A Review of Methods for Identifying Extract Method Refactoring

Omkarendra Tiwari
IIIT Allahabad, India
E-mail: omkarendra@iiita.ac.in

Vrinda Yadav
IIIT Nagpur, India
E-mail: vyadav@iiitn.ac.in

Abstract

Extract method is one of the most popular and versatile refactoring. It is primarily applied to improve the design of methods by mitigating code smells such as long method, code clone, and feature envy. In recent past, various methods for identifying extract method refactoring have been proposed. However, an established performance hierarchy among them is lacking as the proposed approaches have been evaluated on different benchmarks.

This paper evaluates the approaches in a common setting to identify various parameters and their impact on performance, with a goal to help users to identify a tool best suited to their requirement and aid researchers to make an informed decision while designing and evaluating a new approach. Existing approaches are evaluated over a common benchmark consisting of five open-source software studies with a focus on understanding the impact of evaluation settings over performance of approaches. Our experiment shows that standardization in value selection for evaluation parameters is crucial. It is observed that most of the approaches are sensitive to top-n suggestions parameter. Further, tolerance parameter used is not generalized. Our finding in performance trend measured in precision, recall, and F-measure deviate from the earlier results.

1. Introduction

Extract Method refactoring is applied to decompose a long method in order to gain benefits such as improved readability, reusability, ease of feature extension, and lower code duplicability [1, 2]. Its application can be viewed as two part process (i) identification of a task (a subset of statements in the method), and (ii) extraction of the task or extract method opportunity (EMO) as a separate method. The second part is automated and is supported in IDEs, Eclipse for example. Hence, in the recent past, researchers have proposed various techniques to automate the identification of tasks. However, little is known about practitioner's or standard approach to this problem, for example how a set of statements is isolated as a task, what metrics are used as indicators, is there a minimum or maximum size threshold over newly extracted method, awareness and selection criteria of existing tools etc. Moreover, there exists a knowledge gap in state-of-the-art approaches.

Approaches proposed in the last decade vary in design philosophy, underlying technique, benchmarks, and evaluation criteria used. This makes it difficult for users to clearly identify best tool/method in each aspect of performance. Further, unavailability of a large benchmark makes it difficult for researchers to correctly measure the state-of-the-art. Hence, we observe a requirement for establishing a standard evaluation setup and an analysis of the recent approaches.

This study focuses over implementation (tool) and evaluation setting of the proposed approaches; learn early lessons and apply them to design benchmarks and obtain results that can be used by both researchers and practitioners. In this paper, we present an evaluation of state-of-the-art approaches over a common benchmark. Further, we discuss a set of guidelines for configuration and metrics for evaluation. Aim of the study is to bring forward standard practices which may lead to establish the merit of new research methods over existing methods. Key aspects of focus for our study are (i) benchmarks used in evaluation, and (ii) configurations and metrics used in evaluation.

Since, earlier existing approaches used different benchmarks and tool configurations for evaluation, an analysis in common setup was required to understand 'true' performance hierarchy, this paper makes the following contributions:

- A review of state-of-the-art approaches and evaluation practices on a common benchmark
- Pluto: A new synthetic benchmark
- Guidelines for developing and evaluating new approaches

It can be noted that this is the first work aimed at studying the diverse evaluation criteria, and attempts to bring

DOI: 10.18293/SEKE2023-199

out standard practices along with a common benchmark to help users to select a tool as per their requirements and researchers to be better informed about the state-of-the-art.

Rest of the paper is organized as follows: Section 2 lists a set of aspects of interest for researchers/developers/users when developing/selecting an extract method refactoring approach; also these aspects are the focus of this study. Evaluation of existing approaches and experimental setup is discussed in Section 3. Finally, conclusion and future work is discussed.

2. Extract Method Refactoring: Aspects of Concern

Multiple studies have been conducted to understand application and benefits of refactoring in general but little is known about extract method refactoring [1, 3]. Added with subjective nature of this refactoring, its automation becomes a challenging task. We note that human feedbacks are crucial for development of an effective and robust tool. However, there are aspects of formulation and evaluation of the refactoring approaches that are independent of expert's feedback. So, these two aspects can be explored independently and later, findings related to the other aspect can be augmented.

```
    void FiboPrime() {
       int i, n, a, b, t;
A.     printf("Value of N:");
B.     scanf("%d",&n);          // Input [A,B]
C.     a = 0;
D.     if (n ==1)
E.        printf("Nth Term:%d",a);
F.     else {
G.        b = 1;
H.        for (i=3; i <=n; i++){
I.           t = a + b;
J.           a = b;      // Compute Nth Term [C-K]
K.           b = t;
           }
         }
L.    printf("Nth Term:%d",b);//Show Nth Term [L]
M.    for (i=2; i<=b/2;i++){
N.       if (b%i == 0)
O.          break;    // Divisor Computation [M-O]
      }
P.    if (b<=1 || i <=b/2)
Q.       printf("Not Prime");
R.    else            // Prime Checking [P-S]
S.       printf("Prime");
    }
```

Figure 1: A Program for Computing Fibonacci Prime

Now, we present aspects of extract method refactoring that are central to the study. It includes recent approaches, challenges in developing an automated solution, and evaluation setup to establish the performance hierarchy of tools.

2.1 Recent Approaches/Tools

Various refactoring approaches for Long Method code smell use techniques such as clustering [4] [5], control flow graphs [6, 7, 8] and program slicing [9], [10]. However, in this section and the paper, we restrict discussion around a few recent approaches that have been accompanied with their implementation (tool) and have evaluated their performance over open-source software studies.

One of the early extract method refactoring approach available as a tool is *JDeodorant*, proposed by Tsantalis and Chatzgeorgious [11, 12]. It uses complete computational slice to identify a task that can be extracted as a separate method. The computed slice may result in duplicate statements in extracted and base method. It is available as an Eclipse plug-in and have been used for performance evaluation in recent approaches.

Silva et al. [13] proposed, *JExtract*, a new block-based method for generating an exhaustive list of suggestions that are ranked before presenting to the user. The approach uses a web-based system *MyWebMarket* to identify the satisfactory configuration for the tool, which then is used in evaluating the approach via application over two open-source software studies, *JUnit* and *JHotDraw*. Authors synthetically created extract method opportunities(EMOs). The evaluation shows that the approach can achieve a high recall at low precision. Also, the results show the configuration that produces high precision lowers the recall.

Charalampidou et al. [14] proposed, *SEMI*, a clustering based approach for identifying extract method opportunities. The approach forms cluster of coherent statements based on the presence of a common variable, object, method name etc. This method lowers the final suggestions to the developer/user by first grouping the identified extract method opportunities and then ranking them. The approach groups similar opportunities based on overlapping statements, then finds one which offers most benefit in terms of cohesion if refactored. Similar to JExtract, it also generates an exhaustive list of suggestions.

Xu et al. [15] proposed, *Gems*, a machine-learning based probabilistic model for predicting extract method opportunities. Given the pair of the refactoring candidate and the method the approach extracts informations such as loop, size, invocation, type and variable access etc.

Tiwari and Joshi [16] proposed, *Segmentation*, a clustering based approach that aims at maximizing the precision with a manageable recall. Further, the proposed approach restricts the number of suggestions generated by implementing the policy of forming distinct clusters.

Shahidi et al. [17] proposed a method to identify, and mitigate long method code smell by application of extract method refactoring. Further, to ensure modularity the refactored code is analyzed for feature envy. Their proposed

method relies on expert's opinion to evaluate the performance of the approach, in contrast to evaluation strategies followed by recent approaches [12, 13, 14, 16] that use synthetic benchmark.

Since, aforementioned approaches have followed varying benchmarks and evaluation criteria, a user may find it difficult to select the most suitable tool as per the requirement.

2.2 Subjectivity

Consider the Fibonacci Prime program shown in Figure 1. The program illustrates the subjective nature of this refactoring, and also exhibits the challenges in coming up with a solution. The program contains multiple subtasks packed in one method to achieve the task of Fibonacci prime computation. These subtasks are annotated on the right in the figure.

One can decompose the given method into two methods (i) statement block *A-L* corresponding to *Fibonacci term computation*, and (ii) remaining statements for *Prime checking*. Such a decomposition of the program would result in a *Fibonacci Method* with restricted resuability as it contains *input-statement* and *display-statement* within it. Though, such a decomposition is functionally correct but may not always be desirable.

Another strategy for decomposition could be to decompose statement block *A-L* into three parts (i) *Input*, (ii) *Fibonacci term computation*, and (iii) statement *L* for *display*. This decomposition lead to extraction of a reusable method implementing Fibonacci term computation that can be invoked with no modifications. Invocation of this method would require input parameters and returns computed Fibonacci term.

Subjectivity in decomposition adds to challenges in evaluating the performance of automated approaches. For this reason, researchers use a parameter called *tolerance* when matching the automatically identified task to ground truth. We discuss more on tolerance in next section.

2.3 Benchmarks

A benchmark is crucial in evaluating external behavior of the approach, its performance or usability. It also helps in gaining an insight into the improvement made by earlier approaches, and scope for future developments. Evaluation of multiple approaches over a common benchmark enables researchers to compare the approaches and rank their performance. Further, public availability of the benchmarks allow others to replicate the results and also assess the merits of their new approaches against the state-of-the-art.

Recent approaches have used three kind of benchmarks (i) Open-source software along with feedbacks from the developer, (ii) Open-source software studies, where extract method opportunities were introduced by inline refactoring, and (iii) A software from Industry. Since, this paper is focused on the aspects where direct human intervention is not present or it is restricted; the benchmark used for our study is of second kind.

2.4 Tool Configurations and Performance Metrics

Tool's configuration is crucial in their performance report. Most of the approaches use precision, recall, and F-measure for evaluating their performance. Given that most of the approaches have shown excellent performance over either recall or precision, a change in their default or prescribed configuration would affect their 'true' performance. Not only the configuration but also the performance strength is crucial for a user while selecting tool for identifying refactorings. A few parameters that are used in recent approaches to tune their tools include *top-n suggestions* and *tolerance*.

The parameter top-n suggestions restricts the count of suggestions generated by a tool. This parameter directly affects both the precision and the recall of an approach. Generally, tools provide ranked suggestions. So, for example, if a tool's first suggestion is always best match with existing refactoring opportunity in a method then considering more than one suggestions would decrease its precision. However, it can be noted that even for a method with exactly one refactoring opportunity, there are two functionality that may be extracted. For example, method shown in figure 1, a tool may identify Fibonacci or Prime as an extract method refactoring opportunity or both. Thus, it seem that value for this parameter should be greater than two.

The other parameter of interest is tolerance, which sets limit to maximum difference/mismatch between the extract method refactoring opportunity present in a method and suggested refactoring generated by the tool. If the difference between the two is within the threshold the suggestion is considered valid and counted for precision with respective tolerance limit. Finding an appropriate threshold is crucial. In literature absolute and relative threshold has been used.

3. Evaluation

This section discuss evaluation setup and analyses various configurations and their impact over performance.

3.1 Research Questions

The study is focused on quantitatively assessing the sensitivity of the approaches to experimental setup, to report findings that would be helpful for a user to make an

informed decision while selecting a tool for their source code and performance criteria.
RQ1 Does different performance parameters (precision, recall, and f-measure) have a correlation between approach?
RQ2 How does the configuration affects the performance and usage of an approach?

3.2 Experimental Setup

To assess the performance sensitivity to different experiment settings, we prepared a set of open-source software studies with extract method refactoring candidates, a set of values for tolerance threshold, downloaded and installed tools of recent approaches, and fixed precision, recall, and F-measure as parameters to measure performance.

- *Tools* In this study we use the implementation (tools) provided by the authors of the respective approaches. Further, as part of this study, of five approaches discussed in Section 2.1, we include three approaches/tools namely, JExtract, SEMI, and Segmentation. Since, this is an early phase of studying the state-of-the-art of identifying extract method refactoring, we excluded JDeodorant as it has been evaluated in multiple studies and have been outperformed on different performance measures.

  We note that exclusion of the two tools (based on slicing and machine learning) lowers the diversity of the study in terms of techniques. However, included tools follow similar technique for clustering– functional-blocks-based cluster–, which adds to uniformity in evaluation. Further, included tools collectively are among the leading performers in all three performance metrics, *precision*, *recall*, and *F-measure*.

- *Benchmark* To analyze the aspects of the refactoring least associated or dependent on expert feedback, we use synthetic benchmark (type (ii) benchmarks discussed in Section 2.3) to evaluate the selected approaches. Moreover, synthetic benchmark is better equipped for a controlled experiment. [13] created two synthetic open-source software studies(OSS) *JUnit* and *JHotDraw*; both the OSSs were used for evaluation of multiple approaches [13, 14, 16]. We follow the same procedure to extend the set of OSSs in used in the evaluation by including *Mockito*, *EventBus*, and *Javapoet*; collectively named as *Pluto* [18]. The OSS studies used in the paper are listed in Table 1. All five OSSs contain a total of 173 EMOs, which makes it one of the largest benchmark in terms of EMO count. Further, the table shows that individual software studies contain methods with varying range mean-size, which adds to the diversity of the benchmark.

Table 1: Synthetic Benchmark for Extract Method Refactoring

Name	OSS studies	#EMOs	Mean Method Size (LoC)
Pluto-1.0	Mockito-3.3.8	46	12.85
	EventBus-3.2.0	25	20.48
	JavaPoet-1.12.1	21	17.40
JExtract[13]	JUnit-3.8	25	18.04
	JHotDraw-5.2	56	14.98

- *Tolerance* As we discussed earlier, task of *Fibonacci term computation* in Figure 1 is associated with *input* and *display* subtasks. So during extraction of it, inclusion of either or both subtasks results in lower reusability as some developers may have a different source for *input* or they may not be willing to *display* result on console. In such cases, where inclusion or exclusion result in valid functional-block but its desirability is subjective, researchers use an evaluation parameter called as *tolerance*.
  In literature, two methods for selecting a value for tolerance is proposed (i) absolute values in range 1-3 [16] and (ii) values relative to method size in 1%-3% [14]. Now, in case of absolute tolerance value, if the statement difference between task marked desired by expert/ground-truth and tool's suggestion is one-statement then the suggested task is classified as match with tolerance 1. Similarly, difference of 2-statements will be classified as a match with tolerance 2, and so on. In this study, we use absolute tolerance values as for the benchmark used the mean method size and mean EMO size are 7.88 and 17.40 statements, respectively.

- *Tool Configuration* Approaches offer various options for fine tuning the analysis process. We use default configuration except for *top-n* suggestions option. We recorded output for top-3 and top-5 settings; these two settings have been used in literature, so we aim to analyse its impact on performance. A higher value for this setting may increase recall but would lower the precision and F-measure. Whereas, a lower value such as *n=1* can be too restrictive.

3.3 Results and Discussion

This section presents and discusses results obtained by application of the considered tools over five OSS studies. Table 2 and 3 show performance of the approaches in terms of precision, recall and F-measure for *top-n* suggestions set

https://doi.org/10.6084/m9.figshare.20206382.v1

Table 2: Performance for Top 5 suggestions

Tools	Tolerance	Precision	Recall	F measure
JExtract (794)	1	18.91	86.13	31.00
	2	19.42	88.44	31.84
	3	19.67	89.60	32.26
SEMI (492)	1	13.62	38.73	20.15
	2	18.29	52.02	27.07
	3	19.92	56.65	29.47
Segmentation (194)	1	10.88	12.14	11.48
	2	23.83	26.59	25.14
	3	35.23	39.31	37.15

Table 3: Performance for Top 3 suggestions

Tools (Suggestions)	Tolerance	Precision	Recall	F measure
JExtract (502)	1	28.06	80.92	41.67
	2	30.06	86.71	44.64
	3	30.46	87.86	45.24
SEMI (365)	1	16.44	34.68	22.30
	2	22.47	47.40	30.48
	3	24.93	52.60	33.83
Segmentation (192)	1	10.99	12.07	11.51
	2	24.08	26.44	25.21
	3	35.60	39.08	37.26

to 3 and 5. Best performance entry is highlighted by underlining for each configuration setting (that is each row). Now, we analyze the results and answer the research questions formulated before.

3.3.1 Performance hierarchy (RQ1)

- *Suggestions* Segmentation is most conservative approach for generating refactoring suggestions. It provides 1.10 suggestions per EMO (top-3 suggestions). The same for JExtract and SEMI is 2.88 and 2.09, respectively. Thus, strategy used by Segmentation can be applied by existing/new approaches to restrict the suggestions generation.

- *Precision and Recall* JExtract remains top-performer for *recall*, whereas top performer for Precision vary between JExtract and Segmentation. For tolerance 1, JExtract is top performer in both configurations, whereas for tolerance 3, Segmentation provides best precision.

- *F-measure* A high recall with comparable precision JExtract provides high F-measure except in case of top-5 suggestion configuration at tolerance 3.

3.3.2 Impact of configuration (RQ2)

- *Top-n Suggestions* We observe that a preferable value for this configuration parameter is 3 (between 3 and 5). Table 2 and 3 shows that gain in recall is 2-5% but loss in precision and F-measure is 5-10% for SEMI JExtract. JExtract exhibits most and Segmentation least sensitivity.

- *Tolerance* We note that both the absolute and relative (percentage based) criteria for tolerance are not generalizable. Absolute tolerance allows greater flexibility when EMO sizes are smaller (which is often the case) and it is too strict for large EMOs (such as 30+ lines). On the other hand, relative tolerance based on method size is not appropriate for large methods; as large methods does not guarantee relatively larger EMOs. Thus, for same sized EMO in different methods the tolerance would vary greatly (for example, tolerance for a method with 100 and 500 statements will be 1-3 and 5-15 statements respectively). For the benchmark studied, we computed median EMO and median method size for methods with 50 or more statements and found that median EMO size is 26.5 and median method size is 78.

- *Default Configuration* We observe that facilitating a user to set a default configuration settings for a session would make the process faster. For example, in this study, we needed to execute *identify EMOs* operation for different methods in same class. In such cases, resetting configuration for each method consumes additional time. Minimizing the number of clicks could be used as an indicator to design user configuration setting interface.

3.4 Threats to validity

One of the main threats to validity to our study is benchmark. It is comparatively smaller in size that is number of OSS studies included. Inclusion of additional studies would provide necessary diversity in EMOs and methods in terms of size and structure. Further, association between the individual OSSs and approaches is not evaluated, which may differ from overall performance. Finally, inclusion of additional approaches would enrich the performance hierarchy.

4. Conclusion

We presented an analysis of state-of-the-art approaches over an extended benchmark. The study shows that evaluation parameters such as benchmark and configuration settings are crucial in establishing true performance comparison of multiple approaches. Further, we show and discuss how values chosen for *top-n* and *tolerance* parameters can result in biased performance. Some results, such as best approach for recall or conservative suggestion generator, reaffirm the earlier findings. However, some other results, spe-

cially for tolerance, need to be reproduced for larger benchmark.

References

[1] D. Silva, N. Tsantalis, and M. T. Valente, "Why we refactor? confessions of github contributors," in *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*. ACM, 2016, pp. 858–870.

[2] M. Kim, T. Zimmermann, and N. Nagappan, "A field study of refactoring challenges and benefits," in *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*. ACM, 2012, p. 50.

[3] E. R. Murphy-Hill, C. Parnin, and A. P. Black, "How we refactor, and how we know it," *IEEE Trans. Software Eng.*, vol. 38, no. 1, pp. 5–18, 2012. [Online]. Available: https://doi.org/10.1109/TSE.2011.41

[4] X. Xu, C.-H. Lung, M. Zaman, and A. Srinivasan, "Program restructuring through clustering techniques," in *Source Code Analysis and Manipulation, 2004. Fourth IEEE International Workshop on*. IEEE, 2004, pp. 75–84.

[5] A. Alkhalid, M. Alshayeb, and S. Mahmoud, "Software refactoring at the function level using new adaptive k-nearest neighbor algorithm," *Advances in Engineering Software*, vol. 41, no. 10, pp. 1160–1178, 2010.

[6] A. Lakhotia and J.-C. Deprez, "Restructuring programs by tucking statements into functions," *Information and Software Technology*, vol. 40, no. 11, pp. 677–689, 1998.

[7] R. Komondoor and S. Horwitz, "Effective, automatic procedure extraction," in *11th IEEE International Workshop on Program Comprehension, 2003.* IEEE, 2003, pp. 33–42.

[8] ——, "Semantics-preserving procedure extraction," in *Proceedings of the 27th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, 2000, pp. 155–169.

[9] A. Abadi, R. Ettinger, and Y. A. Feldman, "Fine slicing," in *Fundamental Approaches to Software Engineering*. Springer, 2012, pp. 471–485.

[10] H. S. Kim, Y. R. Kwon, and I. S. Chung, "Restructuring programs through program slicing," *International Journal of Software Engineering and Knowledge Engineering*, vol. 4, no. 03, pp. 349–368, 1994.

[11] N. Tsantalis and A. Chatzigeorgiou, "Identification of extract method refactoring opportunities," in *Software Maintenance and Reengineering, 2009. CSMR'09. 13th European Conference on*. IEEE, 2009, pp. 119–128.

[12] ——, "Identification of extract method refactoring opportunities for the decomposition of methods," *Journal of Systems and Software*, vol. 84, no. 10, pp. 1757–1782, 2011.

[13] D. Silva, R. Terra, and M. T. Valente, "Recommending automated extract method refactorings," in *Proceedings of the 22nd International Conference on Program Comprehension*. ACM, 2014, pp. 146–156.

[14] S. Charalampidou, A. Ampatzoglou, A. Chatzigeorgiou, A. Gkortzis, and P. Avgeriou, "Identifying extract method refactoring opportunities based on functional relevance," *IEEE TSE*, vol. 43, no. 10, pp. 954–974, 2017.

[15] S. Xu, A. Sivaraman, S.-C. Khoo, and J. Xu, "Gems: An extract method refactoring recommender," in *2017 IEEE 28th International Symposium on Software Reliability Engineering (ISSRE)*. IEEE, 2017, pp. 24–34.

[16] O. Tiwari and R. Joshi, "Identifying extract method refactorings," in *15th Innovations in Software Engineering Conference*, 2022, pp. 1–11.

[17] M. Shahidi, M. Ashtiani, and M. Zakeri-Nasrabadi, "An automated extract method refactoring approach to correct the long method code smell," *Journal of Systems and Software*, vol. 187, p. 111221, 2022. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0164121222000048

[18] O. Tiwari, "Pluto-a synthetic benchmark for extract method refactoring," 2022. [Online]. Available: https://doi.org/10.6084/m9.figshare.20206382.v1