerns by means of counting the most used methods.

Section *B* exhibits concern definitions available in the library, but users have the possibility to add or delete concerns by means our library manager. Users must choose at least one definition to start the mining process.

Section *C* exhibits options related with the clustering step. Users may activate or deactivate this step, filter getter and setter methods, set the Levenshtein value, and perform manual control for annotated concerns identified with our clustering algorithm.

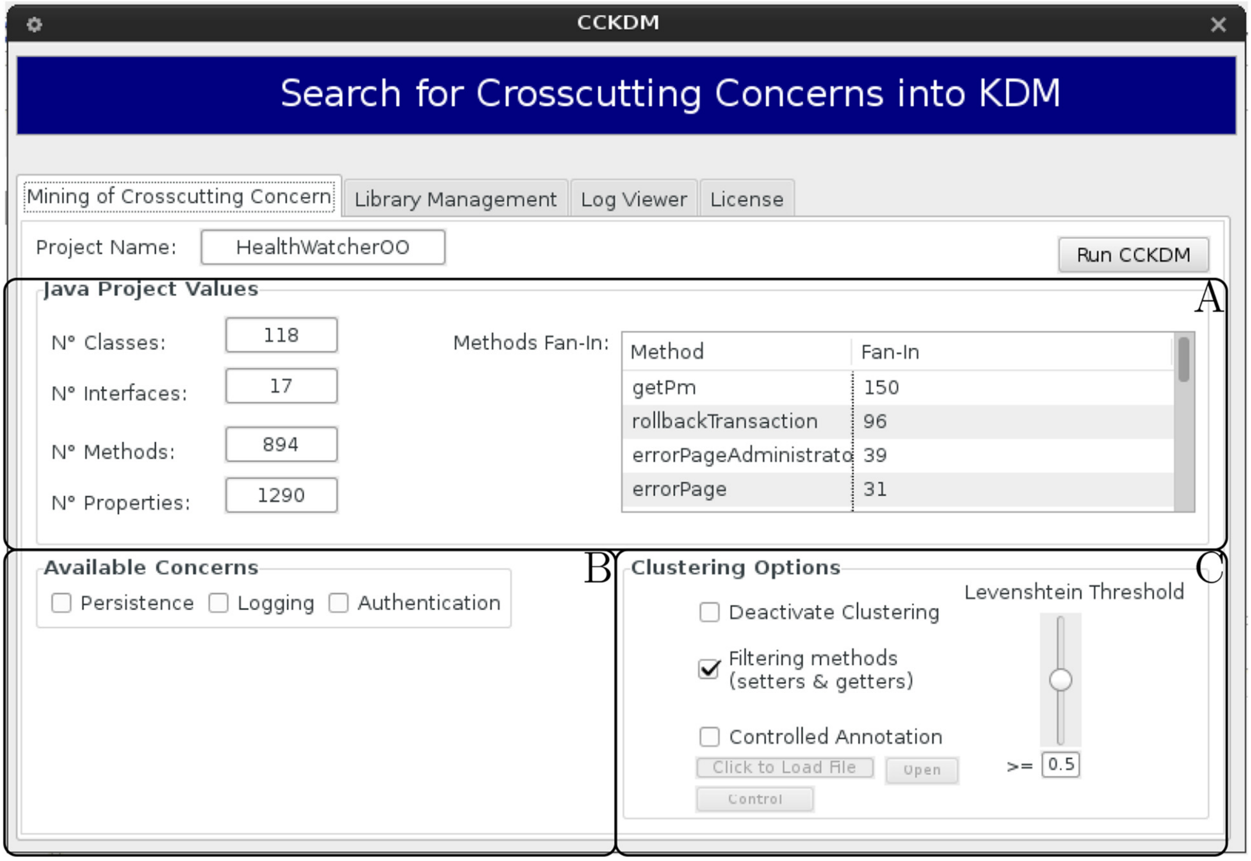
In Fig. 10, we can identify two sections denoted by the letters *D* and *E*. Section *D* allows users to add new API-based concern definitions in the following way: [*ConcerName*], [*PackageName*], [*ClassElement*...]. Section *E* shows loaded concerns where users can delete elements.

Another important feature to be commented is the log viewer tab of the main window. It presents the result of the process after the mining process is over, and it is conformed by two log files. The library log file shows all the elements identified and annotated by the first part of the process, and the cluster log file shows the centroids with identified and annotated clustered elements.

Usage process

The user starts the process by choosing an eclipse project which contains the source code or by choosing a KDM file. If the user starts the process by choosing an eclipse project, the Modisco plugin will discover and create the associate KDM model.

As long as the user has triggered the process, the collect data activity begins. The tool performs OCL queries and Java model queries (JMQ) over the KDM model to get some code structures and store them into a database


CCKDM
Search for Crosscutting Concerns into KDM
Mining of Crosscutting Concern
Library Management
Log Viewer
License
Run CCKDM
A
B
C

Project Name: HealthWatcherOO

Java Project Values

N° Classes: 118

N° Interfaces: 17

N° Methods: 894

N° Properties: 1290

Methods Fan-In:

Method	Fan-In
getPm	150
rollbackTransaction	96
errorPageAdministrato	39
errorPage	31

Available Concerns

□ Persistence □ Logging □ Authentication

Clustering Options

□ Deactivate Clustering

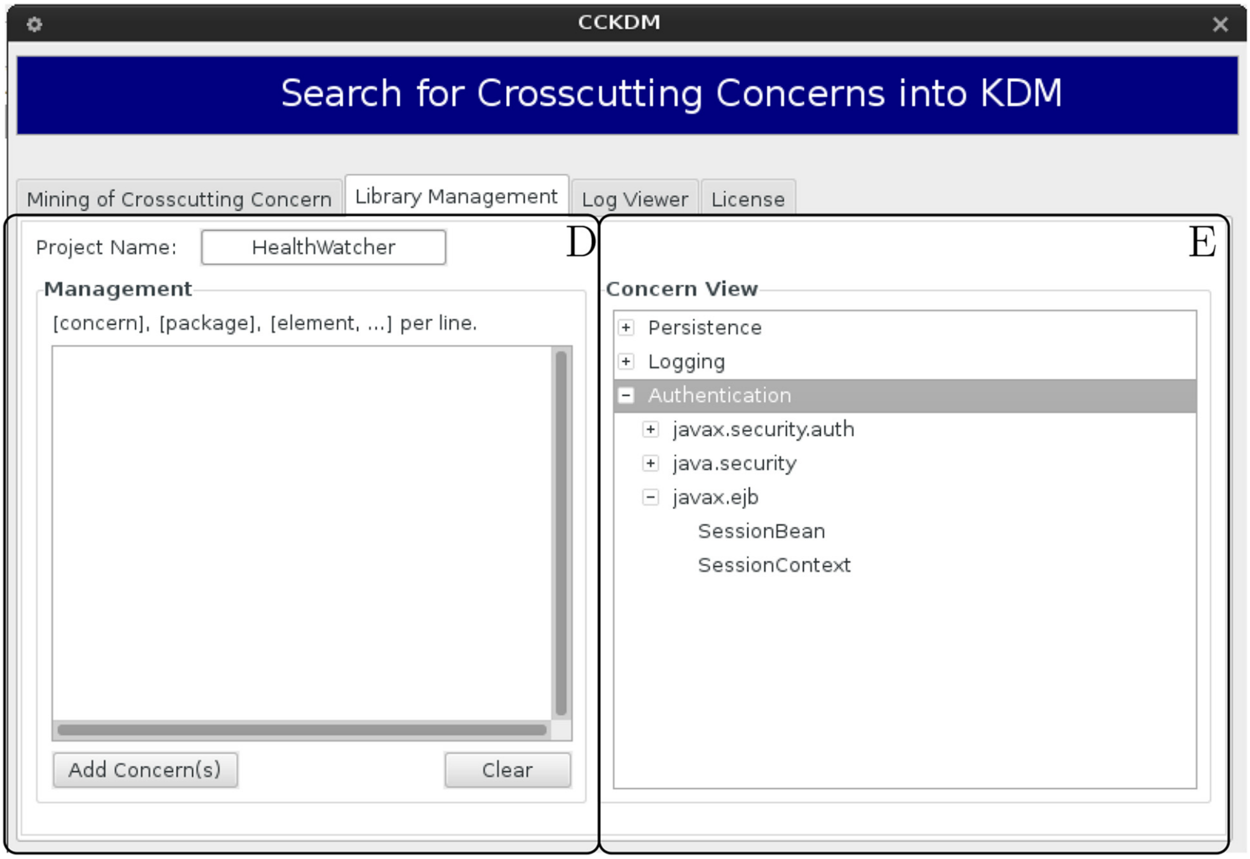
☑ Filtering methods (setters & getters)

□ Controlled Annotation

Click to Load File | Open | Control

Levenshtein Threshold >= 0.5

Fig. 9 Main window


CCKDM
Search for Crosscutting Concerns into KDM
Mining of Crosscutting Concern
Library Management
Log Viewer
License
D
E
Add Concern(s)
Clear

Project Name: HealthWatcher

Management

[concern], [package], [element, ...] per line.

Concern View

- Persistence
- Logging
- Authentication
  - javax.security.auth
  - java.security
  - javax.ejb
    - SessionBean
    - SessionContext

Fig. 10 Concern library management

(Derby DB). When this activity is over, the main window is shown to the user.

In the main window region *C*, the user can select some parameters for the clustering algorithm. The first parameter provided by the tool is to activate/deactivate the clustering capability. The second parameter is to filter getter and setter methods in a way that the names of these methods will not be taken into account by the clustering algorithm. The third parameter enables the controlled annotation. That means users can choose which of the identified method, and property must be annotated into the KDM file. Finally, to indicate the level of similarity between the centroids and method/property names, the *Levenshtein* distance is used. If the *Levenshtein* value is closer to 1.0, then the compared strings are more similar; on the other hand, if it is closer to 0.0, then the compared strings are less coincident.

The user also must select at least one concern of the region *B* in the main window to proceed with the concern mining process. Region *A* of the main window presents some information of the loaded project, such as the number of classes, number of interfaces, number of methods, and properties. These information can provide an idea of the size of the project to be analyzed. Our tool also provides information about the fan-in value of methods, where high value may indicate the presence of crosscutting concerns.

Once the user has selected the concerns to be identified, it can start the process of concern mining by triggering the button *Run CCKDM*. The first part of the process finalizes with the identification of concern seeds by using Derby DB along with the concern library (an XML file generated by our tool). Specifically, this activity consists in identifying and getting all the method names and property names which implement a given concern, matching entries of the concern library with APIs used by the application. The retrieved data is annotated into the KDM model by using BaseX (an XML database) where XQuery queries are performed.

The second part of the process corresponds to the clustering step, and it is performed by default but users may deactivate it. After identifying concerns by means of our concern library, our tool complements the search of concerns with a modified *K*-Means string clustering algorithm using the *Levenshtein* distance. All the strings identified by the concern library are the initial centroids of our cluster algorithm. Then, property names and method names that were not identified by the concern library are clustered in some centroid depending on the similarity of the strings. The concern seeds are annotated into the KDM model as we described above.

Finally, the last step of the process is an alternative activity. If the user performed the second part of the process, one of the created logs is the CSV cluster log. This log can be annotated by users to have better control on the identification process. They can indicate if the identified string name belongs or not to a concern. For that, users mark with an X letter into a CSV file as depicted in Fig. 8. After that, an annotated KDM file and logs are generated again.

Results and discussion

As previously said, the experimental study we have conducted encompasses two analysis. The first one aimed to identify what *levenshtein* values contribute the most for elevating precision and recall values. The second one intended to investigate the recall and precision of our approach. To conduct this second one, we have compared our recall and precision results with oracles and also with two other existing mining techniques found in literature: *XScan* [20] and *Timna* [21].

XScan identifies certain groups of code units that potentially share some crosscutting concerns and recommends them for creating and updating aspects. Those code units, called concern peers, are detected based on their similar interactions (similar calling relations in similar contexts, either internally or externally).

Timna is a framework for enabling automatic combination of aspect mining analyses. Firstly, each method of an application is tagged manually with the tag "candidate", if it is determined to be a good candidate for refactoring or "not a candidate", if it is not. Secondly, Timna applies four aspect mining techniques separately. These techniques are fan-in analysis, no parameters, code clone and pairings. Thirdly, some classification rules are applied, taking into account the computed attributes to decide whether the method implements a concern or not.

In Table 1, we summarize the applications under study. As we can see, they have a reasonable size (KLOC $\leq$ 9K) which make them suitable to perform a manual analysis of concerns to calculate the precision and recall metrics.

The HealthWatcher is a real health complaint system developed to improve the quality of the services provided by health care institutions. The system has a web-based user interface for registering complaints and performing several other associated operations. PetStore models an e-commerce application where customers can purchase pets online using a web browser. ProgradWeb is an academic management system currently used by the Federal University of São Carlos, Brazil.

Table 1 Applications software under study

	System	LOC	Classes	Methods	Properties
1	HealthWatcher v10	8K	118	894	1290
2	PetStore v1.3.2	9K	228	1917	3002
3	ProgradWeb	5K	133	254	4182

The next sections present both analysis we have conducted. In the "Analysis of levenshtein values" section, the *levenshtein* values analysis is shown, and in the "Recall and precision analysis" section, the recall and precision analysis is presented.

Analysis of levenshtein values

As expected, our clustering algorithm gets different results depending on which *levenshtein* value is chosen. If *levenshtein* value is closer to 0, then false positives increase affecting the precision. On the other hand, if *levenshtein* value is closer to 1, then false negatives increase, affecting recall. For example, in HealthWatcher, there is a variable called *con* related with the database connection; if we take it as a centroid with a *levenshtein* value of 0.1, the algorithm will cluster variables like *city, bairroOcorrencia, and facade*. If the *levenshtein* value is 0.9, the algorithm will not cluster variables because there are no similar variables at this level.

Clearly, high and homogeneous values of precision and recall would be optimal, but due to heuristic nature of our clustering algorithm and because method and property names may vary, it is difficult to get similar values in different applications. Therefore, the idea behind this analysis is to determine which *levenshtein* values are more suitable to apply when using our approach.

In Table 2, we show the results of our clustering algorithm taking into account five *levenshtein* values. In this case, we are considering just the results of the algorithm without taking into consideration the concern library. That is why many values are not so good. We took out dissimilar words manually to perform our clustering analysis. For example, if the concern library-based mining pointed out the word "connection" twice, then we discarded this repetitive word. So, the chosen words/terms were the centroids to the cluster algorithm which we ran for different *levenshtein* values. Notice that the first three *levenshtein* values (0.3, 0.4, and 0.5) have presented the more homogeneous values.

In order to help us in the identification of the most significant *levenshtein* values, we have built a box-plot which can be seen in Fig. 11. This graphic was generated with the statistical software R and uses the precision and recall as a variable of study for each one of the five representative thresholds with the data in Table 2.

We can see that there are some differences in the medians. For example, average thresholds 0.3, 0.4, and 0.5 have higher values of precision and recall than thresholds 0.6

Table 2 Precision and recall for persistence among different levenshtein values

	Levenshtein values (only clustering algorithm)									
	0.3		0.4		0.5		0.6		0.7	
Systems	Precision (%)	Recall (%)	Precision (%)	Recall (%)	Precision (%)	Recall (%)	Precision (%)	Recall (%)	Precision (%)	Recall (%)
HealthWatcher v10	34.31	100	41.53	58.03	46.68	52.23	46.02	46.45	46.74	43.76
PetStore v1.3.2	37.35	100	38.08	68.29	41.14	50.88	50.43	47.82	36.00	8.60
ProgradWeb	49.03	100	28.47	47.24	29.23	36.77	25.58	6.18	45.45	6.64

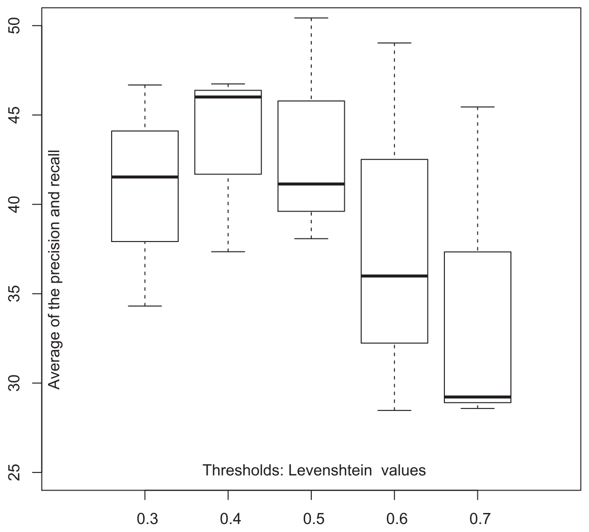


Fig. 11 Box plot for precision and recall according to the given thresholds

and 0.7. We can also observe that the variability of thresholds 0.6 and 0.7 is greater than 0.3, 0.4, and 0.5. Hence, based on this conclusions, we took into account thresholds 0.3, 0.4, and 0.5 to perform comparisons with others concern mining techniques.

Recall and precision analysis

In order to analyze the recall and precision of our approach, we have conducted two sub-analysis.The first one was focused on comparing its results with some oracles, and the second one was intended to compare it with existing mining techniques available in the literature.

Precision and recall are well-known metrics usually employed in the concern mining field to evaluate approach effectiveness:

- Precision is the ratio of the number of true positives retrieved to the total number of irrelevant and relevant code elements retrieved. It is usually expressed as a percentage.

$$P = \frac{TruePositives}{TruePositives + FalsePositives} \tag{1}$$

- Recall is the ratio of the number of true positives retrieved to the total number of relevant code elements in the source code. It is usually expressed as a percentage.

$$R = \frac{TruePositives}{TruePositives + FalseNegatives} \tag{2}$$

Oracle-based analysis

To conduct the oracle-based analysis, we applied our approach for mining persistence and logging concerns in HeathWatcher v10 and in PetStore 1.3.2. After the application, we have calculated the recall and precision of our approach by comparing its results with the oracles found in the literature [38, 39] and with our own manual analysis.

The first three columns of Tables 3 and 4 show the recall and precision results of our approach when mining persistence (Table 3) and logging (Table 4). As commented, in order to calculate the precision and recall values, we needed an oracle as a base for the comparison. Fortunately, both systems, HealthWatcher v10 and PetStore 1.3.2 were available [38, 39] with the persistence and logging concerns clearly identified. The process of checking and calculating these metrics were very time-consuming because we needed to compare the annotated source code (oracle) with the annotated XML files (KDM instance).

Observing Table 3, it is possible to see that using a 0.3 *levenshtein* we got 100 % of precision and recall analyzing HealthWatcher, that is, there are no false negatives or positives. However, notice we got 95 % of precision analyzing PetStore. This happened because our clustering algorithm have recognized more similar words in PetStore than in HealthWatcher increasing the number of false positives, as most of these words (properties and method names) had no relation with the concern.

For logging concern, the values are presented in Table 4. With a value of 0.3 *levenshtein*, our approach reached 100 % of effectiveness for precision and recall for the two systems without finding false negatives or positives. For *levenshtein* values of 0.4 and 0.5, the precision still remains within a good level as in Table 3, but the recall decreases. That is normal because higher values of *levenshtein* implies that strings will match more precisely and as a consequence will generate false negatives.

The scenario changes for PetStore in Table 3; while the *levenshtein* value increases, the precision increases and the recall decreases; that is because on one hand, false negatives increase and on the other hand, false positives decrease. In Table 4, the recall for PetStore slightly decrease but it still is a good value.

The recall is sensitive to false negatives, and on the other hand, precision is sensitive to false positives. If the *levenshtein* values are higher, then false positives increase, and if the *levenshtein* values are lower, then false negatives increase.

Obviously, our cluster algorithm failed in some cases because although some variables are similar to the centroids; the semantic is completely different. For example, the centroid variable *statement* is related to persistence, but it also clustered the variables *nameFont*, *nameY*, *names*, *timer*, *paint*, and *context*, which are related to the interface layer. However, it also clustered correctly the variable *oafee* that is a persistence field.

It is clear that our cluster algorithm helps to find variables which are related with a particular concern, but it is

Table 3 Comparison values of precision and recall for persistence

	Persistence effectiveness analysis						Comparative analysis			
	CCKDM-0.3		CCKDM-0.4		CCKDM-0.5		XScan		Timna	
Systems	Precision (%)	Recall (%)	Precision (%)	Recall (%)	Precision (%)	Recall (%)	Precision (%)	Recall (%)	Precision (%)	Recall (%)
HealthWatcher v10	100	100	100	80	100	76,11	N/A	100	N/A	N/A
PetStore v1.3.2	95	100	95.73	84.15	98.79	75.44	N/A	N/A	93.80	N/A

N/A not available

Table 4 Comparison values of precision and recall for logging

	Logging effectiveness analysis					
	CCKDM-0.3		CCKDM-0.4		CCKDM-0.5	
Systems	Precision (%)	Recall (%)	Precision (%)	Recall (%)	Precision (%)	Recall (%)
HealthWatcher v10	100	100	100	80.8	100	79
PetStore v1.3.2	100	100	90.3	85.2	88.6	82.2

not foolproof. Nevertheless, empirically, we can say that the algorithm adds value to the whole solution but we have to be careful to choose the right *levenshtein* value and complement with other techniques taking account the semantic of the variable.

Comparison with existing mining techniques

In the second analysis, we have compared the recall and precision of our approach with two other existing approaches, named XSCan and Timna, however just for the persistence concern. ProgradWeb was discarded because we did not find approaches that have used this system to analyze precision and recall metrics.

The last two columns in Table 3 aims at supporting a comparison with XScan and Timna approaches. Regarding XScan and Timna, we have relied on the precision and recall data available in publications about these tools. Unfortunately, these publications just presented the recall value of HealthWatcher using XScan and the precision value of PetStore using Timna. When we compare the precision and recall values of CCKDM with XScan and Timna, it is possible to see it reaches similar or equal values. Therefore, we can say that there are indications that CCKDM has similar effectiveness.

Based on our analyses, the following generalizations can be delineated:

- It seems possible to adapt any source code mining technique to KDM, as the program element layer do represent all source code details.
- As can be seen in our analyses, good recall and precision values were obtained using our combined technique for mining persistence. Therefore, this can enable other groups to proceed researching on concern-oriented modernizations. Clearly, we cannot guarantee the same level of recall and precision, but maybe it is possible to keep improving these metrics by using other mining techniques or even mining other type of concerns.
- Since good recall and precision values could be obtained mining persistence and logging, we claim that other concerns, which are strongly supported by APIs, can reach the same levels. The persistence and logging Java APIs provides very good staring point for the mining. After that, our clustering algorithm expands the initial set. So, the initial seeds are totally dependent on the key words from the API. However, we cannot quantify that because software engineers can use just part of an API to implement a system.
- The use of naming conventions increase the effectiveness of the precision of our clustering algorithm because more variables could be clustered and identified with a particular concern. On the other hand, the absence of naming conventions impacts negatively the precision metric because the semantic of a variable may change completely, and if it is clustered, there is no certainty that the variable implements the concern. In fact, the data illustrated in Table 2 supports our state because HealthWatcher and PetStore systems follow good practices in naming conventions, and values of precision does not change drastically among of the five *levenshtein* values. On the other hand, the precision for ProgradWeb are not so good among of the five *levenshtein* values, and in the manual checking, we observe that it does not follow a consistent naming convention.

Threats to validity

The lack of representativeness of the subject programs may pose a threat to external validity. We argue that this is a problem that all software engineering research, since we have theory to tell us how to form a representative sample of software. Apart from not being of industrial significance, another potential threat to the external validity is that the investigated programs do not differ considerably in size and complexity. To partially ameliorate that potential threat, the subjects were chosen to cover a broad class of applications. Also, this experiment is intended to give some evidence of the efficiency and applicability of our implementation solely in academic settings. A threat to construct validity stems from possible faults in the implementations of the techniques. With regard to our mining techniques, we mitigated this threat by running a carefully designed test set against several small example programs. Similarly, XSCan and Timna have been extensively used within academic circles, so we conjecture that this threat can be ruled out.

Related work

Concern mining or aspect recommendation has been a popular research topic in recent years. Static mining and history-based mining are two major techniques based on source code analysis. The static technique analyzes source code of a version of software to extract seeds of concerns. A fan-in value, which is the number of unique callers of each method/function, was first introduced by Marin and others [7] and further generalized by Zhang and others [29] to propose clustering-based fan-in analysis (CBFA). The history-based mining technique was first adopted by Breu and others [40], who proposed history-based aspect mining (HAM). HAM clusters methods/functions that add or remove a call to the same method/function and groups together methods/-functions that are called by the same cluster as concern seeds.

Lengyel et al. [41] proposes a semi-automatic approach to identify crosscutting constraints. The approach uses several algorithms to support the detection of the crosscutting constraints in meta-model-based model transformations. The input of the approach is a transformation (transformation rules and a control flow model), and the expected output is the list of the crosscutting constraints separated as aspects. Van Gorp et al. [42] proposed a UML profile to express pre- and post-conditions of source code refactorings using object constraint language (OCL) constraints. The proposed profile allows that a CASE tool to (*i*) verify pre- and post-conditions for the composition of sequences of refactorings and (*ii*) use the OCL consulting mechanism to detect bad smells such as crosscutting concerns.

The differential of our approach described herein in relation to the others is that our approach mines crosscutting concerns by using KDM instead of another models or source code. It is important to note that to the best of our knowledge, there is no previous research that addresses mining crosscutting concerns by using KDM model as input.

Rodriguez-Echeverria et al. [11] present an approach to modernize legacy web systems into rich internet applications (RIA). This process starts with the transformation of the source code into a KDM instance, using the user interface information extracted from the system. The KDM model is then mined and refined by searching for RIA patterns, which are stored in a repository. Annotations are then introduced in the KDM to signal the identified patterns. After the pattern recognition and signaling, the KDM is ready to be restructured according to the identified patterns. This work presents some similarities with our approach in the way how the concern identification is performed. Both approaches use a KDM model and a pattern repository to identify concerns. In our case, this repository is the API-based library. We also perform annotations, but they do not present the way how this is performed and which tag is used. The main differences are that we use a middle representation before start with the concern identification and a combination of OCL, JMQ, and SQL queries while their work uses predefined QVT pattern expressions to mine the KDM model.

Ricardo Pérez-Castillo et al. [12] propose a reengineering process that follows model-driven development principles to recover Web services from legacy databases. The proposed process takes a legacy relational database as to be transformed into a PSM model, according to a SQL-92 meta-model. This work uses a UML model as PIM while we use a KDM model. The PIM is generated using a SQL-92 meta-model as PSM so it is SQL language-dependent which does not allow to discover web services on non-SQL repository. KDM represents the entire legacy system so in that sense their proposal is restricted to the used PSM meta-model.

Boussaidi et al. [4] present an approach that helps constructing distinct architectural views from legacy systems. The first step of the approach aims at selecting the most relevant concepts and relationships of KDM that are relevant for the targeted view. The second step aims at revealing the system's structure by means of various pattern-driven clustering algorithms to decompose the system. The third step of the approach enables user to modify and adjust the resulting view and documented the results using a KDM model. This work also has some similarities with our approach. First of all, they select relevant KDM entities and relationships which could be related with the concerns of interest. We also perform a similar activity by identifying methods and properties which may implement certain concerns. Then, they apply several clustering algorithms to reconstruct the desired architecture. In our case, we apply our concern mining process to identify crosscutting concerns. Finally, we annotate the concerns into the KDM model. This work does not present annotations but a KDM extension.

Conclusions

In this paper, we have presented a concern mining approach to be used in the context of architecture-driven modernization (ADM). The goal is to provide modernization engineers a methodology and a tool that assists them in identifying crosscutting concerns in KDM models, enabling an aspect-oriented modernization to be performed. Besides the practical characteristic of this work, the process we have presented should also serve as general guidelines to those who need to create mining approaches for other domains/contexts. This is the first work in concern mining area that uses a standardized model in the context of ADM to perform search of concerns, and we believe that ADM standards will be widely used in a near future because is an OMG initiative, and it has the support of several important IT organizations.

Our mining technique employs a combination of a concern library and a string clustering algorithm. The required input of the approach is a KDM instance representing the legacy system, and the output is the same KDM with the concerns clearly annotated. As we have shown, it is possible to adapt a concern mining approach source code oriented to a concern mining approach model oriented. We also have shown that the precision and recall values remained good.

In terms of design, we have sought the simplicity. For example, although OCL is a powerful language to perform queries over MOF models and meta-models, it is not so intuitive to manage when we must perform complex queries. That is why our approach uses OCL simple queries and a SQL database to perform complex queries. We argue that people which may maintain the tool will be more familiarized with SQL queries than OCL. It is important to note that OCL queries are reusable and SQL queries too if the database model is not modified.

Another advantage of using a database is the persistence of the data. The entire data model in conjunction with the KDM file can be ported and reused without generating the KDM file and searching for code structures again if no modifications are performed to the source code. Finally, it is clear if we annotate the KDM file, then it could not serve as source of entry to our approach because the new attribute called "concern" is not part of KDM. To use our annotated KDM file, the KDM meta-model must be extended to support this new feature which is a trivial task by using Eclipse model development tools (MDT).

As it was previously presented, we have employed a relational DB in our solution. We claim the creation of relational subsets of KDM is a canonical activity in many contexts. As relational databases are a well accepted technology, many modernization engineers may prefer to mine relational database than a model. The creation of relational subsets from KDM is usually guided by the information to be queried. The first step is to identify which are the metaclasses that hold the information of interest. The second one is to filter out, from these metaclasses, the attributes that do not contain relevant information. After that, one can create corresponding tables for the metaclasses chosen. A possible work is to generate automatically a relational DB from KDM. As this is already done in many UML tools, we believe it is almost the same to do that for KDM.

Although we have evaluated our approach just using Java applications, applying it for applications implemented in other languages just require to update the concern library with APIs provided by other languages. For example, the library <*sql.h*> support the implementation of persistence concern in C++. Clearly we also need a tool, like Modisco, to generate a KDM instance from C++ applications. We remark the easy way to add new concerns to the concern library which makes it an extensible tool. The central resource of our mining approach is a library based on APIs; the clustering algorithm takes the firstly identified elements as its centroids, then other similar elements (which are not API-dependent) are also identified. That is the reason the precision and recall values reach so good values when compared to other techniques. Nevertheless, we believe it is possible to improve the values of the metrics even more combining our static analysis with a dynamical approach. We also are conscious that few approaches are conducted in just one step. In most of the cases, the software engineer needs to go back in the process to update results or to provide new information. An important point of our approach is the manual filtering step, allowing to filter out elements which were erroneously identified.

During the development of our approach, several observations were noticed. Although *Modisco*™is a robust framework and provides a well-documented API, it is still under strong development so some implementations of CCKDM may need to be updated in the near future. Another issue related with *Modisco*™is that the process for generating the KDM file could take several hours if the program is huge and in some cases, the KDM file is never generated. In fact, that is the main reason why we could not use JHotDraw [43] for our experimental study because *Modisco*™was not able to generate the KDM.

In the future, we plan to improve our annotation feature, for example methods could implement more than one concern. In that case, it is necessary to annotate them with multiple tags. We also are interested to investigate and identify other types of concerns like design patterns and architectural styles. To do that, other packages of the KDM specification must be studied, taking into account semantic issues. Finally, in terms of functionality extension, we are interested in integrate CCKDM with a visualization tool.

Competing interests

The authors declare that they have no competing interests.

Authors' contributions

This work is part of DSM master thesis and VVC was his advisor. DSM carried out most of the investigation, implementation, experiments, and writing. VVC gave important ideas to the project and also contributed in the writing and revision of this manuscript. RSD gave important ideas about CCKDM tool design and the implementation of the cluster algorithm and suggested a way to carry out the experiments; he also gave an important contribution in writing and revising the paper. All authors read and approved the final manuscript.

Acknowledgements

Daniel Santibáñez would like to thank the financial support provide by CAPES. Rafael Serapilha Durelli would like to thank the financial support provided by FAPESP, 2012/05168-4. Valter Vieira de Camargo would like to thank FAPESP, 2012/00494-0. We also would like to thank the Universidade Federal de São Carlos (UFSCar) for giving the ProgradWeb system and be used in the experiment.

Author details

[1]Departamento de Computação, Universidade Federal de São Carlos, Caixa Postal 676—13.565-905, São Carlos, Brazil. [2]Instituto de Ciência Matemáticas e de Computação—ICMC, Universidade de São Paulo, São Carlos, SP, Brazil.

Received: 30 July 2014 Accepted: 10 June 2015
Published online: 01 August 2015

References

1. Ulrich WM, Newcomb P (2010) Information systems transformation: architecture-driven modernization case studies. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA
2. Chikofsky EJ, Cross II JH (1990) Reverse engineering and design recovery: a taxonomy. IEEE Softw 7(1):13–17
3. Canfora G, Di Penta M, Cerulo L (2011) Achievements and challenges in software reverse engineering. Commun ACM 54:142–151
4. Boussaidi GE, Belle AB, Vaucher S, Mili H (2012) Reconstructing architectural views from legacy systems. In: Reverse Engineering (WCRE), 2012 19th Working Conference On, Kingston, Ontario, Canada. pp 345–354. http://dx.doi.org/10.1109/WCRE.2012.44
5. Perez-Castillo R, de Guzman IG-R, Avila-Garcia O, Piattini M (2009) On the use of adm to contextualize data on legacy source code for software modernization. In: Proceedings of the 2009 16th working conference on reverse engineering. WCRE '09. IEEE Computer Society, Washington, DC, USA. pp 128–132
6. von Detten M, Meyer M, Travkin D (2010) Reverse engineering with the reclipse tool suite. In: Proceedings of the 32nd ACM/IEEE international conference on software engineering—volume 2. ICSE '10. ACM, New York, NY, USA. pp 299–300
7. Marin M, Deursen AV, Moonen L (2007) Identifying crosscutting concerns using fan-in analysis. ACM Trans Softw Eng Methodol 17:3–1337
8. Durelli R, Santibáñez DM, Anquetil N, Delamaro ME, Camargo VV (2013) A systematic review on mining techniques for crosscutting concerns. ACM SAC, Coimbra, Portugal
9. Bhatti MU, Ducasse S, Rashid A (2008) Aspect mining in procedural object oriented code. In: Proceedings of the 2008 The 16th IEEE international conference on program comprehension. IEEE Computer Society, Washington, DC, USA. pp 230–235
10. Santos BM, Honda RR, de Camargo VV, Durelli RS (2014) Kdm-ao: An aspect-oriented extension of the knowledge discovery metamodel. In: Software Engineering (SBES), 2014 Brazilian Symposium On, Maceio, Alagoas, Brasil. pp 61–70. http://dx.doi.org/10.1109/SBES.2014.20
11. Rodríguez-Echeverría R, Conejero JM, Clemente PJ, Preciado JC, Sanchez-Figueroa F (2012) Modernization of legacy web applications into rich internet applications. In: Proceedings of the 11th international conference on current trends in Web engineering. ICWE'11. Springer, Berlin, Heidelberg. pp 236–250
12. Pérez-Castillo R, de Guzmán IGR, Caballero I, Piattini M (2013) Software modernization by recovering web services from legacy databases. J Softw Evol Process 25(5):507–533
13. Deltombe G, Goaer OL, Barbier F (2012) Bridging kdm and astm for model-driven software modernization. In: SEKE, San Francisco Bay, USA. pp 517–524
14. Pérez-Castillo R, de Guzmán IGR, Caivano D, Piattini M (2012) Database schema elicitation to modernize relational databases. In: Maciaszek LA, Cuzzocrea A, Cordeiro J (eds). ICEIS (1). SciTePress, Wroclaw, Poland. pp 126–132. http://dblp.uni-trier.de/db/conf/iceis/iceis2012-1.html
15. Mainetti L, Paiano R, Pandurino A (2012) Migros: a model-driven transformation approach of the user experience of legacy applications. Lecture notes in computer science (including subseries lecture notes in artificial intelligence and lecture notes in bioinformatics) 7387 LNCS:490–493. cited By (since 1996) 0
16. Tabares MS, Moreira A, Anaya R, Arango F, Araujo J (2007) A traceability method for crosscutting concerns with transformation rules. In: Proceedings of the early aspects at ICSE: workshops in aspect-oriented requirements engineering and architecture design. EARLYASPECTS '07. IEEE Computer Society, Washington, DC, USA. p 7
17. Mesbah A, van Deursen A (2005) Crosscutting concerns in j2ee applications. In: Proceedings of the seventh IEEE international symposium on Web Site evolution. IEEE Computer Society, Washington, DC, USA. pp 14–21
18. Marin M, van Deursen A, Moonen L (2004) Identifying aspects using fan-in analysis. In: Proceedings of the 11th working conference on reverse engineering. IEEE Computer Society, Washington, DC, USA. pp 132–141
19. Neto AC, de Medeiros Ribeiro M, Dosea M, Bonifacio R, Borba P, Soares S (2007) Semantic dependencies and modularity of aspect-oriented software. In: Assessment of contemporary modularization techniques, 2007. ICSE Workshops ACoM '07. First International Workshop On. pp 11–11
20. Nguyen TT, Nguyen HV, Nguyen HA, Nguyen TN (2011) Aspect recommendation for evolving software. In: Proceedings of the 33rd international conference on software engineering. ICSE '11. ACM, New York, NY, USA. pp 361–370
21. Shepherd D, Palm J, Pollock L, Chu-Carroll M (2005) Timna: a framework for automatically combining aspect mining analyses. In: Proceedings of the 20th IEEE/ACM international conference on automated software engineering. ACM, New York, NY, USA. pp 184–193
22. Force OAT (2012) Why do we need standards for the modernization of existing systems? http://adm.omg.org/legacy/ADM_whitepaper.pdf
23. Bianchi A, Caivano D, Marengo V, Visaggio G (2003) Iterative reengineering of legacy systems. IEEE Trans Softw Eng 29:225–241
24. OMG Object Management Group (OMG) architecture-driven modernisation. http://www.omgwiki.org/admtf/doku.php?id=start
25. Perez-Castillo R, de Guzmán IG-R, Piattini M (2011) Knowledge discovery metamodel-iso/iec 19506: a standard to modernize legacy systems. Comput Stand Interfaces 33(6):519–532
26. Kellens A, Mens K, Tonella P (2007) A survey of automated code level aspect mining techniques. Springer, Berlin, Heidelberg
27. Mens K, Kellens A, Krinke J (2008) Pitfalls in aspect mining. In: Proceedings of the 2008 15th working conference on reverse engineering. IEEE Computer Society, Washington, DC, USA. pp 113–122
28. Cojocar GS, Czibula G (2008) On clustering based aspect mining. In: Intelligent Computer Communication and Processing, 2008. ICCP 2008. 4th International Conference On, Cluj-Napoca, Romania. pp 129–136. http://dx.doi.org/10.1109/ICCP.2008.4648364
29. Danfeng Z, Yao G, Xiangqun C (2008) Automated aspect recommendation through clustering-based fan-in analysis. In: Proceedings of the 2008 23rd IEEE/ACM international conference on automated software engineering. IEEE Computer Society, Washington, DC, USA. pp 278–287
30. von Detten M, Becker S (2011) Combining clustering and pattern detection for the reengineering of component-based software systems. In: Proceedings of the Joint ACM SIGSOFT conference—QoSA and ACM SIGSOFT symposium—ISARCS on quality of software architectures—QoSA and architecting critical systems—ISARCS. QoSA-ISARCS '11. ACM, New York, NY, USA. pp 23–32
31. Bruneliere H, Cabot J, Jouault F, Madiot F (2010) Modisco: A generic and extensible framework for model driven reverse engineering. In: Proceedings of the IEEE/ACM International Conference on Automated Software Engineering. ASE '10. ACM, New York, NY, USA. pp 173–174. http://doi.acm.org/10.1145/1858996.1859032
32. Ceccato M, Marin M, Mens K, Moonen L, Tonella P, Tourwé T (2006) Applying and combining three different aspect mining techniques. Softw Q J 14(3):209–231
33. Warmer J, Kleppe A (2003) The Object Constraint Language: Getting Your Models Ready for MDA. 2nd edn. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA
34. Habela P, Kaczmarski K, Stencel K, Subieta K Ocl as the query language for uml model execution. In: Bubak M, van Albada G, Dongarra J (eds). Computational Science – ICCS 2008. Lecture Notes in Computer Science. Springer Vol. 5103. pp 311–320. http://dx.doi.org/10.1007/978-3-540-69389-5_36
35. Figueiredo E, Cacho N, Sant'Anna C, Monteiro M, Kulesza U, Garcia A, Soares S, Ferrari F, Khan S, Castor Filho F, Dantas F (2008) Evolving software product lines with aspects: an empirical study on design stability. In: Proceedings of the 30th international conference on software engineering. ICSE '08. ACM, New York, NY, USA. pp 261–270
36. Han J (2005) Data mining: concepts and techniques. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA
37. Levenshtein V (1966) Binary codes capable of correcting deletions, insertions and reversals. Sov Phys Doklady 10:707
38. McFadden RR, Mitropoulos FJ (2013) Survey of aspect mining case study software and benchmarks. In: Southeastcon, 2013 Proceedings of IEEE, Fort Lauderlade, Florida, USA. pp 1–5. http://dx.doi.org/10.1109/SECON.2013.6567402
39. Vitek J (ed) (2008) ECOOP 2008 - Object-Oriented Programming, 22nd European Conference, Paphos, Cyprus, July 7-11, 2008, Proceedings.

Lecture Notes in Computer Science, Vol. 5142. Springer, Paphos,Cyprus. http://dx.doi.org/10.1007/978-3-540-70592-5

40. Breu S, Zimmermann T (2006) Mining aspects from version history. In: Proceedings of the 21st IEEE/ACM international conference on automated software engineering. IEEE Computer Society, Washington, DC, USA. pp 221–230
41. Lengyel L, Levendovszky T, Angyal L (2009) Identification of crosscutting constraints in metamodel-based model transformations. In: EUROCON 2009, EUROCON '09. IEEE, St. Petersburg, Russia. pp 359–364. http://dx.doi.org/10.1109/EURCON.2009.5167656
42. Van Gorp P, Stenten H, Mens T, Demeyer S (2003) Towards automating source-consistent uml refactorings. In: Stevens P, Whittle J, Booch G (eds). «UML» 2003 - The Unified Modeling Language. Modeling Languages and Applications. Lecture Notes in Computer Science. Springer, San Francisco, CA, USA Vol. 2863. pp 144–158. http://dx.doi.org/10.1007/978-3-540-45221-8_15
43. Marin M, Moonen L, Deursen AV (2007) An integrated crosscutting concern migration strategy and its application to jhotdraw. In: Proceedings of the Seventh IEEE International Working Conference on Source Code Analysis and Manipulation. SCAM '07. IEEE Computer Society, Washington, DC, USA. pp 101–110. http://dx.doi.org/10.1109/SCAM.2007.4