I.J.Modern Education and Computer Science, **2013, 11, 57-63**
Published Online November 2013 in MECS (http://www.mecs-press.org/)
DOI: 10.5815/ijmecs.2013.11.08

Aspectual Analysis of Legacy Systems: Code Smells and Transformations in C

Zeba Khanam, S.A.M Rizvi
Department of Computer Science, Jamia Millia Islamia.
Email:zebs_khan@yahoo.co.in

Abstract — This paper explores the various code smells or the so called bad code symptoms present in procedural C software. The code smells are analyzed in the light of aspect oriented programming. The intention is to handle the code smells with aspect oriented constructs as it offers more versatile decomposition techniques than the traditional modularization techniques, for software evolution and understandability. The code smells are described at the function and program level. The code smells are followed by the aspect oriented transformations that may be required in order to improve the code quality.

Index Terms — Code Smells, Aspect Oriented Programming, Refactoring, Code Transformations.

I. INTRODUCTION

The code smells helps to reacquire a better understanding of bad design in the code and subsequently leads to good design solutions by revealing sensible evolutions to those solutions. Code smells do not represent an error in the code—they are not technically or syntactically incorrect and don't currently prevent the program from functioning. Instead, they indicate weaknesses in design or code that may hinder further code development or modification or increase the risk of bugs or failures in the future. Bad smells are useful in suggesting undesirable solutions that should be removed or corrected. But extraction of the code smells from the source code cannot be achieved randomly. Though many researchers have worked on exploring the code smells [8] [11] [13] [16] in object oriented software, the procedural code smells have not been investigated in detail. The term code smell was first introduced by Fowler and he detected the presence of code smells wherever the code was problematic and unmanageable and suggested that the code should be transformed. Though there are a number of related studies that investigates the applicability of aspect-oriented techniques to various (domain specific) crosscutting concerns have been performed. The work described by [10][14] have discussed the advantages of using aspect oriented software development (AOSD) such as code duplication and improved cohesion and have proposed guidelines for preparing the code for isolating concerns and performing the necessary transformations. [9] [13] also investigated the transformations and refactorings required to make the code more efficient and understandable. Similarly, [7] have also assessed the impact of crosscutting concerns on software quality. He developed a new technique called prune dependency analysis to locate the source code that implements a concern, i.e., *concern location*. His work also contributed to the development of a suite of metrics for quantifying crosscutting concerns. The work done by [15] explored the refactorings to improve different quality attributes of Fortran programs. The refactoring and transformations described in their work basically targeted the improvement of performance and maintainability of the system.

This paper presents the code smells specific to procedural software developed using C, explored from the aspect specific perspective. The code smells are inspired by the smells presented in [8].

The paper begins with an introduction to the criteria for evaluation of the code smells. Section III discusses the automatic analysis and reverse engineering of the system for the extraction of the code smells .Two case studies are used for the purpose of code analysis. The next section presents the code smells along with the transformations that would remove the code smells. .Finally we present the work related to this area and the last section presents the conclusion.

II. CRITERIA FOR EVALUATION OF CODE SMELLS

We present the criteria for evaluating the code smells in the procedural code. Code smell is any symptom that indicates that something is wrong with the code. It is considered generally an indicator that the code should be refactored, transformed or the overall design should be reexamined. The term appears to have been coined by Kent Beck, Smells are defined only in terms of general, subjective criteria that are dependant upon the software, the intuition of the developer, the type of software development methodology used etc, which makes them difficult for automatic identification. The criteria evaluation is for guiding the legacy maintainer in analyzing a legacy system and applying AOSD transformations. We have followed the level based refactoring for the purpose of organizing the refactorings in a sequential manner: The code smells are investigated on two levels: Program level and Function level. The code smells are characterized by the level-based transformations; we classify and organize the code smells and their transformation in a sequential approach.

Each level is investigated to primarily detect three kinds of smells

(1) To identify the crosscutting concerns in C.
(2) Minimize code duplication that may exist in different forms and could be removed through different solutions and can also be detected automatically.
(3) To identify the places in code that requires improvement and may be accomplished easily by AspectC that in general may not be possible with the normal C constructs.

III. AUTOMATED ANALYSIS OF CODE SMELLS

Analysis of a source code to identify bad smells can be a typically complicated task if performed manually. This necessitates the use of code analysis tool that speeds the process of comprehension for programs that are large and unfamiliar by automating the browsing and analysis of the code. Though tools are significant in speeding up the analysis process but the identification of bad smells is finally done manually as there aren't any standards set to classify the code as written in good or bad style. The quality of the code is very much dependant on the perception and the knowledge of the developers involved in analyzing the code and also the type of environment the project is designed for. Therefore, bad smells are identified by analyzing the projects manually as well as using automated tools.

The software is reverse engineered for a detailed analysis and the detection of badly written code. For the purpose of detecting code smells we analyzed a few case studies using reverse engineering tool. In this context, five projects have been reverse engineered and analysis is performed on a few more to exactly picture the bad smell in the code. These systems have been analyzed using an automated code analyzer Imagix 4D. Though, we don't claim that the tool would detect all sorts of code smells but to great extent it extracts different types of code duplication existing in the software and the discrepancies in the code that serves as an indicator to bad code symptoms. The tool is used basically to reverse engineer the software to study and analyze the potential problems in the development and maintenance of software at any level from the detailed program logic of an individual function to its high level architecture. The software metrics computed from the tool helps in determining the quality of code under consideration and also indicates the places in the software where a code smell may be detected.

For the purpose of elaborating the reverse engineering process we have shown a few snapshots from the case studies that we have analyzed for detecting the code smells.

IV. CASE STUDIES

In this paper we have depicted 2 case studies for the purpose of analysis (1) The Student Information System and (2) Mobile Store Information System. The Student Information System (SIS) is an information system for supporting the management of student record keeping. It is implemented in C and comprises about 5000 lines of code. The SIS performs all the functions related to record management such as addition, deletion, updation, searching and maintenance of student records. The records are maintained in the form of structures and are written and read from the files that maintain the records.

Mobile Store Information System (MSI) is a project that is designed for maintaining the customer records of the mobile store and comprises approx 9000 lines of code. It keeps the basic details of the customer such as name, phone number, address, connection type. The customer records can be added deleted and also manipulated using the software. The record can be searched using different keywords.

The projects are reverse engineered and the snapshots are shown in the figures below.

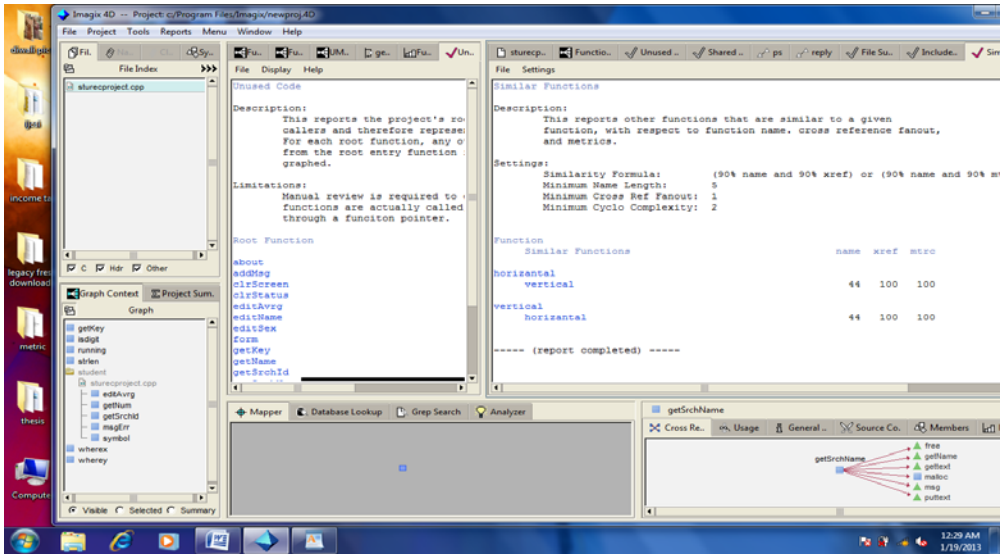

Figure 1. Highlighting similar functions in the code

The reverse engineering of the system reveals a number of flaws existing in the system coding. For example the snapshot in Fig 1 highlights the similar function in the code. Two functions performing similar task should be inspected for the relevance of their existence. There are a number of functions that are actually performing task similar to other functions in the project. These functions need to be identified, since if there is not much difference in the functions and the functionalities could be merged or could be achieved with a fewer number of functions then the complexity of the code can be reduced.

Similarly, the snapshot depicted in Fig 2 taken from SIS reports unclear sub expression that is in need of refactoring or transformation and represents a bad code symptom. The reverse engineering of these case studies have lead to the detection of discrepancies in the source code, thus this assists in searching for code smells. The code smells can be detected with the help of these results easily.

The dangling else if, missing compound statements, the similarity of 2 functions of the code, the frequent usage of memory allocator functions are traced easily using the software and this indicates that at these places the code may need modifications. These discrepancies found in the code help us easily target the functions and statements in the software that are in need of a change and represent a bad code symptom that helps in the identification of the code smells.

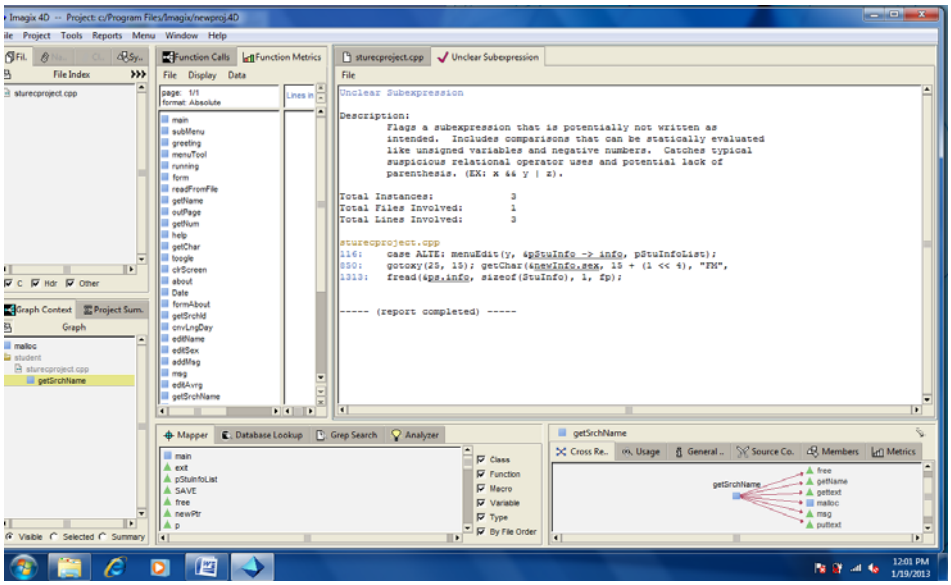

Figure 2. Depiction of unclear sub expression (SIS)

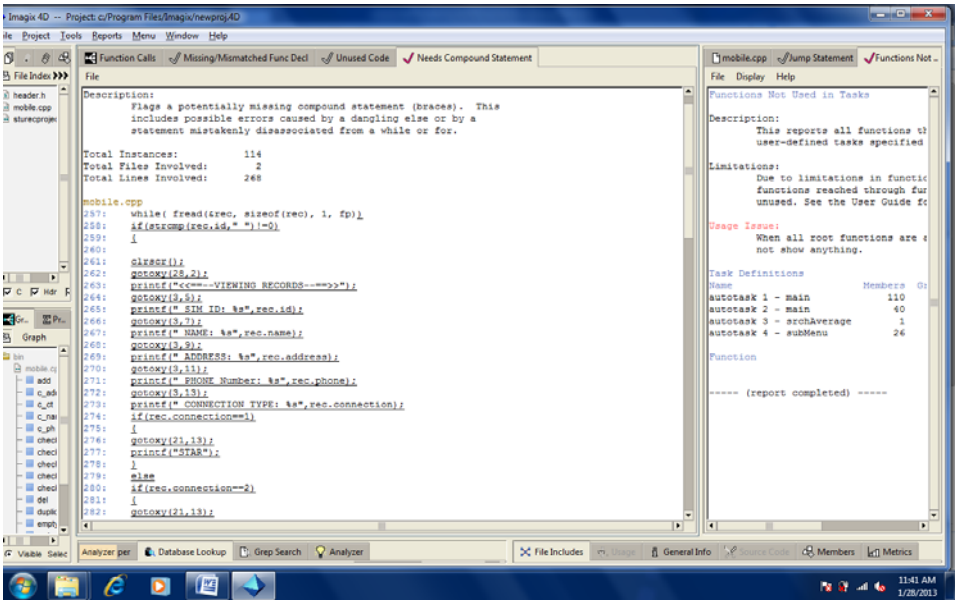

Figure 3. Dangling if else loops in Mobile Information System

V. CODE SMELLS

The detection of code smells is done in levels as it eases the job of simplifying the code and detecting the code smell. This section presents the code smells at different levels. The code smells are extracted at 2 levels (1) Functional Level and (2) Program Level.

A. Functional Level Code smells

The function-level code smells are detected first because a function is generally the smallest complete unit of any program or software that is assigned an independent task .Therefore, it is advised to determine the number of code smells in the lower stage of a program and then move on to the detection at the program level The reverse engineering of the software eases the task of code smell detection.

1. Long methods: Long methods have always been a bad code symptom. Therefore whenever a long method is encountered efforts are required to decompose it into smaller manageable pieces of code [8].The well known solution is to use Extract Method by extracting a chunk into its own method. A function to be considered too long depends on many different factors. For example, it may not have too many lines of code but might be handling different tasks or concepts, such functions could be considered long. It also differs from the perception of one programmer to the other. A long method is generally reduced by "extract method" refactoring, however in certain situations even this technique may not work. For instance, a function may involve a number of local variables performing different task in the function, method extraction may not be possible, in this case the local variables could be defined as fields of aspect and the their task may be decomposed to a function called in the advice. The aspects in aspect oriented programming serves as a powerful tool if the behavior of the code is to be altered in some way. For example if the feature is scattered across several functions: If there is a segment of code that is scattered throughout the program at different places causing code duplication then such fragments if possible should be extracted into an advice that could be captured with the same pointcut rather than multiple function calls. This would result in code reduction and uniform behavior is inserted with the help of an advice without bothering to place a function call everytime the code is moved to an aspect.

2. Very small functions: As are long methods a major reason to refactor so can a very small function be termed as a bad code symptom specifically if they are called numerous times in the program. These functions should be converted into an advice if appropriate joinpoints are available.

3. A function is almost non existent due to insignificant task assigned to it: The code symptom is related to the above described symptom with a slight difference. The case describes a function that handles very less task though it is supposed to hold a greater responsibility. The extra task assigned to the function can be added using an advice.

4. Function not required in the current context: There are certain functions that may not be required in all kind of scenarios or in the current context but may be required sometimes in future. Thus the call to such functions can be skipped in that case. Skipping call to a function is not possible through general C constructs but if an around advice is used then a function call can be skipped.

For example, the functions that are required for the purpose of debugging are required only during that time to trace the execution of the program but does not actually contribute to the working of the system. Thus, such functions are required at that time only but may be required in future also thus deleting them after that may not serve the purpose ,therefore the calls to such functions should be skipped when not required and then when required the call should be revoked. This is not possible with the traditional C constructs but could be achieved easily using around advice in AspectC.

5. The value of function parameters change frequently: The function argument needs to be modified differently at several places. There may be a scenario where a function is called numerous times (around 50-100) in a project, then in that case if the argument values passed to a function has to be modified differently at different calling sites, then manipulating at so many different places manually is a tedious job. But this could be achieved easily using *args() and around()* advice .The modification of the arguments in the function is achieved by defining appropriate join points.

6. Part of a function is related to a concern that is to be moved to an advice and joinpoints need to be combined: If a join point is formed by combining many joinpoints (for example by using && or || operator) and is used with multiple advice then extract it into a pointcut or introduce a named pointcut that describes the join point.

Program level Code Smells

Since the application size is fixed, therefore after the detection of the code smells at the functional level it becomes easier to determine and rectify the code smells existing in the whole program. A few code smells existing on the program level are described as follows:

7. Duplicated code: If code duplication exists at program level i.e across different functions then they are suitable candidates for crosscutting concerns and so can be captured using a proper advice by defining a suitable pointcut.

8. Crosscutting concerns with no proper join points: This code smell results in continuation with the above described symptom. At times the crosscutting concerns do exist but they exist in different code scenarios where they cannot be captured using similar joinpoints.In that case a joinpoint needs to be created at all those points so that they could be captured with very few joinpoints, hence not necessitating the need of extracting the code into a function. This could be achieved using a set pointcut.

9. Complex conditional code: Complex conditional code does hamper the readability of the code. As has been described in [8] that similar conditional statements are duplicated a number of times in different functions. Therefore extracting the conditional statements into an advice would reduce the code complexity considerably and also the code duplication. But this may not be possible in all sorts of conditional statements. It is necessary to ensure that the conditional statements are governed by similar statements (for example a particular integer variable or a string).Thus if the conditional statements are not in the proper format the code has to be refactored in order to bring them into proper shape and thereafter the conditional statements can be extracted to an advice using the set pointcut.

10. When a single updation causes changes in several functions: As pointed by Beck and Fowler this symptom appears when a particular change made to the system may cause changes at many different places in the system. When the changes are scattered at different locations, they are hard to find and it is easy to miss an important change. The fragments and members related to such concerns can be refactored using The *Extract Feature into Advice* refactoring .But the refactoring should only be performed if it simplifies the code not if the code becomes more complex and hard to understand.

11. Presence of cross cutting idiomatic exception handling code: Systems developed with procedural languages typically resort to popular return code idioms for implementing exception handling, as advocated by the usage of the well-known *return code* technique in many C programs and operating systems. The study by [2] explores that such idioms are not scalable and compromise correctness [3].The crosscutting concerns in procedural languages are implemented explicitly using more primitive means, such as naming conventions and coding idioms [3] when aspects are not involved. Such techniques are preferred as they do not require special-purpose tools or languages, are easy to use, and allow developers to recognize the concerns in the code readily. But there are certain drawbacks of such techniques as they are prone to errors and they make concern code evolution time consuming and often lead to code duplication and increases the debugging complexity. The idioms-based approach has been turned into a full-fledged aspect-oriented approach [3] and the results show that the adoption of aspect oriented approach can lead to significant improvements in source code quality. But the try () and catch () pointcuts provided by AspectC can be used to explicitly handle the errors as is done in object oriented languages.

12. Structure needs to be modified: If a structure is to be modified with new data members then the modifications done to the structure can be achieved without modifying the original structure in the presence of aspects. The modification of a structure causes subsequent modifications at different places in the program has to be achieved where the new members of the structure are to be used. The introduction of additional members and then adding their respective functionalities is a complex task and once merged with the code it's not easy to trace the new additions in the system. Their are situations when a member might be introduced for certain specific task and may not be required all the time, in that case it would be better to introduce the member separately through an intype pointcut using AOP approach. But addition of new data members should be done using intype() pointcut, only if the functionalities provided by the new data members could be captured through proper joinpoints else it would introduce unnecessary complexity. The introduction of intype() pointcut is required when manipulation is to be done in a single structure or multiple structures.

TABLE I. THE CODE SMELLS AND THE TRANSFORMATIONS

Symptoms	**Refactorings**
Duplicated code	Extract Fragment into advice
Long methods	Extract method into advice
Feature scattered across several functions	Extract Feature into aspect
Cross cutting concerns with no proper joinpoints	Introduce a set pointcut
Data Clumps (Similar data items are defined at many places)	Extract data members into a structure, Introduce parameter structure object
Very small functions	Add functionality with advice
When a single updation causes changes in several methods (Shotgun surgery)	Move methods to advice, create around advice

Function not required in the current context	Skip calls to function using around advice
Part of a function is related to a concern that is to be moved to an advice and joinpoints need to be combined.	Create a named pointcut, combine pointcut (depending upon the situation)
The value of function parameters change frequently	Modify the function arguments with around advice ,args and proceed()
A function performs very less task	Assign task to Lazy function
Values to a specific function call is changed frequently in the control flow of another function	Introduce dynamic crosscutting using cflow() pointcut
Gotos elimination	Eliminate with proper advice using a set pointcut.
Presence of cross cutting idiomatic exception handling code	Extract the code using try() and catch() pointcuts
Complex conditional code	Extract conditional to advice

We have discussed the code smells specific to C in the light of AspectC. The code smells detected have been handled using the AspectC.

B. RELATED WORK

Aspect-oriented software development (AOSD) [12] aims at improving the modularity of software systems, by capturing the scattered crosscutting *concerns* in a well-modularized way. In aspect-oriented programming languages this is achieved by adding an extra abstraction mechanism, called an *aspect*, on top of existing modularization mechanisms such as functions, classes and methods. One of the earliest works was by Coady *et. al.* [5] in the area of applying aspect orientation to a C based system software . AOP was used to refactor *prefetching code* in the FreeBSD OS kernel. The usage of AspectC depicted several advantages such as independent development of the prefetching modes and overall improved comprehensibility. But the work does not focus on a general approach for isolating crosscutting concerns.

Other researchers had investigated the utility of applying AOP to various crosscutting concerns. [10] dealt with exception detection and handling code in a large Java framework. Both works discuss advantages of using AOSD, such as reduced code duplication and improved cohesion, and discuss some particular limitations of using AspectJ. One of the earliest studies was conducted by [14] for preparing the code for isolating concerns and performing the necessary restructurings and concluded that the aspect solution does reduce the code size.

Bruntink et al. present their experiences of [1] [2] [4] solving crosscutting concerns in embedded C code to using aspect oriented programming. [2] developed a domain-specific language (DSL) for parameter checking. This lead to the development of other aspect oriented languages [6] that worked on the similar line of bringing aspect oriented software development to the C programming language.

The code smells have been investigated by other researchers such as [8] [11] [16] in which they focused on code smells related to object oriented software. The object oriented code smells, C transformations and refactorings have also been investigated by [9]. In her PhD thesis she had developed a refactoring tool for C programs that allow refactoring on Cpp directives, though most of the refactoring tools do not support the preprocessor directives. But the work focuses on extending the refactorings to preprocessor directives and focuses on refactorings like “Extract File”, “Rename Macro”, “Inline File” etc.

C. CONCLUSIONS

In this paper we highlighted a few code smells in systems implemented using C. The code smells are specifically related to the crosscutting concerns in the system and their maintenance using aspect oriented approach. The kinds of code smells that are meaningful for C is very different than those existing in object oriented systems. Most of the literature about code smells and the corresponding transformations and refactoring is concentrated in object-oriented languages, and transformations in the inheritance hierarchy. Various attempts have been made by different researchers in the area of C transformations and code smells. A number of other researchers have also worked on systems developed with imperative style of programming but only in parts. The kinds of refactorings described by Garrido are very different too, as none of the refactorings focus on crosscutting concern. Existing legacy procedural software maintenance approaches do not explicitly consider maintenance of crosscutting concerns, that increases the complexity of a system as a result the addition or updation of crosscutting concerns causes degradation in the structure of the system, reduces the quality factors such as maintainability and understandability of the system.

We have presented several bad code symptoms that could be dealt with aspect orientation with a much lesser effort than is required by the traditional c constructs. In order to assist and ease up the process of code smell detection we have done automatic analysis of the system by reverse engineering the software that helps in locating the sources where a badly written code may be found. Therefore the code smells described in this paper highlights the scenario that might be difficult to maintain with the traditional C constructs and so aspect oriented constructs (AspectC) serves as a tool to handle the code smells and the poorly written code with greater ease and efficiency that may not be possible otherwise. Regardless of the actual programming model these new code smells will be of great help for the maintainers in the extraction of aspects and their implementation in improving the code understandability and efficiency.

REFERENCES

[1] M. Bruntink, A. van Deursen, and T. Tourw´e. “An initial experiment in reverse engineering aspects from existing applications”, Proceedings of the Working Conference on Reverse Engineering (WCRE), *pages 306–307*. IEEE Computer Society, 2004.

[2] M. Bruntink, A. van Deursen, and T. Tourw´e. “Isolating Idiomatic Crosscutting Concerns”, Proceedings of the 21ST International Conference on Software Maintenance (ICSM). IEEE Computer Society, 2005.

[3] M. Bruntink, A. van Deursen, and T. Tourw´e. “Discovering Faults in Idiom based Exception Handling”, Proceedings of the 28th international conference on Software engineering Pages 242-251, 2006.

[4] M. Bruntink, A. van Deursen, R. van Engelen, and T. Tourw´e. “An Evaluation of Clone Detection Techniques for Identifying Crosscutting Concerns”, Proceedings of the International Conference on Software Maintenance (ICSM), pages 200–209. IEEE Computer Society, 2004.

[5] Y. Coady, G. Kiczales, M. Feeley, and G. Smolyn “Using AspectC to Improve the Modularity of Path-Specific Customization in Operating System Code”, Proceedings of the Joint European Software Engineering Conference (ESEC)and 9th ACM SIGSOFT International Symposium on the Foundations of Software Engineering (FSE-9), pages 88–98. ACM Press, 2001.

[6] B. Adams and T. Tourw´e. “Aspect-Orientation in C: Express Yourself”, Proceedings of the AOSD Workshop on Software-engineering Properties of Languages for Aspect Technologies (SPLAT). Aarhus University, March 2005.

[7] Marc Eaddy An Empirical Assessment of the Crosscutting Concern Problem PhD thesis.Columbia University.2008.

[8] M. Fowler, K. Beck, J. Brant, W. Opdyke, and D. Roberts. Refactoring: Improving the Design of Existing Code. Addison-Wesley, 1st edition, 1999.

[9] Garrido, A.: Program Refactoring in the Presence of Preprocessor Directives. PhD thesis, University of Illinois at Urbana-Champaign. 2005.

[10] M. Lippert and C. V. Lopes. “A study on exception detecton and handling using aspect-oriented programming”, Proceedings of the 22th international conference on Software engineering (ICSE), pages 418 – 427. IEEE Computer Society, 2000.

[11] Kerievsky J. Refactoring to Patterns, Addison-Wesley, 2004.

[12] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda,C. Lopes, J.-M. Loingtier, and J. Irwin. “Aspect-oriented programming”, In Proceedings of the Europeen Conference on Object-Oriented Programming (ECOOP), volume 1241 of Lecture Notes in Computer Science, pages 220–242. Springer-Verlag, 1997.

[13] Monteiro, M.P. Refactorings to Evolve Object - Oriented Systems with Aspect-Oriented Concepts. PhD thesis, Universidade do Minho, Portugal 2005.

[14] G. C. Murphy, A. Lai, R. J. Walker, and M. P. Robillard. “Separating Features in Source Code: Exploratory Study”, Proceedings of the International Conference on Software Engineering (ICSE), pages 275–284. IEEE Computer Society, 2001.

[15] Méndez,M. Fortran Refactoring for Legacy Systems, MSc Thesis, Computer Science School, Universidad Nacional de La Plata. Available at http://hpclinalg.webs.com/FRLS.pdf 2011.

[16] Wake W, Refactoring Workbook, Addison Wesley, 2004.

Dr. S.A.M Rizvi is an Associate Professor and former Head, Department of Computer Science, Jamia Millia Islamia (Central University).His area of specialization is Software Engineering and MIS. He is a senior life member of Computer Society of India (CSI), IEEE, ISCA, and IEA. He has authored 6 books and a number of articles in refreed journals and conference proceedings. Some of his recent publications include articles in Ubiquitous computing and Communication Journal, Journal of Opt. Communication, ACM, IROCS, IJSE, IJCA, IEEE Xplore.

Ms. Zeba Khanam is a doctoral candidate in Jamia Millia Islamia (Central University), New Delhi. She is also a faculty member at the department of Computer Science JSS Academy, Noida and is also working on software development projects. Her research interests are Software Re engineering and Reverse Engineering and are teaching and also working in projects of .NET framework. Her recent publications include articles in IEEE Xplore, International Journal of Computer Applications, LNCS proceedings, JSCSE.

