

# Software Refactoring Using Genetic and Heuristic Algorithms

Edwin Kaburu  
Department of Computer Science  
Seattle University  
Seattle, USA  
[ekaburu@seattleu.edu](mailto:ekaburu@seattleu.edu)

Kenny Halim  
Department of Computer Science  
Seattle University  
Seattle, USA  
[khalim@seattleu.edu](mailto:khalim@seattleu.edu)

Pranavi Taneti  
Department of Computer Science  
Seattle University  
Seattle, USA  
[ptaneti@seattleu.edu](mailto:ptaneti@seattleu.edu)

Christopher Downing  
Department of Computer Science  
Seattle University  
Seattle, USA  
[downingchris@seattleu.edu](mailto:downingchris@seattleu.edu)

**Abstract**— Software refactoring is a process that involves improving a software system’s internal behavior along with its structures while maintain the software system’s external behavior. Manual refactoring a software system is a complex and a time-consuming affair. In this paper, we have presented a literature review of five research papers on software automation. These research studies within these papers will utilize genetic and heuristic techniques to improve maintainability, modularity, and understandability of software systems. Thereby reducing challenges faced by developers or engineers, such as accidental complexities and maintenance, during a software development life cycle. We formulate future research prospectives and applications through analyzing the different morphological refactoring techniques constructed.

## I. INTRODUCTIONS

Automation is the process of minimizing human efforts. Adaptation of automation has been attributed to major advancements in sciences and technologies which have challenged and subsequently changed our understanding of the natural and metaphysical phenomenon. It is from these advancements that we have been able to restructure and automate complex systems processes embedded within institutions, entities, and individuals. Some of these automated processes involve industries utilizing robotics within plants to develop products, self-checkout commerce machines, and autonomous airplanes and vehicles. Most of the aforementioned automated system processes are partially developed with software system code. Therefore, the importance of automation within software engineering and development pertains to the case that software code is generally created and maintained by humans. Likewise given the intricate nature of complexity involved with computing systems relied upon by society, software quality robustness along with maintenance cannot entirely be achieved manually. Our research literature initiative focus has been on automated techniques which encapsulate software refactoring complexities to grant software maintainability and understandability. As such we analyze, [1], [2], [3], [4], [5] and provide our own culminations on their results of adaptability within software engineering. Our research goals are objectively to understand the utilization of genetic and heuristic algorithms within software refactoring automation of computing software systems. The paper is structured as follows, a genetic and heuristic algorithms fundamental overview, a

synopsis along with inference remarks on [1] [2] [3] [4] [5], and future directions with conclusions.

## II. GENETIC ALGORITHMS

### A. Fundamentals Overview

For this section we will be looking into software refactoring with genetic algorithms. Genetic algorithms are a subset of Evolutionary algorithms inspired by the biological process of population evolution over a period of time. A single generation is composed of individuals who are consecutively subjected through a fitness function test to find the best possible individuals. A crossover operation and random mutations among the selected individuals will occur to increase randomness of the next iterative generation. *Figure 1* from [1] and *Figure 2* from [6], show the genetic algorithm pseudocode and pipeline activity respectively.

**Input:** Number of generations  $nb$   
**Output:** Population  $p$

```

1  $i \leftarrow 0$ ;
2  $p \leftarrow \text{initialPopulation}()$ ;
3 while  $i < nb$  do
4    $p' \leftarrow \text{SELECT}(p)$ ;
5    $\text{CROSSOVER}(p')$ ;
6    $\text{MUTATE}(p')$ ;
7    $p \leftarrow p'$ ;
8    $i \leftarrow i + 1$ ;
9 end
10 return  $p$ 

```

Fig. 1. Genetic Algorithmn PseudoCode [1]

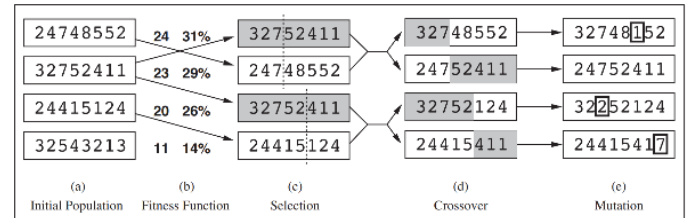


Fig. 2. Genetic Algorithmn Workflow Pipeline [6]

### B. A Genetic Algorithm Based Approach for Automated Refactoring of Component Based Software

In “A genetic algorithm-based approach for automated refactoring of component-based software”, the premise is finding an efficient technique to refactor components within software systems. Its emphasis is that component based bad smells along with their characteristics cannot be grouped or encapsulated together with the characterization of object-oriented smells. Likewise, techniques employed towards refactoring object-oriented bad smells such as collapse hierarchy and extract class, are not admissible to components due to additional collective of classes, and interactions.

*a) Methodology:* Within the research paper, some of the bad smells described which encapsulated characteristics of components are ambiguous interface, connector envy, scattered parasitic functionality, component concern overload, and overused interface [1]. To refactor these component bad smells, a refactoring pipeline technique would be employed to accumulate refactoring operations which are admissible. The refactoring pipeline technique decomposition initially defines and formulates a detection rule for component smells along with presenting a refactoring pattern. The remaining processes within the pipeline involves extracting a source code model from the code and exploring the solution space with a genetic algorithm. Some of the possible refactorings operations for components that result from the pipeline would be pull up interface, extract interface, push and extract component [1].

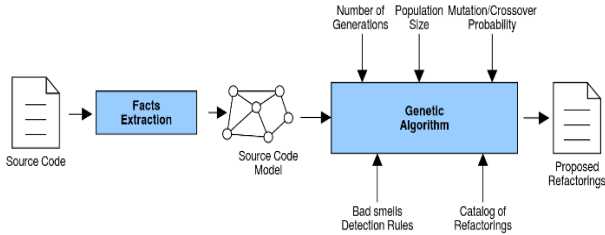
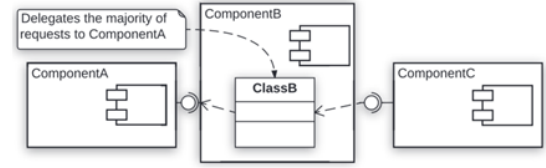


Fig. 3. Component Refactoring Pipeline [1]

Each of the component bad smells are defined along with a detection rules composition to identify if a component contains the associated smell. For instance, the connector envy smell is defined by the researchers as a component delegating a majority of its requirements to other components. Its detection rule is a summation of calls internally and externally relative to the component, over total number of calls, from which its cohesion will be high due to its delegation behavior.




$$CE(C) = \frac{1}{2} \cdot \left( \underbrace{\frac{\sum_{i \in C}^{j \neq C} (|SOC(i, j) \cup SOC(j, i)|)}{\sum_{i \in C}^{j \neq C} (|SOC(i, k) \cup SOC(k, i)|)}}_1 + \underbrace{LCC(C)}_2 \right) \quad (2)$$

where:

- $LCC(c)$  denotes the cohesion of the classes belonging to the component  $c$  according to the commonly used *loose class cohesion* metric proposed by Bieman and Kang in [34].

Fig. 4. Connector Envy Illustration and Detection Rule [1]

In terms of semantics for the source code model extraction, an extraction engine would be utilized to transform source code or metadata into a model. The extraction engine according to the researchers would depend on a few parameters such as the component model, programming language and manifest configuration file. The semantics for the genetic algorithm comprises of individuals decomposed structurally to a genotype, composed of refactoring techniques operations, and phenotype containing the source code model [1]. The phenotype is “obtained by performing the sequence of refactorings on the initial source code model in the order that is given in the genotype” [1]. The fitness functions, of the genetic algorithm will select noteworthy individuals for the crossover or mutation, its formulation will be a summation of all the bad smell detection rules.

Phenotype	Genotype
 Source Code Model	pullInterface(C5,c1)
	extractComponent(C6, {c1,c2,c3})
	embedComponent(C3,C2)
	extractInterface(C3.i1,{s1,s2})
	embedComponent(C1,C4)

$$fitness(S) = \sum_{C \in S} \left( AI(C) + CE(C) + CCO(C) + \sum_{i \in C, p} OI(i) \right) + \sum_{S' \in partitions(S)} SPF(S') \quad (6)$$

where:

- $partitions(S)$  denotes the set of partitions of  $S$  where an indirect path exists that connects all the components of the partition.

Fig. 5. Source Code Model And Fitness Function [1]

*b) Results:* The researchers employed their technique onto the components contained within Eclipse Link. Their findings overall were the fitness value of the genetic algorithm did not increase software size but decreased it, in subsequent generations.

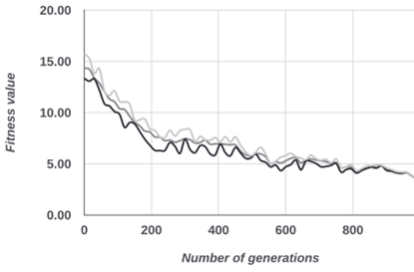


Fig. 6. Number of Generations to Fitness Value [1]

The structural evolution of Eclipse Link's components contains few dependencies with an increase in cohesion. In the before and after optimization images, there are fewer links to the *core component*, some components were removed while others were combined or relinked.

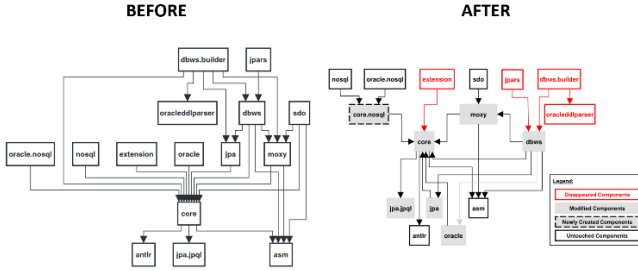


Fig. 7. Component Comparison [1]

*c) Takeaway:* From the research conducted, we view genetic refactoring components is persistently effective because it takes into consideration the context of the software system topology. Based on the results, the refactoring technique process is suitable in maintaining software quality and simplifying software system's architectural component structural design, through identifying and removing duplicate or dead components as well as refactoring used components.

### C. On the Use of Genetic Programming for Automated Refactoring and The Introduction of Design Patterns

It is a laboring process to maintain an object-oriented design in a software. Hence in , *On the use of genetic programming for automated refactoring and the introduction of design patterns*, the objective is to address that problem through genetic programming to identify sets of refactoring to apply to a software. Also, to answer the question, "does genetic programming help with regard to software refactoring." Metrics and design patterns are both methodologies used in the evaluation of a software design quality and providing a driven solution to a recurrent problems in software system. However, adopting these two metrics needs a lot of investment in terms of labor and intelligence, which is why this paper addresses those concerns by automating the use of metrics to generate refactoring strategies that will establish design patterns.

Metrics are used in software engineering to measure specific characteristics of software, through providing a snapshot or profile of design changes over time. The Quality Model for Object-Oriented Design (QMOOD) is a metric suite that includes 11 metrics for evaluating object-oriented design

quality. QMOOD is amenable to automated evaluation, making it ideal for use in software engineering tools. The suite measures abstract quality characteristics, and a real-value weight can be assigned to each characteristic to specify its relative importance. QMOOD has been validated through an individual quality characteristic evaluation and an overall quality estimation. Design patterns are small, reusable solutions to common design problems, and a subset of the Gamma design patterns, including Abstract Factory, Adapter, Bridge, Decorator, Prototype, and Proxy, can be applied to object-oriented software designs. The process of applying a design pattern to a software design is known as design pattern instantiation.

*a) Methodology:* The solution representation for software refactoring has two components: a design graph and a transformation tree. The transformation tree contains nodes representing small changes to software design called Mini transformations, which can create design patterns. Mini transformations are defined as functions which are expressed concisely as nodes in the transformation tree. These nodes gather input from child nodes representing elements of the software design being analyzed. The QMOOD metric suite is used to evaluate the quality of refactored designs, which includes a variety of metrics for analyzing the complexity of object-oriented hierarchies and the cohesion of classes. The experiments involve applying Remodel, an approach for improving software design using genetic programming, to a set of 58 published software designs. The first experiment establishes a baseline using only the QMOOD metric suite to determine fitness, while the remaining experiments determine optimal values for the coefficients in the fitness function. Each experiment considers several values for a single coefficient, and once the best value is determined, it is used for the remaining experiments. The resulting fitness function is then applied to the large case study. The experiments were conducted on a large cluster of PCs running SuSE Linux Enterprise Server 10, and the prototype GP was constructed using the Evolutionary Computation for Java (ECJ) framework. Each experiment comprised 100 generations, with tournament selection used and only the most-fit individual at the end of each run considered. REMODEL was applied to each design five times using a unique random seed for each run to obtain an unbiased sample.

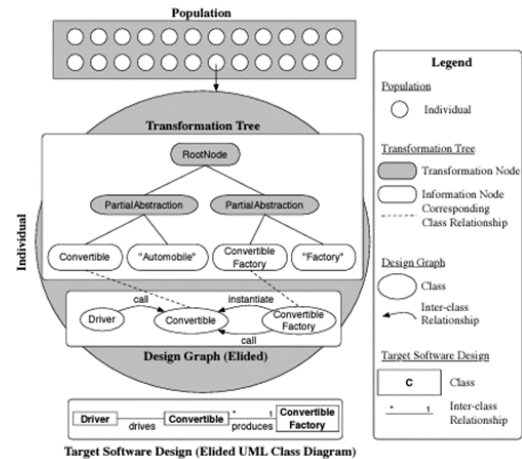


Fig. 8. Relationships Among GP Elements [2]

*b) Results:* The paper presents the results of four experiments and a case study on the effectiveness of the Remodel tool in introducing design patterns in software designs. The experiments involved applying Remodel to a set of published software designs and determining optimal coefficient values for the fitness function. The case study applied REMODEL to a large Web-based software system and evaluated its ability to construct design pattern instances in the presence of existing instances. The results showed that REMODEL can introduce design patterns in a diverse set of software designs and can construct instances of design patterns in the presence of existing instances. The average number of transformation nodes in the best individuals in each run was 4.50. Also, an average of 12 new design patterns evolved throughout the process. Hence, REMODEL is capable of simultaneously improving software quality metrics and introducing design pattern instances.

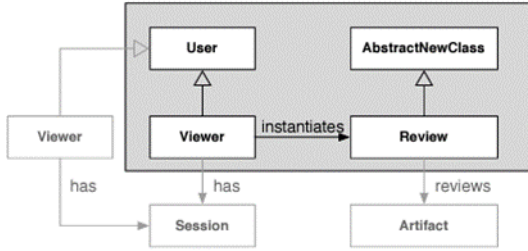


Fig. 9. Evolved Design Pattern Instance [2]

Pattern	# of Instances
Abstract Factory	2
Adapter	13
Bridge	3
Composite	0
Decorator	2
Prototype	26
Proxy	15

Fig. 10. Evolved Pattern Instances in Case Study [2]

*c) Takeaway:* To conclude, this paper introduces an automated refactoring approach called REMODEL that combines genetic programming, software engineering metrics, and mini transformations to introduce design patterns in existing software designs. The approach can simultaneously improve the quality of a software design with respect to metrics as well as automatically introduce design pattern instances. Ongoing research aims to explore the impact of different metrics, domain-specific design patterns, and design change mechanisms with different levels of abstraction on different refactoring problems.

#### D. Refactoring Fat Interface Using Genetic Algorithm

The objective of this paper was to produce a genetic algorithm that would outperform random refactorings of a set of interfaces using the metric of IUC or “Interface Usage Cohesion”, which measures the average proportion of an interface that is used by an implementing class. A “Fat Interface” is defined in other source papers cited by Romano et

al., as a bloated interface with too many functions. Fat Interfaces can and often violate the Interface Segregation Principle (ISP) which says that clients should not be forced to depend upon interface methods that they do not actually invoke. Fat Interfaces are linked to large numbers of changes, which causes problems to clients using an interface changing frequently.

$$IUC(i) = \frac{\sum_{j=1}^n \frac{used\_methods(j,i)}{num\_methods(i)}}{n}$$

Fig. 11. Interface Usage Cohesion (IUC) calculation [3]

*a) Methodology:* The refactoring technique involves an API split into many sub-APIs each with a subset of the methods from the fat interface. Therefore, each client has a new API where they implement a higher proportion of the methods from their interface. The result is straightforward that they depend on a smaller interface, and actually use more (percentage-wise) of the thing they depend on. The number of sub-APIs must also be minimized, to avoid the situation where each client has its own unique API. In this hypothetical, each implementing class has the exact interface that matches what they need. The downside in this situation is that there are too many interfaces in the system, and the likelihood of sharing interfaces is extremely low.

The basics of genetic algorithms are fundamentally covered within this paper, but for this experiment the steps were as follows. The Genetic Algorithm starts with the current set of interfaces, and their IUC values are calculated. The next step is to create more solutions using evolutionary operators: selection, crossover, and mutation in that order. First, we pair up sets of solutions to perform a crossover on. Then the results of the crossover are randomly mutated at different points to provide a new set of solutions. These new solutions are run through the IUC calculation again, and solutions that have higher values are considered to be an improvement. This process can be repeated iteratively. Each new set of solutions is considered a “generation”.

The authors compared a genetic algorithm to two other methods for control. The first being completely random mutations and the second being a variation of local search called “Simulated Multi-Objective Annealing”. The annealing method takes the same form as the mutations in the genetic algorithm to maintain consistency. Solutions are randomly mutated from a single starting solution and the IUC values are re-calculated. If the solution is an improvement, then that change is accepted. For the random approach it is as it seems, randomly generated solutions are repeated the same number of iterations as the other two methods and the solution with the best IUC is chosen at the end.

*b) Results:* The results show that a genetic approach to fat interfaces outperforms both other approaches in a statistically-significant manner using both Mann-Whitney p-value and Cliff Delta metrics. The results also show that as the number of methods, invocations, and clients increase, the degree to which the genetic approach outperforms increases in most cases (the



exception is the number of methods only showed this correlation when comparing genetic to random).

c) *Takeaway*: The conclusion is that using genetic algorithms to solve the issue of Fat Interfaces was decently successful on the sample of Java projects analyzed. The next step for the authors would be showing code base owners how to integrate something like this. Automated changes to a codebase, even when increasing a metric that seems as good as IUC, decreases understandability (at least initially) among the developers, because it introduces changes that no one wrote into the code. However having simpler, more coherent interfaces will lead to more useful and resilient APIs overall.

### III. HEURISTIC ALGORITHMS

#### A. Fundamentals Overview

Heuristic algorithms are a collection of search algorithms which possess additional information to find solutions more efficiently than uniformed algorithms. This additional information is represented as,  $h(n)$ , which underestimates the cost from the current state to the solution or goal. Some of the common uniformed algorithms are Depth-First, Breadth-First and Dijkstra search, only consider,  $g(n)$ , which represents the total cost from the start or root to the current state. Some of the common heuristic algorithms are Greedy Best First, Memory Bounded Heuristic, IDA \*, Recursive Best First and A \* search, which consider both  $h(n)$  and  $g(n)$ , to a composition of  $f(n) = g(n) + h(n)$ .

#### B. Detecting Occurrences Of Refactoring With Heuristic Search

In “Detecting Occurrences of Refactoring with Heuristic Search”, the premise is finding a compound sequence of refactoring operations used to transform a software system from one version to another. Within the research paper it is acknowledged that software is constantly evolving to adapt to changing dynamics and requirements. The main contention addressed is developers must understand behavioral changes of a modified software systems to ensure reliability, understandability, functionality, and robustness. Exacerbating matters of understandability is the case that only an unsubstantial number of modifications are documented or archived, leading to impure refactoring [4].

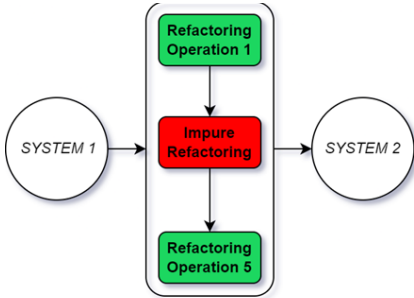


Fig. 12. Impure Refactoring Illustration, certain refactoring operations history is lost or not archived or saved.

a) *Methodology*: To reconstruct the chain an exponential number of potential refactoring operations combinations would

exist, and a heuristic search algorithm, A\*, is employed to find the optimal refactoring sequence combination. The refactoring occurrence detection technique is decomposed of a difference calculation, expansion, refactoring operator selection from a priority queue and performing the selected refactoring operation within graph search space. The graph search space structure comprises of refactoring operators,  $O$ , along the transitional edges and nodes with  $n_0 = P_{old}$  marking the old version of the system and  $n_m = P_{new}$  representing the current version [4]. A\* traversal’s evaluation through the graph search is based on,  $f(n) = g(n) + h'(n)/\alpha(o)$ , with  $g(n)$  representing the total number of performed refactorings from  $n_0$  to the current state.

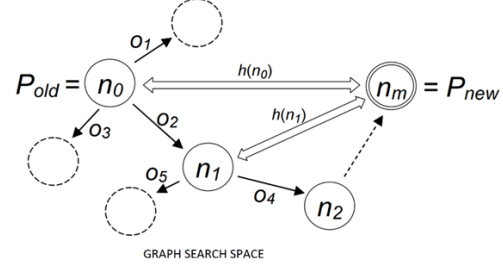


Fig. 13. Graph Search Space containing nodes are program versions, and operators are transitions to those states [4].

The difference calculation portion will formulate a heuristic distance,  $h'(n)$ , through finding structural differences among the Abstract Syntax Trees (AST) of two different versions of the software system. The specified operators modifying the AST will be represented into edit scripts composed into add (node, position), remove (node, position), change (before, after, position), and move (node, from, to) [4]. For the expansion portion of the occurrence detection technique, it will utilize the difference calculation to identify potential refactoring operation candidates for a given state within the search space. The refactoring operators will be arranged based on  $\alpha(O)$ .  $\alpha(O)$ , as described by researchers, which represents the likelihood of a refactoring operation to be performed which, “is determined by the rate of satisfied refactoring sequences” [4]. Therefore, an operator with a lower likelihood will have a lower priority reducing its chances of being selected.

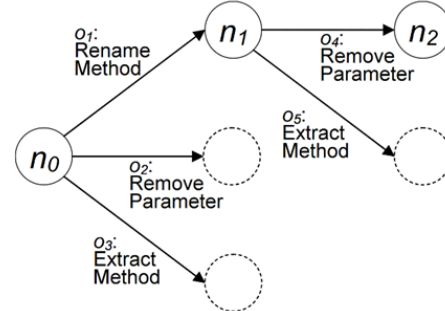


Fig. 14. A\* Search Tree Overview Result [4]

b) *Results*: The researchers’ techniques were applied to a single case involving two different versions of an archived project’s class. The refactoring modifications history to a method, calculateDistance (), was able to be traced to two

refactoring operators: rename-method, and remove-parameters. The procedure to find the two operators is illustrated with the search tree and functionality table. The functionality table's columns are structured with the numbers of iterations, structural differences calculation, and potential refactoring operators arranged within the queue based on the evaluation function. From this in the first iteration rename-method was invoked since its evaluation was highest, followed by remove-parameter in the second iteration.

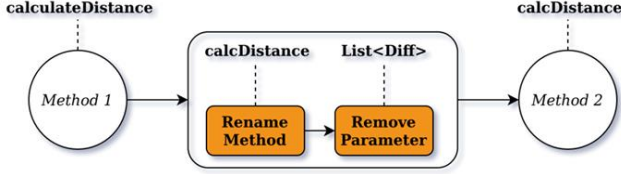


Fig. 15. calculateDistance Transformation to calcDistance.

Table 1. Experimental results: differences, derived refactorings, and their evaluations

th	Differences	w	h'	Operators in the queue	$\alpha$	h	f
1	$d_1$ : change( $C, m_1, m_2$ )	1	4	$o_1$ : Rename Method( $C, m_1, m_2$ )	1	4	4
	$d_2$ : remove( $C, m_1, p$ )	1		$o_2$ : Remove Parameter( $C, m_1, p$ )	1	4	4
	$d_3$ : remove( $C, m_1, block$ )	1		$o_3$ : Extract Method( $C, m_1, block$ )	0.5	8	8
2	$d_4$ : remove( $C, m_2, p$ )	2	3	$o_4$ : Remove Parameter( $C, m_2, p$ )	1	3	4
	$d_5$ : remove( $C, m_2, block$ )	2		$o_5$ : Extract Method( $C, m_2, block$ )	0.5	6	7
	$d_6$ : Extract Method( $C, m_1, block$ )	2		$o_6$ : Extract Method( $C, m_1, block$ )	0.5	8	8
3	-	-	0				

$C$ : DistanceCalculator,  $m_1$ : calculateDistance,  $m_2$ : calcDistance,  $p$ : The parameter List<Diff>

Fig. 16. In depth table showcasing priority queue, operators differences calculation, along with number of iterations [4].

c) *Takeaway*: From research conducted, traceability of performed or impure refactoring is possible, allowing for software systems to be understandable by developers or engineers based on historical information.

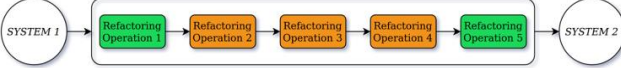


Fig. 17. Impure refactoring operations retrieval.

### C. Detecting Model Refactoring Opportunities With Heuristic Search

In this paper, the work is based on detecting and correcting design defects within Model-Driven Engineering (MDE) and proposes an automated approach to detect the defects in the design process. Model-Driven Engineering (MDE) is an approach to design, implement and deploy software development through a series of models. While implementing the Model-driven Engineering (MDE) we must keep some things on our mind like building appropriate models, evolving them and maintaining the model quality. Among them model maintenance is different as the model is improved by adding new functionalities. Generally effort, time and money are required for the total project. So, this paper proposes an approach to be more efficient and cost effective in improving the model quality.

The author proposes an automated approach to detect the refactoring opportunities related to design defects in models, using Genetic Programming to automatically generate rules that detect defects. In this approach the main aim is to relieve software designers from these manual tasks of detecting defects.

The authors evaluated their approach on two large class diagrams and successfully detected the majority of defects in the sample class diagrams.

The author defines two important concepts that are relevant to the proposed automated approach: Design Defects and Quality metrics. Design defects are situations in which a software design has a negative impact in its development. The author focuses on defects that appeared at the model level and in class diagrams, which are Blob, Functional Decomposition, and Poor Usage of Abstract Class. Quality metrics are a measurement that helps assess the quality of the software such as evolvability and reusability, and focuses on calculation on class diagrams such as the number of associations, aggregations, dependencies, generalizations and aggregation hierarchies. The Genetic Programming algorithm explores the search space of detection rules to find a set of rules that detects the defects in the examples, such as using a combination of selection, mutation and crossover operators.

a) *Methodology*: The methodology of this paper is that the author examined 2 open-source Java projects (GanttProject and LOG4J) and analyzed their libraries to find specific defects like Poor Usage of Abstract Class, Blob and Functional Decomposition. After using the Heuristic search technique, the defects are detected automatically. From this experiment, we can conclude that the technique was able to find design defects with good precision and recall score. The classes that are mentioned in the diagram are the compared classes between the 2 projects GanttProject and LOG4J are shown in Fig. 18, the 'x' indicates that the defects are found in the classes.

b) *Results*: The proposed technique appears to be effective in detecting defects in software designs. However, there is one limitation. That is it is time-consuming as the tasks are manually inspected. Since the rules are generated by using heuristic search techniques the results may vary. Finally, the execution time is scalable, and it may depend on the number of metrics used and the size of the class diagrams.

c) *Takeaway*: The conclusion is that by using Genetic Programming we generate defect detection rules based on examples of design defects. The evaluation results show that the approach was successful in detecting various types of design defects in large models.

System	Precision	Recall
GanttProject	Blob : 100%	100%
	PC : 83%	91%
	FD : 91%	94%
LOG4J	Blob : 87%	90%
	PC : 84%	82%
	FD : 66%	74%

Fig. 18. Detection results[5].

Class	F.D.	Blob	P.C
GanttGraphicArea		x	X
GraphicPrimitiveContainer			
GanttOptions		x	
ResourceLoadGraphicArea			X
GanttGanttCaver			X
GanttDialogPerson			X
NewProjectWizard	X	x	X
GanttTree		x	
GanttProject		x	
TimeUnitGraph	X		
GregorianTimeUnitStack	X		X
TipsDialog			X
RecalculateTaskCompletionPercentageAlgorithm	X		
TaskHierarchyManagerImpl	X		X

Fig. 19. Results obtained from the GanttProject[5].

#### IV. SYNTHESIS OF CORE PAPERS

The research initiative conducted within [1], [2], [3], [4], and [5], was to utilize genetic and heuristic refactoring techniques to encapsulate refactoring complexities granting software maintainability and understandability. Along with formulating design pattern instances to improving software quality as demonstrated in [1], [2], [3] and [5]. The studies conducted within the papers also focused on improving code smells detection and appropriately seeking relevant refactoring operations or techniques through defined metrics, to objectively limit accidental complexities.

*a) Effectiveness:* In our analysis, we have seen that using genetic and heuristic algorithms can be very effective in proposing refactorings in the cases we looked at. These include refactoring fat interfaces, component-based bad smells, class diagrams, design patterns, and method-based bad smells. The common methodology in proposing an algorithm is to define a metric or set of metrics that are important and propose an algorithm to maximize the improvement to those metrics. A key part of the process here is identifying whether or not the metrics we have chosen are important, and later having a way to check if the refactoring process had a positive effect. Of course, the metrics are improved, as that was the entire goal, but as we have learned in our class reducing bad smells is not always guaranteed to improve the software quality.

Something to consider here is that the authors of papers like these are selecting a metric themselves, and then building the problem space around that metric. Authors may start with an algorithm in mind, or in other words they may have a solution before they have a problem. The process of selecting when to apply these types of refactorings must be considered, because modifying a system automatically without manual review may not be the best approach in some cases. We can apply algorithmic refactoring to many pieces of code, but the more important factor (assuming algorithmic refactoring becomes common) is to decide *when* to apply and accept the results of this type of refactoring.

*b) Applicable Use Cases:* The research papers did not divulge into practically applications use cases, as their implications were within theoretical and empirical inquiries. Open-source software are widely used within society, to maintain and expand their capabilities numerous commits and reviews occur frequently to adapt to changing dynamics. We identified that the refactoring heuristics techniques specified in [4], can be applied to version control where traceability is important in ensuring management of software systems. To the contributions of the software systems, they can understand how

a conclusion was reached through analyzing sequence of refactoring operations changes. Computing environments are diverse and expanse, for instance currently there are plentiful distributed, operating systems and architectures. Portability is a concern inflicting numerous software systems, however we find that the genetic refactoring concepts and techniques could be applicable to identifying and optimizing software components in adapting to the host computing environments. This would alleviate development effort to maintaining portability, while maintaining a consistent and concise external behavior across environments.

#### V. OVERVIEW OF CURRENT STATE

The current state of the art for genetic algorithm/programming in software refactoring involves using these approaches to automate the improvement of software design, based on various design metrics and objectives. These approaches have shown promising results in improving software design and reducing technical debt in software systems. One approach is to use genetic programming to generate a sequence of refactorings that improve the design of a software system. This involves defining a set of candidate refactorings and using genetic programming to search for a sequence of refactorings that improves a fitness function. The fitness function measures the quality of the software system, based on various design metrics. Another approach is to use genetic algorithms to search for the optimal values of coefficients in a fitness function used for software refactoring. One last approach is to use multi-objective genetic algorithms to optimize multiple conflicting objectives in software refactoring. This involves defining multiple fitness functions that measure different aspects of the software design, such as cohesion, coupling, and complexity, and using a multi-objective genetic algorithm to search for a set of solutions that optimize all the objectives simultaneously.

#### VI. FUTURE DIRECTIONS

In terms of future directions, we perceive improvements to automated software refactoring using genetic and heuristic algorithms to include incorporating more domain specific knowledge within use cases and architectures. Improvements within fitness or evaluations functions to yield better refactoring outcomes would be an interesting area of research such as incorporating machine learning techniques. The manner along with performance of automated refactoring through genetic or heuristic techniques, in large distributed or dynamic computing systems, would be an interesting and potentially inevitable exploration considering the usage of internet applications and their services.

#### REFERENCES

- [1] Kebir, S., Borne, I., & Meslati, D. (2017). A genetic algorithm-based approach for automated refactoring of component-based software. *Information and Software Technology*, 88, 17-36G.
- [2] Jensen, A. C., & Cheng, B. H. (2010, July). On the use of genetic programming for automated refactoring and the introduction of design patterns. In *Proceedings of the 12th annual conference on Genetic and evolutionary computation* (pp. 1341-1348).
- [3] Romano, D., Raemaekers, S., & Pinzger, M. (2014, September). Refactoring fat interfaces using a genetic algorithm. In *2014 IEEE International Conference on Software Maintenance and Evolution* (pp. 351-360). IEEE.

- [4] Hayashi, S., Tsuda, Y., & Saeki, M. (2008, December). Detecting occurrences of refactoring with heuristic search. In 2008 15th Asia-Pacific Software Engineering Conference (pp. 453- 460).IEEE.
- [5] Ghannem, A., Kessentini, M., & El Boussaidi, G. (2011, November). Detecting model refactoring opportunities using heuristic search. In Proceedings of the 2011 Conference of the Center for Advanced Studies on Collaborative Research (pp. 175-187).
- [6] Russell, S. J., & Norvig, P. (2022). 4.1 Local Search Algorithms And Optimization Problems. In Artificial Intelligence A modern approach (3rd ed., pp. 139–148). essay, Pearson Education Limited.