IJCSE International Journal of Computer Sciences and Engineering Open Access
Survey Paper Vol.-6, Issue-11, Nov 2018 E-ISSN: 2347-2693

Scaling and Testing Refactoring Preconditions in Refactoring Engines

Padakanti Divya[1*], Karanam Madhavi[2]

[1]Department of Information Technology, GRIET, Hyderabad, India
[2] Department of Computer Science and Engineering, GRIET, Hyderabad, India

[*]*Corresponding Author: divyajalender@gmail.com, Tel.: 8897203208*

Available online at: www.ijcseonline.org

Accepted: 22/Nov/2018, Published: 30/Nov/2018

Abstract— Demonstrating refactoring sound as for a formal semantics is viewed as a test. Designers compose test cases to check their refactoring implementations. However, it is troublesome and time expending to have a decent test suite since it requires complex sources of info (programs) and a prophet to check whether it is conceivable to apply the transformation. In the event that it is conceivable, the subsequent program must save the perceptible conduct. There are some computerized strategies for testing refactoring motors. In any case, they may have impediments identified with the program generator (comprehensiveness, setup, expressiveness), automation (sorts of prophets, bug classification), time utilization or sorts of refactoring that can be tried. This paper stretches out past system to test refactoring engines. It likewise clarifies the enhancement expressiveness of the program generator for testing more kinds of refactoring's, such as Extract Function. Moreover, developers simply need to determine the information's structure in an explanatory dialect. They may likewise set the system to skip some continuous test contributions to enhance performance. This additionally assesses strategy in 18 kinds of refactoring implementations of Java and distinguishes 35 bugs identified with aggregation blunders, behavioral changes, and overly strong conditions. This paper thinks about the effect of the skip on the time utilization and bug detection in this proposed method. By using a skip of 25 in the program generator, it decreases in 96%the times to test the refactoring implementations while missing only 3.9% of the bugs. In almost no time, it finds the principal failure related to aggregation blunder or behavioral change.

Keywords: Refactoring, overly strong preconditions, automated testing, program generation

I. INTRODUCTION

Characterizing and executing refactoring's is a nontrivial task since it is hard to characterize all preconditions to ensure that the transformation protects the program conduct. In fact, proving refactoring rightness for whole dialects such as Java and C comprises a test [1]. Thus, refactoring engines may have bugs [2], [3]. By and by, developers of refactoring motors utilize tests to assess the refactoring implementations. However, testing refactoring motors isn't trivial since it requires complex data sources, such as programs, and an oracle to characterize the right coming about the program or whether the transformation must be rejected. Physically composing test cases may be expensive, and in this way, it might be hard to make a good test suite considering all the dialect develops.

Scientists have proposed various automated methods for testing refactoring engines [3], [4], [5], [6].They automate four noteworthy strides of the testing procedure: (I) creating test inputs; (ii) applying the refactoring implementation; (iii) checking the yield accuracy; (iv) and classifying the identified failures into distinct bugs. In spite of the fact that these systems have identified various bugs in refactoring engines, it remains a question whether they scale to distinguish more bugs without impressive effort.

To reduce the time to test the refactoring implementations, this paper actualizes a technique to avoid some continuous test inputs [8]. Back to back programs created by DOLLY will, in general, be fundamentally the same as, possibly distinguishing a similar kind of bug. Thus, developers can set a parameter to avoid a few programs to reduce the time to test the refactoring implementations. By skirting these programs, this proposed technique can reduce the Time to First Failure (TTFF), decreasing the developer inactive time [8].

The proposed technique utilizes an arrangement of automated prophets to evaluate the rightness of the transformations identified with assemblage mistakes, behavioral changes, and overly strong conditions. In the wake of recognizing the failures, the technique utilizes an arrangement of automated bug categorizers to classify every single failing transformation into distinct bugs. For simplification, the new technique utilizes the term transformation to allude to a refactoring or a failing transformation.

Here evaluated 18 kinds of refactoring implementations of Just Add Refactoring Tools (JRRT) [9], Eclipse JDT (Java) and Eclipse CDT (C). 76 (53 new bugs) bugs in a total of 49 bugs identified with assemblage blunders, 17 bugs identified with behavioral changes, and 10 bugs identified with overly strong conditions. Among those bugs, 28 bugs in refactoring's connected inside function level.

The time utilization and bug detection have been analyzed in this proposed technique. By utilizing a skip of 25 in the program generator, it reduces in 96% the time to test the refactoring implementations while missing just 3.9% of the bugs. Moreover, by utilizing this equivalent skirt the proposed strategy locate the first failure as a rule in almost no time. In this way, the refactoring motor developer can discover a bug in the refactoring implementation generally rapidly, settle it, run the proposed technique again to discover another bug, et cetera. Before a release, tool developers can run the technique without the jump to locate some missed bugs.

Proposed system:

This technique proposes Disabling Preconditions (DP), a new technique to recognize overly strong preconditions in refactoring implementations by disabling preconditions. From now on we allude to disabling preconditions as the way toward forestalling to report messages to the client, raised by the preconditions. A message is accounted for when a precondition is unsatisfied. Proposed technique automatically create various programs as test inputs, utilizing JDOLLY. For each created program, we endeavor to apply the transformation utilizing the refactoring implementation that is being tried. At the point when the refactoring implementation rejects a transformation, it reports a message to the client depicting the issue. For every kind of message, Proposed technique distinguish code fragments identified with the precondition that yields the message. There might be various preconditions identified with each message, yet for effortlessness, we consider, for each refactoring implementation, one precondition per message in our technique. Next, Proposed technique modify the refactoring implementation to cripple the code fragments that kept the refactoring application. This technique proposes the DP changes to encourage and systematize the way toward modifying the code to permit disabling preconditions.

II. LITERATURE SURVEY

So are et al., [3] propose a Java program generator called JDOLLY for exhaustively making programs. By using JDOLLY, fashioners can show the amount of some Java constructs and confinements for the made programs by using Compound [7], a formal detail vernacular. They used JDOLLY to make more than 100,000 projects. Though JDOLLY can lessen the effort for delivering Java programs, it just makes programs with straightforward system bodies (only a solitary clarification), which isn't adequate to test refactoring's inside technique level. Moreover, altogether making programs, for few Java fabricates, can require an impressive measure of time.

[2] M. Vakilian and R. Johnson, "Alternate refactoring paths reveal usability problems", Current Integrated Development Environments (IDEs) bolster numerous refactoring's. However, programmers incredibly underutilize automated refactoring's. Ongoing examinations have connected customary ease of use testing strategies such as studies, lab studies, and meetings to discover the ease of use issues of refactoring tools. Nonetheless, these procedures can recognize just specific kinds of ease of use issues. The critical incident technique (CIT) is a general procedure that reveals ease of use issues by dissecting disturbing client connections. The strategy adjusts CIT to refactoring tools and demonstrates that other refactoring paths are indicators of the ease of use issues of refactoring tools. It characterizes another refactoring path as a sequence of client communications that contains undoing's, revealed messages, or rehashed summons of the refactoring tool.

[3] M. Mongiovi, R. Gheyi, G. Soares, L. Teixeira, and P. Borba, "Making refactoring safer through impact analysis", As of now most developers need to apply for manual advances and utilize test suites to enhance certainty that transformations connected to protest arranged (OO) and aspect-situated (AO) programs are right. Notwithstanding, it isn't easy to do manual reasoning, due to the nontrivial semantics of OO and AO dialects. Moreover, most refactoring implementations contain various bugs since it is hard to set up all conditions required for a transformation to conduct safeguarding. In this article, the new technique proposes a tool (Safe Refactor Impact) that investigates the transformation and creates tests just for the strategies impacted by a transformation distinguished by change impact analyzer (Safira). contrast the Safe Refactor Impact and the past tool (Safe Refactor) as for rightness, performance, the number of strategies passed to the automatic test suite generator, change inclusion, and the number of pertinent tests produced in 45 transformations. Safe Refactor Impact recognizes behavioral changes undetected by Safe Refactor. Moreover, it reduces the number of techniques passed to the test suite generator.

[4] G. Soares, R. Gheyi, E. Murphy-Hill, and B. Johnson, "Comparing Approaches to Analyze Refactoring Activity on Software Repositories", A few approaches have been utilized to examine proof on how developers refactor their code, whether refactoring's exercises may decrease the quantity of bugs, or enhance developers' profitability. Be that as it may, there is some negating proof in past investigations. Here recognize submitted

conduct protecting transformations in software repositories by utilizing manual examination, submit messages or dynamic investigation. Others center around distinguishing which refactoring's are connected between two programs by utilizing manual examination or static investigation. In this paper, look at the three changed approaches based on a manual investigation, submit a message (Ratzinger's approach) and dynamic examination (SAFE REFACTOR's approach) to recognize whether a couple of forms decides a refactoring, as far as behavioral protection.

[5] G. Soares, R. Gheyi, and T. Massoni, "Automated behavioral testing of refactoring engines", Refactoring is a transformation that saves the outside conduct of a program and enhances its interior quality. More often than not, arrangement mistakes and behavioral changes are maintained a strategic distance from by preconditions decided for each refactoring transformation. Be that as it may, to formally characterize these preconditions and exchange them to program checks is a rather mind-boggling task. Practically speaking, refactoring motor developers ordinarily actualize refactoring's in a specially appointed way since no rules are accessible for assessing the accuracy of refactoring implementations. Accordingly, even standard refactoring engines contain critical bugs. This paper presents a technique to test Java refactoring engines. It automates test input generation by utilizing a Java program generator that thoroughly creates programs for a given extent of Java affirmations. The refactoring under test is connected to each produced program. The technique utilizes Safe Refactor, a tool for distinguishing behavioral changes, as a prophet to evaluate the accuracy of these transformations. At long last, the technique classifies the failing transformations by the kind of behavioral change or assemblage blunder presented by them.

[6] S. Negara, N. Chen, M. Vakilian, R. Johnson, and D. Dig, "A comparative study of manual and automated refactorings", Regardless of the tremendous achievement that manual and automated refactoring has appreciated amid the last decade. Understanding the refactoring practice is critical for developers, refactoring tool manufacturers, and analysts. Numerous past approaches to consider refactorings are based on looking at code previews, which is loose, inadequate, and does not permit noting research questions that include time or think about manual and automated refactoring. This paper displays the first expanded experimental investigation that considers both manual and automated refactoring. This examination is empowered by proposed technique calculation, which derives refactorings from constant changes.

III. PROBLEM STATEMENT

In existing work utilizes Differential Testing to automatically recognize transformations dismissed by refactoring engines because of overly strong preconditions (DT technique). It automatically produces various programs as test inputs utilizing JDOLLY, a Java program generator. Next, it applies the equivalent refactoring to each test input utilizing two distinct implementations and thinks about the two outcomes. The technique utilizes SAFEREFACTOR to automatically evaluate whether a transformation protects the program conduct. SAFEREFACTOR automatically evaluates whether two variants of a program have a similar conduct via automatically producing experiments just for the normal techniques impacted by the change. To utilize this technique, developers require access to something like two refactoring engines. Be that as it may, it must be utilized if both refactoring engines execute the equivalent refactoring.

IV. IMPLEMENTATION PROCEDURE

Detecting Overly Strong Preconditions

In this section, the proposed technique to recognize overly strong preconditions in refactoring implementations utilizing the DP prophet. The Proposed technique gets as

info a refactoring implementation, the DP changes used to permit disabling the preconditions, and a few parameters to design DOLLY, such as skip, scope, and extra limitations. Each precondition checks whether the transformation may present a particular issue in the program, which can result in gathering mistakes or behavioral changes. The technique restores the adjusted refactoring implementation and all transformations that yield an arrangement of overly strong preconditions in the first refactoring implementation.

The main steps of the technique.

Step 1: Next, the refactoring implementation under test endeavors to apply the transformations to each created program. On the off chance that the refactoring implementation rejects a transformation, the messages will be gathered and answered to the client.

Step 2: For every kind of message, the refactoring implementation code reviewed and physically recognize the code fragments identified with the precondition that raises it. One assumption ought to be made, for each refactoring implementation, that there is one precondition identified with every kind of message. Then, one adjustment ought to be done such that the refactoring implementation code by adding If explanations to permit disabling the execution of the distinguished precondition utilizing the DP changes

Step 3: The objective is to apply the transformation as opposed to detailing the message again.Once the technique changes the refactoring implementation code to permit automatically disabling the preconditions, and evaluate them. For every transformation dismissed by the refactoring implementation, it automatically endeavors to apply a similar transformation again with a debilitated precondition

Step 4: In the event that the refactoring implementation rejects the transformation and reports another message, it rehashes the procedure by disabling more preconditions until the point when the refactoring implementation applies a transformation. On the off chance that the altered refactoring implementation applies the transformation and the subsequent program protects the program conduct as indicated by SAFE REFACTOR IMPACT, then the technique classifies the arrangement of impaired preconditions as overly strong

Step 5: Otherwise, it breaks down the following rejected transformation. When the classification is over a precondition as overly strong, it won't be evaluated again with other sources of info produced by DOLLY that yield a similar message. Calculation 1 condenses the fundamental advances. Next, this paper clarifies in more subtle elements the way toward disabling the preconditions.

V. DETECT OVERLY STRONG PRECONDITION TECHNIQUE

Input: refactoring implementation R, skip, scope, constraints, timeLimit, DP changes

Step 1. progs= DOLLY.generate(skip, scope, constraints);
progs' = Ø; . A set of pairs of programs and messages
msgs= Ø; . A set of all messages reported by R
Step 2. foreachprog∈progs do
msg= R.canApplyRefactoring(prog); .canApplyRefactoring yields one message, for simplicity,
ifR cannot apply it
ifmsg≠Øthen
progs'.add(hprog, msgi);
msgs.add(msg); . For simplicity, it does not show that it removes some names and keywords from msg
map= Ø; . A set of all mappings of messages to preconditions
Step 3.1. Create a class: public class ConditionsR{ public static void enableConditions() {} };
Step 3.2. foreachmsg∈msgs do
Step 3.2.1. Identify how msgis represented in R; .Specific for each refactoring engine
Step 3.2.2. Create a fresh public static boolean field (cond) in ConditionsR. Add cond= true in enableConditions;
Step 3.2.3. map.add(hmsg, condi); . It relates each message to a condition
Step 3.3. Identify how to prevent reporting messages to user in R; .Specific for each refactoring engine
R' = R; R' will contain the modified refactoring implementation
Step 3.4. foreachmsg∈msgs do
Step 3.4.1. places= Identify all places in R that can prevent reporting msgto user;
Step 3.4.2. foreach place ∈places do
R' = applyDPChange(DPChanges, R', place, msg, map); . Add if (ConditionsR.cond) {place}. Specific for each ref. engine
transformations= Ø; . A set containing all transformations applied by R'
Step 4. foreachhprog, msgi∈progs' do
Step 4.1. ConditionsR.enableConditions(); . It enables all preconditions
Step 4.2. ConditionsR.(map.getCondition(msg)) = false; . It disables a condition related to msg
Step 4.3. msg= R'.canApplyRefactoring(prog);
ifmsg∈msgs then
go to Step 4.2;
else if msg = Øthen
transformations.add(hprog, R'.applyRefactoring(prog)i); . It saves a transformation that does not yield a message
else
continue; . For simplicity, it does not focus on disabling preconditions related to messages not reported in Step 2
result= Ø;
Step 5. foreach t ∈transformations do
ifSAFEREFACTORIMPACT(t.input,t.output, timeLimit).hasSameBehavior() then
result.add(t); . It saves a behavior preserving transformation applied by R'[25]

VI. DP CHANGES IN ECLIPSE

Eclipse actualizes a class (Refactoring Status) that stores the result of the preconditions checking operation. It contains methods,such as addError, addEntry, addWarning, createStatus, createFatalErrorStatus,createErrorStatus, and createWarningStatus.

Those strategies get a message and other contentions, portraying a particular issue distinguished amid the precondition checking. The strategies began with make restore a Refactoring Status Protest. The messages are stored in the refactoring. Properties record. A field from the Refactoring Core Messages class speaks to them. They can be specifically gotten to by a field call or through a variable, parameter of the strategy, or the arrival of a technique called.

The refactoring implementations of Eclipse check the status of a transformation, in a Refactoring Status protest, in the wake of assessing the preconditions. If it contains some notice or mistakes messages, Eclipse rejects the transformation and reports the messages to the client. This paper proposes the Eclipse DP changes by breaking down the littlest code fragment, which requirements to incapacitate for maintaining a strategic distance from the motor to

include a new blunder or cautioning status in a Refactoring Status object. DP Change 2 keeps Eclipse from announcing mistake messages.

VII. RESULTS EVOLUTIONS

The proposed technique chose up to 10 refactoring implementations from Eclipse JDT 3.7, NetBeans 7.0.1. Afterward, a new form was released with enhancements and bug settling (which likewise call JRRTv2); this new form was additionally subject to analysis. Table 1 demonstrates all evaluated refactorings. The evaluated refactoring's center around a delegate set of program structures. Moreover, a study did demonstrate the Eclipse JDT refactoring's that Java developers utilize most: Rename, Move Method, Extract Method, Pull Up Method, and Add Parameter. Four of these are evaluated in this article. The Move Method refactoring was not bolstered by NetBeans When that this article was composed.

Table 1: Summary of scope and constraints for each refactoring

Refactoring	Scope (P - C - F - M)	Main constraint
Rename Class	2-3-0-3	some class
Rename Method	2-3-0-3	some Method
Rename Field	2-3-2-1	some Field
Push Down Method	2-3-0-4	some c:Class k someSubClass[c] and someMethod[c]
Push Down Field	2-3-2-1	some c:Class k someSubClass[c] and someField[c]
Pull Up Method	2-3-0-4	some c:Class k someParent[c] and someMethod[c]
Pull Up Field	2-3-2-1	some c:Class k someParent[c] and someField[c]
Encapsulate Field	2-3-1-3	some Field
Move Method	2-3-1-3	some c:Class k someTargetClassField[c] and someMethodToMove[c]
Add Parameter	2-3-0-3	some Method

Scope = Package (P) - Class (C) - Field (F) - Method (M).

Table 2 synopses the experiment results. Segments Program and Time demonstrate the number of programs created by JDOLLY for each refactoring and the normal time for testing the refactoring implementations from each engine. Columns Comp. error., Behav. cha. and Overly strong demonstrates the total number of transformations connected by Eclipse, Net-Beans, JRRTv1, and JRRTv2 that delivered gathering errors, behavioral changes, and that were not connected due to overly strong conditions, respectively. Considering all refactorings, JDOLLY produced 153,444 programs, and new technique distinguished 43,235 transformations with assemblage blunders, 27,597 ones with behavioral changes, and 70,832 that were not connected due to overly strong conditions. Even, however, Eclipse, JRRT, and NetBeans have their own test suites, new technique recognized 120 (likely) remarkablebugs.

Table 2: Overall experimental results

Refactoring	Program	Time(h)	Comp. error.	Behav. cha.	Overly strong
Rename Class	15322	6.7	4368	160	4528
Rename Method	11263	6.9	2290	1713	4003
Rename Field	19424	29.3	894	1834	2728
Push Down Method	20544	11.9	13579	3312	16891
Push Down Field	11936	6	7231	119	7350
Pull Up Method	8937	7.3	3867	1363	5230
Pull Up Field	10927	8.6	1726	785	2511
Encapsulate Field	2000	2.5	472	1220	1692
Move Method	22905	10.3	1321	12289	13610
Add Parameter	30186	34.69	7487	4802	12289
Total	153444	124.19	43235	27597	70832

Table 3 outlines the bugs answered to Eclipse JDT, Net-Beans and JRRT. new technique distinguished 34 overly powerless preconditions in Eclipse. Albeit every one of them was acknowledged by the Eclipse developers, 16 of them were named as copied. Up until now, they have settled only two of them. In NetBeans, new technique recognized 51 overly powerless preconditions. Net-Beans group has officially acknowledged 30 of them and settled 7 bugs. In the interim, here 24 overly frail preconditions to JRRTv1, from which 20 were acknowledged and settled (4 of the bugs were not viewed as bugs because of a shut world assumption of JRRT developers)it additionally announced more 11 bugs to JRRTv2, from which 6 were acknowledged and settled. JRRT group additionally fused experiments into their test suite.

The proposed technique did not find overly strong preconditions in NetBeans but identified 16 ones in Eclipse.

Table 3: Summary of reported bugs.

Engine	Submitted	Accepted	Duplicated	Not accepted	Not answered	fixed
Eclipse	34	34	16	0	0	2

VIII. CONCLUSION

Bridges the bugs answered to Eclipse JDT, Net-Beans and JRRT. new technique recognized 34 overly frail preconditions in Eclipse. Albeit every one of them was acknowledged by the Eclipse developers, 16 of them were marked as copied. Up until this point, they have settled only two of them. In NetBeans, new technique distinguished 51 overly feeble preconditions. Net-Beans group has officially acknowledged 30 of them and settled 7 bugs. In the interim, here 24 overly powerless preconditions to JRRTv1, from which 20 were acknowledged and settled (4 of the bugs were not viewed as bugs because of a shut world assumption of JRRT developers)it likewise revealed more 11 bugs to JRRTv2, from which 6 were acknowledged and settled. JRRT group likewise consolidated experiments into their test suite.

IX. REFERENCES

[1] M. Schafer, T. Ekman, and O. de Moor, “Challenge proposal: verification of refactorings,” In PLPV, 2008, pp. 67–72.

[2] G. Soares, M. Mongiovi, and R. Gheyi, "Identifying overly strong conditions in refactoring implementations," in ICSM, 2011, pp. 173–182.

[3] G. Soares, R. Gheyi, and T. Massoni, "Automated behavioral testing of refactoring engines,” IEEE Transactions on Software Engineering, vol. 39, pp. 147–162, 2013.

[4] B. Daniel, D. Dig, K. Garcia, and D. Marinov, “Automated testing of refactoring engines,” in FSE, 2007, pp. 185–194.

[5] M. Gligoric, T. Gvero, V. Jagannath, S. Khurshid, V. Kuncak, and. Marinov, “Test generation through programming in UDITA,” in ICSE,2010, pp. 225–234.

[6] M. Gligoric, F. Behrang, Y. Li, J. Overbey, M. Hafiz, and D. Marinov, on reals"Systematic testing of refactoring engines software projects," inECOOP, 2013, pp. 629–653.

[7] D. Jackson, “Software Abstractions: Logic, Language, and Analysis.Revised edition. “The MIT Press, 2012.

[8] V. Jagannath, Y. Lee, B. Daniel, and D. Marinov, “Reducing the costs of bounded-exhaustive testing,” in FASE, 2009, pp. 171–185.

[9] M. Sch¨afer and O. Moor, “Specifying and implementing refactorings," in OOPSLA, 2010, pp. 286–301.

[10] D. Jackson, I. Schechter, and H. Shlyahter, “Alcoa: the Alloy constraint analyzer,” in ICSE, 2000, pp. 730–733.

[11] M. Mongiovi, R. Gheyi, G. Soares, L. Teixeira, and P. Borba, “Makingrefactoring safer through impact analysis,” SCP, 2014, In press.

[12] G. Soares, R. Gheyi, D. Serey, and T. Massoni, “Making program refactoring safer,” IEEE Software, vol. 27, pp. 52–57, 2010.

[13] W. Mckeeman, “Differential testing for software,” Digital TechnicalJournal, vol. 10, no. 1, pp. 100–107, 1998.

[14] E. Torlak and D. Jackson, “Kodkod: A relational model finder,” inTACAS. Wiley, 2007, pp. 632–647.

[15] G. Soares, R. Gheyi, E. Murphy-Hill, and B. Johnson, “comparing approaches to Analyze Refactoring Activity on Software Repositories, "JSS, pp. 1006–1022, 2013.

[16] W. Opdyke, “Refactoring Object-Oriented frameworks,” Ph.D. dissertation, the University of Illinois at Urbana-Champaign, 1992.

[17] L. Tokuda and D. Batory, “Evolving object-oriented designs with refactorings,” ASE, vol. 8, pp. 89–120, 2001.

[18] A. Garrido and R. Johnson, “Refactoring C with conditional compilation," in ASE, 2003, pp. 323–326.

[19] A. Garrido and R. E. Johnson, “Analyzing multiple configurations of a program,” in ICSM, 2005, pp. 379–388.

[20] F. Steinmann and A. Thies, “From public to private to absent: RefactoringJava programs under constrained accessibility,” in ECOOP, 2009, pp.419–443.

[21] P. Borba, A. Sampaio, A. Cavalcanti, and M. Cornelio, “Algebraic reasoning for Object-Oriented programming,” “SCP, vol. 52, pp. 53–100,2004.

[22] L. Silva, A. Sampaio, and Z. Liu, “Laws of Object-Orientation with reference semantics,” in SEFM, 2008, pp. 217–226.

[23] H. Li and S. Thompson, “Testing ErlangRefactorings withQuickCheck,” in IFL, 2008, pp. 19–36.

[24] M. Vakilian and R. E. Johnson, “Alternate refactoring paths reveal usability problems,” in ICSE, 2014, pp. 1–11.

[25] Melina Mongiovi Member, Rohit Gheyi, Gustavo Soares, Márcio Ribeiro, Paulo Borba, "Detecting overly strong preconditions in refactoring engines" IEEE 2017.

[26] Geeta Bagade, Shashank Joshi "Analysis of Aspect-Oriented Systems: Refactorings using AspectJ" International Journal of Computer Sciences and Engineering,Vol.4, Issue .5, pp.76-80, May-2016

[27] Nagaveni, A. Ananda Rao, P. Radhika Raju, "Testing Refactoring Implementations of Object-Oriented Systems"International Journal of Computer Sciences and Engineering, Vol.6 , Issue.7,pp.530-534, Jul-2018

Authors Profile

Padakanti Divya is currently pursuing Masterof Technology from Gokaraju Rangaraju Institute of Engineering and Technology, Hyderabad. She has pursued Bachelor of Technology in 2016 from Balaji Institute of Technology and Science, Warangal. Her main research work focuses on Software Engineering.

*Karanam Madhavi***,** working as a Professor in Computer Science and Engineering Department, Gokaraju Rangaraju Institute of Engineering and Technology. She has completed her B.E in 1997, M.Tech from JNTUA in 2003 and awarded Ph.D. from JNTUA in 2013. She has 19 years of teaching experience. She has published several papers in reputed international journals and international conferences. Her research interest includes software engineering, Model Driven Engineering, Data Mining, and Mobile software engineering.

