Semi-automatic Concern Annotation Using Differential Code Coverage

Matúš Sulír and Jaroslav Porubän
Department of Computers and Informatics
Faculty of Electrical Engineering and Informatics
Technical University of Košice
Letná 9, 042 00 Košice, Slovakia
Email: {matus.sulir,jaroslav.poruban}@tuke.sk

***Abstract*—Concern annotations are source code annotations over language elements, expressing concerns related to a particular peace of code. While elements can be annotated manually by programmers, it is a time-consuming activity. We present an approach of semi-automatic method annotation using differential code coverage, focusing on end-user features. We compare the results of semi-automatic annotation of a sample program with manually performed annotation. In our preliminary implementation, we achieve a precision and recall over 40%, identify the shortcomings and name differences.**

I. Introduction

A. Concerns

Each nontrivial software contains many *concerns* – programmer's intents behind a particular piece of code [1]. For example, a diagramming application may contain code related to on-screen drawing, vector arithmetics, file formats, etc. When a program is written in a traditional programming language, these concerns are tangled together. This is unavoidable since one piece of code can pertain to multiple concerns [1].

B. Features

Some of the concerns are related to the problem domain and are directly visible to application end users. Such concerns are usually called *features*.

Feature location is an act of finding all occurrences of a particular concern. It is one of the most frequently performed actions during program maintenance [2]. However, when concerns are not explicitly recorded in the source code, this task can be onerous.

C. Concern Annotations

For the mentioned reasons, we devised a notion of *concern annotations* in [3]. For each concern, a Java annotation type is created by a programmer. For example, the drawing concern is represented by an annotation type `@Drawing`. Subsequently, each class, member variable and method related to this concern is marked with this annotation. Marked elements are called *annotation occurrences*.

This approach has a major advantage over natural-language source code comments: All occurrences of a concern can be unambiguously found using standard IDE (Integrated Development Environment) features like Find Usages. Compared to other code-tagging methods, annotations do not require any additional plugins as they are a core feature of the Java language (and present also in other languages, e.g., as C# attributes [4]).

Concern annotations can be parametrized, e.g., `@Drawing(target=Target.SCREEN)`. However, for the purpose of automatic concern tagging, we will only consider annotations without parameters.

There are three types of concern annotations [3]:

- domain annotations (concepts related to the problem domain), e.g. "searching students",
- design annotations – usually instances of design patterns and other implementation decisions,
- maintenance annotations like `@TODO("finish later")`.

In this paper, we are interested only in domain annotations as they present program features visible to the end user.

D. Differential Code Coverage

Differential code coverage [5] is based on an idea of running an application two times – first with a particular feature utilized, second time without this feature – while collecting sets of executed methods. By subtracting the second set from the first one, we should get a set of methods specific to the given feature.

E. Hypothesis

In this article, we present the first draft of a method for semi-automatic annotation using differential code coverage. We assess its usefulness and try to list some of its drawbacks.

Our main **hypothesis** is: A semi-automatic approach of concern annotation using differential code coverage is a suitable replacement for manually performed concern annotation.

We have broken the hypothesis into three smaller research questions:

- **RQ1:** Is it feasible to use differential code coverage to annotate methods in the source code with concern annotations?

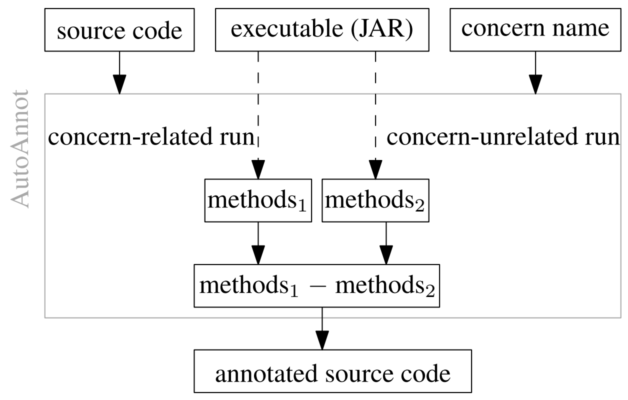

Fig. 1. Annotation process for a specific concern. Solid arrows represent automated processes, dashed partially manual ones.

- **RQ2:** To what extent do automatically created annotations overlap with manually created ones for the same project?
- **RQ3:** What are the main reasons of differences between semi-automatic and manual annotation?

II. Semi-automatic Concern Tagging

An overview of the whole process for one specific concern is in Fig. 1. The process can be also run with a list of multiple concern names supplied. In this case, the AutoAnnot part is repeated for each concern.

A. Creating Annotation Types

First, the programmers must create an annotation type for each concern they recognize in a program. This process is not automated and remains the same as in our previous work.

B. Differential Code Coverage

In this section, we will apply the idea of differential code coverage [5] to determine a set of methods pertaining to a particular concern and designated to be annotated with the given annotation type.

1) Instrumentation: The program is augmented so that execution traces are collected when it is run. We used BTrace[1]. It was run as a Java agent – a program which can instrument the bytecode at runtime. BTrace uses a file with Java-like syntax, specifying which actions should be performed e.g. on each method call.

We configured BTrace to log a fully-qualified name of each called method to a file. A fully-qualified name unambiguously identifies the method – it includes a package and class name, method name and parameters types. Each method is included only once in a log file, even if it is called multiple times. We also included constructors in the logs.

[1]http://kenai.com/projects/btrace

2) Execution: For each concern, a user executes the program two times:

- once with the particular concern utilized – a *concern-related run*
- and once in a manner when the concern is not utilized – a *concern-unrelated run.*

For example, if the concern of interest is “file saving”, the user first creates a simple drawing in an application and saves it to a file; in the second run (s)he creates the drawing again but discards the changes.

We now have two log files for a particular concern – one containing the methods from the first run, one from the second.

3) Subtraction: Let us call the set of methods executed for concern c during the first run $M_1(c)$, during the second one $M_2(c)$. Then the set of methods most specific to the concern c is:

$$M(c) = M_1(c) - M_2(c)$$

This way, only methods specific to a particular concern are included in the result, leaving out utility methods and methods common to multiple concerns.

C. Writing Annotations

Once a set of methods corresponding to a particular concern is computed, each of these methods is annotated. The original source code is parsed, modified and written back using the Roaster library[2].

There was a problem with Java’s anonymous classes – mapping from the class during runtime to a class in the source code was ambiguous. For this reason, we chose not to annotate methods inside anonymous classes.

III. Comparison with Manual Tagging

In this section, we will investigate how are results of semi-automatic concern annotation similar to human-performed annotation and how they are different.

A. Method

1) Materials: We performed this study on EasyNotes[3] – a 2500 LOC (lines of code) desktop Java application for bibliographic note-taking.

2) Manual Annotation: We used manually-annotated source code from our previous experiment [3]. The procedure was as follows: Seven participants were asked to create an annotation type for any concern recognized in the application. Then they marked elements with concern annotations they thought are relevant. Next, we selected annotation types (concerns) recognized by at least two participants (26 types total) – let us call them “shared concerns”. Finally, we merged all annotation occurrences (marked by at least one person) for all shared concerns into one project.

For this study, we selected only 9 concerns we clearly categorized as domain annotations because only this type can

[2]http://github.com/forge/roaster
[3]http://github.com/MilanNosal/easy-notes

be recognized during the application runtime. For instance, a user can easily run an application and click the GUI (graphical user interface) controls related to “searching”. On the other hand, it makes no sense to talk about running application in a way that involves “coding by convention” or “unused code”.

3) Semi-automatic Annotation: Again, we used the same 9 domain-related concerns recognized by participants of the previous experiment.

We created a simple utility, AutoAnnot, to assist in the process of semi-automated annotation.

For every concern, the AutoAnnot utility started the program two times. A researcher used the program (by clicking, typing, etc.) as described in section II-B2. The decision what a particular concern comprises was subjective – left on the researcher. Meanwhile, our utility recorded executed methods.

Annotations of methods resulting from the differential code coverage were written into the source code by AutoAnnot.

4) Comparison: For both manually and automatically annotated source code, we extracted the mapping from concerns to their occurrences into XML files using the SSCE NetBeans plugin[4]. The resulting XML files were compared by a small script.

B. Quantitative Results

1) Annotation Count: First, let us compare the number of methods annotated manually and semi-automatically (Table I, column Annotated methods count). The mean number of annotated methods is only slightly higher for the semi-automatic approach. This means differential code coverage does not produce excessively large lists of methods. The number of methods per concern (2–17) is acceptable for manual inspection by a developer when needed.

2) Overlap: To study overlap, we decided to define a set of manually tagged methods for a specific concern as a “gold standard” – methods which we consider perfectly relevant to that concern. Note that this definition is certainly not ideal. Manual annotation was performed by multiple participants, some of who did not know the system in detail. That being said, the results are in Table I, column Overlap.

Given the set of methods *auto* as a result of semi-automatic tagging and $manual$ of manual tagging, we define (adapted from [6]) precision as:

$$precision = \frac{|auto \cap manual|}{|auto|} \cdot 100\%$$

and recall as:

$$recall = \frac{|auto \cap manual|}{|manual|} \cdot 100\%$$

Intuitively, the higher the precision, the less false positives (methods tagged automatically, but not manually) there are. The higher the recall, the less false negatives (methods tagged manually, but not automatically) are present.

Finally, F_1 score is defined as a harmonic mean of precision and recall.

[4]http://github.com/MilanNosal/sieve-source-code-editor

The mean precision is 42.14%, recall is slightly higher (44.83%). While not ideal, we consider the F_1 score of 40.69% sufficient, taking into account the limitations of manual tagging.

C. Qualitative Results

Now we will inspect a set of false positives for a concern with the lowest precision – Links and a set of false negatives for a concern with the lowest recall – Tagging.

1) Low Precision Causes: The `Links` concern signifies links to local files (e.g., article PDFs) and other resources in bibliographic records. There were 16 methods annotated automatically, but not manually.

Fourteen such methods were in classes `LinksFilter`, `EditLinksPanel` and `ShowLinksPanel`. These methods were not tagged manually because the whole classes they belong to were tagged by participants. It is not certain what annotating a class in respect to its methods means. Do absolutely all methods of the class pertain to the given concern? Or does it hold for a majority of methods? Probably the most elegant solution is to refrain from annotating classes and annotate only methods instead.

Two methods were in the class `DynamicCollectionPanel`. This class is a custom UI (user interface) component not directly related to the domain concept of links, but used exclusively to manipulate links in the EasyNotes application.

2) Low Recall Causes: Notes in the EasyNotes application can contain tags (conceptually similar to tags in some blogs). This concern is marked by the `Tagging` annotation. There were 7 methods annotated manually for this concern, but not automatically.

Sometimes it is difficult, if not impossible, to avoid executing some methods in the second (concern-unrelated) run. For example, the method `NotePanel.setTags` is called each time an empty note-adding dialog is shown because a newly created note contains the “new” tag by default. The method `Note.getTags` is called every time a list of notes is displayed, even if the tag list of each note is empty. The constructor `Note()` is called for each note added through the dialog or loaded from a file.

Other times, a user may miss the functionality during the first (concern-related) run. For instance, the tagging concern was not obvious from a note filter (an item in a drop-down list) called “not used”. It displays all notes which do not contain tag “new”. This way, the method `Note.isUsed` was missed. Even more subtle case is the method `Note.isNew`, which is called when a user hovers the mouse cursor over a note for a while – this is the time when a text for a tooltip is generated.

The methods `Note.addTag` and `removeTag` were obviously annotated manually because of their name. However, they are not used (called) in the program at all. Thus, if a programmer wants to get an overview of the current application state, semi-automatic tagging can be superior to manual one even if the recall says the opposite.

TABLE I. A COMPARISON OF MANUAL AND SEMI-AUTOMATIC ANNOTATION RESULTS

Concern	**Annotated methods count**		**Overlap**		
	Manual	**Semi-automatic**	**Precision [%]**	**Recall [%]**	**F_1 score [%]**
Citing	5	2	50.00	20.00	28.57
Filtering	13	9	33.33	23.08	27.27
Links	4	17	5.88	25.00	9.52
Note adding	7	5	60.00	42.86	50.00
Note deleting	2	2	100.00	100.00	100.00
Note editing	6	10	20.00	33.33	25.00
Notes loading	5	10	40.00	80.00	53.33
Notes saving	6	8	50.00	66.67	57.14
Tagging	8	5	20.00	12.50	15.38
Mean	**6.22**	**7.56**	**42.14**	**44.83**	**40.69**

D. Threats to Validity

Now we will present threats to validity according to guidelines [7].

1) Construct Validity: We used precision, recall and F_1 score to measure overlap between a manual and semi-automatic approach. These metrics are often used to express classification accuracy [6] in areas like machine learning, where human-performed recognition is considerably superior to the machine-performed algorithm. However, in our study it is not clear what what constitutes a good set of methods for a given concern. In our previous work [3], we formulated a hypothesis that the code author is the best annotator. Nevertheless, this hypothesis is not yet confirmed.

2) Internal Validity: Participants of our previous experiment (manual annotation) were asked to annotate methods, types (classes, enums, etc.) and member variables. Using semi-automatic annotation, we only took methods into account. This is probably the most important threat to validity as it might lower the recall significantly.

3) External Validity: We performed the comparison only on one particular project. The source code is small: 2.5 kLOC in 33 classes, excluding annotations. The study should be repeated on a larger project, preferably from another domain and using other technologies, to confirm the generality of results.

4) Reliability: The execution step of differential code coverage for all concerns was performed entirely by one researcher. The results are therefore affected by his subjective decisions, mainly during the execution phase of differential code coverage.

IV. RELATED WORK

A. Concerns

Revelle et al. [8] performed case studies regarding the identification of concerns in source code. However, tagging was performed fully manually. They also present a list of concern tagging guidelines. Sadly, these guidelines are not fully applicable for us, since they created an Eclipse plug-in allowing to tag and color-mark the source code with character granularity. Standard Java does not allow annotating arbitrary statements below the method level.

B. Differential Code Coverage

The idea of subtracting code coverage results was originally described by Wilde et al. [9]. In contrast to our approach, the traced application was not executed by a user interactively. They used test cases written in a form of a source code instead. While their method is more easily reproducible, running the application interactively is less time-consuming when only one or a few executions are required.

In [10], Wilde and Casey called their method “software reconnaissance”. Sherwood and Murphy [5] performed an experiment using this approach, giving it a more expressive name “differential code coverage”. Furthermore, instead of executing test cases, the participants also ran an application interactively. They present an Eclipse plug-in, Tripoli, which displays results of differential code coverage graphically. Contrary to our approach, they did not persist the results in any way. Moreover, as annotations are a standard Java feature, our approach does not require any plugin – just an IDE.

C. Execution Tracing

MUTT [11] is a tracing tool which allows to start and stop capturing the list of executed methods with a button. This idea could be applied in connection with differential code coverage, which could possibly rise the precision.

D. Aspect Mining

Aspect mining [12] aims to identify cross-cutting concerns in source code. It is therefore focused on concerns unrelated to the primary functionality of a program, like logging or session management [13]. Furthermore, aspect mining itself only deals with finding the corresponding locations in source code and does not specify a persistence method of the findings.

E. Automatic annotation

Joy et al. [14] use an approach where macros are inserted into source code, causing collection of timing and power information at runtime. The collected information are then back-annotated to the source code. In contrast to our approach, they did not recognize any concerns or features in the software.

V. Conclusion and Future Work

A. Conclusion

In this article, we presented a simple approach for semi-automatic concern annotation. The input of a our demo program, AutoAnnot, is an unannotated source code, the executable file of an application to be annotated and a concern name (or a list thereof). The user runs the program two times – once using the features of interest (a concern-related run) and then purposely not using them (a concern-unrelated run). AutoAnnot traces the sets of methods executed and subtracts the second set from the first one. The resulted set of methods is marked with Java annotations, overwriting the original source code. This demonstration answers **RQ1**: it is feasible to use differential code coverage for semi-automatic concern annotation.

To answer **RQ2**, if we take manually created annotations as a "gold standard", both precision and recall are more than 40%, producing the F_1 score of almost 41%.

Answering **RQ3**, the most prominent reason or false positives (methods annotated just automatically) is an unclear definition of what language elements should have concern annotations – whether just methods or also classes and member variables. One of the main reasons of false negatives are the inability to avoid executing some methods in a concern-unrelated run; and forgetting to include some functionality in a concern-related run.

Regarding the main **hypothesis**: Our approach definitely has its potential. However, before actually being used as a replacement of manual tagging, it must be improved.

B. Future Work

One of disadvantages of our approach is that after each source code change, all affected concerns must be re-annotated – the program must be manually run again. A possible solution is to allow performing concern-related and concern-unrelated runs as test cases. This way, the method would be fully automated and it could be executed automatically as a part of a build process. This could also eliminate the inability to exclude some behavior in the second run to some extent.

Complementing differential code coverage with a button starting and resuming trace collection, or even fully replacing it with these approach, would be an interesting future work.

Types (like classes and enums), member variables, methods and method parameters can be annotated in Java. From these elements, only methods have a notion of "execution". While we could also intercept member variables reading and writing, we observed that these actions are often performed without any relevance to the concern executed. For example, in a constructor, Java initializes all member variables either to `null` or to a defined value. Thus, some form of filtering would be required to capture only concern-relevant member variable reads and writes. We decided to leave this for future work.

Acknowledgment

This work was supported by VEGA Grant No. 1/0341/13 Principles and methods of automated abstraction of computer languages and software development based on the semantic enrichment caused by communication.

References

[1] J. Porubän and M. Nosáľ, "Leveraging program comprehension with concern-oriented source code projections," in *3rd Symposium on Languages, Applications and Technologies*, ser. OpenAccess Series in Informatics (OASIcs), M. J. V. Pereira, J. P. Leal, and A. Simões, Eds., vol. 38. Dagstuhl, Germany: Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2014, pp. 35–50.

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

[3] M. Sulír and M. Nosáľ, "Sharing developers' mental models through source code annotations," in *Computer Science and Information Systems (FedCSIS), 2015 Federated Conference on*, Sept 2015, pp. 997–1006.

[4] R. Sepeši and M. Nosáľ, "Towards developer-friendly annotation-based code generation," in *Engineering of Modern Electric Systems (EMES), 2015 13th International Conference on*, June 2015, pp. 1–4.

[5] K. D. Sherwood and G. C. Murphy, "Reducing code navigation effort with differential code coverage," Department of Computer Science, University of British Columbia, Tech. Rep., September 2008.

[6] T. Fawcett, "An introduction to ROC analysis," *Pattern Recognition Letters*, vol. 27, no. 8, pp. 861–874, 2006, ROC Analysis in Pattern Recognition.

[7] P. Runeson and M. Höst, "Guidelines for conducting and reporting case study research in software engineering," *Empirical Software Engineering*, vol. 14, no. 2, pp. 131–164, 2009.

[8] M. Revelle, T. Broadbent, and D. Coppit, "Understanding concerns in software: insights gained from two case studies," in *Program Comprehension, 2005. IWPC 2005. Proceedings. 13th International Workshop on*, May 2005, pp. 23–32.

[9] N. Wilde, J. Gomez, T. Gust, and D. Strasburg, "Locating user functionality in old code," in *Software Maintenance, 1992. Proceedings., Conference on*, Nov 1992, pp. 200–205.

[10] N. Wilde and C. Casey, "Early field experience with the software reconnaissance technique for program comprehension," in *Software Maintenance 1996, Proceedings., International Conference on*, Nov 1996, pp. 312–318.

[11] D. Liu and S. Xu, "MuTT: A multi-threaded tracer for Java programs," in *Computer and Information Science, 2009. ICIS 2009. Eighth IEEE/ACIS International Conference on*, June 2009, pp. 949–954.

[12] R. S. Durelli, D. S. M. Santibáñez, N. Anquetil, M. E. Delamaro, and V. V. de Camargo, "A systematic review on mining techniques for crosscutting concerns," in *Proceedings of the 28th Annual ACM Symposium on Applied Computing*, ser. SAC '13. New York, NY, USA: ACM, 2013, pp. 1080–1087. [Online]. Available: http://doi.acm.org/10.1145/2480362.2480567

[13] M. Ceccato, M. Marin, K. Mens, L. Moonen, P. Tonella, and T. Tourwe, "A qualitative comparison of three aspect mining techniques," in *Program Comprehension, 2005. IWPC 2005. Proceedings. 13th International Workshop on*, May 2005, pp. 13–22.

[14] M. Joy, M. Becker, W. Mueller, and E. Mathews, "Automated source code annotation for timing analysis of embedded software," in *Advanced Computing and Communications (ADCOM), 2012 18th Annual International Conference on*, Dec 2012, pp. 12–18.