ISSN (Online) : 2278-1021
ISSN (Print) : 2319-5940

International Journal of Advanced Research in Computer and Communication Engineering
Vol. 3, Issue 5, May 2014

Detection of Optimal Refactoring Plans For Resolution of Code Smells

PANDIYAVATHI.T[1], MANOCHANDAR.T[2]

Student M.E. (SWE), Anna University, Chennai, India[1]

Assistant Professor, VRS College of Engineering & Technology, Arasur, India[2]

Abstract: Bad smells can be detected using various kinds of automated tools. The problem behind this is clear, where the smell being refactored may have dependency in increasing or resolving some other kind of smell which in turn results in increased effort and time. A smell being resolved may affect the presence of an existing smell or introduces some more conflicts into the system. The works discussed in the literature leads to lot of human effort and enormous amount of maintenance time. In order to reduce the manual work load and to obtain the better source code for easy maintenance and to obtain a better refactoring sequence this work proposes optimal refactoring plans that enhances detection and sequencing of bad smells. The selected code smells are sequenced to avoid RIPPLE EFFECT. The refactoring methods that have to be applied to the source code are also ordered based on the fitness criteria using a genetic algorithm.

I. INTRODUCTION

The design of software systems can exhibit several problems which can be either due to inefficient analysis and design during the initial construction of the software or sometimes may be due to software ageing since software quality degenerates with time. According to Fowler[1], design problems appear as "bad smells" at code or design level and the process of removing them is called refactoring where the software structure is improved without any modification in the behavior. It can be briefly defined as "Restructuring of internal structure of object oriented software to improve the quality while the software's external behavior remains unchanged" – Fowler[1]

Refactoring improves the design of software and makes software easier to understand. It also helps us to find bugs. Bad smells can be detected using various kinds of automated tools. The problem behind this is clear, where the smell being refactored may have dependency in increasing or resolving some other kind of smell which in turn results in increased effort and time. A smell being resolved may affect the presence of an existing smell or introduces some more conflicts into the system.

PROPOSED WORK

In our approach only one tool is used for calculating the metrics. Based on the metrics of the given source code, defects in the source code is detected and the stored refactoring list elements are called based on the detected smell. This sequence is given as input to the genetic algorithm which gives the proper ordered sequence to perform refactoring which reduces human effort and also gives optimal sequence for refactoring. Complexity of the current approach lies in finding the fitness function based on which the crossover and mutation in the genetic algorithm are done.

It is suggested that often manual refactoring will be the most effective one among all the others. Since it increases the time factor, we detect the smells using different strategy and finally apply the sequence of refactoring methods to the code which involves manual checking along with the defect resolution.

II. FEASIBILITY ANALYSIS

Using Development history is a resolution technique where the past refactoring can be used to do the current plans since there is higher probability of the already refactored code to inject new smells. The technique used is multi-objective evolutionary algorithm that adapts non-dominated sorting genetic algorithm (NSGA-II)[2].

A Monitor-Based Instant Software Refactoring framework[3] is developed to conduct more refactoring in which changes in the source code are instantly analyzed by a monitor running in the background. If smells are introduced, monitor by itself invokes smell detection tools to inform the developer to resolve the smells. This facilitates instant refactoring decisions being made as soon as the smell is been detected. This solution is found to reduce the total number of smells by 51 percent.

Identification of generalization refactoring [4] in the code allows to related classes, shared functions with interfaces and implementations in java. The refactoring rules can be indentified using conceptual relationship, implementation, similarity, structural correspondence and inheritance hierarchies. This helps in resolving the smells that are highly related, using the tools generated using this approach.

BAD SMELLS

The key issue can be solved by a kind-level scheme that arranges the detection and resolution sequences of different kinds of bad smells. Arranging detection and resolution sequences[5] can be done by analyzing **the relationship among different bad smells**. Based on the analyzed sequence, smells are detected and resolved using several kind of automated tools like JDeodorant(Feature envy), PMD(duplicate code) based on the type of smell. This greatly minimizes human effort but the tool may miss some bad smells in some cases. Since there exists only 8 important code smells being analyzed, in the proposed system some more smells can be introduced and sequencing is done to improve the quality of the code. Detection tool can be used in detecting a specific kind of smell. Comparison between the refactoring without sequencing and with sequencing can be made to confirm that the proposed approach provides better results than simply refactoring the code without considering the relationship between the smells. Batch model of refactoring is used where the system is thoroughly refactored at one attempt. Metrics can also be calculated to look into the results of the proposed system. This approach can be evaluated on application in future work for validation. In the proposed system smells namely move method, move field and dead code can be added as additional smell detection and evaluation.

To specify what kind of source code should be restructured, Fowler [1] proposed the concept of bad smells. They proposed and described 22 bad smells in object-oriented systems. They also associated refactoring rules with these bad smells, suggesting how to resolve these bad smells. Bad smells in specific domains have also been proposed. Srivisut and Muenchaisri defined some bad smells in aspect-oriented software, and proposed approaches to detect them. Van Deursen Test Smells indicating problems in test code. The impact of bad smells has also been analyzed.

Lozano[5] assessed the impact of bad smells, i.e., the extent to which different bad smells influence software maintainability. They argued that it is possible to analyze the impact of bad smells by analyzing historical information. With the impact in mind, it is possible to assess code quality by detecting and visualizing bad smells. Van Emden and Moonen[6] implemented a code browser for detecting and visualizing code smells, and assessed the quality of code according to the visual representation. Detecting bad smells is critical and time-consuming. Therefore, automating detection is essential. Tsantalis[7] proposed an approach to identifying and removing type-checking bad smells which is implemented in an prototype tool named JDeodorant. Fokaefs[8] proposed an Eclipse plug-in to identify and resolve feature envy bad smells. Clones, one of the most common bad smells, have been investigated for a long time, and dozens of detection algorithms have been proposed to detect them. Moha proposed a language for formalizing bad smells, and a framework for automatically generating detection algorithms for the formalized bad smells.

RELATIONSHIPS AMONG BAD SMELLS

Relationships among bad smells have also been investigated. Wake classified bad smells into two categories: bad smells within classes and bad smells between classes. Meszaros [10] classified test smells into code smells, behavior smells, and project smells. Mantyla[11] analyzed the correlations among bad smells by investing the frequency with which each pair of bad smells appears in the same module. They found that bad smells within the same category are more likely to appear together. The work aimed to simplify the comprehension of bad smells, instead of refactoring activities.

Pietrzak and Walter[12] investigated the intersmell relationships to facilitate the detection of bad smells. They argued that detected or rejected bad smells might imply the existence or absence of other bad smells. Their work aimed to simplify the detection of bad smells, whereas our work focuses on the detection and resolution sequences of different kinds of bad smells.

GENETIC ALGORITHM

Genetic algorithm was first proposed by Goldberg et al in 1989. In the computer science field of artificial intelligence, a genetic algorithm is a search heuristic that mimics the process of natural selection. This heuristic is routinely used to generate useful solutions to optimization and search problems. To insure the detection of maintainability defects, several automated detection techniques have been proposed by Moha. The vast majority of these techniques rely on declarative rule. In these settings, rules are manually defined to identify the key symptoms that characterize a defect. These symptoms are described using quantitative metrics, structural, and/or lexical information. For example, large classes have different symptoms like the high number of attributes, relations and methods that can be expressed using quantitative metrics.

Beside the previous approaches, one notices the availability of defect repositories in many companies, where defects in projects under development are manually identified, corrected and documented. However, this valuable knowledge is not used to mine regularities about defect manifestations, although these regularities could be exploited both to detect and correct defects.

In this paper, we propose to overcome some of the above-mentioned limitations with a two-step approach based on the use of defect examples generally available in defect repositories of software developing companies:

(1) Detection-identification of defects, and
(2) Correction of detected defects.

Instead of specifying rules manually for detecting each defect type, or semi automatically using defect definitions, we extract these rules from instances of maintainability defects. This is achieved using Genetic Programming (GP).

We generate correction solutions based on combinations of refactoring operations, taking in consideration two objectives:

(1) Maximizing code quality by minimizing the number of detected defects using detection rules generated
(2) Minimizing the effort needed to apply refactoring operations.

Thus, we propose to consider refactoring as a multi-objective optimization problem instead of a single-objective approach.

In all these previous work discussed above, we have briefly discussed the resolution sequences of bad smells, but no evaluation or discussion was presented. In this paper, an approach known as Genetic algorithm is presented to evaluate the metrics through which the code is going to be analyzed. Furthermore, the need for resolution sequences is illustrated.

III. REFACTORING STEPS INVOLVED

1. Perform pair wise analysis among each selected code smells.
2. Draw directed graph based on the analysis made above
3. Apply topological sorting to obtain ordered code smells.
4. Generate detection rules using combinations of metrics and thresholds
5. Collect refactorings methods to be processed after the defect detection
6. Generate/ frame list of possible refactoring methods like pull up, move method, extract method
7. Apply natural evolution techniques like genetic algorithm with input as the outcomes of steps 4,5,6
8. Perform crossover and mutation along with the elitism property in the above algorithm
9. Obtain Optimal solution with sequenced refactoring plans

Module 1- Sequencing Code Smells
Study of smells selected for the problem and analyzing its complexities, Pair wise analysis, Generate DAG and Sequence the code smells using topological sorting algorithm

Module 2- Detection Of Smells Using Automated Tools
Select fragments of code, Inject smells into the code, Use automated tools to detect the smells in the code where PMD detects dead code, duplicate code, long method, long parameter list and Checkstyle detects feature envy and finally check the ease of tools for detection of the code smells.

Module 3- Metric Calibration And Refactoring Plans
It aims on implementing our current ideas on detecting the code smells. We use a tool named “metrics” which is an eclipse plugin to find the metrics in the code which is followed by metrics calibration where the detected/calculated metrics is compared against the threshold values. Based on the result appropriate method is called and code smell is detected. If there is no smells in the code, then the resultant array will be empty. This array will be given as input to the next module. Initially generate Design defects rules and then generate list of refactoring plans

Module 4- Extracting Optimal Refactoring Solution
Use the results from module 3 as input to the Genetic Algorithm. Encoding involves conversion of Array to a feasible input value in which the processing is going to be done and Selection involves selecting individuals for the population using tournament selection method or Roulette wheel selection and finally evaluate the individuals through fitness function.

a. Perform crossover
b. Perform mutations
c. Generate optimal refactoring solution

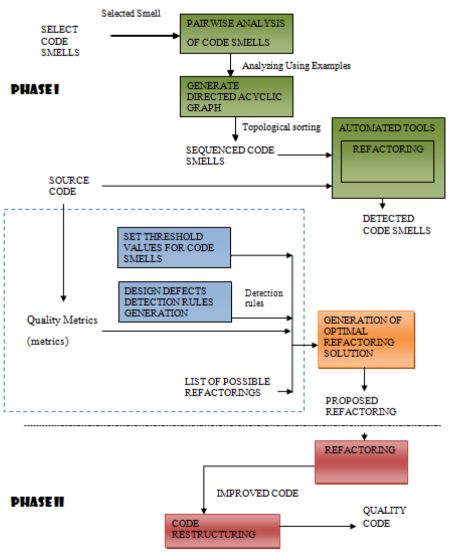


Fig. Architecture Diagram

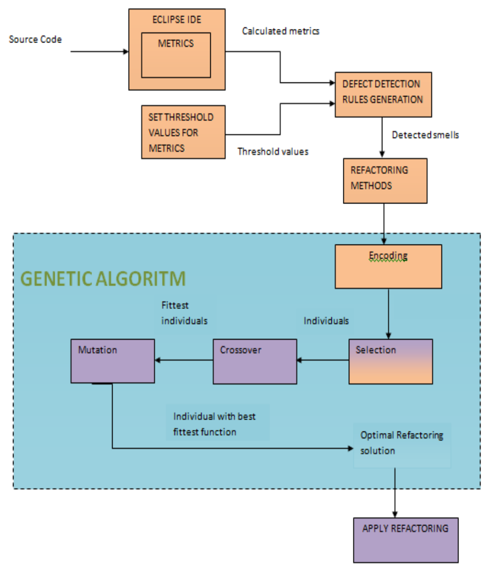


Fig. Genetic Algorithm

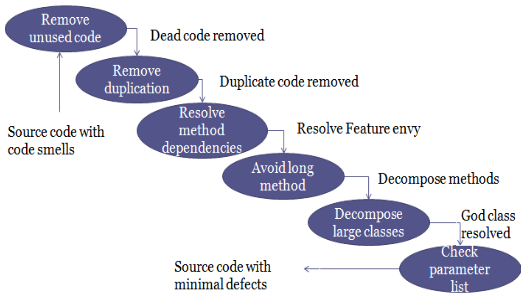


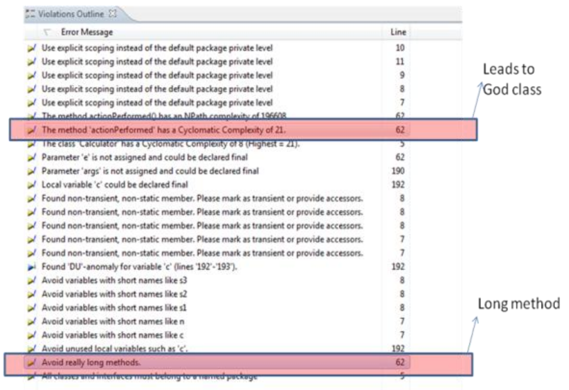
Fig. Generated Sequence

Sequenced Code Using Topological Sorting:
Dead code→ Duplicate code→Feature envy→long method→ god class→ long parameter list

IV. DETECTION OF CODE SMELLS USING AUTOMATED TOOLS

Tools used: pmd,checkstyle

1. Inject smells into the code
2. Use automated tools to detect the smells in the code


Violations Outline

Error Message	Line
Use explicit scoping instead of the default package private level	10
Use explicit scoping instead of the default package private level	11
Use explicit scoping instead of the default package private level	9
Use explicit scoping instead of the default package private level	8
Use explicit scoping instead of the default package private level	7
The method actionPerformed() has an NPath complexity of 196608	62
The method 'actionPerformed' has a Cyclomatic Complexity of 21.	62
The class 'Calculator' has a Cyclomatic Complexity of 8 (Highest = 21).	5
Parameter 'e' is not assigned and could be declared final	62
Parameter 'args' is not assigned and could be declared final	190
Local variable 'c' could be declared final	192
Found non-transient, non-static member. Please mark as transient or provide accessors.	8
Found non-transient, non-static member. Please mark as transient or provide accessors.	8
Found non-transient, non-static member. Please mark as transient or provide accessors.	8
Found non-transient, non-static member. Please mark as transient or provide accessors.	7
Found non-transient, non-static member. Please mark as transient or provide accessors.	7
Found 'DU'-anomaly for variable 'c' (lines '192'-'193').	192
Avoid variables with short names like s3	8
Avoid variables with short names like s2	8
Avoid variables with short names like s1	8
Avoid variables with short names like n	7
Avoid variables with short names like c	7
Avoid unused local variables such as 'c'.	192
Avoid really long methods.	62
All classes and interfaces must belong to a named package	5

Fig. Output Of Automated Tool(Long Method & God Class)

Flaws in using tools:

1. Tools used here can find the code smells that exist in the code. But not all the code smells are detected using a single tool
2. There are more than 22 code smells as proposed by Fowler and plug-in can detect not more than 5 smells in a code

Steps:

a. "Metrics" runs on Eclipse IDE, where the result from this tool is taken as input to our work.
b. We use netbeans to run our java program. In our program design rules are generated by setting threshold values to each metric value.
c. Give the output generated from the tool as input to the program
d. Design rules with conditional statements are checked and appropriate refactoring solution is stored to the array
e. Null value is pushed into the array if the metric value doesn't exceeds the threshold value
f. Resultant array is the input to be fed into the genetic algorithm

V. EXTRACTING OPTIMAL REFACTORING SOLUTION

The solution generated using the genetic algorithm is applied to the source code by using Eclipse IDE to do the corresponding refactorings. Genetic Algorithms (GAs) are an iterative approach which is described as analogous to evolutionary processes for solving search and optimization problems. We find the individuals and combine them to create a population with higher fitness.

Sample Input: [EM, EC, MM, PDM, 0]
1011011100010110011011011001000001100001011001000001001101100 11

Sample Output:
11110000000000000000000000000000000000011111101
0010100100101101000

Problem

Problem exists in the algorithm since the input is converted to binary strings, after the computation of the algorithm on the bit strings, due to crossover and mutation the number of strings and the sequence is changed. Because of this, illegal results are produced which has been found on latter stage. Due to this limitation, studies are done and an crossover technique which avoids this problem is proposed.

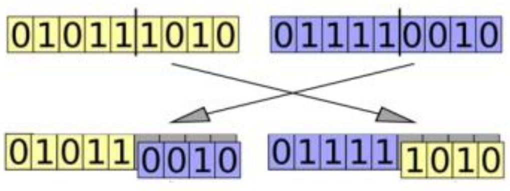


Fig. One-Point Crossover

Solution

The crossover technique which avoids the production of illegal children is analyzed and found to be PARTIAL MAPPED CROSSVER technique.

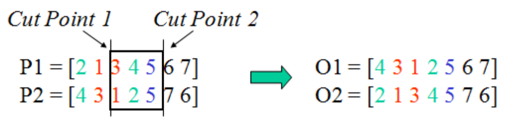


Fig. Partially-Mapped Crossover

VI. CONCLUSION

In this work, the optimal sequence for refactoring is generated using genetic algorithm. Some problem exists while doing Encoding and Crossover in genetic algorithm. However the problem will be handled using techniques available. In previous work, scheduling of the code smell is done. Refactoring of the code smells solely depends on the tool and in case of existence of code smells ever after refactoring leads in increasing the human effort.

To avoid these problems we amended,

1. Topological sorting algorithm for sequencing the major selected code smells
2. Automated tools to define that the usage of tools has many flaws which has to be solved
3. Metric calibration where the metrics calculated from the source code is used for detecting the defects and finding their related refactoring methods from the list of refactoring methods stored in an array list
4. Genetic algorithm which takes the array as input and thereby encode, select individual, do crossover and mutation and finally produces the optimal solution to the problem. Number of iterations needed for getting the optimal solution is also obtained using this approach.

5.1. LIMITATIONS

We have encountered a problem with the illegal child generation in the genetic algorithm. The main reason for encoding from array of strings to bit strings is for easy computing of genetic operators and during study it is found that for optimal solution generation using binary encoding is the better way. But this condition holds bad for our solution domain. This resulted in rework of the genetic algorithm by assigned some char values or numbers to the refactoring methods.

Problem is therefore analyzed and the crossover technique which produce legal children is found to be **partially-mapped crossover technique** and later on this technique is been implemented and results are obtained along with the fitness values.

```
31204; Fit = 105.62002441935006
02134; Fit = 105.62002441935006
12304; Fit = 103.93666659677497
03214; Fit = 103.93666659677497
12304; Fit = 103.93666659677497
30124; Fit = 105.07115674260575
12034; Fit = 108.75444221231609
32104; Fit = 95.83511235190642
30214; Fit = 108.75444221231609
30214; Fit = 108.75444221231609
32104; Fit = 95.83511235190642
12304; Fit = 103.93666659677497
30124; Fit = 105.07115674260575
30124; Fit = 105.07115674260575
```

SAMPLE INPUT:	EM,EC,0,0,MM
EXPECTED OUTPUT:	0,EC,0,EM,MM
ACTUAL OUTPUT:	0,EC,EM,0,MM

VII. RESULTS

INDIVIDUAL	FITNESS	NUMBER OF OCCURENCE
30214	108.75	14
31204	105.62	15
02134	105.62	18
32104	95.83	8
12304	103.93	9
03214	103.93	12
21034	105.07	4
12034	108.75	4
01234	95.83	3
30124	105.07	6
12034	108.75	4

Fig. Output Of Genetic Algorithm

tp(correct result)	fp(unexpected)
6	3
fn(missing)	tn(correct absence of result
1	4

ANALYSED RESULTS

1. **Precision:** fraction of retrieved docs that are relevant = P(relevant|retrieved) Precision P = tp/(tp + fp) P=66.66
2. **Recall:** fraction of relevant docs that are retrieved= P(retrieved|relevant) Recall R = tp/(tp + fn) R=85.71
3. **Accuracy:** A=(tp+tn)/(tp+tn+fp+fn) = 71.428 f-score= 2*(P.R)/(P+R) = 74.99

GRAPH REPRESENTATION

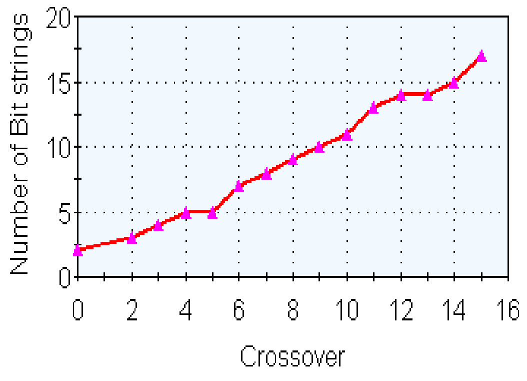


Fig. Cause Of Illegal Children

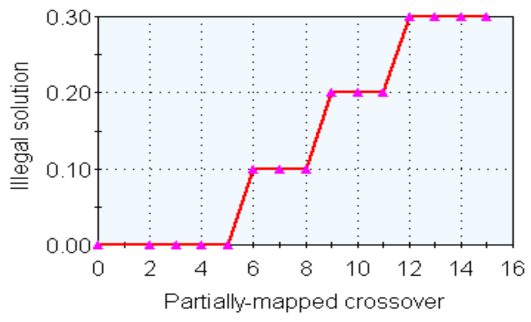


Fig. Usage of Partially mapped crossover

REFERENCES

[1] Martin Fowler, Kent Beck (Contributor), John Brant (Contributor), William Opdyke, don Roberts(2002), "Refactoring: Improving the Design of Existing Code".
[2] Ali ouni, Marouane Kessentini, Houari Sahraoui, Mohamed Salah Hamdi, "The Use of Development History in Software Refactoring Using a Multi-Objective Evolutionary Algorithm", proc. GECCO'13 pages 1461-1468 and ACM 2013.
[3] Hui Liu, Xue Guo and Weizhong Shao, "Monitor-Based Instant Software Refactoring", IEEE Transactions of Software Engineering 2013.
[4] Hui Liu, Zhendong Niu, Zhiyi Ma, Weizhong Shao, "Identification of generalization refactoring opportunities", Automated Software Engineering: Volume 20, Issue 1 (2013), Page 81-110
[5] A. Lozano, M. Wermelinger, and B. Nuseibeh, "Assessing the Impact of Bad Smells Using Historical Information," Proc. Ninth Int'l Workshop Principles of Software Evolution: In Conjunction with the Sixth ESEC/FSE Joint Meeting, pp. 31-34, 2007.
[6] E. van Emden and L. Moonen, "Java Quality Assurance by Detecting Code Smells," Proc. Ninth Working Conf. Reverse Eng., pp. 97-106, 2002.
[7] N. Tsantalis, T. Chaikalis, and A. Chatzigeorgiou, "Jdeodorant: Identification and Removal of Type-Checking Bad Smells," Proc. 12th European Conf. Software Maintenance and Reeng., pp. 329-331, Apr. 2008.
[8] M. Fokaefs, N. Tsantalis, and A. Chatzigeorgiou, "Jdeodorant: Identification and Removal of Feature Envy Bad Smells," Proc. IEEE Int'l Conf. Software Maintenance, pp. 519-520, Oct. 2007.
[9] M. Mantyla, J. Vanhanen, and C. Lassenius, "Bad Smells—Humans as Code Critics," Proc. IEEE 20th Int'l Conf. Software Maintenance, pp. 399-408, Sept. 2004.
[10] G. Meszaros, xUnit Test Patterns: Refactoring Test Code. Addison-Wesley, 2007.
[11] M. Mantyla, J. Vanhanen, and C. Lassenius, "A Taxonomy and an Initial Empirical Study of Bad Smells in Code," Proc. Int'l Conf. Software Maintenance, pp. 381-384, 2003.

[12] B. Pietrzak and B. Walter, "Leveraging Code Smell Detection with Inter-Smell Relations," Proc. Seventh Int'l Conf. Extreme Programming and Agile Processes in Software Eng., pp. 75-84, June 2007.
[13] T. Mens, G. Taentzer, and O. Runge, "Analysing Refactoring Dependencies Using Graph Transformation," Software and Systems Modeling, vol. 6, no. 3, pp. 269-285, Sept. 2009.
[14] H. Liu, L. Yang, Z. Niu, Z. Ma, and W. Shao, "Facilitating Software Refactoring with Appropriate Resolution Order of Bad Smells," Proc. Seventh Joint Meeting of the European Software Eng. Conf. and the ACM SIGSOFT Symp. the Foundations of Software Eng., pp. 265- 268, 2009.
[15] http://www.eclipse.org/downloads
[16] http://pmd.sourceforge.net/cpd.html
[17] Core Java vol-1 Fundamenals,Eighth edition by Cay S.Horstmann and Gary Cornell
[18] Hui Liu, Zhiyi Ma, Weizhong Shao and Zhendong Niu, "Schedule of Bad Smell Detection and Resolution: A New Way to Save Effort", IEEE Transactions on Software Engineering, vol.38, no.1, pp.220-235, Jan.-Feb. 2012
[19] http://www.myreaders.info/html/artificial_intelligence.html
[20] Integrating Code Smells" Detection with Refactoring Tool, a report submitted by Kwankamol Nongpon August 2012
[21] Tools - http://www.Sourceforgenet.com- pmd, checkstyle
[22] Automated refactoring: a step towards enhancing the comprehensibility of legacy software systems by Isaac D. Griffith.

BIOGRAPHIES

T. PANDIYAVATHI was born in Tamilnadu, India in 1991. She moved to Villupuram where she completed her Schooling by 2008 and Bachelor of Degree in University college of Engineering Villupuram(A constituent college of Anna university Chennai) in 2012. She is pursuing Masters in Software Engineering in CEG, Anna University Chennai. Software quality, Agents, Data mining and Data structures are her interests.

T. MANOCHANDAR was born in Tamilnadu, India in 1988. He received B.E. in Electronics and Communication Engineering from Kamban Engineering College (A constituent college of Anna University, Chennai) in 2009 and M.E. in Communication Systems from Prathyusha Institute of Technology and Management, Tiruvallur (A constituent college of Anna University, Chennai)in 2012.He is interested in the fields of Wireless Communication, Neural Networks, Image Processing and Embedded Systems. He is an associate member of the IRED and life member of ISTE and IAENG. He is Assistant Professor of Electronics and Communication Engineering in VRS college of Engineering and Technology, Arasur (A constituent college of Anna University, Chennai).