# Enhancing Cohesion Metrics in Refactoring with Code Smells

**J. Kaviarasi*, P.R. Jayanthi**

*Abstract* – **The increasing importance being placed on software measurements has led to an increased amount of research in developing new software measures. The main objective is to increase the value of cohesion in effective manner when compared to the refactoring plans using Featureous tool. The problem with the existing method is that the metrics were not given higher priority and so we aim in increasing the metrics efficiency for cohesion and coupling. The automated process of refactoring using Featureous tool which runs on net beans platform gives ambiguous results. To avoid these kinds of result and in order to increase the cohesion metrics for the purpose of reusability, maintenance and ease of access to the existing software code, a new approach is incorporated along with the existing works.**

*Index Terms* – **Cohesion, Refactoring, Ambiguity, Coupling.**

## I. BACKGROUND AND MOTIVATION

As computing resources continue to grow, efficiency falls further behind another concern when writing code, though. That concern is the cleanness of the code itself. Mostly, this boils down to readability and comprehensibility. Without readability and comprehensibility, we cannot reuse our code. In this paper, we seek to provide an efficient approach to the development of software computational platform for the currently very active research domain of enhancing cohesion metrics in software code. The purpose of this paper is to bring together a community of software engineering practitioners and researchers who have an interest in developing an effective software code. The idea of the project is concerned with:

- Get sample code and inject smells into the code.
- Use automated tolls to check the cohesion metrics in the code.
- Implementing Swarm algorithm.

By software engineering tool that provides some automated support for the software engineering process. Within each level of support, we can find differing breadths of support

- Individual tools that support one particular task.
- Workbenches or toolsets that support a number of related task.
- Environments that support that whole or at least a large part of the development process.

## II. INTRODUCTION

### A. Cohesion

Cohesion is a degree to which a function performs a single well defined task and dependencies prevails within the class and coupling is the degree of interaction between modules. We say that an entity is cohesive if it performs a single, well defined task and everything about it is essential to the performance of the task. An important goal in design is to try to ensure that each entity we design (class, method, system) exhibits the highest possible level of cohesion. The issue in low cohesion leas to complex interface, changes influence in other modules often, cannot be reused and less comprehensibility.

### B. Featureous tool

The goal of feature-centric analysis is to help identifying and understanding the implicit boundaries, overlaps and dependencies among feature implementation. In order to be feasible for large code bases, feature-centric analysis requires an efficient method of establishing feature-code traceability links.

A programmer has to annotate it's entry points, the method through which a program's control flow enters implementation of a feature. As these methods are the only ones that need to be annotated, our approach is feasible for large and unfamiliar code bases [3]. After annotating feature-entry points, Featureous instruments a program with a tracing aspect at load time. The aspect identifies methods, classes and objects used by individual feature at run-time. The captured traces are shown in the feature explorer window of Featureous as depicted in fig 1.



ISSN: 2278 – 1323 

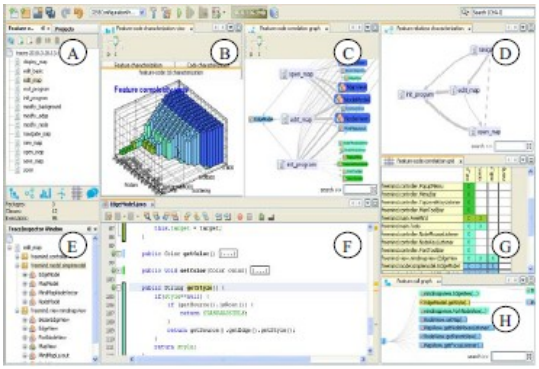


Fig 1 Overview of the user interface of Featureous

### C. Analytical views

Featureous provides a number of analytical views for investigating the captured traceability links. The feature and computational-unit characterization view (B) uses three-dimensional bar chart to visualize the scattering (X axis) of individual features (Z axis) and tangling of their computational units (Y axis). Thereby it summarizes the complexity of feature-code correspondences. The coloring scheme assigns green to computational units used exclusively by a single feature, dark blue to core units used by most of the features and two shades in-between to any intermediate units. The feature-code correlation graph view (C) and feature-code correlation grid view (G) correlate features and computational units allows for investigating how concrete structural units of a legacy program implement features. In figure 2, we present an example usage of the feature-class correlation graph, where we compare how the four decompositions of the KWIC application implement features. The four decompositions of KWIC: (I) implicit invocation, (J) shared data, (K) abstract data type, (L) pipe and filter.

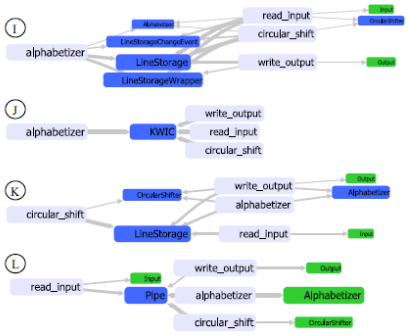


Fig 2 Feature-class correlation

The four features of KWIC (“alphabetizer”, ”circular_shift ”, “read_input”, “write_output”) in figure 2 are completely tangled with each other in the shared data design (J). In contrast, the pipe and filter design (L) grants the most independence to feature implementations, having only the pipe class shared among features. This decomposition also exhibits lowest average values of class-granularity scattering and tangling metrics, i.e. 0.10 and 0.08 respectively. The next view of Featureous called feature relations characterization (D) visualizes dynamic relations (based on object instantiation and co-usage) among features.

This allows an analyst to identify run-time dependencies between features. Such dependencies are an important factor to consider when investigating deployment –time functional customizability of an application and identifying excessive dependencies not reflected in program’s problem domain.

Finally, the feature-code traceability (E), feature call graph (H) and code-feature traceability (F) views provide the grained navigable links between features and source code. Herein, it is possible to list and navigate units contributing to a feature’s call graph, as well as features contributing to concrete fragments of source code being edited.

## III. REFACTORING AND FEATUREOUS

Program refactoring is transforming a program to improve readability, structure, performance, abstraction, maintainability which is not applied in practise as much as desired [2]. One deterrent is the cost of detecting candidates for refactoring and of choosing the appropriate refactoring transformation. Featureous tool demonstrates feasibility of automatically finding places in the program that are candidates for specific refactoring. Featureous enhances the Netbeans IDE for Java with feature-oriented perspective on source code.

Featureous makes it possible to easily establish traceability links between user-visible features and individual units of source code like packages, classes and methods. This is done by tracing execution of program’s code, as a user, or a test suite, is triggering it’s features.

Fig 3 Tracing features



ISSN: 2278 – 1323 

```
<aspectj>

    <aspects>
        <concrete-aspect name="dk.sdu.mmmi.aspects.AnagramGame5" extends="dk.sdu
            <pointcut name="packages" expression="within(com.toy.anagrams..*)"/>
        </concrete-aspect>
    </aspects>

    <weaver>
        <exclude within="! com.toy.anagrams..*"/>
        <include within="com.toy.anagrams..*"/>
    </weaver>

</aspectj>
```

Fig 4 Select and trace the project

In order to start annotating features we have to configure the sources of project to be of format jdk 5, in the Java class. Annotate the @FeatureEntryPoint("gui"), and the nextTrialActionPerformed method which will modularize the "generate_word" feature. We needed an additional feature "gui" for the infrastructural classes and classes unrelated to "generate_word". Featureous by executing from Netbeans automatically opens the latest traces which contain a mapping between the features that have annotated and the packages, classes and methods that get executed as a result of entering their corresponding feature-entry point method.

```
@FeatureEntryPoint("gui") private void initComponents() {
    java.awt.GridBagConstraints gridBagConstraints;

    mainPanel = new javax.swing.JPanel();
    scrambledLabel = new javax.swing.JLabel();
    scrambledWord = new javax.swing.JTextField();
    guessLabel = new javax.swing.JLabel();
    guessedWord = new javax.swing.JTextField();
```

Fig 5 Annotate the method

The structural view of the source code of our project shows packages, the classes and the runtime call relation between classes. Each package, class method is colored using the coloring scheme of Featureous that tells about the number of features it is used by. Although the actual algorithm for assigning colors is more complicated, the scheme can be intuitively understood as assigning green to units used by a single feature, whereas dark blue means "used by meny features". As for using this information to create a new program structure, single-feature classes are good candidates for moving to a dedicated feature-modules, which classes used by many features are good candidates including in reusable core/infrastructural modules. Furthermore, the diagram allows for (un)collapsing packages to hide or show contained classes and (un)collapsing classes to show the methods that were used by features at runtime.

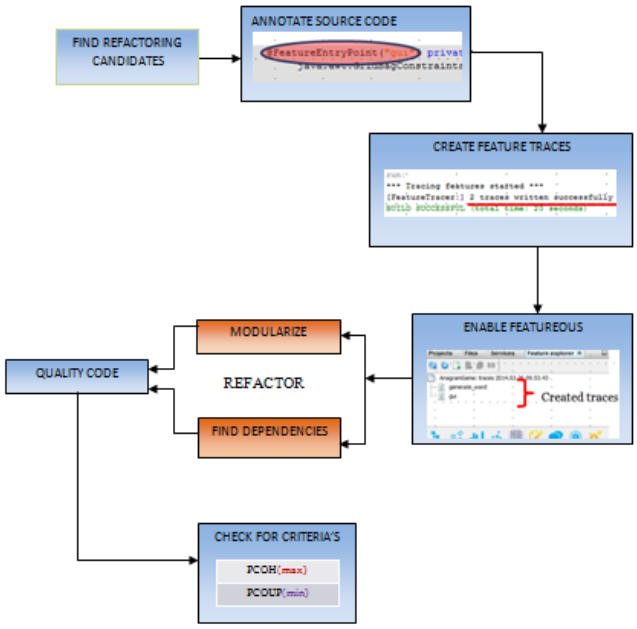


Fig 6 Refactoring by modularization

## IV. PROBLEM DOMAIN

Technical systems are getting larger and more complex where global control is hard to define and program, and larger systems lead to more errors. The typical problem to be solved is high coupling and low cohesion by refactoring.

In automatic re-modularization, the refactoring candidate will be annotated by the tool support which gives quality code as an output but not an efficient one [1]. Because of inefficient moving of class and results, even the manual check with dependencies is not suitable for code efficiency.

## V. SWARM ALGORITHM

Through a solely decentralized and reactive system, where "the whole is greater than the sum of the parts", swarms exhibits behaviours that coordinate on a global scale. Deriving inspiration from nature, swarm theory leverages the self-organization that emerges from the interactions of multiple agents to evoke a group-level behaviour that is beyond the capability of any single member of the swarm [7]. Swarm algorithms are most useful for problems that are amenable to an agent-based decomposition, have a dynamic nature, and do not require time-limited or optimal solutions. Swarm behaviour can be simulated by a type of multi-agent algorithm. The most popular swarm algorithm is the algorithm of boids developed by Craig Reynolds in 1986.


ISSN: 2278 – 1323 

The boids algorithm consists of three simple rules to interact with other boid agents:

- Cohesion : going to the centre of the surrounding agents.
- Separation : going away from other agents.
- Alignment : heading towards the same direction of other agents.

Each of those three rules has different distance range. Typically cohesion has the largest range, alignment has the second largest and the separation has the smallest, but depending on the intended behaviours this order can be changed.

A swarm algorithm is a balance of ability and numbers. For example, assume an arbitrary task to be completed. On one end of the spectrum, a single complex agent capable of performing the task must possess all the intelligence and skill required to complete the task. This enables the agent to quickly and efficiently address to the task, but at a cost of robustness and flexibility. If any of the required components of the agent fails, the agent cannot complete the task. Additionally, when the ability of the agent needs to be expanded, the complexity of the agent will be increased, thus possibly causing unforeseen issues and reducing the maintainability of the agent. Finally, if the task is dynamic, a single agent may have a difficult time of keeping up with the changing environment.

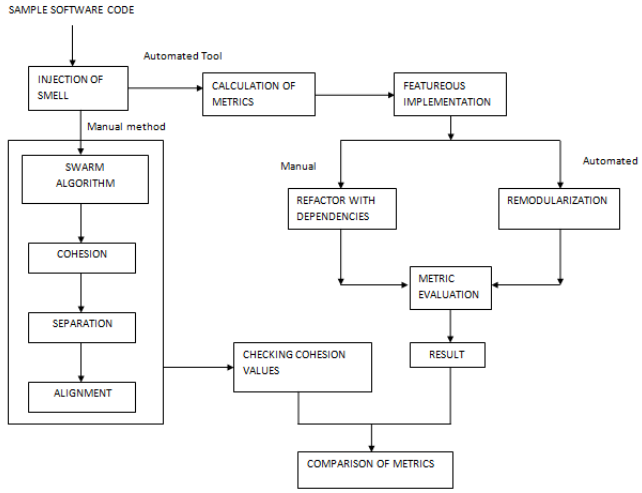


Fig 7 Architecture diagram

At the other end of the spectrum, for the same task, assume that we are given a non-homogeneous swarm of agents who collectively have the ability to accomplish the task. The swarm contains agents with redundant abilities, thus some duplicate work will be performed but with the benefit that a malfunctioning or damaged agent does not preclude the accomplishment of the task. The complexity of each swarm agent is much less than that of the single centralized agent, thus maintenance and replication is much easier. Finally, though swarms may not always optimally perform small or static tasks, as the size of dynamicism of a problem grow, a swarm can scale with the problem with the addition of new agents.

## VI. CONCLUSION

We have encountered problem with an existing work in which the cohesion and coupling methods are not discussed in depth. Also, the metrics were not given higher priority. It is suggested that often manual refactoring will be the most effective one among all the others. Since it increases the time factor, different strategy is used in our work to increase the metrics efficiency by increasing the cohesion level. It can be evaluated by comparing the cohesion values implemented by swarm algorithm.

## REFERENCES

[1] T.Pandiyavathi, Manochandar, "Smell Detection Sequencing And Usage Of Optimal Refactoring Plans", Vol 3, Issue 5, May 2014.

[2] Martin Fowler, Kent Beck (Contributor), John Brant (Contributor), William Opdyke, Don Roberts (2002), "Refactoring: Improving the Design of Existing Code".

[3] Yoshio Katako, Michael D.Ernst, William G.Griswold, David Notkin, "Automated Support for Program Refactoring using Invariants", University of Washington.

[4] 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, Jan-Feb 2012.

[5] Core Java Vol 1, Fundamentals, Eighth edition by Cay S.Horstmann and Gary Cornell.

[6] Tools-http://www//SourceForgenet.com-pmd, checkstyle.

[7] Andrea Gasparri, "Swarm Aggregation Algorithm for Multi-Robot Systems", June 2013.

## BIOGRAPHIES

**J.KAVIARASI** was born in Viluppuram, Tamilnadu, India in 1990. She completed her Schooling by 2008 and Bachelor Of Degree in Tagore Engineering College in 2012. She is pursuing Masters in Computer Science and Engineering in Mailam Engineering College, Mailam(A constituent college of Anna University Chennai).

**P.R.Jayanthi** was born in Tamilnadu, India in 1983. She received Masters in Mailam Engineering College(A constituent College of Anna University, Chennai). She is an Associate Professor Of Computer Science and Engineering in Mailam Engineering College(A constituent college of Anna University Chennai).



ISSN: 2278 – 1323 