A Role-based Crosscutting Concerns Mining Approach to Evolve Java Systems Towards AOP

Mario Luca Bernardi  
Department of Engineering  
University of Sannio, Italy  
mlbernar@unisannio.it

Giuseppe Antonio Di Lucca  
Department of Engineering  
University of Sannio, Italy  
dilucca@unisannio.it

ABSTRACT

A key step in the evolution of a Java system towards the aspect oriented paradigm is the identification of crosscutting concerns that need to be refactored.

This paper proposes an approach to identify concerns and the crosscutting among them in existing Java systems. A meta-model is defined to represent concerns as sets of Type Fragments (where a Type Fragment is a portion of a Type in terms of its members, properties and relationships). The approach exploits the concept of Role: each Role is associated to a concern and the system source code is analyzed to find the Type Fragments implementing it. All the Roles that contribute to implement a same semantic concern are grouped together by a clustering algorithm based on a combination of a structural and a lexical distance. Each cluster of Roles (and thus the Type Fragments associated to them) is assigned to a single more abstract concern. Crosscutting is detected looking for scattering and tangling of Type Fragments within the identified concerns. The structural information about the Type Fragments assigned to each concern and the crosscutting relationships among the concerns can be used to drive the refactoring towards aspects. The results from a case study where the approach has been applied to several software systems are presented and discussed.

Categories and Subject Descriptors:
D.2.7 [Software Engineering]: Distribution, Maintenance, and Enhancement—Restructuring, reverse engineering, and reengineering.

General Terms: Design, Documentation, Experimentation.

Keywords: Software Evolution, Reverse Engineering, Aspect Mining, Aspect Oriented Programming, MOF

1. INTRODUCTION

The adoption of the Aspect Oriented Programming (AOP) is growing in the development of complex systems to better manage the crosscutting concerns (such as Persistence, Logging, Synchronization). Indeed, AOP allows the separation and encapsulation of the crosscutting concerns into the “aspects” [11], in a better and easier way than traditional programming approaches. Crosscutting concerns are usually implemented by code scattered and tangled across more components (e.g., classes, methods in existing Object Oriented systems) and this can produce a high level of duplication [2] that makes the crosscutting concerns’ code hard to maintain, evolve and reuse. This is also fostering the evolution/migration of existing Object Oriented (OO) systems towards AOP ones to eliminate, or reduce at a minimum, the scattered and tangled code implementing the crosscutting concerns; for example OO systems coded by Java are transformed into AOP ones by using AspectJ [11], one of the most popular AOP languages. A key issue to efficiently and effectively drive such an evolution/migration process is to provide maintainers with useful tools for the automatic identification of the code components implementing the crosscutting concerns to re-engineer into aspects. In this paper an approach to identify the components implementing the static crosscutting concerns associated to Roles implemented by super-imposition in Java systems is proposed.

The concept of Role is a significant one in the OO paradigm and various definitions have been provided for it [26, 15, 19, 21], such as: a role models visible properties of an object; an object can play different Roles simultaneously; a Role is bound to one or more objects (called the players of the Role); a Role specifies requirements for its players that can be implemented by several classes so the players need not be instances of the same class (different objects can thus perform the same role in different ways). Analogies between object Roles and aspects (i.e. crosscutting concerns) has been highlighted and discussed by some authors with controversial results: Kendall, in [10], describes AOP as a promising approach for role models; Hanenberg et al., in [24], come to the conclusion that roles and aspects are too different, while in [6] Hanenberg et al. consider role based programming as a special case of AOP.

The proposed approach is based on the fact that usually inheritance, interfaces implementation and type nesting, are used as a way to implement object roles. Indeed, Roles (like crosscutting concerns) are often implemented by super-imposition in OO (Java) systems, [7, 29]: each entity in the system that need to play a well defined Role is forced to

---

1 Static crosscutting is related to the system static structure expressed as a type hierarchy based on inheritance, implementation and type nesting relationships.
implement an Interface (examples are the implementations of Persistence and Logging concerns). The use of interfaces to model roles has been studied in literature and several aspects mining and refactoring approaches are based on their analysis to produce a list of candidate methods to encapsulate into aspects [7, 29]. The superimposition of (secondary) roles can introduce both scattering and tangling, at class level. Scattering is due to the injection of the declarations of methods into all the classes implementing a certain role; tangling will exist when, such a scattered role implementation, is also tangled with other roles (i.e. concerns) implemented by the same classes. Type containment relationship is another way to introduce tangling and scattering at class level: a type implementing a role can contain another type implementing a role associated to another concern.

In the proposed approach each role is (initially) associated to a separate concern and the analysis of the type hierarchy of the system is performed and exploited to identify the roles and the crosscutting in role implementations. A MOF\textsuperscript{2} meta-model has been defined to represent concerns as sets of Type Fragments (i.e. a portion of a Type in terms of its members, properties and relationships). Traversing the system hierarchy, the roles (i.e. the associated concerns) are identified and an instance of the proposed meta-model is created. The instantiated model defines a concern for each role along with all the Fragments of the types in the system that can assume such role. Since the focus is on the static crosscutting the type hierarchy analysis is performed at member/method granularity, i.e. methods are seen as black-boxes and their internal code is not analyzed but just their declarations are considered. A Type Fragment is a portion of a type made by (some of) its members/methods and its relationships (inheritance, implementation, and containment). All the Roles that contribute to implement a same semantic concern are grouped together by means of a clustering algorithm based on a combination of a structural and a lexical distance. Each cluster of Roles (and thus the Type Fragments associated to them) is assigned to a single more abstract concern. Thus the main concerns of the system are identified together with all the Type Fragments making up them. Crosscutting is detected looking for scattering and tangling of Type Fragments within the identified concerns.

The proposed approach is the starting step of a wider process aiming at re-engineering a Java system to an AOP one. Indeed the structural information about the Type Fragments assigned to each concern and the crosscutting relationships among the concerns can be used to drive the existing refactoring approaches [4, 17]. In this paper we do not discuss about how such a refactoring can be performed. Anyway the structural information, about the concern composition by the type fragments, is useful to reduce the needed effort because the focus for the refactoring can be limited to the identified Type Fragments to define the related aspects into which encapsulate them.

The paper is structured as follows. Section 2 discusses background on crosscutting concerns while in Section 3 the meta-model defined to represent concerns at class level is described. Section 4 presents the approach to identify Roles in a Java system and to assign Type Fragments to concerns.

Section 5 describes the concerns’ clustering while Section 6 describes how to perform the crosscutting analysis given a concern model instance. Section 7 illustrates a case study conducted on some Java systems. In Section 8 relevant related works are discussed and some comparisons with the proposed approach are made. Finally Section 9 contains conclusive remarks and briefly discusses future work.

2. BACKGROUND
A concern has been defined in several ways [20, 5, 18, 27, 4]. Informal definitions all share a key point: a concern is a human-oriented expression of computational intent. More formal definitions see the concern as a set of elements within the program’s (abstract) specification.

We refer to a concern as the behaviour specified by a set of portions of types (in terms of methods, properties and relationships) playing a role associated to it.

Several definitions have been proposed also for the concept of “crosscutting” [16, 14, 4], for which there is a lack of a widely-accepted definition.

Some of them are less restrictive with respect to the scattering and tangling properties that the elements associated to concerns must satisfy. However, a misconception is to consider crosscutting as a structural property of a single concern, while it’s rather a relationship between representations of at least two concerns: one crosscutting the other (and vice versa). This is an important issue: code that is either only scattered or only tangled can still be modularized in a satisfactory way within object oriented paradigm [13, 30, 8]. Hence the crosscutting relationship about two concern representations is a property of code elements that are scattered and tangled in such a way that it is not possible to modularize them within traditional composition/decomposition constructs. In the proposed approach to identify static crosscutting concerns the first definition of crosscutting as provided by Kiczales [14] and its equivalent formalization provided by Ostermann [16] has been considered. This definition views crosscutting as a symmetrical, not reflexive and not transitive relationship among representation of concerns.

Given two concerns \(A\) and \(B\) and a crosscutting relationship \(\cong\), it holds that[16]:
\[
A \cong B \Rightarrow B \cong A \tag{1}
\]
\[
A \not\cong A \tag{2}
\]
\[
A \cong B \land B \cong C \not\Rightarrow A \cong C \tag{3}
\]

Hence the analysis of crosscutting among concerns requires to address three main issues: (i) the representation of the concerns within the system at different level of granularity (e.g. class and method level); (ii) the assignment of code components to the concerns they contribute to implement; (iii) the identification of scattering and tangling among these components. To have a complete crosscutting representation this process should be performed at both class and method. In this paper only concerns and crosscutting at class level are considered leaving the dynamic crosscutting identification (at method level) as future work.

3. A META-MODEL TO REPRESENT CONCERNS AT CLASS-LEVEL
Both static and dynamic crosscutting concerns can be found in an OO system. As already said, the implementation of static crosscutting by OOP languages (such as the Java one) is usually made by super-imposition and inheritance/implementation declarations on types.

\textsuperscript{2}Meta Object Facility, see http://www.omg.org/mof
Conversely dynamic crosscutting is implemented by means of interleaving code lines within the bodies of different methods thus containing similar blocks of code statements related to the involved concerns. These code blocks are scattered among the several methods and tangled with the code implementing the specific responsibility of each method.

Since the focus is on static crosscutting, the model explicitly represents the structure of the system in terms of concerns made up of fragments of types, their members and their relationships.

Figure 1 shows the meta-model, as a UML class diagram, representing the static structure of a Java system in terms of its Concerns. The model is centered on the concept of ReferenceTypeFragment defined as a portion of a Type. The Java system is modeled as a set of Types (i.e. Array, Value and Reference Types) where Reference Types (i.e. Interfaces and Classes, and Interfaces implemented by Classes) are composed by Reference Type Fragments (that can be Class- or Interface- Fragments). A Reference Type Fragment is in turn composed by Fields and Methods. A ReferenceType can inherit from another ReferenceType as well as can contain another ReferenceType (e.g. an inner class). A Concern is represented by mean of a set of ReferenceTypeFragments (in the paper, for brevity, we refer to them also as Type Fragments). In model instances, relationships on types are mapped to the Type Fragments associated to one of the concerns. The meta-model has been defined as a MOF model and implemented by means of the EMF framework\(^3\). As the focus is on the static crosscutting, the model is at class granularity, i.e. a method is seen as a whole black-box (just its declarations is considered, while its internal code is not). The model is able to represent different concerns separately viewing the complete system as the composition of all the concerns. An analysis of an instance of the model allows to identify the crosscutting among the concerns due to the tangling and scattering of the ReferenceTypeFragments associated to each concern.

\(^3\)The Eclipse Modeling Framework, http://www.eclipse.org/modeling

4. **ROLE-BASED IDENTIFICATION OF STATIC CONCERNS**

An instance of the concern meta-model is generated according to the two main steps described in the following sub-sections. The generation of the instance of the meta-model is made on the base of the underlying assumption that, at class level, the finest granularity of a concern implementation\(^4\) is by means of type fragments implementing a single role. This assumption was always verified in the experiments carried out, although more experiments should be performed in order to assess its general validity.

4.1 **Identification of Roles**

The approach is pivoted on the concept of *Role*. Since the goal is to identify the code components acting as Roles in OO class-based systems, a definition of role closer to their implementations in the source code was adopted. A Role is defined as any reference type that introduces, in all paths from a root to a leaf of the type hierarchy graph, the declaration of a new set of members. Such definition follows in the second category within the Steimann classification [25] (roles represented as gen/spec). This definition is valid for any object-oriented class-based language supporting multiple inheritance of interfaces and single class inheritance. According to this definition, Roles are identified by traversing each type hierarchy in the system, from the root to its leaves, to find the types that declare along it new members. Figure 2-(a) shows an example consisting of three hierarchies rooted in the interfaces “Command”, “Undoable” and “Logger”. Following the paths from the three roots down to leaves, the roles showed in Figure 2-(b) are identified. The three root interfaces are all roles as well as their direct implementing classes (UndoableCommand and the two inner classes “UndoActivity”). Indeed, all of them declare a new set of members while the concrete classes implementing Logger (“ConsoleLogger” and “FileLogger”) and the concrete Command classes (like “MoveCommand” and “CopyCommand”) do not introduce any members and hence aren’t roles.

4.2 **Determining the Type Fragments Implementing the Roles**

To create an instance of the meta-model all the type fragments implementing each role have to be identified. Initially the concerns associated to the roles contains no type fragment. The association between concern and roles is made regardless of any semantic meaning about them. Just the structural information, extracted by the traversal of the type hierarchy, is exploited. At the end of the traversal each concern will contain the type fragments of all types that implements a single role. So the system can be represented, by means of the meta-model, as a set of concerns each made by the type fragments contributing to a single role implementation.

Figure 3 reports the skeleton of the algorithm implementing the two steps to create an instance of the meta-model: it performs a traversal of the hierarchy type graph starting from leaves and going up to the roots. Let $T$ be the

\(^4\)Concerns can be also found at a finer granularity if methods bodies are taken into account (i.e. as sequences of statements).
set of the types defined in the system, leaves(T) be the set of the leaves in the type hierarchy graph and \( F(t) \) be the type fragment of a type \( t \). Moreover let \( sup(t) \) and \( sub(t) \) be respectively the set of super- and sub-types of the type \( t \). Each type \( t \) encountered during the traversal is analysed to find what roles it implements (this operation is called \texttt{implementedRoles()} in Figure 3: it verifies if sets of new members are introduced by the type and computes the implemented roles). For each role \( r \) implemented by \( t \), a type fragment \( tf \) is created and added to the concern \( c \) associated to the role \( r \). Figure 2-(c) shows the results of the algorithm execution on the example of Figure 2-(a), for the concern associated to the “Undoable” role: starting from the leaves (the two inner classes “UndoActivity”, the concrete Logger classes and the concrete Commands), fragments are created for each type that can perform the “Undoable” role. The two UndoActivities classes produced two fragments each including just the \texttt{undo()} method declared by the interface fragment of the “Undoable” role itself.

The resulting concern model instance could be at a too detailed level: many of the concerns are likely made by the Type Fragments used to implement not one, but several roles. Thus a step is needed to cluster roles that actually contribute to implement a same more abstract concern by merging together the concerns at a lower level of abstraction.

5. **CLUSTERING OF ROLES**

Among the identified concerns there will be concerns associated to roles that model quite specific properties or abstraction and concerns associated to roles that model common general properties and abstractions. Usually the latter consist of a number of fragments larger than the first.

It is possible to have several concerns’ structures (i.e., roles and the corresponding Type Fragments implementing the concerns) that contribute to realize a more wider (abstract) concern (i.e., a concern associated to roles related to a main abstraction implemented in different contexts).

Groups of such concerns’ structures (i.e., roles), can be merged together by a Hierarchical Agglomerative Clustering (HAC) algorithm [9] based on a distance among roles that takes into account both structural and lexical information extracted from the system’s type hierarchy graph. This algorithm requires pre-processed data and produces sets of clusters in order of decreasing similarity. The input to the clustering process is a dissimilarity matrix. The numerical value of each element in the matrix is the distance between
two roles and is based on both structural and lexical information extracted from the hierarchy type graph.

Given two roles $i$ and $j$, the distance $d(i, j)$ is defined as the weighted sum of two contributions as follows:

$$d(i, j) = \alpha \ast StructuralDistance(i, j) + (1 - \alpha) \ast LexicalDistance(i, j)$$

The $\alpha$ weight can be exploited to balance structural and lexical contribution to the overall distance value.

### 5.1 Structural Distance among roles

The Structural Distance (SD) between two roles is calculated traversing the hierarchy type graph and it’s based on the analysis of inheritance and implementation relationships. A sub-role is considered at the minimum distance from the super-role that it implements or inherits from if no other sub-role is derived from the sub-role itself.

The distance among a super-role $i$ and a sub-role $j$ is proportional to the Number of the Added Methods (NAM) declared by $i$ and inherited by $j$ normalized with respect to the maximum NAM value of the other super-roles of $j$. This because $j$ should be clustered with the super-role that contributes the most to specify its external interface. The overall computed distance is modulated by the normalized number of children (NNOC$^5$) of each super-role since the more a super-role is inherited or implemented, the more it should be considered as a shared property or functionality of the system that’s worth to consider as a separate concern. Roles that are not connected by any direct path (i.e. are not involved in any direct relationship) are considered at maximum structural distance.

Given two roles $i$ and $j$, such that $j$ is sub-role of $i$, the distance can be calculated by the following expression:

$$SD(i, j) = 1 - \frac{NAM(i)}{\max(NAM(sup(j)))} \ast NNOC(i)$$

where $sup(j)$ is the set of the super-types of the role $j$. The structural distance is expressed in the range $[0, 1]$.

![Figure 4: An example to calculate distances.](image)

Table 1: SD distance matrix for the example in Figure 4.

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td>Logger</td>
<td>*</td>
<td>1.0</td>
<td>0.3</td>
<td>1.0</td>
<td>1.0</td>
<td>1.0</td>
<td>1.0</td>
<td>1.0</td>
<td>1.0</td>
</tr>
<tr>
<td>Command</td>
<td>0.3</td>
<td>*</td>
<td>0.5</td>
<td>1.0</td>
<td>1.0</td>
<td>1.0</td>
<td>1.0</td>
<td>1.0</td>
<td>1.0</td>
</tr>
<tr>
<td>LoggedCommand</td>
<td>0.5</td>
<td>0.5</td>
<td>*</td>
<td>1.0</td>
<td>1.0</td>
<td>1.0</td>
<td>1.0</td>
<td>1.0</td>
<td>1.0</td>
</tr>
<tr>
<td>UndoableCommand</td>
<td>0.5</td>
<td>0.5</td>
<td>0.5</td>
<td>*</td>
<td>1.0</td>
<td>1.0</td>
<td>1.0</td>
<td>1.0</td>
<td>1.0</td>
</tr>
<tr>
<td>Undoable</td>
<td>0.5</td>
<td>0.5</td>
<td>0.5</td>
<td>0.5</td>
<td>*</td>
<td>1.0</td>
<td>1.0</td>
<td>1.0</td>
<td>1.0</td>
</tr>
<tr>
<td>UndoActivity</td>
<td>0.5</td>
<td>0.5</td>
<td>0.5</td>
<td>0.5</td>
<td>0.5</td>
<td>*</td>
<td>1.0</td>
<td>1.0</td>
<td>1.0</td>
</tr>
<tr>
<td>UndoActivity</td>
<td>0.5</td>
<td>0.5</td>
<td>0.5</td>
<td>0.5</td>
<td>0.5</td>
<td>0.5</td>
<td>*</td>
<td>1.0</td>
<td>1.0</td>
</tr>
<tr>
<td>UndoActivity</td>
<td>0.5</td>
<td>0.5</td>
<td>0.5</td>
<td>0.5</td>
<td>0.5</td>
<td>0.5</td>
<td>0.5</td>
<td>*</td>
<td>1.0</td>
</tr>
</tbody>
</table>

5The normalization is performed with respect to the maximum value of Number of Children (NOC) within the entire system.

just roles and relationships are showed, while near the name of each role it is reported the number of method the role declares.

Table 1 reports the calculated SD matrix. The distance among roles not involved in inheritance or implementation relationships (like for “Command” and “Logger” roles) is 1 (the maximum value). Conversely, the distance between a sub-role and its unique implemented super-role is equal to the minimum value (like for “Undoable” and the three “UndoActivity”). In the other cases, when a sub-role implements or inherits from several super-roles, the distance depends on the number of the added method (as for roles “Command” and “UndoableCommand”). The SD distance privileges the aggregation to the super-role that gives the most contribution to the external sub-role interface. Therefore the distance among LoggedCommand and Command is lower (0.5) than the distance among LoggedCommand and Logger (0.9).

### 5.2 Lexical Distance among roles

The Lexical Distance (LD) is based on the analysis of the prefixes and suffixes in the names of the roles. A common substring in two roles’ names is qualified as a body, as a prefix or as a suffix substring and the different kind of substrings are weighted in a different way. Common prefixes are considered as an indicator of concern similarity since they usually “qualify” the prefix (e.g. “DefaultGraphListerner” and “MultiGraphListerner” should be clustered in the same concern since they are both listeners for different kind of graphs). Prefixes are considered in a different way depending on the presence of a common suffix. If there isn’t a common suffix then the common prefix is considered as an indicator of concern dissimilarity. This is because the common prefix is qualified by a different suffixes in two, or more, role names. Conversely if there is a common suffix the common prefix is considered an indicator of concern similarity. As an example “GraphModel” and “GraphListener” roles likely belong to two different concerns and “Graph” prefix weight as a dissimilarity indicator; while considering the role names “GraphModelListener” and “GraphEditorListener”, the presence of the common “Listener” suffix makes the common prefix “Graph” an indicator of concern similarity and therefore the distance should be lower. Let be $ED(x, y)$ the normalized edit distance between the role names $x$ and $y$ and $S(i, j)$ a boolean digit indicating the presence of a common suffix in the role names $i$ and $j$. The LD distance is calculated by the following expression:

$$LD(i, j) = S(i, j) \ast (1 - \frac{|c(i, j)|}{\min(|i|, |j|)}) + (1 - S(i, j)) \ast (ED(p(i, j), e) + ED(i, j))$$

As an example of the SD distance matrix calculation let us consider the system type hierarchy in Figure 4, where
Table 2: LD distance matrix for the example in Figure 4.

where $\epsilon$ is the empty string, $c(i,j)$ is the concatenation of common sub-strings of $i$ and $j$, $p(i,j)$ is the prefix of strings $i$ and $j$ (or $\epsilon$ if does not exists) and $\bullet$ is the operator returning the string length. $LD(i,j)$ is expressed in the range $[0, 1]$.

Table 2 shows the LD distance matrix calculated for the example in Figure 4.

A complete-linkage HAC clustering algorithm, based on the dissimilarity distance, has been defined to cluster the roles, i.e. the concerns associated to them. Such conservative choice was selected after having experimented with most of the intra-cluster distance available (single, weighted, centroid and V linkages) obtaining comparable results. An automatic cut based on $R^2$ and $RMSSTD$ values has been chosen to select the best trade-off between number of cluster and their internal cohesion.

With reference to the example in Figure 4 the following clustering resulted: \{Undoable, UndoAdapter, UndoActivity, UndoActivity, UndoActivity\}, \{Command, LoggedCommand, UndoableCommand, UndoActivity\} and \{Logger\}. LoggedCommand could be clustered both with Logger or Command concerns since it is a sub-role of both.

5.3 Merging of Concerns

Once the roles that have to be clustered together and assigned to a single concern are identified, a merging algorithm is applied to the model instance. The merge of two concerns $C_x$ and $C_y$ creates a new concern $C_z$ in place of $C_x$ and $C_y$ by merging their type fragments looking at fully qualified names. When a type fragment $f$ with a given name exists only in one of the two merged concerns $C_x$ and $C_y$, the merge operation is trivial since it simply adds $f$ to the resulting concern $C_z$. When two type fragments $f \in C_x$ and $g \in C_y$ have the same fully qualified name, a new type fragment $h \in C_z$ is created by joining the properties (methods, fields and relationships) of $f$ and $g$.

More formally, given a type fragment $f$, let be $m(f)$ its members (fields and methods), $i(f)$ the set of types inherited by $f$ and $im(f)$ the set of interfaces implemented by $f$; the merge operation between two type fragments $f$ and $g$ is defined as the operation of creating a new type fragment $h = f + g$ satisfying the following properties:

$$m(h) = m(f) \cup m(g)$$
$$im(h) = im(f) \cup im(g)$$
$$in(h) = in(f) \cup in(g)$$

The merge operation behaves in different ways depending on the kind of type fragments\(^6\). Hence the $in(x)$ set specifies zero or more inherited interfaces if $x$ is an interface fragment, or zero or one super class if $x$ is a class fragment.

A simple merging example is reported in Figure 5. Each concern $C_x$ and $C_y$ defines several type fragments involved in some relationships. Interface Fragments, as $I$, owned only by one concern, are directly added to the merged concern $C_z$. Fragments like $X$ and $Y$ are merged since they have the same fully qualified name (e.g. they are fragments in different concerns of the same type). In the figure the concern $C_z$ includes all the $X$ and $Y$ fragments composing $C_x$ and $C_y$ along with the implementation and inheritance relationships (respectively towards the interface fragment $I$, and towards the class fragment $X$).

6. CROSSCUTTING ANALYSIS OF CONCERNS

A crosscutting matrix is built and analysed to identify pairs of crosscutting concerns. The matrix is computed by tracing down the relationships among the type fragments located in different concerns. The model instance is traversed to identify, for each pair of concerns, scattering and tangling at fragment level.

The matrix is symmetric (since symmetric is the adopted crosscutting definition) and reports for each pair of scattered concerns\(^7\) the number of type fragments generating tangling (such fragments are those type fragments of both concerns that refers to the same type). Therefore two concerns $C_x$ and $C_y$ are crosscutting among them if and only if: (i) they both contain more than one type fragment (hence generating scattering) and (ii) there are at least two type fragments $f \in C_x$ and $g \in C_y$, with the same fully qualified name (this means that the two concerns are introducing members or relationships on the same type, thus generating tangling).

7. CASE STUDY

To verify and validate the feasibility and effectiveness of the proposed approach it was applied to some java software systems. A prototype tool was developed, on top of the Eclipse JDT platform, to automatically perform and support the several steps of the approach. The tool allows to automatically recover and to instantiate the model described in the Section 4, by finding the roles and associating them to elements of the same kind: both interface fragments or both class fragments.\(^8\)

\(^6\)Since the type fragments are merged by fully qualified names, the merge operation is defined only for type fragments.

\(^7\)The scattered concerns are, within our model, the concerns containing at least more than one type fragment.
Table 3: Summary of the analyzed systems.

<table>
<thead>
<tr>
<th>System</th>
<th>#Classes</th>
<th>#Interfaces</th>
<th>#Methods</th>
<th>LOC</th>
<th>#Roles</th>
<th>#Clustered Concerns</th>
</tr>
</thead>
<tbody>
<tr>
<td>JGraph</td>
<td>71</td>
<td>17</td>
<td>1249</td>
<td>13359</td>
<td>26</td>
<td></td>
</tr>
<tr>
<td>JHotDraw 6.x</td>
<td>298</td>
<td>48</td>
<td>2939</td>
<td>2559</td>
<td>83</td>
<td>30</td>
</tr>
<tr>
<td>JHotDraw 7.x</td>
<td>417</td>
<td>49</td>
<td>1571</td>
<td>63202</td>
<td>96</td>
<td>28</td>
</tr>
<tr>
<td>JCP NIO</td>
<td>461</td>
<td>32</td>
<td>7358</td>
<td>75480</td>
<td>83</td>
<td>27</td>
</tr>
</tbody>
</table>

Table 4: An excerpt of the concerns found in JHotDraw 6

<table>
<thead>
<tr>
<th>Concerns</th>
<th>Clusters Roles</th>
<th>#?</th>
</tr>
</thead>
<tbody>
<tr>
<td>Persistence</td>
<td>Trollable</td>
<td>52</td>
</tr>
<tr>
<td>Undo/Redo</td>
<td>Tunnable, TunnableAdapter, RunnableAdapter</td>
<td>43</td>
</tr>
<tr>
<td>Figures</td>
<td>Figure, AbstractFigure, PolygonFigure, TextFigure, AbstractLineDecoration, CompositeFigure, GeometryShape, DecoratorFigure, RectangleFigure, TextAreaFigure, ConnectionFigure</td>
<td>39</td>
</tr>
<tr>
<td>Event Handling</td>
<td>Tool, AbstractTool, CreationTool, ConnectionTool, ActionTool, SelectionTool, ToolTip, AreaTracker, TextAreaTool</td>
<td>37</td>
</tr>
<tr>
<td>Tooling</td>
<td>AbstractHandle, Handle, LocatorHandle, NullHandle, ChangeHandle</td>
<td>34</td>
</tr>
<tr>
<td>Entities Handles</td>
<td>AbstractHandle, Handle, LocatorHandle, NullHandle, ChangeHandle</td>
<td>16</td>
</tr>
<tr>
<td>Commands</td>
<td>Command, AbstractCommand</td>
<td>16</td>
</tr>
<tr>
<td>Views</td>
<td>StandardView, MainView, DrawingView</td>
<td>13</td>
</tr>
<tr>
<td>Editors</td>
<td>DrawingEditor, MDI_DrawingApplication, SplitPaneDrawingApplication, FigureSelection</td>
<td>12</td>
</tr>
<tr>
<td>Entities Holders</td>
<td>DelegatingResourceId, TextHolder, CommandHolder</td>
<td>11</td>
</tr>
<tr>
<td>Connectors</td>
<td>Connector, AbstractConnector, ChopBoxConnector</td>
<td>9</td>
</tr>
<tr>
<td>Entities Enumms</td>
<td>FigureEnumeration, HandleEnumeration</td>
<td>8</td>
</tr>
<tr>
<td>Painting</td>
<td>Painter</td>
<td>8</td>
</tr>
<tr>
<td>Layouting</td>
<td>Layout, SimpleLayout</td>
<td>8</td>
</tr>
<tr>
<td>Desktop</td>
<td>SplitPaneDesktop, Desktop</td>
<td>6</td>
</tr>
<tr>
<td>Storage Formats</td>
<td>StorageFormat, DefaultStorageFormat</td>
<td>4</td>
</tr>
<tr>
<td>Content Production</td>
<td>AbstractFigureProducer, FigureDataContentProducer, AttributeFigureContentProducer, ColorContentProducer</td>
<td>4</td>
</tr>
<tr>
<td>Drawable elements</td>
<td>Drawing, StandardDrawing</td>
<td>3</td>
</tr>
<tr>
<td>Factories</td>
<td>CollectionFactory</td>
<td>3</td>
</tr>
<tr>
<td>Animation</td>
<td>Animatable, Layoutable</td>
<td>3</td>
</tr>
<tr>
<td>Figure Helpers</td>
<td>RecyclableHelper</td>
<td>3</td>
</tr>
<tr>
<td>Figure Traversing</td>
<td>FigureVisitor</td>
<td>3</td>
</tr>
<tr>
<td>Version Requesters</td>
<td>VersionRequester</td>
<td>3</td>
</tr>
<tr>
<td>Constraints</td>
<td>PointConstraint</td>
<td>2</td>
</tr>
<tr>
<td>Layoutable elements</td>
<td>Layoutable</td>
<td>2</td>
</tr>
</tbody>
</table>

Figure 6: “Connectors” concern internal structure

... robust graph visualization library targeted at extensibility and performances. The second and third systems are two major versions (6.x and 7.x) of the JHotDraw graphical editor framework designed to show good use of design patterns. The last system is the portion of the Java foundation classes related to the I/O handling.

7.1 Discussion of the Results

Due to space constraints, in the remaining of the section, a detailed discussion of the results will be provided just for JHotDraw 6. The role identification step for JHotDraw produced 83 roles, 48 related to interfaces and 35 to classes.

Before the clustering step, roles that were leaves in the type hierarchy graph were filtered out: they are of low interest because their static structure is well modularized. Moreover, as leaves, they do not determine any new concern since they are grouped with their super-roles by the clustering algorithm. After the clustering and a validation step, 30 concerns were identified. Table 4 lists an excerpt of the clustered concerns, ordered by descending number of type fragments (the omitted roles are the ones implemented by the lowest number of type fragments). For each identified concern, the instantiated model provides the internal structure in terms of type fragments, their relationships and their members. The largest concerns, in terms of the number of type fragments are Persistence, Undo/redo, Figures, and Event handling that, all together are associated to 34 roles on a total of 83 (about 40% of the total). As an example, the Figure 6 shows the internal structure of the concern “Connectors”.

7.1.1 Validation

A validation to assess the quality of the clustering was done performing “by hand” the clustering of roles by inspecting the source code and type hierarchy graph. In three
cases the clustering algorithm grouped together roles that were clustered in a different way "by hand".

In Table 4 the roles grouped into the "wrong" cluster are reported in bold and crossed out. The most relevant case is for FigureChangeAdapter role, grouped with the Undoable role but implementing a completely different concern (the Event Handling one). The same was for the Layoutable role that was grouped with the Animatable one. In these cases a manual correction was needed to split the concerns in two different ones (Editors and Layoutable Elements). The wrong automatic grouping was due to the fact that the lexical distance contribution privileged the aggregation because of the equal suffix.

Summarizing, the manual validation step identified 30 concerns; of these, 28 concerns were also identified by the automatic approach and the two missing ones (Figure Selection and Layoutable) were obtained removing the roles clustered by mistake from the bad clusters (Undo/Redo and Animatable). The FigureChangeAdapter role that was clustered with Undo/Redo by a mistake was instead moved to the Event Handling concern.

### 7.2 Crosscutting Analysis

Given two concerns, their type fragments can be compared searching from tangling and scattering. In particular, tangling occurs when two concerns declares a member (field or method) or a relationship (inheritance, implementation or containment) into a type fragment with the same fully qualified name. Scattering for a concern occurs when its members are declared in several type fragments of the system. In order to find crosscutting among two concerns, scattering and tangling are not sufficient. Indeed, they must be related: scattered fragment of each concern must be tangled in at least one module. The model instance is traversed to verify which concerns satisfy such properties and for what elements.

Figure 7 reports the crosscutting matrix for JHotDraw resulted from the crosscutting analysis (the matrix just reports an excerpt of all the identified concerns; due to the space constraints just the main concerns are listed). For JHotDraw 6.0, the concerns Persistence and Event handling are responsible for the crosscutting relationships with the largest extent. In particular Persistence crosscuts 8 other concerns; the most relevant crosscutting is with the concerns Figures (31 type fragments), Event Handling (11 type fragments), and Layouting (5 type fragments). This information can be used to analyse the most entangled concerns when planning a refactoring task. For each cell of the matrix the type fragments involved in the crosscutting relationships are known (in terms of members and relationships). Therefore an analysis of most scattered concerns along with the identification of where in the system such components are to be found could be easily conducted. The reported results were verified and validated by code inspection. The analysis of the crosscutting relationship between Commands and Undo/Redo (due to 6 fragments) showed that the fragments of Undo/Redo are inner class fragments nested in the class fragments of the Commands concern. This actually generated tangling: among the members of the enclosing fragments (associated to the Command concern) there are the inner class members associated to the Undo/Redo concern (because of the inheritance towards UndoableAdapter or the implementation of Undoable interface, both roles of Undo/Redo concern).

The model instance can be examined also to compare concerns: for example to gather information on the diffusion of a concern into another or to look at the type fragments that are common to two or more concerns (performing intersection of their sets of type fragments), in order to focus on crosscutting only for selected components or concerns. As an example in Figure 8 is showed the portion of concern Connectors in common with Persistence one. This tell us what are the components of the Connectors concern in which there are also elements of the Persistence one (and vice-versa). More complex analysis on model can be performed, filtering on type fragments, on types or packages and focusing on the concerns of interest. It’s also interesting to compare this kind of (class level) analysis with others at statement level. In particular in [12] the analysis performed using fan-in metric shows that there is a consistent invocation of the Undo concern from within Command concern elements (that are also tangled with other functionality). Our approach obviously miss such kind of dynamic crosscutting but it is able to find, for the

---

8Dynamic crosscutting concerns crosscut the run-time control flow, specifying additional behaviour that is executed when predefined events occur (during program execution).
same two concerns, static crosscutting due to their structure: as discussed before each concrete Command has an inner class playing the Undoable role. The two approaches, from this point of view, have a high degree of complementarity: they could both be used to build a complete crosscutting concern representation. Looking at the matrix in Figure 7 the concerns that are not involved in static crosscutting relationships with any other concerns (and hence are, structurally speaking, well modularized) can be identified too. For instance looking at Entities Handling, Desktop and Painting it can be noted that they are well modularized being involved in very few crosscutting relationships. Of course this means that they are well modularized only from a static point of view. The analysis of method bodies is likely to reveal any dynamic crosscutting due to scattering and tangling of statements related to different concerns.

7.3 Limits of the Approach

To the aim of building a complete crosscutting representation, one of the main limit of the approach regards the granularity of the considered elements. Since statements in methods are not taken into account dynamic crosscutting is not detected and not represented. Such limitation could be addressed integrating dynamic analysis techniques within the proposed static approach. This is an important point: many dynamic approaches suffer the lack of a precise analysis of structural crosscutting. The integration of dynamic analyses with the proposed static approach could provide not only a more complete crosscutting representation (that takes into account of both structural and behavioral crosscutting) but also an improvement of the precision of dynamic crosscutting mining approaches. For this reasons future work will be devoted to experiment with the integration of dynamic crosscutting mining techniques.

Another important limit of the approach is related to the assumption made on the finest concern granularity. We assumed that, at class level, the lowest granularity of the concern is a single role implementation (made up of the type fragments implementing it). We verified such assumption in the analysed systems finding very few cases in which it was not verified. In particular in JHotDraw it resulted not verified only for a single case: the DrawingView class that does not implement Figure but re-declare all the interfaces of Figure by itself. This is due to a bad design since DrawingView behaves like a Figure but it is not substitutable with it. We believe that such cases are more frequent for systems that are not designed and developed following good OO practices and doesn’t enforce the “programming to the interface” rule. More experiments should be performed to assess the range of validity of such assumption.

8. RELATED WORK

Aspect mining techniques aim to make as much as possible automatic the identification of crosscutting concerns.

Most techniques perform source code analyses and manipulations (static techniques ) while others focus on executing the code manipulating the data gathered at run-time (dynamic techniques). There are also hybrid techniques that use both structural and run-time information to improve the quality of those approaches.

In [22] Serban et. al. presented a k-means based clustering algorithm specific to aspect mining. Their approach consid-
10. REFERENCES


