2015 IEEE 23rd International Conference on Program Comprehension

Exploring the Use of Concern Element Role Information in Feature Location Evaluation

Emily Hill
Drew University
Madison, NJ, USA
emhill@drew.edu

David Shepherd
ABB Corporate Research
Raleigh, NC, USA
david.shepherd@us.abb.com

Lori Pollock
University of Delaware
Newark, DE, USA
pollock@cis.udel.edu

***Abstract*—Before making changes, programmers need to locate and understand source code that corresponds to specific functionality, i.e., perform concern or feature location. Numerous concern and feature location techniques have been proposed, but to the best of our knowledge, no existing techniques or evaluations report information on *what role* a code element plays in the larger concern. In this paper, we report on two case studies that investigate two hypotheses on how evaluation studies of concern location techniques can be strengthened by utilizing concern role information: (1) by increasing agreement among human annotators for gold set establishment and (2) by providing richer information about the elements ranked as relevant by concern location techniques, which could help further improve the tools.**

We conducted a case study of 6 Java developers annotating 3 concerns with role information. When the developers understood the task description, pairwise agreement increased by 20%, 25%, and 135% for the 3 concerns over a prior concern location study without role information. Our findings also suggest that there may be core element roles that need to be annotated by humans, but that the remaining roles may be automatically derived, which could facilitate more reliable concern location benchmarks in the future. We also conducted an exploratory study of the element roles represented in results returned by a state of the art feature location tool. The results of these two studies suggest that integrating concern element role information into evaluations can help to strengthen both the gold set establishment and the analysis of results returned by various tools.

I. Introduction

During software maintenance, programmers spend considerable time locating and understanding code related to the maintenance task. Numerous feature and concern location techniques have been developed to help ease this developer burden [1], [2], [3], [4], [5], [6], [7], [8], [9], [10], [11]. Although the notions of feature and concern are often viewed as synonymous, in this work, we use the broader definition of a concern [12]: "anything that stakeholders of a software project may want to consider as a conceptual unit." These include features, nonfunctional requirements, design idioms and implementation mechanisms such as logging and caching.

The variety of feature location techniques has motivated research to better support evaluation and comparison [13], [14]. Existing evaluations only consider an element's *relevance*, and not what *role* it plays in the concern. However, some elements play a key role in implementing the concern's functionality, whereas others are important in understanding how the concern interacts with its surrounding source code context.

In a previous short paper [15], we proposed the notion of differentiating program elements of a concern by the roles that they play with respect to the concern's implementation. We investigated the feasibility of such roles by manually analyzing a small set of concerns, focusing on concerns that can be described by a precise verb phrase (VP), which not only includes a verb and direct object, but also an indirect object (e.g., not just "add a song", but "add a song to a playlist"). We proposed a classification of roles that program elements play in a concern and manually examined whether individual annotators in a gold set for concern location agree on certain roles more than others. Our results indicated that human agreement aligns with some of the roles that we proposed.

Based on the initial results, we believe that evaluation studies of concern location techniques can be strengthened in two ways by utilizing concern role information: (1) by increasing agreement among human annotators for gold set establishment and (2) by providing richer information about the elements identified and ranked according to relevance by concern location techniques, which could help to direct improvements to the tools in the future.

This paper provides significant contributions beyond our preliminary work by:

1) conducting an in-depth case study focused on whether knowledge of concern element role information increases annotator agreement, and
2) exploring whether annotating feature location results with concern element roles reveals new insights to further improve feature location tools.

We refined the initial concern element roles through the authors' manually analyzing 8 additional concerns. This paper also advances the understanding of concerns beyond results from prior concern annotation studies. Robillard, et al. [16] investigated the agreement among developers, while Murphy, et al. [17] compared the opinions of a newcomer with the code's author. Neither study investigated the nature of a concern's elements or differentiated among the different roles that program elements might play in comprehension.

II. Motivation

In this section, we describe our insights from prior work [15], [16] as well as give an example concern showing that not all concern elements are included for the same reason.

978-1-4673-8159-8/15 $31.00 © 2015 IEEE
DOI 10.1109/ICPC.2015.23

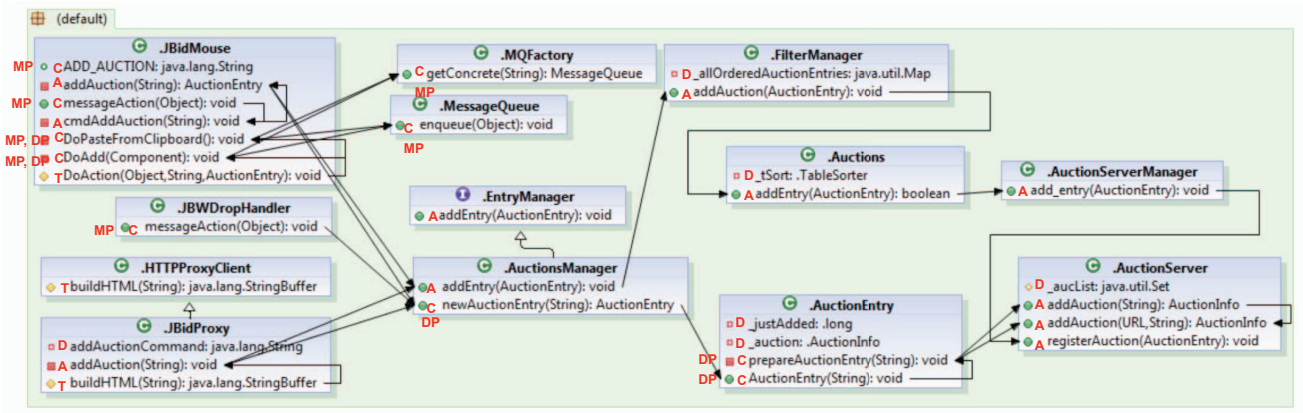


Fig. 1: Program elements and roles for the example "add new auction to local system" concern. Each element is annotated with its role in red. Edges indicate structural relationships such as calls (solid-head arrows) and inheritance (open-head arrows).

A. Insights from Previous Studies

A prior empirical study of concern location collected and analyzed 3 independent annotations for 16 search tasks, annotated by 23 distinct developers [16]. From this work, we make the following observations that motivate the current work.

Low average agreement. The average pairwise agreement of any two annotators was very low—just 34%, and the range was 0–82%. Because this experiment was conducted with experienced, professional developers, we do not believe that agreement would significantly increase if this experiment were repeated with a different set of developers. We also do not believe agreement would significantly improve this experiment were repeated with a different set of concerns, since the experiment involved 16 concerns from 4 different systems.

Low agreement due to loose definition of concern. One main source of disagreement appeared to be subjectivity surrounding what is a concern. Some subjects considered only methods that actually modified the target data structures, whereas other subjects annotated UI elements, such as a button that was pushed to trigger the concern. We believe that we can reduce this source of disagreement by defining roles of individual concern elements and educating annotators about the roles. In prior work, we studied a subset of these concerns [16] and found that an initial version of our roles helped to explain annotator disagreement [15].

Low agreement due to not following structural paths. Another source of disagreement involved control flow "islands", where the related concern elements were either not connected by control flow or the connections were difficult to recognize, such as complex data flow or interwoven library code. We observed that when annotators found one item in a control flow group, they would typically find others, but might not make the jump to discover other related elements if the structural connections were not as obvious or well-supported by the IDE (e.g., through extremely long control flow paths or complex data flow relationships). We noticed that these islands often coalesced at the beginning of a concern (i.e., where it was triggered) and at the end (i.e., where the main action work was done). We believe these roles will naturally lead concern annotators to include *both* the beginning and ending elements of a concern, rather than omitting one.

B. Example Annotated Concern

We use a concern in jBidWatcher to demonstrate how program elements take on different roles. jBidWatcher is an auction bidding, sniping, and tracking tool for online auction sites such as eBay or Yahoo. It includes a unique sniping feature that allows the user to place a bid in the closing seconds of an auction. Before a user can bid on an auction, they must add the auction to the user view and system data structures. We define the verb phrase (VP) for this concern to be "add new auction to local system." Figure 1 shows the methods and fields that implement this concern.

Nodes are added to the concern for a number of reasons. Some methods trigger the execution of the concern, such as the "do" methods in JBidMouse or the messageAction method in JBWDropHandler. Some methods are relatively generic, such as MessageQueue and MQFactory, which process all user-initiated actions. Although generic, these methods communicate (or *connect*) information from the triggers to methods that implement the concern's actions and are critical to comprehending the control and data flow within the concern.

The action of adding an auction culminates in updating several internal data structures: the set of auction entries managed by the system (FilterManager._allOrderedAuctionEntries), the table of auctions displayed by the user interface (Auctions._tSort), and the list of auctions managed by the internal auction server (AuctionServer._aucList). These are some of the *data* or results of the concern.

Although creating a new AuctionEntry object is not obviously part of the add auction concern, since it can be precisely described by its own VP, creating a new auction object culminates in adding that AuctionInfo to the system with the addAuction methods in class AuctionServer, and thus is still relevant to adding an auction.

This example demonstrates that a program element may be included as part of a concern for different reasons, and

```
/**
 * @brief Auctions must be registered when they are added, so that
 * the auction server can keep a list of auctions that it is
 * managing.  This is used when storing out the list of auctions per
 * server.
 *
 * @param ae - The AuctionEntry to add to the server's list.
 */
public void registerAuction(AuctionEntry ae) {
  synchronized(_aucList) { _aucList.add(ae); }
}
```

Fig. 2: Example action node that uses an API library call to implement the VP, "add auction," despite having a seemingly unrelated method name.

```
public boolean parseAuction(AuctionEntry ae) {
 loadSecondaryInformation(_htmlDocument);
 try {
   if(JConfig.queryConfiguration("show.images", "true").equals("true")
     if(!_no_thumbnail && !hasThumbnail()) {
       MQFactory.getConcrete("thumbnail").enqueue(this);
     }
   }
 } catch(Exception e) {
   ErrorManagement.handleException("Error handling thumbnail loading",
 }
```

Fig. 4: Example trigger node for the "get thumbnail" concern that is not directly triggered by the user through the UI, but indirectly triggered through adding or loading an auction.

that implementing the main action is only one reason. In Figure 1, some concern elements are included because they trigger, or initiate, the concern action ('T'), whereas other elements execute the action of adding an auction by creating AuctionEntry objects and adding them to the internal data structures of the system (indicated by 'A'). The data structures and fields that provide input data to the concern's action or are updated as a result of the action are indicated by 'D', with connecting elements that communicate between different parts of the concern labelled 'C'.

III. Concern Element Roles

We are targeting concerns that can be precisely described by a verb phrase (VP) that includes a verb, direct object, and supporting indirect objects or modifying phrases. The action, or verb, is key to determining whether program elements (i.e., methods and fields) should be included in the concern. Direct objects are useful in identifying fields, since field names do not typically refer to actions or verbs. The indirect objects or modifying phrases help differentiate nodes to be included in the concern by indicating concern boundaries and helping to determine when one concern has become another.

Even if a concern can be precisely described by a VP, sometimes the concern's implementation does not use the VP's exact words. For example, the action *delete* might be implemented as the synonym *remove*, and the concept of *adding* may include *creating* a new item. An objective definition of concern roles should not be so fragile as to require exact word matching in the source code. In the definitions below, we use the concept of *similarity*, where two words or phrases are semantically equivalent (i.e., synonyms).

To develop our concern element role definitions, the first two authors manually located 12 concerns described by well-defined VPs in 3 open source Java programs [16]. The first 4 concerns informed our initial role definitions of action, trigger, result and connector nodes [15]. We further refined these definitions by annotating 8 additional concerns. For each concern, we agreed on a descriptive VP before independently locating the concern and annotating role information. We then discussed our annotations until we reached a consensus both in what elements implement the concern and their roles. We then briefly justified the element's role and it's relevance to the concern. After locating and annotating these 8 concerns, we reviewed our definitions and carefully analyzed all our annotations, using the justifications to help ensure our definitions were consistent. From this research process, we defined 5 distinct roles of nodes in a structural representation of a concern: action, trigger, initializer, data, and connector.

Action Nodes (A): Any method that directly implements the concern's verb phrase. The name need not explicitly refer to the VP, since method naming can be arbitrary (especially for overridden methods) or buggy [18]. Action nodes can also serve as trigger points (see trigger nodes, below).

In practice, action nodes can implement the VP in multiple ways. For instance, a VP can be implemented by directly manipulating primitive types (e.g., for the phrase "remove person" a method could set an array value to null to remove that person). Alternatively, a method can call external libraries to implement the VP (e.g., for the phrase "connect to database" an action node could call the connect method in a database library). Finally, a method can call other methods within the same system (e.g., for the phrase "open file" calling FileOpener.open). In this situation, the calling node is only considered an action node if the call and surrounding code are the major theme of the method (i.e., not just a side effect supporting the method's unrelated main action).

In Figure 1, most of the action nodes are easily determined from their names, which have a strong relationship to the VP, "add new auction to local system". The exception is registerAuction in AuctionServer, which is responsible for updating the list of auctions managed by the internal auction server (_aucList), and very clearly contains the phrase "add auction" in its method body (see Figure 2).

Trigger Nodes (T): Any method or field that triggers the execution of an action node, either directly or through connector nodes, but does not implement the concern's VP. Trigger nodes usually contain a reference to the concern's VP, and serve as an entry point into the concern from outside the concern (see Figure 3). If a node is a trigger for the concern but also implements the concern's VP, then the node is considered to be an action node.

Trigger nodes are typically manifested in two ways depending on whether the trigger is part of the user interface (UI). First, methods that respond to UI events, such as button presses, are often trigger nodes as they typically send a message to a command queue or signal an event listener to

```
protected void DoAction(Object src, String actionString, AuctionEntry whichAuction) {
  Component c_src;
  if(actionString.equals("Save")) DoSave(c_src);
  else if(actionString.equals("Load")) DoLoad(null);
  else if(actionString.equals("Add")) DoAdd(c_src);
  else if(actionString.equals("Delete")) DoDelete(c_src, whichAuction);
```

Fig. 3: Example trigger node for many user-triggered events in jBidWatcher, including adding and deleting an auction.

execute an action, yet do not implement the action themselves. Similarly, for non-UI features, the main public methods for an API often serve as a trigger point. When outside clients call this API, a command queue is often notified, but the trigger method does not implement the concern itself. Another non-UI example includes enqueuing a message on an event queue, to be handled elsewhere in the system (see Figure 4).

Initializer Nodes (I): Any method or field that initiates a relationship between a trigger node and an action node. Unlike the other nodes, initializers are executed *before* the concern is executed, rather than after the concern is triggered. This is most often manifested in methods that register UI listeners (e.g., connecting a UI button to an action handler), command queue listeners (e.g., listening for an event), or initializing an event message to be sent when the concern is triggered.

Data Nodes (D): Data is passed into, within, and out of a concern using a number of mechanisms. Data nodes include any fields and simple get or set methods involved with passing information into, within, or out of a concern.

We have identified 4 broad categories of data nodes that may be used to implement a concern. *Result nodes* include any field that has a similarity to the concern's VP that is altered by an action method node as a result of an action related to the VP. In contrast, *Exception result nodes* can occur when an action ends with an unintended result, such as an exception being thrown. *Input nodes* include any field that has a similarity to the concern's VP that is used as input to an action, either directly or through a series of trigger and connector nodes. Finally, some fields are used as both data input and results within a concern, and may be changed many times throughout a concern's execution. Typically, such fields are *flags* that indicate different stages of a concern's execution. Flags typically have similarity to the concern's VP.

Connector Nodes (C): Any method or field that structurally connects two concern nodes in the program structure graph, and is used to pass data between the two identified concern nodes. Connector nodes most often connect triggers to actions. They are often manifested as pure message passers **(MP)** by simply passing data or messages within the concern, and/or as data preparers **(DP)** by modifying and preparing data for use in subsequent action methods. Connectors can include generic methods that are shared by many concerns in a system. In this case, the connector's parameters will have a strong relationship to the verb or direct object of the concern's VP. Otherwise, generic methods are typically not included.

In Figure 1, methods that transform the data or create objects from the data to be added from the add auction request trigger are considered to be DP connectors, such as preAuctionEntry, newAuctionEntry, and AuctionEntry. In contrast, methods that simply pass along auction data without modifying it or indicate that the add auction event has occurred are considered to be MP connectors (e.g., ADD_AUCTION, messageAction, getConcrete, and enqueue). Some methods are responsible for transforming the data as well as passing along a message, and are labelled as DP and MP connectors.

IV. Case Study: Annotator Agreement

We hypothesize that knowledge of the roles defined in the prior section will increase agreement among developers annotating concerns, which will help both evaluation of concern location techniques and provide information for deeper understanding of concerns during software maintenance tasks. Our study investigates the following research questions:

- RQ1 Does knowledge of roles increase annotator agreement?
- RQ2 How clear are the role definitions?
- RQ3 Do concerns share similar distribution of roles?

To investigate the above research questions, we compare with concerns annotations from a prior study where each concern was annotated by 3 different developers [16]. Because concern location is a difficult and time-intensive task, especially on an unfamiliar code base, we chose a subset of the concerns to be annotated by a new set of developers with knowledge of roles.

A. Study Design

From our experience in locating and annotating roles for 12 concerns, we appreciate how difficult and time consuming annotating concerns can be. We attempted to streamline the process and reduce developer annotation effort wherever possible. In the prior study [16], annotators were given no training and asked to spend 20-30 minutes locating a single concern. In contrast, for the current study, we gave an hour long training session and asked participants not to exceed 5.5 hours in locating and annotating the 3 concerns. Thus, we allowed participants close to two hours to locate and annotate each concern, rather than 30 minutes.

Subject Concerns: From the set of previously annotated concerns, we randomly selected three concerns from the same program to limit the amount of initial setup work for the participants. The tasks needed strong verb phrases, and the prior version of the program needed to be executable (e.g., jBidWatcher is dependent on an external web site, eBay, and prior versions do not currently execute as intended). This left us with two possible programs: jajuk and freemind. We randomly chose jajuk and 3 of its 4 concerns.

Number of Software Developers in Study					
No. Years	**Programming Experience**	**Industry Experience**	**Perform Maintenance**	**Perform Maintenance on Code Not Authored**	**Frequency**
10+ years	3	1	3	0	**Daily**
5-9 years	3	3	2	1	**Weekly**
1-4 years	0	0	1	4	**Monthly**
< 1 year	0	2	0	1	**Yearly**

TABLE I: Participant Developer Characteristics: number of years of experience (left) and maintenance frequency (right)

The randomly selected program, jajuk, is an open source Java music organizer similar to iTunes. The three randomly selected concerns were given in the same order to all participants: sorting a collection by genre (C8), adding a song to a playlist by dragging and dropping (C7), and playing a song from the Logical Perspective pane (C5). The program and concern descriptions given to participants were not changed in any way from the original study [16].

Participants: Six participants with extensive Java programming and/or industry experience were recruited through author contacts. Table I shows characteristics of our study participants. The distribution of years of programming and industry experience for each subject is displayed on the left of the table, and the frequency they perform maintenance tasks to the right.

Procedure: All six participants attended a 1-hour webinar tutorial on how to locate concerns and annotate the elements with roles. All of the definitions were accompanied by examples taken from jBidWatcher, which had no relation to jajuk, the program used in the study. Participants were encouraged to ask questions during the webinar and when annotating.

In addition to the webinar, participants were given the slides for the webinar, a written description of the roles similar to Section III, the task descriptions, and a spreadsheet template to record their annotations. The jajuk source code was distributed as an exported Eclipse project, and Eclipse was the recommended IDE (although its use was not required). After the webinar, participants were asked not to exceed 5.5 hours in locating and annotating the 3 concerns. At the conclusion of the study, participants were asked to complete an exit survey that collected information about the difficulty of the 3 tasks, the clarity of the role definitions, as well as the demographic information depicted in Table I.

The experimental materials used in the study are available at http://lee.cs.montclair.edu/~hillem/cer/, including a video recording of the webinar.

Measures: To investigate the three research questions, we employed a number of measures. For analyzing agreement among annotators, we compared pairs of concern annotations using Jaccard's coefficient for comparing two sets. Given two sets of program elements A and B:

$$agreement = \frac{|A \cap B|}{|A \cup B|}$$

We apply two sample t-tests to determine significant differences between the pairwise agreement of no roles and roles. When subsets used in a comparison are not normally distributed, or the variances are non-homogenous, we apply the non-parametric Mann-Whitney U-test instead.

While percent agreement gives an estimated agreement overlap between any two participants, it does not account for spurious agreement due to chance. Reliability statistics such as kappa or alpha are commonly used in the content analysis community to compliment percent agreement [19], [20]. We use Fleiss's kappa in our analysis, rather than the more traditional Cohen, since we have more than two annotators. We restrict the set of annotations to those program elements that were annotated by at least one subject in either study. Because our second study with role training has more annotators than the original study, it is invalid to use an alpha test such as Cronbach or Krippendorff, since increases in alpha are more likely to occur with more participants.

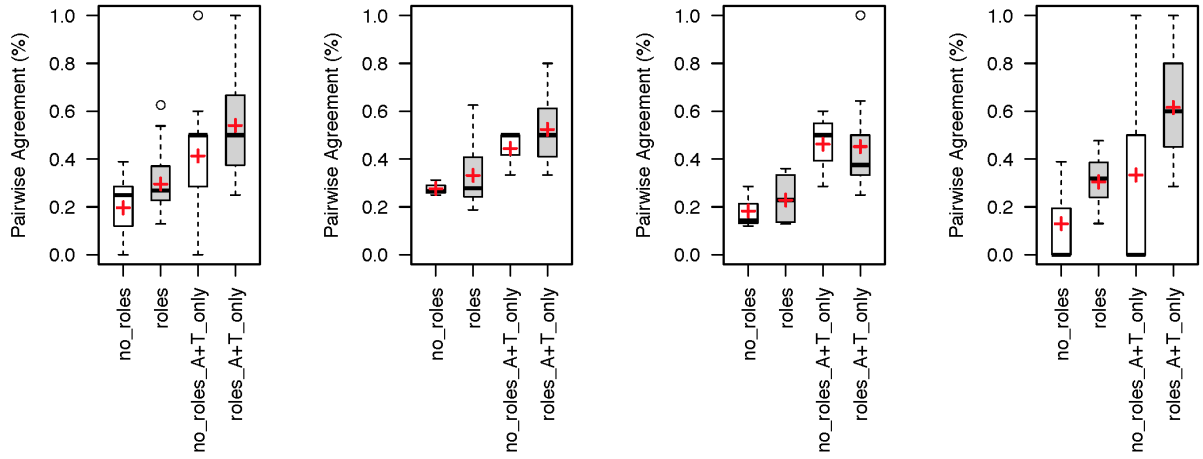
To determine how clear the role definitions are, we analyzed the number of annotation pairs categorized by role. We assume that if pairs of participants tended to use the same role when including an element in a concern, then the role definition is clear. If pairs of participants both include the same element with differing roles, we assume the definitions to be unclear. In addition, we used data from the exit survey, which collected responses on a 7-point Likert scale, to determine the relative clarity of the definitions for particular roles. In analyzing the relative distribution of roles across multiple concerns, independent of concern size, we calculated the percent of annotated elements categorized by role for each concern.

Statistical Power and Outliers: Although a sample size of 9 participants may appear small for statistical testing, we are not performing any statistical analysis on each subject's data as a whole; rather, we are analyzing agreement among pairs of participants for each concern (40 with roles, 9 without), as well as the number of annotations (308 total) and the number of annotated program elements (149). Thus, our data sample sizes are sufficiently large to perform statistical analysis.

There was one annotation for which a participant located a different concern from all the 8 other participants across both studies. Instead of locating the code to add a song to the playlist from the logical tree view (C7), the participant located the code for adding a song to the playlist from the table view. While the VP is similar, these concerns execute different code, only overlapping in two methods. The remaining concern elements were not present in any of the other 8 annotations. Because this subject clearly misunderstood the task description, we have removed this particular concern annotation from our results and analysis.

Concern	Description	All elements			Actions & Triggers (A+T) only		
		Agreement w/o Roles	*Agreement w/Roles*	*Percent Increase*	*Agreement w/o Roles*	*Agreement w/Roles*	*Percent Increase*
C8	Sort collection by genre	27.64	33.18	20.04%	44.44	52.35	17.80%
C7	Add song to playlist by drag and drop	18.29	22.85	24.93%	46.19	45.21	-2.12%
C5	Play song in Logical Perspective	12.96	30.44	134.88%	33.33	61.57	84.73%

TABLE II: Pairwise agreement without (w/o) and with (w/) foreknowledge of roles and the percent increase.



(a) Agreement overall (b) C8: Sort by genre (c) C7: Add to playlist (d) C5: Play song

Fig. 5: Percent of pairwise agreement between no knowledge of roles (no_roles), knowledge of roles (roles), and the subset of action and trigger roles only (A+T_only). The box plots show the median, quartiles, and mean (+).

	C8		C7		C5	
	None	Roles	None	Roles	None	Roles
All	0.223*	0.207*	0.102	0.16*	-0.0002	0.274*
A+T only	0.462*	0.233*	0.459*	0.173*	0.192	0.354*

TABLE III: Fleiss's kappa reliability measure without (None) and with foreknowledge of roles (Roles) for each concern. Starred* values are statistically significant ($p < 0.05$).

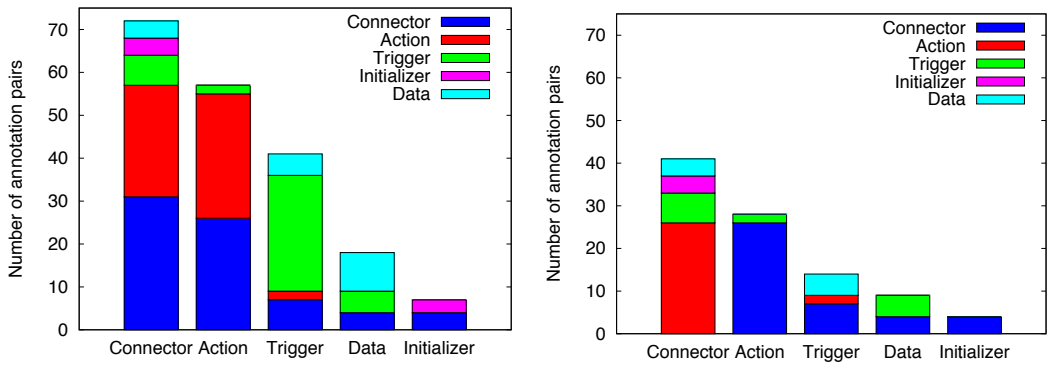
B. Results and Analysis

In all, the 6 participants annotated 209 elements with roles, along with 99 annotations from the 3 participants from the prior study without roles. The most common annotation was connectors (86), followed by actions (44), data (41), triggers (25), and initializers (13). Two of the six participants never annotated an element as an initializer; each participant annotated at least one element with the remaining roles.

RQ1: Does knowledge of roles increase agreement?: To analyze agreement among annotators, we compared pairs of concern annotations using Jaccard's coefficient for comparing two sets (see Section IV-A). Table II presents an overview of the pairwise agreement of the three concerns annotated without roles (from the original study [16]) and with roles (our current study). Figure 5 presents this same data as a box plot where the thick horizontal line represents the median, the plus represents the mean, and the box indicates the interquartile range (i.e., the difference between the first and third quartiles). The whiskers extend to the maximum and minimum values of the range, with outliers indicated by ○. Figure 5 includes pairwise agreement for the A+T_only subset of program elements. Based on participant feedback that action and trigger roles were important to understanding a concern, we measured the agreement for the subset where at least one participant marked an element as an action or a trigger. Agreement improves both for the original data set (no_roles) and the current data set (roles) when only analyzing actions and triggers.

To verify these conclusions, we apply the two sample t-test and the non-parametric U-test to determine significant differences between the pairwise agreement with and without knowledge of roles. Knowledge of roles has significantly higher agreement than no_roles (t-test, $p < 0.05$). When only considering elements categorized as actions or triggers by at least one participant, the action and trigger subset with knowledge of roles (roles_A+T_only) significantly outperforms roles when all elements are considered (U-test, $p < 0.05$). No other comparisons were statistically significant.

We observe similar trends when considering inter-annotator reliability. Table III shows Fleiss's kappa for the three concerns. For concerns C7 and C5, reliability increases with knowledge of roles. Furthermore, restricting the set of annotations to just potential actions and triggers uniformly increases reliability. Because the agreement when considering only actions and triggers increased significantly, both statistically



(a) All pairs

(b) Only pairs that mismatch

Fig. 6: Number of annotation pairs categorized by role. Chart (b) is a subset of the data presented in chart (a).

and practically, we believe these two roles are particularly important in understanding a concern.

RQ2: How clear are the role definitions?: To investigate whether the roles are clearly defined, we analyze pairwise agreement of role annotations. For every pair of annotations in the study, we take the intersection of concern elements and count the number of role annotation pairs (action-action, action-trigger, action-connector, etc.). We hypothesize that if both participants mark a program element with the same role, then that particular role's definition is clear. Concretely, we measure the number of matching roles and the number of mismatching roles for each pairwise comparison across all participants. We count a match if both participants include the program element and mark it as the same role (e.g., P1 action, P2 action), and a mismatch if they both include the same program element but mark it a different role (e.g., P1 action, P2 trigger).

Figure 6(a) shows the number of annotation pairs for each role, categorized by role. For example, consider the 'Action' bar. Given the key in the top right, we can see that 26 actions were also categorized as connectors, 29 elements were categorized as action by at least 2 participants, and 2 actions were also categorized as triggers. From this information, we can determine the percent of matching annotation pairs (action-action, trigger-trigger, etc) and non-matching pairs (action-trigger, action-connector, etc). Notice that connectors, shown in the first column, appear as part of 31 matching pairs but also in 41 non-matching pairs, giving them the highest percent of mismatches (57%). We also see that actions, triggers, and data have lower mismatch rates of 49%, 34%, and 50%. Note that data also has a mismatch rate of 57%, but that this rate is dominated by connectors.

Figure 6(b) shows only the subset of mismatching role annotation pairs, with matching pairs removed. Notice how connectors (dark blue) dominate the mismatch counts for action, trigger, data, and initializer. The number of mismatches decreases significantly when connectors are not considered, reducing to 7%, 21%, 35%, and 0%, respectively. When mismatches caused by connectors are disregarded, participants most often agree on actions and triggers, which likely indicates that these role definitions are clear. Note that both data and initializers have low mismatch rates when disregarding connectors, but have insufficient data to draw further conclusions.

Survey Question	μ	$\pm\sigma$
How clear were the study instructions?	6.17	0.75
How helpful was the webinar [training]?	6.00	0.63
How clear was the description of:		
concern element roles?	4.67	1.03
action nodes?	5.67	1.03
trigger nodes?	5.83	0.98
initializer nodes?	5.33	1.21
data nodes?	5.00	1.67
connector nodes?	3.83	1.47
How would you rate the difficulty of:		
Task 1 [C8]?	4.50	1.38
Task 2 [C7]?	4.33	1.21
Task 3 [C5]?	4.33	1.37

TABLE IV: Mean (μ) and standard deviation ($\pm\sigma$) for survey responses on a scale of 1 to 7, where 1 is low (unclear, unhelpful, or easy) and 7 is high (clear, helpful, or difficult).

We believe connectors are most often involved in a mismatch because the definition for connectors is not as clear as that of the other roles. This hypothesis is further supported by the participant exit survey data. Table IV shows the results of the closed-form questions in the survey, which were evaluated on a 7-point Likert scale. This survey investigated the clarity of each role's definition as understood by the participants. Notice that when asked, "How clear was the description of connector nodes?" participants responded with the lowest mean score of the entire survey, at 3.83, whereas the average of the other roles' scores was 5.46. Participants clearly had trouble understanding the connector role definition.

We also asked participants open-ended questions about how the concern element role definitions could be improved. Four of the participants wanted a more precise definition of connector nodes. One participant wanted connectors better differentiated from action nodes, while another wanted connectors to be differentiated from triggers, and a third wanted better examples to differentiate between MP and DP connectors. These answers

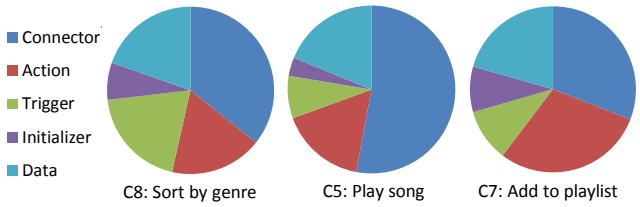


Fig. 7: Percent of annotated elements by role for each concern

reinforce our quantitative findings regarding the clarity of the connector role definition.

RQ3: Do concerns share similar distribution of roles?: The final research question investigates whether the three concerns in the study share similar role distributions. In contrast to RQ2, where we counted *pairs* of role annotation categories, to answer this question, we counted the *raw number* of role annotations per category for each concern, shown in Figure 7. Note that we counted the annotations irrespective of agreement. For example, if the same element was annotated as an action, trigger, and initializer by different subjects, it will count toward the pie slices for all three of those roles. Figure 7 shows that the distribution of roles is relatively consistent, with actions representing 21% of the annotations on average, connectors 40%, triggers 13%, data 20%, and initializers 7%.

Notice in Figure 7 that both C7 and C8 show a similar role distribution. The largest variance is the balance between actions and triggers. In contrast to the other two concerns, C8 has a higher percentage of triggers due to its trigger implementation. While C7 has a single triggering user action (i.e., dragging a song to the playlist), C8 has three different triggering actions. Varying number of triggers is exactly the kind of variation we expect across concerns. It is often the case that the same concern can be triggered through many different UI elements (e.g., a menu item and a button).

Notice that C5, while otherwise similar in distribution to C8 and C7, has a larger percentage of connectors. In many cases, we observe that much of the connector code flows through libraries, which are not included in our concern annotations. However, C5 involved a FIFO class which processed play commands, and thus many of C5's connectors were non-library elements, inflating its connector percentage.

We hypothesize that with a large enough sample of concerns, we would observe a role annotation distribution close to the averages seen across Figure 7. Despite some minor variations, the distributions of roles for these three concerns appear relatively consistent. We plan to confirm this hypothesis in future work by investigating if this trend holds true across multiple Java programs.

C. Threats to Validity

In this study, we compared pairwise agreement and reliability results from the current study using knowledge of roles and training, and a prior study [16]. In the prior study, annotators were asked to spend 20-30 minutes locating a single concern, and no information about roles was provided. In contrast, for the current study, we gave an hour long training session on role definitions and asked participants not to exceed 5.5 hours in locating and annotating the 3 concerns. Thus, we allowed participants close to two hours to locate and annotate each concern, rather than 30 minutes, to accommodate for the additional role information used during annotations.

The challenging and time consuming nature of the task makes it difficult to replicate over many participants, concerns, and software systems. To reduce participant workload as much as possible for this difficult task, we have chosen to focus on a smaller set of concerns (3 from the same software system) and increase the number of annotators per concern from 3 to 6. However, the limited sample precludes overgeneralizing from these results, and the conclusions of our study may not generalize to other Java programs or other programming languages.

Because the concerns were given to the participants in the same order, it is possible that learning effects could be present. Based on the exit survey data in Table IV, we observe that participants rated the tasks as having relatively uniform difficulty. As expected, the first task is slightly more challenging as the participants learn to locate concerns and annotate their roles. Another potential issue is whether certain concerns are much harder to locate than others. In addition to no learning curve, the results in Table IV show that all the tasks were rated with similar mean difficulty and similar variance. Finally, the equivalent means between tasks 2 and 3 demonstrate that locating concerns within the same program does not necessarily improve with more experience.

D. Discussion

As discussed in Section IV-B, agreement is higher among participants with knowledge of concern roles when performing concern location. While this serves to validate our role definitions, we were surprised by the degree to which participants thought our connector role definition was unclear, as shown in Table IV. In the exit survey, they rated this role definition the lowest (below the neutral score of 4), and several asked for improvements on its definition. One subject commented:

> "I think connector is too broad. It seems like a catch-all. The other categories seem like the boundaries of the graph. The entire center of the graph is just 'connectors'."

A look into the annotated concerns supports this view. In our experience, triggers typically demarcate the leading edge of the concern, whereas actions tend to indicate the ending boundary. For example, consider the "add auction" concern in Figure 1. The trigger `JBidProxy.buildHTML()` represents the beginning edge of the concern, and is connected via control flow edges to actions in the `AuctionServer` class. For instance, this trigger is connected to `addAuction(String)` via several connectors like `AuctionsManager.newAuctionEntry()`. Similarly, although harder to follow as dataflow edges have been omitted from the diagram, the trigger `JBidMouse.DoAction` uses the connectors `DoAdd`,

`MessageQueue.enqueue`, and `messageAction` to eventually reach the action `cmdAddAction`. Both of these examples show triggers and actions delineating the outer edges of the concern.

As we began to view concerns in this new way, with actions and triggers as boundary nodes and other roles dominating the center of the graph, we investigated the agreement of these boundary nodes. When we calculated the agreement considering only elements that were marked by at least one participant as a trigger or an action, the agreement increased dramatically (see Figure 5(a)). This surprising increase in agreement led us to reconsider our concern role definitions. After discussion and reconsidering the concerns, which the authors have also previously mapped themselves, we have formed a new hypothesis regarding concern roles, which we plan to investigate in future work: **given the action and trigger elements, all remaining role elements can be derived.**

First, we propose that triggers and actions are the only non-derivative concern roles. That is, the other roles, including initializers, data, and connectors, can be derived if the triggers and actions are known, yet triggers and actions cannot (in the general case) be derived from the other roles. Furthermore, we suggest that these derivative roles can be automatically calculated from the trigger and action nodes. While the details of how to calculate these derivative roles are left to future work, we anticipate that existing control and data flow techniques will be successful. For example, in the "add auction" concern in Figure 1, the `JBidProxy.buildHTML()` trigger is connected through control-flow call edges to the actions in the `AuctionServer` class. In another example, more complex data flow analysis could be used to connect the `JBidMouse.DoAction` trigger to the `cmdAddAction` action.

These results have implications for creation of future concern location benchmarks. If it is possible to automatically and reliably derive connector, data, and initializer nodes, then future concern annotators need only locate a concern's actions and triggers to identify a concern. Since these roles also have the least ambiguity in their definitions and the highest agreement, this could have a significant impact on the reliability of future concern data sets for concern location and comprehension studies.

V. Exploratory Study: Feature Location

In addition to increasing agreement among annotators, we hypothesize that concern element roles can provide new insights to further improve feature location tools. To investigate this hypothesis, we conducted an exploratory study using DocFetcher (http://docfetcher.sourceforge.net), a Lucene-based implementation of a vector space model search tool that utilizes tf-idf weighing. DocFetcher is effectively a front end for Lucene (http://lucene.apache.org/), a popular search library.

We ran the search tool using the following queries taken directly from the concern descriptions: "sort by genre" (C8), "add song to playlist" (C7), and "play song file" (C5). Using annotated role data from Section IV, we calculated the rank

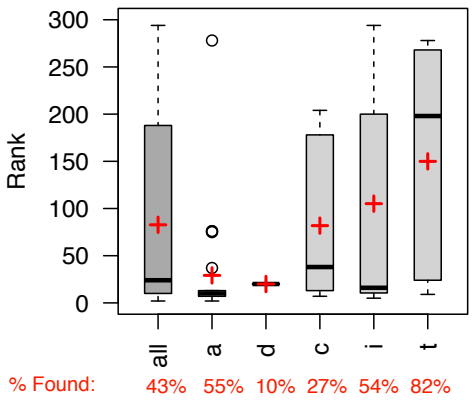


Fig. 8: Rank of annotated elements returned by the search tool, categorized by role and in total (all). Below each role is the percent of annotated elements found by the search tool.

Role	Number Found	Total Number	Percent Found	Recall @ 5	Recall @ 10	Recall @ 20
a	24	44	55%	0.11	0.30	0.43
d	1	10	10%	0.00	0.00	0.10
c	21	77	27%	0.00	0.05	0.10
i	7	13	54%	0.15	0.15	0.31
t	18	22	82%	0.00	0.05	0.05
Total	71	166	43%	0.04	0.12	0.20

TABLE V: Number of methods annotated by study participants that were found by the search tool, categorized by role.

of each method annotated by one of the six study participants. Figure 8 shows the rank of each search result, categorized by role. Below each category is the percent of that role found by the search tool. For example, although the median rank for connector method search results is 38, the box plot in Figure 8 only represents 27% of all the methods annotated as connectors by the study participants. Table V presents the total number of each role found, the percent found, as well as recall values at ranks 5, 10, and 20.

Based on this data, we believe that annotating feature location results with concern element roles reveals new insights to further improve feature location tools. Specifically, we make the following observations:

Actions have better ranks than other roles. For this particular feature location technique, Figure 8 shows that although actions are only found 45% of the time, when they are found, they are in the top 10 half the time (median rank 10). We also observe that initializers, connectors, and triggers are either found infrequently or rank quite high. For instance, the highest trigger rank was 9, and only 25% are in the top 25 (1st quartile rank 24). When connectors are found (just 21 out of 77), the median rank is 40. Although the data role had a low rank (20), only 1 out of 10 data methods were found.

Recall can vary depending on the role. The 'all' box plot in Figure 8 and the 'Total' row in Table V present the results that a feature location evaluation without roles would be based on. However, we can see that such aggregate data does not give the whole picture. Although the overall recall at rank 10 is

12%, the recall for actions is much higher (30%) and triggers lower (5%). Role-based analysis provides an alternative view. **Triggers are the most found role, yet have the worst ranks.** Given the nature of triggers, it is not surprising that they are found by the search technique more than any other role (82%) but that their mean and median ranks are much lower (150 and 198, respectively). Triggers by their definition as entry points into a concern tend to call connectors and actions that have stronger textual clues to the concern. In contrast, actions, as the core of the concern, have the strongest textual relationship to the queries.

Based on these results, the feature location technique could be improved by leveraging call information and reranking the results. Rather than treat all methods in the gold set as the same, role-based interpretation of search results allows us to drill down into the *types* of methods that the search technique is finding, and potentially find new insights to improve the feature location technique.

VI. Related Work

Koenemann, et al. studied developers comprehending code for maintenance, and differentiated 3 tiers of relevance: direct (must be modified), intermediate (studied if interaction with relevant code is important), and strategic (guide comprehension process; points to relevant code) [21]. These relevance tiers are orthogonal to our concern element roles.

In a study of three hand-annotated concerns related to specific change tasks, Murphy, et al. observed that concern boundaries may be difficult to determine [17]. Based on interviews with annotators, the authors observed that the interface between concerns can itself be a concern. The authors conclude that concerns have 3 different types of elements: core behavior, a potentially ambiguous interface that may be a concern in its own right, and a set of execution points that hook into where new functionality may be added to the concern during maintenance. Although the notion of core nodes is similar to our action nodes, the notions of interfaces and execution hooks are different. As previously mentioned, interface methods are part of our trigger and connecting nodes. Because the focus of this work is on annotating existing functionality, we would consider execution hooks to belong to their own concern. This work provides further support that elements may have different roles in concerns, but does not use these roles to decrease agreement and better understand feature location tool results as we have.

Other work has proposed concern tagging guidelines [22]. Two investigators exhaustively identified and located concerns by hand in two software systems. The systems were identified at the source code line level, whereas our work focuses on annotating concerns at the slightly higher granularity of methods or fields. The study found that agreement varied between annotators due to an annotator's understanding of the particular program, experience with the programming language, as well as a poorly defined concern description. The authors identified the following types of concerns: user-triggerable feature, domain-independent functionality, input/output, internal features that are not user-triggerable (such as optimizations or data buffers), and specific language characteristics such as constants, comments, or imported interfaces. Guidelines include data and control dependencies for groups of elements that are deemed part of the concern.

The notion of concern element roles is loosely related to the concept of method stereotypes [23]. Dragan, et al. propose 4 broad categories of method stereotypes: accessor, mutator, collaborators, and creators. Stereotypes are complimentary to our notion of concern element roles. Roles define an element's purpose in comprehending and executing a concern, whereas stereotypes describe a method's purpose in the larger system.

Concern element roles are orthogonal to the idea of roles in object-oriented system design [24], [25]. Object-oriented roles are used to represent dynamically changing entities (i.e., objects) in a system. For example, the idea that a person at a university may be a student, an employee, or a student employee. Each type of person represents a particular role, and these roles may dynamically change over time. In contrast, our notion of concern element roles refers to the role a program element's implementation plays in the implementation of a particular concern of the system and is independent of the system's overall object design or dynamic information.

VII. Conclusion

In addition to refining our initial definitions of the roles that program elements play in concerns, we presented the results from two studies that we conducted to explore how the added role information can potentially improve evaluations of concern location techniques. In a study of 6 Java developers annotating 3 concerns, we found that knowledge of roles increases annotator agreement with statistical significance ($\alpha = 0.05$). This result is even stronger when considering only the action and trigger roles, which typically delineate the upper and lower boundaries of the concern.

Our exploratory study examining the roles of methods returned as search results from a feature location tool did indeed provide more detailed insights into the results. For instance, action elements appear most prominently in the top ranked results, while trigger nodes are the most found but worst ranked elements. The results of these two studies suggest that integrating concern element role information into evaluations can help to strengthen both the gold set establishment and the analysis of results returned by various tools. To help others in role training, the written instructions and video recording of a webinar are shared at http://lee.cs.montclair.edu/~hillem/cer/.

Acknowledgment

We would like to thank our study participants, Martin Robillard for his contributions to early stages of this work, and Sarah Abramowitz for her statistical guidance.

References

[1] B. Dit, M. Revelle, M. Gethers, and D. Poshyvanyk, "Feature location in source code: A taxonomy and survey," *Journal of Software Maintenance and Evolution: Research and Practice*, vol. 25, no. 1, pp. 53–95, Jan 2013.

[2] S. Grant, J. R. Cordy, and D. Skillicorn, "Automated concept location using independent component analysis," in *WCRE '08: Proceedings of the 2008 15th Working Conference on Reverse Engineering*. Washington, DC, USA: IEEE Computer Society, 2008, pp. 138–142.

[3] S. Haiduc, G. Bavota, A. Marcus, R. Oliveto, A. De Lucia, and T. Menzies, "Automatic query reformulations for text retrieval in software engineering," in *Proceedings of the 2013 International Conference on Software Engineering*, ser. ICSE '13. Piscataway, NJ, USA: IEEE Press, 2013, pp. 842–851.

[4] E. Hill, L. Pollock, and K. Vijay-Shanker, "Improving source code search with natural language phrasal representations of method signatures," in *ASE '11: Proceedings of the 26th IEEE International Conference on Automated Software Engineering, short paper*. Washington, DC, USA: IEEE Computer Society, November 2011, pp. 524–527.

[5] S. Lukins, N. Kraft, and L. Etzkorn, "Source code retrieval for bug localization using latent dirichlet allocation," in *WCRE '08: Proceedings of the 15th Working Conference on Reverse Engineering*, 15-18 2008, pp. 155 –164.

[6] A. Marcus, A. Sergeyev, V. Rajlich, and J. I. Maletic, "An information retrieval approach to concept location in source code," in *WCRE '04: Proceedings of the 11th Working Conference on Reverse Engineering (WCRE'04)*, 2004, pp. 214–223.

[7] C. McMillan, M. Grechanik, D. Poshyvanyk, Q. Xie, and C. Fu, "Portfolio: a search engine for finding functions and their usages," in *Proceedings of the 33rd international conference on Software engineering*, ser. ICSE '11. New York, NY, USA: ACM, 2011, pp. 1043–1045.

[8] M. Petrenko and V. Rajlich, "Concept location using program dependencies and information retrieval (depir)," *Inf. Softw. Technol.*, vol. 55, no. 4, pp. 651–659, Apr. 2013.

[9] D. Poshyvanyk, M. Gethers, and A. Marcus, "Concept location using formal concept analysis and information retrieval," *ACM Transactions on Software Engineering and Methodology*, vol. 21, no. 4, 2012.

[10] D. Shepherd, Z. P. Fry, E. Hill, L. Pollock, and K. Vijay-Shanker, "Using natural language program analysis to locate and understand action-oriented concerns," in *AOSD '07: Proceedings of the 6th International Conference on Aspect-Oriented Software Development*, 2007, pp. 212–224.

[11] J. Wang, X. Peng, Z. Xing, and W. Zhao, "Improving feature location practice with multi-faceted interactive exploration," in *Proceedings of the 2013 International Conference on Software Engineering*, ser. ICSE '13. Piscataway, NJ, USA: IEEE Press, 2013, pp. 762–771.

[12] M. P. Robillard and G. C. Murphy, "Representing concerns in source code," *ACM Transactions on Software Engineering and Methodology*, vol. 16, no. 1, p. 3, 2007.

[13] B. Dit, E. Moritz, and D. Poshyvanyk, "A tracelab-based solution for creating, conducting, and sharing feature location experiments," in *IEEE Int. Conf. on Program Comprehension*, 2011.

[14] K. Damevski, D. Shepherd, and L. Pollock, "A case study of paired interleaving for evaluating code search techniques," in *Proceedings of the IEEE Conference on Software Maintenance and Reengineering - Working Conference on Reverse Engineering (CSMR-WCRE)*, 2014.

[15] E. Hill, D. Shepherd, L. Pollock, and K. Vijay-Shanker, "Differentiating roles of program elements in action-oriented concerns," in *Software Maintenance (ICSM), 2013 29th IEEE International Conference on*, 2013, pp. 376–379.

[16] M. P. Robillard, D. Shepherd, E. Hill, K. Vijay-Shanker, and L. Pollock, "An empirical study of the concept assignment problem," School of Computer Science, McGill University, Tech. Rep. SOCS-TR-2007.3, Jun. 2007, http://www.cs.mcgill.ca/~martin/concerns/.

[17] G. C. Murphy, W. G. Griswold, M. P. Robillard, J. Hannemann, and W. Leong, "Design recommendations for concern elaboration tools," in *Aspect-Oriented Software Development*, T. Elrad, S. Clarke, and M. Akşit, Eds. Addison-Wesley, 2004, pp. 507–530.

[18] E. W. Høst and B. M. Østvold, "Debugging method names," in *ECOOP '09: Proceedings of the 23rd European Conference on Object-Oriented Programming*, 2009.

[19] A. F. Hayes and K. Krippendorff, "Answering the call for a standard reliability measure for coding data," *Communication Methods and Measures*, vol. 1, no. 1, pp. 411–433, 2007.

[20] K. Krippendorff, "Reliability in content analysis," *Human Communication Research*, vol. 30, no. 3, pp. 411–433, 2004.

[21] J. Koenemann and S. P. Robertson, "Expert problem solving strategies for program comprehension," in *CHI '91: Proceedings of the SIGCHI conference on Human factors in computing systems*. New York, NY, USA: ACM, 1991, pp. 125–130.

[22] M. Revelle, T. Broadbent, and D. Coppit, "Understanding concerns in software: Insights gained from two case studies," in *IWPC '05: Proceedings of the 13th International Workshop on Program Comprehension*. Washington, DC, USA: IEEE Computer Society, 2005, pp. 23–32.

[23] N. Dragan, M. Collard, and J. Maletic, "Reverse engineering method stereotypes," in *Software Maintenance, 2006. ICSM '06. 22nd IEEE International Conference on*, 24-27 2006, pp. 24 –34.

[24] D. Bäumer, D. Riehle, W. Siberski, and M. Wulf, "The role object pattern," in *Proceedings of Pattern Languages of Programming (PLoP)*, 1997.

[25] G. Gottlob, M. Schrefl, and B. Röck, "Extending object-oriented systems with roles," *ACM Trans. Inf. Syst.*, vol. 14, no. 3, pp. 268–296, Jul. 1996.