2013 IEEE International Conference on Software Maintenance

Differentiating Roles of Program Elements in Action-Oriented Concerns

Emily Hill
Montclair State University
Montclair, NJ 07043
hillem@mail.montclair.edu

David Shepherd
Industrial Software Systems ABB Corporate Research
Raleigh, NC, 27606
david.shepherd@us.abb.com

Lori Pollock and K. Vijay-Shanker
University of Delaware
Newark, DE 19716
{pollock, vijay}@cis.udel.edu

***Abstract*—Many techniques have been developed to help programmers locate source code that corresponds to specific functionality, i.e., concern or feature location, as it is a frequent software maintenance activity. This paper proposes operational definitions for differentiating the roles that each program element of a concern plays with respect to the concern's implementation. By identifying the respective roles, we enable evaluations that provide more insight into comparative performance of concern location techniques. To provide definitions that are specific enough to be useful in practice, we focus on the subset of concerns that are action-oriented. We also conducted a case study that compares concern mappings derived from our role definitions with three developers' mappings across three concerns. The results suggest that our definitions capture the majority of developer-identified elements and that control-flow islands (i.e., groups of elements with little to no control flow connections) can cause developers to omit relevant elements.**

***Index Terms*—concerns; evaluation; software maintenance**

I. Introduction

Programmers who maintain software, such as adding new features, fixing bugs, or other evolution tasks, spend considerable time locating the program elements relevant to the feature (also called a *concern* [1]). Whether due to programming skill level, familiarity with the code, or other reasons [7], programmers do not often agree on what program elements are relevant to a given feature [2]. This causes difficulty in evaluating, comparing, and providing direction for improving feature location techniques.

Existing concern and feature location evaluations consider relevance to be binary; each program element is either judged as part of the concern or not. However, we have observed in our experience that not all program elements of a concern are equal. Some program elements play a major role in implementing the feature's functionality, while others play a less key role, but are important to understand how the feature is implemented and interacts with its surrounding source code context. Our hypothesis is that more informative analysis of feature location techniques can be achieved if they are evaluated with respect to the roles that each program element is playing in the concern and its surrounding context. Additionally, feature location tools could be designed to provide more informative feedback about the elements in the concern.

In this paper, we propose operational definitions for differentiating the roles that each program element of a concern plays with respect to the concern's implementation. Specifically, we describe the characteristics of a program element that takes on the role of an *action, trigger, result*, or *connector* node in a program structure graph representation of a program. In this initial work, we focus on a specific kind of concern: action-oriented concerns [3]. Action-oriented concerns can be specified using a precise verb phrase (VP), which not only includes a verb and direct object, but also an indirect object. For example, not just "add a song", but "add a song to a playlist". Action-oriented concerns may implement user-observable features or be object-oriented, while there are action-oriented concerns that are neither. In the future, we hope to extend this work to a broader set of concerns.

Previous concern mapping studies [2] have investigated the agreement among developers or compared the opinions of a newcomer with the code's author [4]. Neither study has investigated the nature of a concern's elements and differentiated among the different roles the program elements might play. If we can identify the roles of program elements in gold sets, we can study how different feature location techniques identify the different kinds of program elements. For example, some feature location techniques may identify only the action nodes, while another technique also identifies and displays trigger nodes. In addition, with the capability to identify the roles automatically, a feature location tool could provide different display options to the programmer based on the programmer's preferences during maintenance.

This paper makes the following contributions:

- A classification of roles that program elements play in an action-oriented concern with a set of operational definitions that enable precise, objective role-labeling
- A motivating example illustrating the labeling of concern elements with their respective roles
- A preliminary study analyzing human-annotated concerns with respect to roles of each element and annotator agreement

II. Motivating Example

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 data

1063-6773/13 $26.00 © 2013 IEEE
DOI 10.1109/ICSM.2013.51

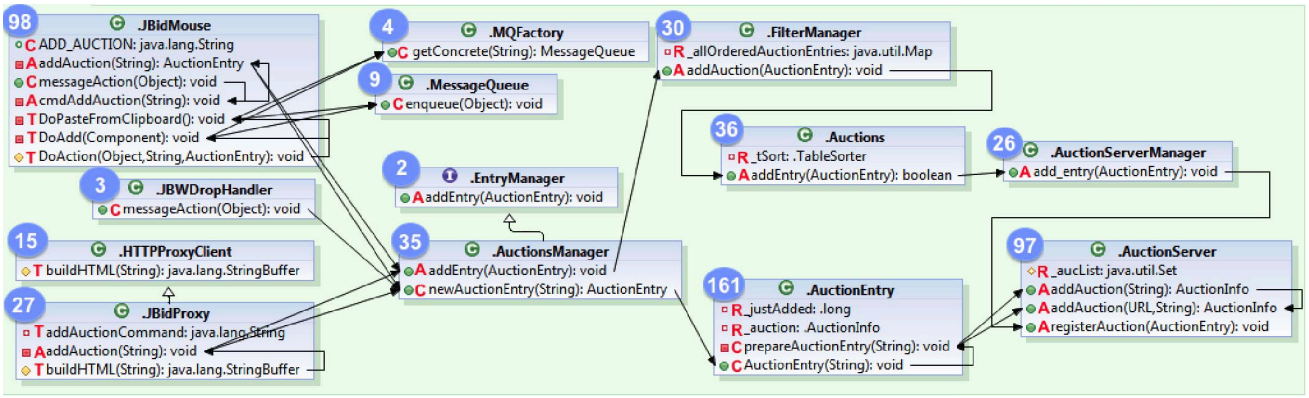

Fig. 1: Program elements and roles for the example "add new auction to local system" concern. Each element (method and field) is annotated with its role in red to the left of the name. The numbers annotating each class in the upper left hand corner are the total number of additional methods and fields in the class, aside from what is displayed in the diagram. Edges indicate structural relationships such as calls (solid-head arrows) and inheritance (open-head arrows).

structures. We define the verb phrase (VP) for this concern to be "add new auction to local system." Figure 1 shows the code that implements the 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 inJBidMouse or JBWDropHandler.messageAction. Some methods are relatively generic, used to process all user-initiated actions, such as MessageQueue and MQFactory. Although generic, these methods communicate (or *connect*) information from the triggers to methods that implement the concern's actions. The action of adding an auction culminates in updating several internal data structures: the set of auction entries being managed by the system (FilterManager._allOrderedAuctionEntries), the table of auctions displayed by the user interface (Auctions._tSort), and the list of auctions being managed by the internal auction server (AuctionServer._aucList). These are some of the *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.

This example demonstrates that a program element may be included as part of an action-oriented concern for different reasons, and that implementing the main action is only one of the reasons. In Figure 1, each element (method and field) is annotated with its role in red to the left of the name. Some concern elements are included because they trigger, or initiate, the concern action (indicated by '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 updated as a result of the action are indicated by 'R', with connecting elements that communicate between different parts of the concern labelled as 'C'.

III. Concern Element Roles

An underlying premise is that action-oriented concerns are described by a VP that includes a verb, direct object, and an indirect object. The action, or verb, is key to determining whether program elements (i.e., methods and fields) should be included in the concern. The direct and indirect objects are also important, with the direct object more prominently used in identification because fields rarely refer to actions and verbs. The indirect object helps further differentiate nodes to be included in the concern by indicating concern boundaries and helping to determine when one concern has become another.

One issue could be when the concern's implementation does not use the verb in the VP to describe the action. For example, *delete* might be implemented as the synonym *remove*, and the concept of *adding* may include *creating* a new item. Plus, an objective definition of action-oriented concerns should not be so fragile as to require exact word matching in the source code. Thus, in the definitions below, we make use of the concept of *similarity*, where two words or phrases are semantically equivalent (i.e., synonyms).

We define four distinct roles of nodes in a structural representation of a concern:

Action Node (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). Action nodes can also serve as concern trigger points (see trigger nodes, below).

Trigger Node (T): Any method 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. If a node is a trigger point for the concern but also implements the concern's VP, then the node is considered to be an action node. This role also applies to fields that are explicitly used to trigger the concern and have a strong relationship with the

concern's VP. We consider initialization of the trigger code to be its own concern.

Result Node (R): Any field that has a similar object to the concern's VP that is altered by an action (method) node as a result of an action related to the VP. For example, an AuctionEntry object is added to the _aucList field as part of the add auction concern in Section II.

Connector Node (C): Any method or field that structurally connects two identified concern nodes (actions, triggers, or result nodes) in the program structure graph with a similar, but not identical, VP to the concern's VP. These nodes do not perform the action, but communicate data and information about the action, enable the action to execute, or support the action. The difference between a connector field and a result field is that the purpose of a connector field is to communicate between different parts of the concern (for example, by connecting a trigger with an action node) and supporting the action's execution in some way, whereas a result field typically stores the results of the action's execution or is otherwise a side effect of the action.

Most of the action nodes in Figure 1 are easily determined from their names alone, but some of the other roles are not as obvious. For example, the action node registerAuction in AuctionServer 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. However, not all methods with a strong relationship to the verb and direct object qualify as action nodes; the indirect object needs to be taken into account as well. For example, the method addAuction in the class MultiSnipe is not adding an auction to the system, but adding an existing auction as part of a special multi-snipe bid process. However, some method names have little relationship to the concern's VP. For example, MQFactor.getConcrete and MessageQueue.enqueue process the generic message queue that connects all user-triggered events to their implementations. Although they are general methods shared by multiple concerns, the add auction concern would not be able to execute without them.

IV. Preliminary Study: Using Roles in Practice

We hypothesize that poor agreement when annotating concerns may be due to annotators favoring certain roles over others. This led us to the following research question:

> ***Research Question**: Do individual annotators tend to include or exclude certain roles over others?*

To investigate this question, we used annotated concerns from a prior study to compare against [2]. Each concern was annotated by 3 different developers. We selected three concerns from two different programs that had strong verb phrases in the concern descriptions given to the annotators: "update auction upon user-trigger," "zoom mind map in and out," and "toggle folded node state." Two of the authors independently annotated the concerns and met to agree on the roles of each method and field. Finally, we applied these roles to the concern annotations created by the 3 subjects. In our analysis, we identified two observations, described below.

***Observation 1:** Action-oriented role definitions capture the majority of program elements that developers annotate.*

When annotating concerns without a clear VP definition, the agreement among annotators has been dismally low, with average agreement only 34% [2]. After annotating three concerns ourselves and comparing our annotations with developer-generated annotations, we determined that our role annotations contained the majority of program elements that the developers included. For instance, in the toggle folded concern, 43 of the 50 elements in our mapping were contained in at least one developer's mapping—an 86% intersection rate. Similarly, in the update auction concern, 32 of 42 elements intersected (76%). We attribute these unusually high intersection rates to the ability of our concern definitions to comprehensively define the roles relevant to action-oriented concerns.

In the last concern, we saw much less intersection between our annotations and the developers', with only 21 of 35 (60%) intersecting. This low intersection rate is explained by examining the non-intersecting nodes. For this concern, developers included six nodes that initialize the trigger code. Recall from Section III that we do not consider trigger initialization code as part of the concern, because they only execute during startup and not during the action's execution. For this concern, developers also included two incorrect triggers, code related to zooming as part of the fit-to-page action, which was intentionally omitted from the concern description. Not considering these initialization nodes and incorrect triggers increases the intersection rate to 82%. In our future work, we will explore whether initialization nodes should be included as part of the role definitions. Although developers included a number of these extra nodes, there were only 3 elements that we included that the annotators missed (91% agreement).

We also found that the role definitions possess inherently good explanatory power. For example, the zoom concern contains two methods, update and updateAll, which seemingly are outside of the zoom concern. In fact, only one developer identified them as relevant. When producing our annotations, we included both methods because they are connector nodes leading to the action node setZoom and result node zoomFactor. Thus, roles can help us reason about why a developer may include certain nodes in their annotations.

***Observation 2:** Differences in annotations are due to control-flow islands within the action-oriented concerns.*

When analyzing developers' concern mappings, we noticed that they were often composed of distinct "islands" where there were no control flow connections or the connections were difficult for a developer to recognize. These islands tended to be organized around trigger nodes or action nodes, with connectors occurring in either type of island, and result nodes predominantly in action-oriented islands (as opposed to trigger-oriented). Some islands contained both trigger and action nodes, if there were strong control flow relationships and method names.

The update auction concern offers a clear example of these islands, with our final annotation comprising three distinct

call graphs. The first island contains user triggers and a flag-setting method, the second contains a timer process to check the flag periodically, and the third the actual UI updating call-chain. These islands have two clear effects on a developers' ability to annotate a concern. First, if developers can find a single element in an island, they typically include multiple elements from that island. For example, in the update auction concern, all subjects found two of the three islands, and all subjects found the second island, which contains 8 elements and includes both action and trigger nodes. However, the annotators found different numbers of elements within those islands: subject P13 found $2/4$ and $3/8$ elements, P10 found $3/4$ and $7/8$ elements, and P12 found $7/8$ and $3/5$ elements. This example illustrates that developers are adept at following clear control flow edges and expanding from a starting point in the code. Second, we observed that even if developers found a significant portion of the overall concern, they were likely to omit *an entire* island, with each developer omitting at least one of the three islands in this concern. For example, P13 found $0/5$ in the third island, P10 found $0/5$ in the third island, and P12 found $0/4$ in the first island. Both the other concerns we studied contain similar islands. For instance, the toggle folded concern contains an island mainly composed of the UnfoldAll class and methods that no participants included because it had no direct control flow links to identified elements. Similarly, the zoom concern contained an island with elements in the MultipleImage class. The developers that skipped NodeView's update and updateAll methods missed the control flow links to the MultipleImage island.

In terms of navigating to islands, developers seemed less likely to follow data flow between fields connecting islands, or control flow where the textual clues are poor. Developers especially had trouble finding methods that implemented interfaces that were called by other parts of the concern. For example, in the "toggle folded node state" concern, annotators found portions of the ToggleFoldedAction and ControllerAdapter island, but completely missed the remaining 3 islands. Both the unfound islands MindMapNode and FoldActionType contain methods called directly by the found island, but developers seemed to have trouble recognizing the relevance of the interface methods, and did not locate the implementing methods. The UnfoldAll island is largely composed of triggers and connectors, and would be missed if the call graph were not explored from ControllerAdapter in the found island.

V. Related Work

Koenemann, et al. studied developers comprehending code for maintenance, and differentiated between 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) [5]. These relevance tiers are orthogonal to 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 [4]. Based on interviews with annotators, the authors observed that the interface between concerns can be a concern. This is in contrast to our notion of triggers and connectors, which we consider to be part of a concern when specified by a precise verb phrase, rather than a change task. 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.

VI. Conclusion and Future Work

In this paper, we presented a characterization of the roles program elements play in action-oriented concerns. We identified four basic concern element roles: action, trigger, result, and connector nodes. We used these role annotation nodes to study formerly annotated concerns with low agreement, finding that the roles help explain the differences between annotations. We believe the preliminary results show promise for improving recommendation and exploration tools by focusing on retrieving the types of nodes and connections that developers seem to have trouble identifying, and that are not currently well-supported in state of the art IDEs. We plan to use the notion of action and trigger islands to study the agreement in the remaining concerns from the prior study [2], and investigate whether trigger initialization code should be added as a concern role type. We hypothesize that roles may also help explain differences in existing code recommendation and exploration tools.

References

[1] M. P. Robillard and G. C. Murphy, "Representing concerns in source code," *TOSEM*, vol. 16, no. 1, p. 3, 2007.

[2] 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/.

[3] 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*, 2007.

[4] 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.

[5] J. Koenemann and S. P. Robertson, "Expert problem solving strategies for program comprehension," in *CHI*, 1991.

[6] N. Dragan, M. Collard, and J. Maletic, "Reverse engineering method stereotypes," in *ICSM*, 2006.

[7] M. Revelle, T. Broadbent, and D. Coppit, "Understanding concerns in software: Insights gained from two case studies," in *IWPC*, 2005.

[8] J. marie Burkhardt, F. Détienne, and S. Wiedenbeck, "Object-oriented program comprehension: Effect of expertise, task and phase," *Empirical Soft. Eng.*, vol. 7, no. 2, 2002.

[9] V. Rajlich and N. Wilde, "The role of concepts in program comprehension," in *IWPC*, 2002.