Evolution of Legacy System Comprehensibility through Automated Refactoring

Isaac Griffith, Scott Wahl, Clemente Izurieta
Computer Science Department
357 EPS Building
Montana State University
Bozeman, MT 59717
{isaac.griffith, scott.wahl}@msu.montana.edu
clemente.izurieta@cs.montana.edu

ABSTRACT

Software engineering is a continually evolving discipline, wherein researchers and members of industry are working towards defining and refining what are known as "best practices." Best practices are the set of known correct engineering techniques that lead to quality software.

When a software artifact is produced, it becomes temporally locked into a single instantiation of these best practices at a given point in time. If such software is not maintained in such a way as to keep it current with the evolution of practice, then there is a good chance that subsequent engineers may not understand the design choices made. There are known techniques, called refactorings, which allow for structural changes to software without altering the outward appearance and behavior, thus maintaining the intent of the original design. Unfortunately, refactoring requires an engineer to both understand the techniques to be applied and the code to which they are applied to. This is not always feasible.

We have developed an automated system utilizing Evolutionary Algorithms to manipulate refactorings correctly without requiring an underlying understanding of the software. This then allows for sustained levels of quality of evolving software systems. The understandability, maintainability, and reusability of the software regenerate as best practices evolve.

Categories and Subject Descriptors

D.2.2 [**Software Engineering**]: Design Tools and Techniques – *computer-aided software engineering (CASE)*.

D.2.7 [**Software Engineering**]: Distribution, Maintenance, and Enhancement – *restructuring, reverse engineering, and reengineering.*

I.2.2 [**Artificial Intelligence**]: Automatic Programming – *program modification*

General Terms

Measurement, Design, Experimentation

Keywords

refactoring; automation; software engineering; code smell; software evolution; measurement

1. INTRODUCTION

Many legacy software artifacts, software systems developed prior to the adoption of UML, can benefit from refactoring. Yet, problems are encountered when current developers are required to maintain and update legacy systems. This usually occurs as a result of original developers no longer being available for maintenance or to field reference questions about the system. A disconnect between current and original engineers develops from the constant imperative to improve software engineering practices [22] and is reflected throughout versions of the evolving software design. Given such disconnects, the original intention and design choices made in legacy software systems may be unrecoverable (in a practical amount of time) by current engineers.

Software maintenance by developers lacking the understanding of the best practices under which the software was developed, or being confronted with a system which exhibits low understandability and maintainability due to poor quality, leads to a large amount of time wasted trying to understand how to integrate and implement new features. We currently have techniques which can be utilized to modernize the code to bring the system up to current best practices —these techniques are refactorings; which are designed to improve the quality of the software [12]. The quality of the software we are concerned with is directly related to the understandability and maintainability of the system. Thus, we have designed a system which provides automated refactoring of a system and focuses on removing code smells while simultaneously improving code metrics known to be linked to the qualities of concern.

The majority of time spent on a project occurs during the maintenance portion of the life cycle [23]. The system proposed herein aims to significantly reduce this portion of the development life cycle, which would lead to substantial time and financial savings. Analysis and enhancement of the architecture/design of a software system is achieved through the systematic application of structural refactorings, in order to enhance and illuminate the intension of the underlying design. This process is first initiated through the extraction of the design from the code level into the refactoring space of the system we have developed.

This paper is organized as follows: A summary and comparison of related work can be found in Section II. Underlying theory and concepts are described in Section III. System design, component subsystems, and data flow diagrams are provided in Section IV. Our methodology is explained in section V. Results of each experimental group and discussion of results are provided in Sections VI and VII, respectively. In Section VIII we describe threats to the validity of this work and the potential solutions to threats. Section IX provides concluding remarks and a discussion of ensuing future work.

2. RELATED WORK

Refactoring has presented itself as a means by which developers can evolve a software system in order to improve overall quality. Refactoring is an intense and time-consuming process which requires that developers understand both how and when to use the techniques. It also requires that they understand the underlying code. This can be seen in the widely available semi-automatic refactoring tools which now adorn most IDE's (e.g., Eclipse IDE, Netbeans, IntelliJ, etc.) [3].

Automated refactoring generally falls into the category of Search Based Software Engineering (SBSE) [19]. The sub-problem of refactoring in SBSE helps address disharmonies found in many applications. These disharmonies, found in legacy systems in the form of module clusters, are addressed by Kastener et al. by refactoring them into feature modules [15]. Perez and Crespo [22] use of graph transformations as a means of describing refactorings was presented as an approach to perform code comparisons when refactoring has been used. O'Keefe and Ó Cinnéide [19], [20] conducted an empirical survey detailing the utilization of search based optimization algorithms to refactor a software system while optimizing a set of metrics (used to measure the improvement of quality in the system). A detailed description of refactoring and search based optimization can be found in their survey [19].

Other approaches involving genetic algorithms and software quality assessment have been recently introduced. First is the work of Shimomura, Ikeda, and Takahashi [26] which focused on the recommendation of refactorings by measuring the overall quality of a codebase using a genetic algorithm. Another approach attempting to remove known duplicate code code smells was the work of Zibran and Roy [27] whose main goal was to solve the constrain problem associated with scheduling the removal of this code smell by applying a constraint programming approach. Another approach was the use of automated refactoring to enhance the ability of a Genetic Programming algorithm to generate code that solve much larger problems, as proposed by Otero, Johnson, Freitas, and Thompson [21].

Along with refactoring, Fowler [12] describes a set of code smells to indicate when refactoring is needed. A large amount of work has also been conducted by both Munro [18] and Roperia [24] in the area of automated code smell detection. Munro proposed a means to quantitatively assess whether source code exhibited traces of known code smells using standard object oriented metrics, such as depth of inheritance tree (DIT), Lines of Code (LOC), etc. [24].

The novelty of our approach is the combination of the code smell detection algorithms and the search based approach for refactoring. Up to this point this approach, to the best of the authors', knowledge has not been tried. This approach is meant to address the need to fully understand the code that needs to be refactored. The context in which this becomes a problem is as follows: In the development of legacy systems, developer turnover is inevitable over the lifetime of the software, yet new features and code maintenance must still be performed. Coupled to this, are the ever changing best practices in software engineering. Keeping up with best practices maintains a team's understanding of the source code because, in general, the modularity of the software is maintained through refactorings. The overall quality of the source code in a production environment is therefore maintained. In our research, this is accomplished by utilizing code smell detection algorithms to first detect code smells, and then produce a set of viable refactorings which eliminate the code smells. The techniques we use reduce the search space of the source code to areas which are in need of refactoring (code smells). This effectively reduces the scope of the search space to a limited set of known refactorings. The problem then becomes finding the optimal order in which to apply the refactorings.

3. BACKGROUND

This research is based on three major concepts: the relationship between code smells and refactoring, the theory and application of genetic algorithms, and the application of software engineering product metrics.

3.1 Code Smells and Refactoring

Refactoring is the incremental redesign of a software artifact. After a process of applying various refactorings in sequence, the refactored code will better conform to agreed upon solutions of software engineering best practices [12] whilst maintaining identical functionality. For example, let's say that an inheritance hierarchy exists within the codebase and that the top-level class defines some abstract method *operation().* After inspection of the code we find that several of the classes (or all of them) implement this method in the same way. What could be done here is that the implementation could be moved up into the parent class and overridden in the classes that implement the method a different way. This is an example of the Pull Up Method refactoring. All this has done is to remove redundancy in the code while simultaneously exploiting the principles of OOP to better the structure of the underlying design.

Alterations specified by a refactoring are designed in such a way that if applied correctly they will not alter the observable effects of the program. Thus, given a unit test designed for the original legacy code, refactored code should still pass the unit test.

Code smells are described by Fowler and Beck [12] as a set of qualitative notions that help indicate when a refactoring is necessary or when to stop refactoring. Code Smells were intentionally designed to not be a heuristic or metric of any sort. Instead, they are meant as a means of inspiration to be coupled with human intuition [12] and are apt for subjective interpretation. Regardless, code smells provide an indication of where and when refactoring is most needed [12], and since algorithms [18] [24] have been designed to detect code smells, we have decided to utilize these to reduce the refactoring search space.

3.2 Genetic Algorithms

Genetic Algorithms (GAs) are an iterative approach which is described as analogous to evolutionary processes for solving search problems [1]. The GA generates a population of potential answers to a problem and measure the fitness (survival ability) of each solution to solve the problem. In order to apply a GA approach, the solution space must be decomposed into smaller sub-problems which can subsequently be combined into an overall solution. We restrict the search to the best solutions found so far and combine them to create a population with higher fitness [25]. During the search, intermediate solutions are represented as strings of binary values or *alleles*, where each value represents the presence or absence of a trait. Through operations like crossover and mutation [1], we can progressively generate strings (new intermediate solutions) with higher fitness levels. Intermediate solutions are re-integrated into the existing population until some pre-determined stopping criteria (i.e., number of iterations or fitness level) is met. The basic algorithm pseudo code is described in Figure 1.

The *initialPopulation()* function takes a maximum size parameter used to generate a population of random individuals with varying levels of fitness. The *evalFitness()* function takes the population of individuals and evaluates their ability to solve the problem using a predefined fitness function. During iteration, the *select()* function is used to select the individuals, meeting some criteria, from the population in order to generate the next generation. The selected individuals are then paired off for crossover. The *crossover()* function selects a random pivot point in a pair of individuals (between alleles) and swaps the alleles after this point, thus generating new offspring. Offspring and parents alike fight for survival during the next iteration.

The *mutate()* function examines each new member of the population and determines whether or not a mutation will occur. Mutations decrease the chances of getting stuck in a local minimum or maximum of the search space. When a mutation occurs, a randomly selected allele is modified to produce a new individual which replaces the previous individual [25].

3.3 Product Metrics

Product metrics [11] serve as surrogates to help assess the quality of software. External quality attributes include reusability, understandability, maintainability, testability and reliability [23] among others. Li and Henry [17] introduced a core set of object oriented metrics (CK suite) that we use as surrogates for quality and to help direct the direction of the GA search for finding better refactoring solutions. The CK suite of metrics includes: Depth of Inheritance Tree (DIT), Weighted Methods per Class (WMC), Lack of Cohesion of Methods (LCOM), Coupling Between Objects (CBO), Number of Children (NOC), and Request For Class (RFC). See Li and Henry [17] for detailed descriptions of each metric.

4. SYSTEM DESIGN

This system has been designed to utilize measurements of selected object oriented metrics (as described in section II) to detect code smells and to influence the direction of the search space by a GA in order to refactor sections of legacy software. The system is broken into three major sections: 1) *input processing*, described in subsection A, which spans source code parsing and initial code graph generation, 2) the *refactoring subsystem*, described in subsections B, C, D, and E, which encompasses the GA and is responsible for gathering measurements described by selected metrics to identify potential code smells, and 3), the *output subsystem*, described in subsection F, which is responsible for combining the final code graph from the refactoring subsystem with the Eclipse Modeling Framework (EMF) [28] to generate refactored Unified Model Language (UML) [30] class diagrams. Figure 2 shows the Data Flow Diagram (DFD) of the system as well as a mapping from each section of the DFD to their corresponding descriptions (provided below). In the following sub-sections we provide detailed descriptions of each component of the refactoring system.

```
procedure GeneticAlgorithm()
define: population (a list of
                      individuals),
begin
  population = initialPopulation(size)
  evalFitness(population)

  do
    parents = select(population)
    children = crossover(parents)
    mutate(children)
    evalFitness(children)
    population = combine(parents,
       children)
  until(stoppingCondition)
end proc GeneticAlgorithm
```

Figure 1. Pseudocode for a general genetic algorithm [1]

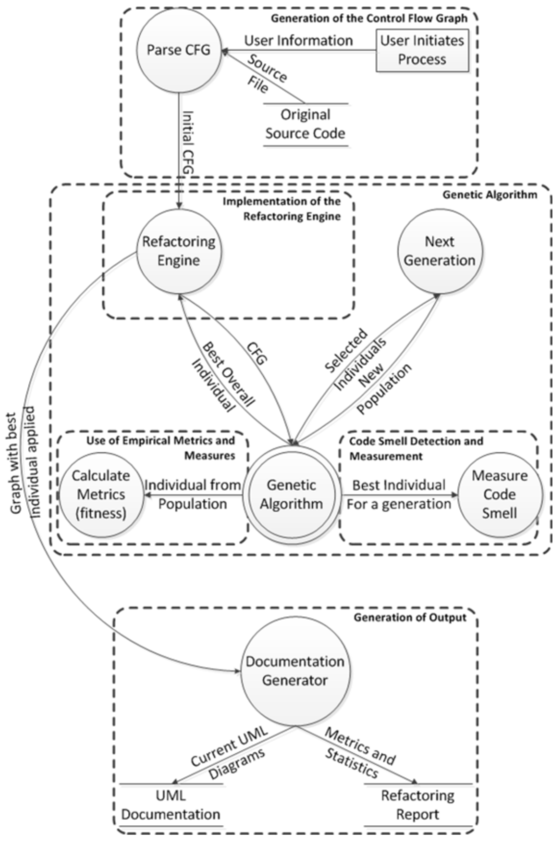

Figure 2. Overall Approach Data Flow Diagram [14]

4.1 Generation of the Control Flow Graph

We use JavaCC [29] to produce a Java language parser that generates a parse tree. The parse tree is then converted into a Control Flow Graph (CFG) representation of the source code. Although we currently only support Java, the system has been designed using the Builder Pattern [13] to allow for the swapping of parsers as well as including new languages.

We use CFGs which provide an ideal structure that captures the necessary information to facilitate the operations involved in refactoring, metrics analysis, and code smell measurements. Each CFG can contain multiples of the following Node types: *PackageNodes*, *ClassNodes*, *MethodNodes*, *AttributeNodes*, and *StatementNodes*. The names of the nodes are self-descriptive; however, we note that StatementNodes are contained within a separate graph in each MethodNode object. We also define multiple edge types. Edge types represent different association types or intra-class connections. Finally, in order to ensure that source code information is not lost after the parse tree is disposed of, the CFG maintains all its references to the nodes and edges of the graph.

4.2 Implementation of the Refactoring Engine

The Refactoring subsystem controls access to the original CFG, provides a gateway between the GA and the Metrics subsystem, and generates a final modified CFG to the Output component. The refactoring subsystem was designed using a combination of the Model-View-Controller (MVC) Pattern [6] and a combined Strategy and Command Pattern [13]. The MVC pattern ensures that the correct view of the CFG is supplied to the Refactoring subsystem, the Code Smell subsystem, and the Output subsystem. The Strategy and Command pattern combination ensures extensibility and the dynamic swap ability of the GA.

Refactorings were selected after investigating current best practices to mitigate code smells [12]. For this project we have selected *Encapsulate Field, Inline Class, Move Field, Move Method, Pull-Up Field, Pull-Up Method, Push-Down Field, Push-Down Method, and Self-Encapsulate Field* [14] and have implemented them according to the descriptions provided by Fowler in [12].

The Refactorings are used after an individual (representing a member of the population in the current iteration of the search) is generated by the GA, but prior to assessing the fitness of that individual [14]. The refactored CFG is then passed to the metrics measurement subsystem and code smell detection subsystem to evaluate the individual's fitness [14]. A favorable evaluation of the individual allows its survival in the next generation of the population.

4.3 Code Smell Detection and Measurement

Code smells represent poorly engineered code. Using metrics to quantify code smells [18]; we can determine how well a legacy system is being maintained and aid the GA with refactoring procedures that improve the fitness and hence the comprehensibility of the overall code in the system. Once the initial CFG has been generated by the Input subsystem, it is passed by the System Controller to the Refactoring Controller and eventually into the Refactoring subsystem. The Refactoring subsystem then starts the GA which performs the initial measurements of all metrics across the CFG as well as performing an initial code smell estimate. When the genetic algorithm reaches a certain code smell threshold level, it passes processing control back to the Refactoring Controller.

The implementation of code smell analysis is based on the work of Munro [18] and Roperia [24]. They suggest several algorithms to aid in finding classes that may exhibit potential code smells such as “Lazy Class” and “Temporary Field”. We have selected to implement detection algorithms for the *Lazy Class* (LYCL), *Temporary Field* (TMPF), *Long Method* (LNGM), *Large Class* (LGCL), and *Shotgun Surgery* (SHOSUR) code smells. Each algorithm has been modified to detect code smells across an entire code base and return a count of code smells detected. This is accomplished through the utilization of the code smell analysis algorithms during the initialization of the GA (during initial fitness measurements across the CFG).

Code smell detection is used to generate the list of refactoring chains. Upon initial measurement of each code smell, if one is found, a decision algorithm is invoked which will lead to the generation of a sequence of refactorings which can potentially remove the found code smell. The decision algorithm determines if the refactoring can be performed, and which other refactorings must be present in the refactoring chain to ensure that the present refactoring's preconditions will be met. Each decision algorithm is based on the prerequisite information for the implemented refactorings as found in Fowler's description of each refactoring [12]. These refactoring chains are placed into a list from which they are randomly selected during the creation or mutation of individuals by the GA.

4.4 Use of Empirical Metrics and Measures

When refactorings are identified as candidates to be applied to a CFG, the genetic algorithm first creates a clone of the CFG and then applies each refactoring in sequential order to the CFG clone. The genetic algorithm then calculates the values of each metric in the CFG clone and combines their value to generate a fitness level for the clone. Li and Henry [17] describe all the metrics used by our implementation of the GA, the means by which they are evaluated, and the quality attributes for which they serve as surrogates.

4.5 Genetic Algorithm

The GA is used to automate the refactoring(s) of legacy source code. Each GA individual represents a sequence of refactoring chains to be applied to a CFG. Each refactoring chain is a sequence of refactorings, where each refactoring is dependent on the sequence of refactorings that are before it in the sequence, and the final refactoring is the actual refactoring required to be performed. Each refactoring is implemented by a corresponding RefactoringCommand.

The GA is designed to use two operators to affect the generation of a new population: mutate and crossover. The mutate operator randomly selects a new refactoring chain from the list of remaining refactoring chains and swaps it with a randomly selected position in an individual. Currently the mutation probability is set to 10%. Crossover is defined to be the random selection of a point in two individuals where the contents between them after that point are swapped, thus forming two new individuals. Due to the potential for invalid refactoring chains to be generated, the crossover operation has been amended to include the ability to crossover different sized individuals. In order to achieve this, a normalization process is applied to the lengths of individuals, where the crossover point becomes a ratio of the length of each individual. If an invalid condition is detected during the processing of a refactoring (such as preconditions not met or missing input) the refactoring is not applied to the CFG and processing continues on the next refactoring.

When the individuals of a population are recombined to form a new population, we select the best member of the population, where the best member is the individual with the highest fitness value. The selected individual's sequence of refactorings is applied to the current CFG and the resulting CFG's code smell is evaluated.

4.6 Generation of Output

Once the refactoring subsystem has completed its tasks and generated the final CFG with the highest fitness level, it passes the CFG to the SystemController component, which invokes the OutputDirector component to generate output. Currently, the output subsystem only generates structural UML class diagrams; however, given the amount of information contained in the CFG, we plan to enhance the output to include source code and UML sequence diagrams. We utilized the Eclipse Modeling Framework (EMF) [28] to provide the UML generation functionality.

5. METHODOLOGY

We elected to develop a small test program as the experimental subject in which to run all experiments [14]. The development of a test program has allowed us to provide a simple but easily extensible platform into which multiple types of code smells can be introduced with minimal work [14]. The benefit of utilizing our own experimental subject is that we can run various experiments across the system and maintain full control of it without the possibility of having multiple developers introduce changes [14].

Experimental subjects were allocated to separate groups. Each group was comprised of the original system and clones of the original code injected with known code smells [14]. Each group in the experiment was injected with different code smells, while leaving the third group as a control group [14]. Experiments were constructed to use the genetic algorithm to generate solutions that actively remove code smells while maintaining the functionality of the subjects.

6. RESULTS

The graphs in figures 4 through 15 display results from the both experimental group and the control group. The dependent variables for the experiment are the code smells and the CK-suite of metrics. Code smells detected include: Lazy Class (LYCL), Long Method (LNGM), Large Class (LGCL), Shotgun Surgery (SHOSUR), and Temporary Field (TMPF) and are depicted in Figures 3, 4, 7, 8, 11, and 12. Metrics include: Coupling Between Objects (CBO), Lack of Cohesion in Object Methods (LCOM), Weighted Methods per Class (WMC), Class Size (CS), Depth of Inheritance Tree (DIT), Response for Class (RFC), and Number of Children (NOC) and are depicted in Figures 5, 6, 9, 10, 13, and 14.

6.1 Group 1 Results

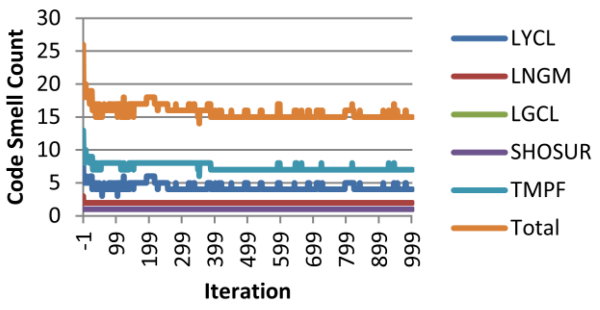

Figure 3. Code smell counts from the fittest individual of a population after every iteration of the genetic algorithm [14]

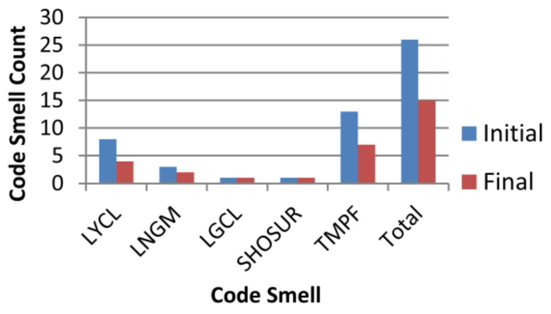

Figure 4. Comparison between smell counts before and after the genetic algorithm processes the source code [14]

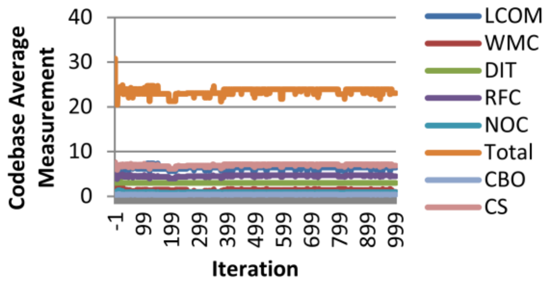

Figure 5. Average CK-metric values of the subject's codebase after every iteration of the genetic algorithm [14]

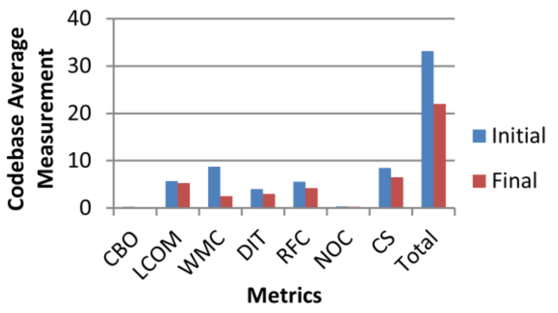

Figure 6. Comparison of initial and final metric measurements after the genetic algorithm processes the source code [14]

6.2 Group 2 Results

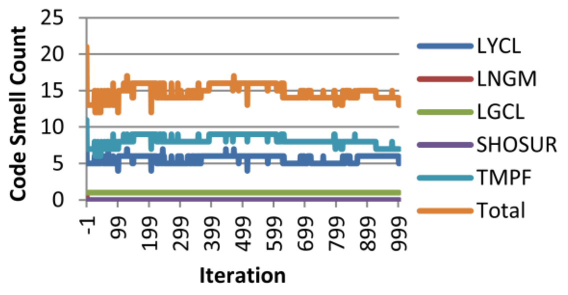

Figure 7. Code smell counts from the fittest individual of a population after every iteration of the genetic algorithm

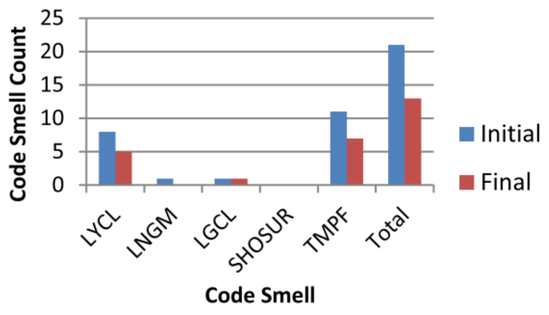

Figure 8. Comparison between smell counts before and after the genetic algorithm processes the source code

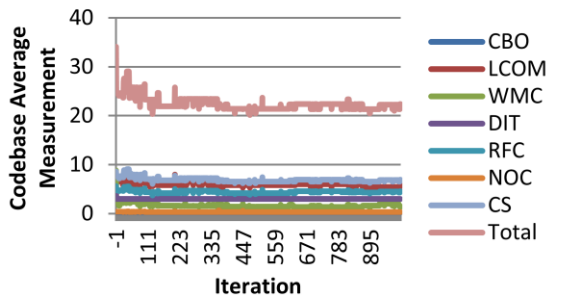

Figure 9. Average CK-metric values of the subject's codebase after every iteration of the genetic algorithm

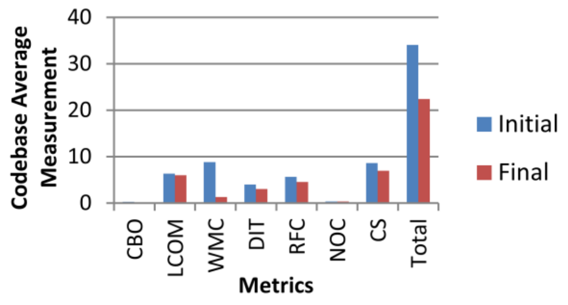

Figure 10. Comparison of initial and final metric measurements after the genetic algorithm processes the source code

6.3 Control Results

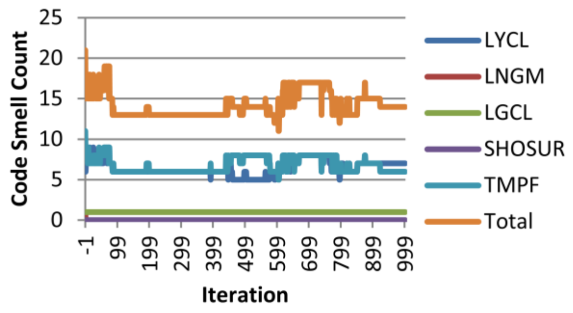

Figure 11. Code smell counts from the fittest individual of a population after every iteration of the genetic algorithm

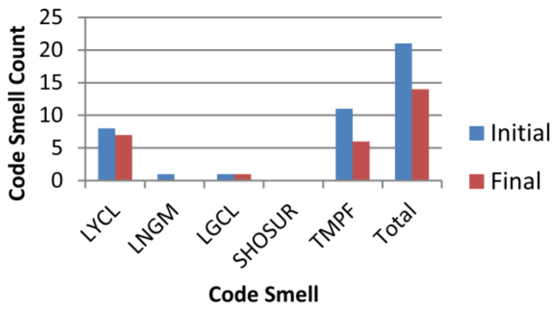

Figure 12. Comparison between smell counts before and after the genetic algorithm processes the source code

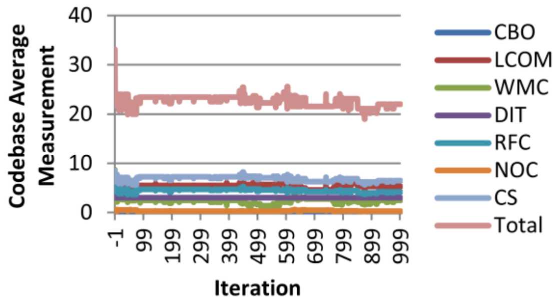

Figure 13. Average CK-metric values of the subject's codebase after every iteration of the genetic algorithm

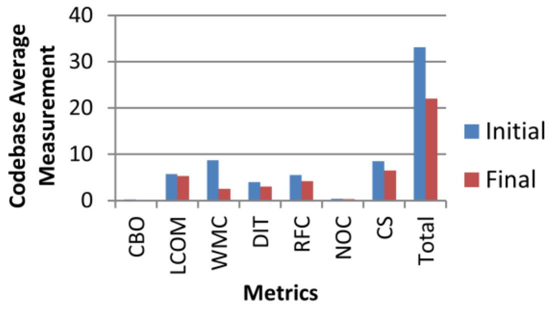

Figure 14. Comparison of initial and final metric measurements after the genetic algorithm processes the source code

7. DISCUSSION AND ANALYSIS

A number of interesting observations can be made from our results. First, when comparing the total smell counts (shown in Figures 3, 7, and 11) with the total codebase average metric counts (Figures 5, 9, and 13) for all three groups, we observe an interesting trend. As the current iteration of the GA increases, both code smell counts (for the fittest individual) and the measured metrics shows a downward trend. Thus by minimizing total remaining code smells we are effectively minimizing total measured metrics as well. This result is also reflected in the comparisons between initial and final measurements for both code smell and metrics for both groups. This is depicted in Figures 4, 6, 8, 10, 12 and 14.

Although, the CFG maintains all the information needed to accurately generate the UML diagram with information such as accessibility, methods parameters, etc., the EMF does not easily display all this information. Currently we are researching other libraries with better representation and display capabilities. Additionally, we have found that when certain refactorings are automated placeholders are required to represent new fields, methods, or classes

8. THREATS TO VALIDITY

We examine four different types of threats to validity: construct validity, content validity, internal validity, and external validity [2], [7], [8].

Construct validity refers to the meaningfulness of measurements and the quality choices made about independent and dependent variables. One must show that the measurements are consistent with an empirical relation system [11]. The dependent variables under consideration are the code smell counts and the values of all the metrics, as measured each iteration. The set of CK-metric measurements and measured code smells serve as surrogates that are clearly related to the comprehensibility aspects of quality. The former represents maintainability, reusability, understandability, testability, and reliability attributes of a system [23], while code smell counts, if viewed as indicators for new refactorings, can be attributed to increases in maintainability and extensibility [16]. Our results are clearly consistent with a real world observation of an empirical relation system.

The results also show a lack of variability in dependent variables. This allows us to clearly see trends in the results without concerns that may obscure the relationships between dependent and independent variables. In order to strengthen the construct validity of this study, additional code smells that guide the GA search should be added. Further, smarter crossover and mutation techniques could also enrich the search for better refactorings.

To have content validity, our choice of code smells and metrics must capture the notion of quality we are trying to improve – comprehensibility of source code. Our choice of metrics was sufficient to make increases in the comprehensibility of source code. But, in order to strengthen the content validity of this research and to validate the metrics selected as surrogates for quality, we should recruit software engineers of varying skill levels and use qualitative techniques to examine refactored code and compare its comprehensibility with the original source code. This could be done at both the design and implementation levels.

Internal validity refers to cause and effect relationships between independent and dependent variables. The independent variable in this study is the current iteration of the GA. Here both dependent variables are confounding, but additional metrics are necessary in order to fully investigate the relationship between them.

External Validity refers to the ability to generalize results. Currently the GA is utilizing refactoring sequences dependent upon initial code smell measurements. In order to strengthen the external validity, we need to investigate the use of this procedure on larger real world systems. Generalization of results in case studies is generally a difficult problem and one cannot infer similar results would be obtained from different types of systems that could span several domains.

9. CONCLUSION AND FUTURE WORK

Using surrogate measures that capture the notion of comprehensibility (maintainability, understandability, reusability and testability), the GA performed satisfactorily and was able to successfully reduce the overall total smell counts found in source code using a suite of metrics and code smell decision algorithms selected to reduce the search space of adequate refactorings. As to whether this truly increases codebase comprehensibility, we cannot say. What we need is a user study which is left to future research.

Whilst results show that automated refactoring is indeed viable and results are very encouraging, comprehensibility enhancements can benefit from further automated and manual processes [14]. Thus, a hybrid approach where automated refactorings help with initial structural changes coupled with expert manual intervention is the most likely approach that would yield finer results. Thus, this is not an attempt to take the human out of the loop, but instead to ease the burden on the human in the first place.

10. REFERENCES

[1] Afenzeller, M., Winkler, S., Wagner, S., and Andreas, B. 2009. *Genetic Algorithms and Genetic Programming: Modern Concepts and Practical Applications*, 1-70. Chapman & Hall/CRC Taylor & Francis Group, Boca Raton, FL.

[2] Bieman, J. M., Straw, G., Wang, H., Munger, P. W., and Alexander, R., 2003. Design Patterns and Change Proneness: An Examination of Five Evolving Systems, In *Proceedings of the 9th International Software Metrics Symposium* (September 2003). METRICS'03. IEEE. Sydney, Australia, 40-49. DOI= http://doi.ieeecomputersociety.org/10.1109/METRIC.2003.1232454

[3] Bodhuin, T., Canfora, G., Troiano, L., 2007. SORMASA: A tool for Suggesting Model Refactoring Actions by Metrics-led Genetic Algorithm, In *Proceedings of the 1st Workshop on Refactoring Tools* (July 30 - August 03, 2007, Berlin). WRT 2007. 23-24.

[4] Bowman, M., Briand, L. C., and Labiche, Y., 2007. Multi-objective Genetic Algorithm to Support Class Responsibility Assignment, *Proceedings of the IEEE International Conference on Software Maintenance* (Paris, France, October 2-5, 2007), ICSM 2007. Paris, France, 124-133.

[5] Budinsky, F., Steinburg, D., Merks, E., Ellersick, R., and Grose, T. J., 2004. *Eclipse Modeling Framework: A Developer's Guide*, 89-280. Pearson Education, Inc. Upper Saddle River, NJ.

[6] Buschmann, F., Meunier, R., Rohnert, H., Sommerlad, P., and Stal, M., 1996. *Pattern-Oriented software Architecture: A System of Patterns*, 25-193. John Wiley & Sons Ltd. Hoboken, NJ.

[7] Campbell, D. and Cook, T. D., 1979. *Quasi-experimentation: Design and Analysis Issues for Field Settings*. Houghton Mifflin Company. Boston, MA.

[8] Campbell, D. and Stanley, J., 1963. *Experimental and Quasi-experimental Designs for Research*. Rand-McNally. Chicago, IL.

[9] Chidamber, S. R. and Kemerer, C. F., 1994. A Metrics Suite for Object Oriented Design. *IEEE Transactions on Software Engineering*, 35, 6 (June 1994), 476-493.

[10] Chisalita-Cretu, C., 2009. A Multi-objective Approach for Entity Refactoring Set Selection Problem, In *Proceedings of the Second International Conference on the Applications of Digital Information and Web Technologies* (London, England, August 4-6, 2009). ICADIWT'09. 790–795. DOI=http://dx.doi.org/10.1109/ICADIWT.2009.5273850

[11] Fenton, N. E. and Pfleeger S. L., 1998. *Software Metrics: A Rigorous and Practical Approach*, Revised, 2nd ed. PWS Publishing Co. Boston, MA.

[12] Fowler, M., 200. *Refactoring: Improving the Design of Existing Code*, 27-100. Addison-Wesley. New York, NY.

[13] Gamma, E., Helm, R., Johnson, R., and Vlissides, J., 1995. *Design Patterns: Elements of Reusable Object-Oriented Software*, 79-345. Pearson Education, Inc. Upper Saddle River, NJ.

[14] Griffith, I., Wahl, S. and Izurieta, C., 2011.TrueRefactor: An Automated Refactoring Tool to Improve Legacy System and Application Comprehensibility, To appear in *Proceedings of the ISCA 24rd International Conference on Computer Applications in Industry and Engineering*.

[15] Kästner, C., Kuhlemann, M., and Batory, D., 2007. Automating Feature-Oriented Refactoring of Legacy Applications. In *Proceedings of the 1st Workshop on Refactoring Tools* (Berlin, Germany, July 30 - August 03, 2007). WRT 2007. 63-64.

[16] Kerievsky, J., 2005. *Refactoring to Patterns*, 9-21. Pearson Education, Inc. Upper Saddle River, NJ.

[17] Li, W. and Henry, S., 1993. Object-oriented Metrics that Predict Maintainability. *J. Sys. Soft.*, 1993, 23, 2, 111–122. DOI= http://dx.doi.org/10.1016/0164-1212(93)90077-B

[18] Munro, M. J., 2005. Product Metrics for Automatic Identification of "Bad Smell" Design Problems in Java Source-code, In *Proceedings of the 11th IEEE International Software Metrics Symposium* (Como, Italy, September 19-22, 2005). METRICS'05. 15. DOI= http://doi.ieeecomputersociety.org/10.1109/METRICS.2005.38

[19] O'Keeffe, M. and Ó Cinnéide, M., 2008. Search-Based Refactoring: an empirical study. *J. Soft. Maint.*, 20, 5, 345-364. DOI= http://dx.doi.org/10.1002/smr.378

[20] O'Keeffe, M. and Ó Cinnéide, M., 2006. Search-based Software Maintenance, In *Proceedings of the 10th European Conference on Software Maintenance and Reengineering* (Bari, Italy, March 22-24, 2006), CSMR 2006, 249-260, DOI=http://dx.doi.org/10.1109/CSMR.2006.49.

[21] Otero, F. E. B., Johnson, C. G., Freitas, A. A., and Thompson, S. J., 2010. Refactoring in Automatically Generated Programs. In *Proceedings of the 2nd International Symposium on Search Based Software Engineering* (Benevento, Italy, September 7-9, 2010). SSBSE 2010.

[22] Perez, J. and Crespo, Y., 2007. A Refactoring Discovery Tool based on Graph Transofrmation. *Proceedings of the 1st Workshop on Refactoring Tools* (Berlin, Germany, July 30 - August 03, 2007). WRT 2007. 7-9.

[23] Pressman, R. S., 2010. *Software Engineering: A Practitioner's Approach*, 7th ed., 613-44. McGraw-Hill. New York, NY.

[24] Roperia, N., JSmell: A Bad Smell Detection Tool for Java Systems, UMI Order Number: UMI Order No. 1466306, California State University.

[25] Russell, S. and Norvig, P., 2010. *Artificial Intelligence: A Modern Approach*, 3rd ed. Pearson Education, Inc. Upper Saddle River, NJ.

[26] Shimomura, T., Ikeda, K., and Takahashi, M., 2010. An Approach to GA-driven Automatic Refactoring based on Design Patterns, In *Proceedings of the 2010 Fifth International Conference on Software Engineering Advances* (Nice, France, August 22-27, 2010). ICSEA'10. 213-218.

[27] Zibran, M. F. and Roy, C. K., 2011. A Constraint Programming Approach to Conflict-aware Optimal Scheduling of Prioritized Code Clone Refactoring, To appear in *Proceedings of the 11th IEEE International Working Conference on Source Code Analysis and Manipulation* (Williamsburg, VA, USA, September 25-26, 2011). SCAM'11.

[28] Eclipse Modeling Framework Project (EMF). <http://www.eclipse.org/modeling/emf/>.

[29] Java Compiler Compiler (JavaCC) <http://javacc.java.net/>

[30] Unified Modeling Language, Version 2.3, Object Modeling Group, 2010, <http://www.omg.org/spec/UML/2.3/>.