48/50 A

Epistemic Retention Refactoring System (ERRS)

Ben Darling, Isaac Griffith, Scott Wahl

Abstract—**Properly understanding and maintaining legacy code is a complicated procedure. Refactoring the software can improve understandability of a system, but it is difficult to accomplish when the original code is not well understood. It is especially difficult in cases where the original developers are no longer available to offer assistance. In order to aid in this procedure, this paper presents a system referred to as the Epistemic Retention Refactoring System (ERRS). This system automatically converts source code files into a graph structure which represents both the class hierarchy and program flow. By applying genetic programming to the graph, automatic refactoring solutions are generated for use by a client. Preliminary results for this system are promising, but require refinement.**

I. Introduction

Over the years there have been many advancements in both software design technique and programming languages. As, software design evolves, this causes a discontinuity between current "best practices" and previously engineered systems. As a system, and its code, grows older the engineers who designed it either move on to new systems or perhaps new companies, thus taking with them there knowledge of the system's internal structure and workings. This leaves new engineering teams at a disadvantage in the areas of system knowledge and understanding required to extract or reuse previously written code. Thus, a system is needed to not only update the internal structure of legacy systems to current "best practices" through refactoring, but also to help reintroduce a level of understandability that was previously lost.

Therefore the problem facing today's engineers is as follows: Given that time to produce new systems is becoming smaller and smaller, the need to reuse previously written code is becoming ever increasing. Yet, without a means to understand code from previous designs, or a lack of engineers familiar with those designs (and the standards from when they where created) a tool is needed which can aid the new engineers. This tool must be able to restructure the code in such a way that it matches the current "best practices" and simultaneously generates code which has an increased property of understandability. That is, the source code produced is more structurally sound and easier to understand, while simultaneously producing the same output.

II. Refactoring

Refactoring is, according to Martin Fowler, "a change made to the internal structure of software to make it easier to understand and cheaper to modify without changing its observable behavior." [3] There are many techniques which fall into the domain of refactoring which can and cannot be applied by a machine (easily). In the scope of this work we have selected some of the easier techniques for proof of concept.

Refactoring itself directly provides a means to accomplish our goal, which is the same goal of refactoring. That is, refactoring is the direct technique required to restructure object oriented code in order to retain or regain lost knowledge of the purpose of that source code. Yet, normally refactorings are performed by developers who actively work on the source code being refactored, and hence have a working knowledge of the codes purpose and design. Within the scope of the problem to be solved here, we assume that the last requirement is not present and that the developer is either new to the project or is attempting to reuse a portion of older legacy code. Therefore, while refactoring can provide more understanding, it would require the developer to understand the underlying code being restructured. In order to prevent inordinate amount time lost, we prescribe that a machine can take over the task of initially restructuring the code into a more easily understandable form from which refactoring can take over. We now move to describe the specific refactoring techniques we choose to implement.

A. Refactoring Techniques

The following is a list of refactoring techniques and their descriptions. Each of these techniques where specifically selected to determine if refactoring can be applied as an application of genetic programming and as an attempt to increase code understandability.

1) **Move Method** - The problem is that a given method, M1(), in Class C1, is used more often by methods of Class C2 than any other class. It stands to reason then that perhaps M1() should have been a method of Class C2 rather than of Class C1. Therefore, the solution is to move method M1() from Class C1 to Class C2.[3]
2) **Move Field** - This problem is similar to the problem solved by Move Method. That is, we have a class C1, with a field F1, which is used more often by another Class C2 rather than its own class. Therefore, the solution to this problem is to move F1 to C2 from C1.[3]
3) **Pull-up Field** - Similar to move field, but the problem is that a Field F1 exists in a class C1 and a Field F2 exists in class C2. Field F1 and Field F2 are identical and Classes C1 and C2 extend the same base class C3. That is, there are two classes in the same inheritance hierarchy, and contain the same field which is not inherited. The solution is to remove the field from the subclasses and move it into the super class.[3]
4) **Pull-up Method** - Similar pull-up field and move method. In this case there are two classes C1 and C2 that both inherit from C3, but separately define identical methods M1 and M2. In order reduce redundancy of

code and increase maintainability M1 and M2 are removed C1 and C2, respectively and their contents shifted to C3. Thus, C1 and C2 still have the method defined but only one instance of the method need be changed if so desired.[3]

The specific refactorings where selected in order to reduce code complexity, increase maintainability, increase reusability, and increase understandability.

B. Code Smells

In order to determine when refactorings should be applied a set of qualitative measure were created by Martin Fowler *et. al.* which are collectively name Code Smells. Each smell provides a description of the problem and a set of possible refactorings to be used in order to remove the smell from the code. The limitation, as it applies to this research, is that what constitutes a smell is not qualitatively available. What is required is contextual knowledge to be distilled into an algorithm means to identify the smell. Another limitation is found in the following fact: for a given code smell there is a set of *possible solutions* for which each entry may be a single refactoring or a sequence of refactorings.

Quantifying code smells to be useful to search-based solutions to the refactoring problem is a novel-approach, but in itself a hard problem. It requires not only contextual knowledge of the code itself, but also of the potential outcome of each refactoring to be placed in a sequence of possible refactorings. If the quantification is provided, then it could be used as a heuristic to be applied when attempting to solve the problem of generating a list of optimal refactorings to be applied. In other research which has attack this problem, there remains an issue which this research attempts to avoid. This is the issue of requiring an engineer who has the necessary understanding and knowledge about the project, to which the refactorings will be applied, who can provide insight to the tool generating the refactorings. The assumption for this research project is that this engineer is either unavailable or does not have the necessary prerequisite understanding and knowledge of the project at hand. Overall, we found that perhaps code smells could be useful in generating an initial set of refactorings to be applied to generate our initial population, we choose not use them.

In lieu of code smells to evaluate the code, there is another way around the problem. Instead of using code smells to evaluate the source code and then apply refactorings, the idea is to instead apply refactorings to maximize a specific set of metrics. Where the metrics should be selected in order to retroactively measure the same characteristics that code smells are designed to exemplify; mainly the maintainability, reusability, and understandability of the source code. If this can be done then selecting the sequence of refactorings that maximizes these metrics over a series of iterations will itself provide a similar effect as applying code smells in the first place. The benefit found in this latter approach is that it does not require an engineer who has the necessary prerequisite knowledge of code smells, refactoring techniques, and contextual understanding of the source code to be restructured.

C. Metrics

The restructuring of a program via genetic programing necessarily requires a means to evaluate the fitness of the solution. This requirement of a means to assess the fitness of the refactorings as applied to the graph structure representing the modified program necessitates that some form of metrics must be used. There are many metrics in existence in the software engineering community and there are those that specifically meet the criterion demanded by the problem we are trying to solve. That criterion is as follows (1) metrics which assess the maintainability of the code; (2) metrics which access the reusability of the code; and (3) metrics which attempt to assess the understandability of the code.

The selection of metrics was an attempt to choose metrics which assess multiple criteria at a time. Once that initial selection was completed a further criteria was put in place which limited the metrics to those that genetic programming could influence. From this we were left with the following metrics:

- Coupling Between Classes(CBO): $\sum Number(C'_{methods} UsingC_{methods}) + Number(C'_{methods} UsingC_{fields})$ where C and C' are classes, and C is the class the metric is applied to.
- Lack of Cohesion in Methods(LCOM): $\prod \frac{\#(edges-1)-\#(nodes)}{edges-1}$ where *edges* is the set edges connecting nodes in a method's CFG and *nodes* is the set nodes of the CFG.
- Weighted Methods per Class(WMC): $\sum CC(m)$ where m is a method in the class.
- Cyclomatic Complexity(CC): $Number(edges) - Number(nodes) + 1$ where the *edges* and *nodes* are the respective sets which comprise a methods CFG.

The use of these metrics in order to determine performance of the refactorings applied requires two conditions. The first is whether maximization or minimization of the function is required. That is, is progress towards maintainability, understandability, and reusability measured by an increase in the metric or a decrease. The second requirement is an understanding that these metrics are designed simply to be applied to a portion of the source code and not the entire project. Thus, there must be a means by which we can normalize the metrics across the entire graph, representing the source code. The normalized metrics are as follows:

- Coupling Between Classes(CBO): $\frac{\sum Number(C'_{methods} UsingC_{methods}) + Number(C'_{methods} UsingC_{fields})}{NumberOfClasses}$ where C and C' are classes, and C is the class the metric is applied to. *NumberOfClass* is the number of classes in the number of classes in the project.
- Lack of Cohesion in Methods(LCOM): $\frac{\sum \prod \frac{\#(edges-1)-\#(nodes)}{edges-1}}{NumberOfClasses}$ where *edges* is the set edges connecting nodes in a method's CFG and *nodes* is the set nodes of the CFG and *NumberOfClasses* is the number of classes in the project.
- Weighted Methods per Class(WMC): $\frac{\sum CC(m)}{NumberOfClasses}$ where m is a method in the class, and *NumberOfClasses* is the number of classes in the project.

These metrics having now been normalized, with exception to Cyclomatic Complexity which when normalized would become simply WMC, the use of the metrics must now be discussed. Coupling between classes is itself and indication of how connected a class is to other classes. Thus, if CBO is too high then a given class is well connect and this creates a lower degree of maintainability. It also indicates lower understandability due to multiple connections between classes. Thus, in order to effectively measure and utilize this metric the program must attempt to minimize the overall CBO. Lack of Cohesion in Methods is an attempt to measure how connected a method is within its own class. If a method is not well connect in its class then it may not belong in that class in the first place. This is assuming that it is not necessarily a static class that is designed to be used as a utility. Thus, if LCOM is low then this is an indication of poor reusability and maintainability, as well as understandability (when the engineer needs to track down the uses in a variety of other classes). Therefore the program will need to maximize the overall cohesion throughout the project. Finally there is Weighted Methods per Class, which is itself (prior to normalization) a metric which is intended to assess the overall complexity of a class. If WMC is very low then this is an indication that the class is less complex. That is, it has been developed such that the methods are less complex and thus perhaps only perform a few tasks at most. If WMC is low then understandability will be increased as well as maintainability, since smaller chunks of code which only perform a small amount of tasks is easier to comprehend and maintain. Thus, the program will need to minimize WMC for the project overall.

III. Genetic Programming

Genetic algorithms are stochastic processes modelled after the process of evolution. Specifically, it attempts to model natural selection and survival of the fittest over a series of generations of a population. For a slightly formal definition, genetic algorithms generally consist of the following:

- Population: As the system is attempting to model the process of evolution for a group of individuals, the genetic algorithm begins with a specified population. For most systems, the number of individuals kept in the population is fixed across generations.
- Selection Criteria: This is the method for selecting which individuals from the population will breed which corresponds to the natural selection portion of evolution.
- Crossover: The crossover step determines how parents are combined to create individuals for the subsequent generation. The specifics of how this is accomplished is highly dependent on the nature of the individuals.
- Mutation: As with real populations, random mutations are inflicted upon the individuals based on a set mutation rate.
- Retention: Once selection, crossover, and mutation are completed, retention describes the strategy used in determining which individuals will remain for the next generation. It is this step which most directly corresponds with survival of the fittest.

In the classical genetic algorithm, individuals in a population are represented by a fixed length bit string such that all members of the population are of the same size. Crossover occurs by selecting an index within the range of the bit string of the parents. The first child obtains the bits from parent one which are before the index and the bits from parent two which are after the index. A second child receives the opposite. Mutation occurs by randomly flipping a single bit.

For this system, individuals are instead composed of a sequence of refactorings. Since the sequences can be of differing length, a modified crossover is used and is described in section 4. Additionally, the fitness proportionate selection criteria is used for determining which individuals are selected as parents in crossover. This procedure works by selecting an individual with a probability proportional to its fitness relative to other individuals. Naturally, this requires an appropriate fitness function as well. The function selected is discussed in section 4.

IV. Algorithms and Experimental Methods

A. Graphs and Parsing

The graph structure we have designed is generated by parsing a given source code file and generating each node of the graph complete with code. The parser chosen to be used for the Java language was generated using the javacc lexical analyzer using the open source java specification available on the javacc tool's website. This specific version produces the code to parse a java source file which is compliant with the Java 1.5 language specification. Using the parser to parse through the code a two-pass system is used where the first pass generates the nodes of the graph and the second pass generates the connections between the nodes. during the second pass the context flow graph (CFG) is also generated. The parsing control class which is provides a given source code directory, and it recursively searches from the base directory and collects a list of file names. It then processes each file through the parse and prepares to generate the graph.

After the code is parsed it can be divided into chunks that represent its overall structure. A graph is the best representation of code because it shows each chunk of code as a node and the edges represent which code segments it is connected to. Classes, statements, fields, and methods all become the nodes and are connected with different kinds of edges such as Inheritance or Aggregation. After the graph is constructed from the original code, we need a way to manipulate it so that the proper refactorings of the code can be made. Nodes can be disconnected and reconnected with a different node or an extra edge can be added on so that the graph can be reorganized properly. Edges do not need to be manipulated directly because there is only a use for an edge that is connected on both sides. Since edges really only contain the data that represents the type of connection, it is easier to just remove unused edges and recreate them where they are necessary rather than keeping track of unused edges. As the GA begins to recommend changes to refactor the code the controller can easily modify the graph, rather than the actual code. This is why the graph structure is important to the refactoring of the code as it would

be very difficult to reorganize the code directly as a series of strings. Once the graph is done being manipulated, the code can be rebuilt simply by extracting the strings of code from the nodes. The new file is created from combining the strings of code and seeing how they connect based on the edges of the graph.

B. Controlled Refactoring

The refactoring of the code stored in the graph structure is performed by a Refactoring Controller. This controller provides graph manipulations which are designed to provide for the refactorings required. In this implementation we have provided for *Move Method*, *Move Field*, *Pull-up Field*, and *Pull-Up Method* as the initial experimental methods. In determining which refactorings to use we determined that simple refactorings, such as Rename Field, Rename Method, etc. are beyond the ability of graph manipulation and either require a system that can incorporate domain and context specific knowledge or requires an engineer to provide additional information. Both of these requirements are beyond the scope of this research.

While conducting research it was realized that, if given a Graph G, consisting of connections between class, methods, and within methods a generated CFG, nearly all refactorings can be represented as a series of graph manipulations. For example Move Method can be accomplished by disconnecting the node representing the method from its class node and connected to the new class node, the same applies for a field. Each of these refactorings is provided by a refactoring controller which acts as the main interface inside of the tool.

The refactoring controller acts as a means through which the Genetic Programming algorithm can interact with both the graph data structure, the refactorings to modify it, and the metrics by which it is evaluated. The metrics provided were selected to provide measures of understandability, reusability, and maintainability. Each of these metric selecting criteria were used because they inherently underlie the core concepts of what an engineer requires in order to understand the code. Thus if a system is designed in such a way as to increase reusability and maintainability then understandability will increase as well due to the well defined structure of the code and ease by which maintenance can be performed. If these all are increase then the ability of a new engineer on the project to learn about the code will increase as well.

C. Genetic Programming

The current iteration of the ERRS program utilizes a specialized genetic programming algorithm. Under this system, individuals in the population consist of a sequence of single refactorings. Thus, the chromosome for a single individual is comprised of swapping criteria consisting of *type*, *childNode*, and *parentNode*. Based upon this formulation, every single swapping criterion, or allele, specificies what type of refactoring a node is undergoing and the resulting set of parents for that node.

As an example, consider simple graph prior to a sequence of Move Method refactorings as shown in Figure 1. An individual with a chromosome of $\{(MoveMethod, X, ClassOne), (MoveMethod, Z, ClassOne), (M$ would cause method X and Z to be moved into $ClassOne$ while B is moved to $ClassTwo$ as shown in Figure 2.

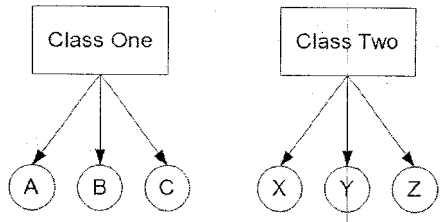

Figure 1. Initial Graph

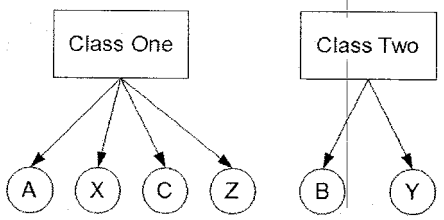

Figure 2. Resulting Graph

Selecting two parents for crossover occurs by fitness proportionate selection. Crossover between individuals occurs by a uniform selection among individual portions of each parent. More specifically, at index position i for two parents, the first child randomly inherits the allele at index i from parent one or two with equal probability. The parent which wasn't selected contributes its allele at index i to the second child. If the two chromosomes are of different lengths, they are effectively made the same length by inserting null swaps (refactorings that don't perform any modifications). Once the crossover is completed, these excess refactorings are removed as they do not change the final result of applying the refactoring sequence.

Once the crossover is completed, based upon the mutation rate, a child may be randomly mutated where a randomly selected allele in its chromosome is modified. These modifications consist of deleting a refactoring, adding a new refactoring, or modifying an existing one. These all occur with equal probability. Once the mutation procedure is complete, an elitist scheme is utilized for selecting the individuals which remain in the population. The original design used a fitness proportionate selection without replacement. However, in order to better track the best individual, and for simplicity, this was modified. As in the standard genetic algorithm, this process was repeated until convergence.

Testing fitness of an individual was accomplished by creating a copy of the source code graph and applying its suggested refactoring to the system. The metrics mentioned above were then calculated based upon the resulting graph. The results of these calculations were weighted such that the weights summed to one and added (or subtracted) based upon the values measure of quality code design.

While performing the experiment, the parameters for the genetic algorithm were tuned manually. Initial tests of the

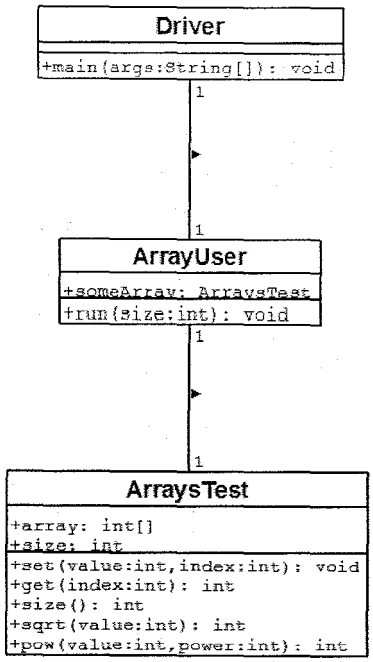

Figure 3. UML Diagram of Test Program One

system were allowed to run for many generations, but as was determined, this was not necessary due to its early convergence. In the final runs, the number of generations was limited to one-hundred. In addition to this parameter, the mutation rate of the system was set at a constant 5% as higher mutation rates did not significantly improve the convergence rate. The population size was kept at 20 and the initial members of the population were set to have a random number of refactoring steps by a uniform distribution over the interval [2, 10].

D. Test Code

To test the efficacy of the system, two test programs were created. They were specifically designed such that there was an interdependency on various mathematical functions and the complexity of the system could be reduced by refactoring. A UML diagram for the first of these test programs can be seen in Figure 3. For this test program, the ArrayUser class utilizes an instance of the ArraysTest class. The ArraysTest class is simply a container class for an array of integers. However, in addition to its basic functionality, the ArraysTest class contains functions for calculating the integer square root and power functions. These functions are not used within the class itself, but they are used by ArrayUser for some simple calculations. If the ERRS system works as expected, it should move the two methods out of the ArraysTest class and place them in the ArraysUser class.

The second of the test programs is slightly more complicated and is shown in Figure 4. In this program, there is a far greater degree of interdependence among the classes. Both TestClassOne and TestClassTwo rely on the MathFns class in order to calculate the integer square root and power

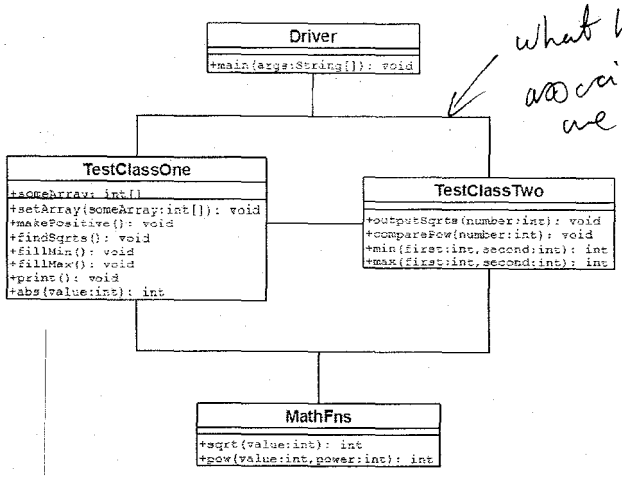

Figure 4. UML Diagram of Test Program Two

functions. In addition to this, TestClassOne uses the functions in TestClassTwo in order to calculate the minimum and maximum of two integer values. TestClassTwo contains a call to the absolute value function in TestClassOne as part of one of its functions. None of these functions are strictly required to be located with TestClassOne or TestClassTwo. Thus, if ERRS can find a suitable refactoring, it should place the min, max, and abs functions in the MathFns class, completely eliminating the link between TestClassOne and TestClassTwo.

V. Results

Due to time constraints, only five trials were attempted over the first test program. Over all five of these trials, the ERRS system recommended moving at least one of the two functions previously identified as problems. In two of the trials, the power function was moved from ArraysTest to ArrayUser. In two different trials, the square root function was moved from ArraysTest to ArrayUser. The the last of the five trials moved both the square root and power function from ArraysTest to ArrayUser, as was desired. However, it also attempted to move the size field and function. This is a valid refactoring, but does break the logical association of the size of the array and the array itself.

In addition to this qualitative analysis, the progression of the best and average fitness of the population was recorded as a function of time. The graphs of this show that the program suffers from early convergence. Given the small number of different possibilities and resulting calculations, the best overall value was found early and the rest of the individuals in the population rapidly converged to this value. This occurred within the first twenty generations for each trial. A representative graph from these trials can be seen in Figure 5.

In trial two, however, the convergence rate of the system was not as swift. Though some trials still converged within the first thirty trials, some still showed improvement at the end of the allotted number of generations as shown in Figure 6. Additionally, the system did not find the expected refactoring for the system. Instead of placing the mathematics functions

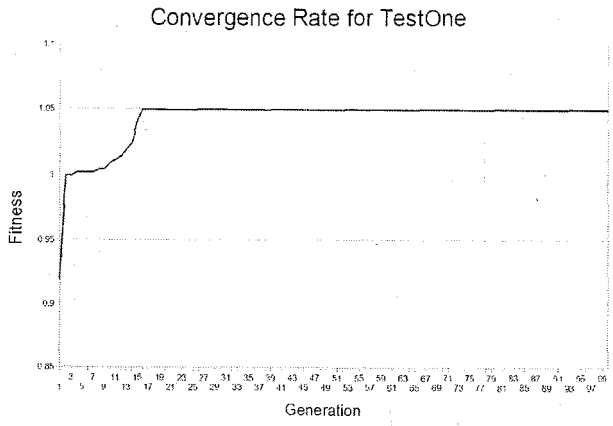

Figure 5. Convergence Rate of Trial 5 on Test Program One

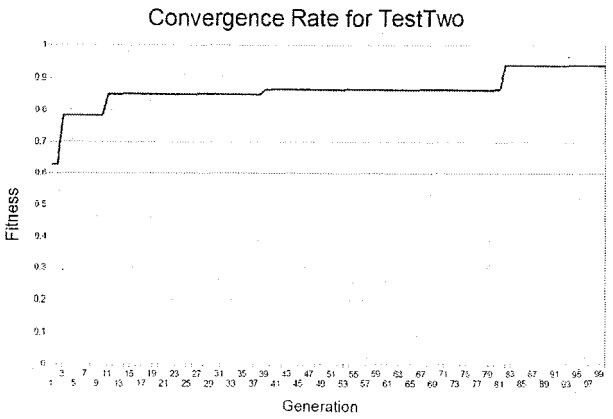

Figure 6. Sample Convergence Rate on Test Program Two

inside the separate math class, the mathematics functions were frequently removed from both the math class and one of the test classes and placed into the other test class.

VI. Discussion

Based on the results from the first test program, early convergence, especially on small programs, is an issue. Whether or not this can be corrected by utilizing genetic diversity principles will have to be tested in future work. Despite this, on the small programs, ERRS did manage to identify methods which did not logically belong in a class and move them to the correct class in the first test program. On the larger program, the ERRS system did not find the expected refactoring, but the resulting refactorings were still well within reason, consolidating the various mathematics functions within a single class. In both cases, the system would sometimes also suggest seemingly strange refactorings, such as moving a function closely tied with a class and its fields to a different class. However, these results were not consistent and seems to be caused by the early convergence of the system. With a stronger fitness function and better parameter tuning, it should be possible to improve on these results. As a proof-of-concept, however, the results are very promising.

VII. Threats to Validity

Unfortunately for the research this program does not actually generate the modified code in any form of output. Thus, even though it may refactor the code within the graph data structure, its output may not truly refactor the code by producing a restructure program that operates the same. If this is true, the results that followed from the operation of the program although relevant may not be able to fully determine the usefulness of the tool.

Another issue is the metrics defined to be normalized. Although they are intended to provide a means of measuring the metric across the entire class, the metrics may include a flaw due to the normalizing factor of using the number of classes in the project. Though this is perhaps only a small issue that can possibly be mitigate by the fact that the metrics themselves are used to only determine whether the genetic programming algorithm is applying viable and useful refactorings.

The issue of the metrics leads to another issue that inherent in the methodology. This issue is based in the methodology in which the metrics are applied after the refactorings, retroactively, instead of proactively in such a way as code smells would be. Yet, using predefined and proved metrics is perhaps more efficient that designing metrics, that can be proactively used, based on code smells.

VIII. Conclusion

Based upon the results obtained, the initial formulation of the ERRS system shows promise in aiding understanding by refactoring legacy code. The current program does face issues with early convergence, but this should be correctable by enforcing genetic diversity or potentially by differing population models (such as islanding). However, there is a considerable amount of additional work to perform in order to obtain a more reliable system.

Immediate future work will consist of a means by which the modified code can be returned as output. This system can take two forms, the first is a means of traversing the graph structure and extracting the code into source files. The second means of accomplishing this task would be to generate UML from the graph structure, by producing a XML Metadata Interchange (XMI) document. Either method allows the newly restructured code to be generated. Along with output is the need to apply more powerful refactoring techniques and better metrics for which to evaluate the code, specifically object oriented metrics beyond coupling and cohesion.

References

[1] Bowman, M., Briand, L. C., & Labiche, Y. (2007, April). Multi-Objective Genetic Algorithms to Support Class Responsibility Assignment. , 1-16.

[2] Chisalita-Cretu, C. (2009). A Multi-Objective Approach for Entity Refactoring Set Selection Problem. *Second International Conference on the Applications of Digital Information and Web Technologies*, 790-795.

[3] Fowler, M. (2000). Refactoring: Improving the Design of Existing Code (pp. 27-170). Addison-Wesley.

[4] Tsantalis, N., & Chatzigeorgiou, A. (2009). Identification of Move Method Refactoring Opportunities. IEEE Transactions on Software Engineering, 35(3), 347-367.

[1] Russell, S. and Norvig, P. (2003). Artificial Intelligence: A Modern Approach (Second Edition). Pearson Education, Inc. Upper Saddle River, NJ.