Final Project Report for Software Architectures

Studied Paper:

DEPICTER: A Design Principle Guided and Heuristic Rule Constrained Software Refactoring Approach

Group G2

Group Members:

Danial Bazmandeh, 97521135 danibazi9@gmail.com
Parisa Alaie, 401723283 amy78rose@gmail.com
Fatemeh Ranjbar, 401723148 fatran95@gmail.com

Table of Contents

Abstract

As we know, the more suitable design patterns a software uses, the higher its scalability, readability and maintainability. The quality of software design is affected by the software evolution cycle. This may be due to the addition of new features or possible future bugs. This made the developers turn to **code refactoring**.

Software Refactoring means improving the internal structure and architecture of the software, without changing its external performance and functions, which leads to improving the maintainability of the software. But the operation of automatic code refactoring is a very challenging task. Because it requires a comprehensive view of the entire software system. For this purpose, recent studies introduced search-based algorithms to facilitate software reconstruction. However, they still have the following major limitations:

1. Searched solutions may violate the design principles, because their Fitness Functions do not directly reflect and measure the degree of compliance of the software with the design principles.
2. Most approaches start the search process from a completely random initial population, which may lead to suboptimal solutions.

In this paper, we aim to develop an effective search-based refactoring approach to recommend better refactoring activities for developers, which can improve the degree of software conformance to design principles as well as the quality of software design.

In this research, we investigated and implemented DEPICTER. To develop and increase the population, we have used the genetic algorithm NSGA II, which uses metrics and design patterns as fitness functions. In addition, DEPICTER uses heuristic rules to improve the quality of the initial population for subsequent general evolutions.

Therefore, the main parts of this project, which have been implemented, include the following three parts:

1. Implementation of genetic algorithm NSGA II
2. Implementation of the heuristic functions
3. Implementation of automated refactoring operation

We used Java language to implement the genetic algorithm. For this purpose, we defined the initial population and population evolution using Fitness Functions. It should be mentioned that because the implementation of fitness functions for our paper was a very difficult process (because we had to find a numerical measure to calculate the amount of dependence between classes and methods) and in coordination with the professor, who left us open for implementation, we used the fit functions for another paper that used this algorithm similar to our paper.

For the heuristic functions part, with the confirmation we got from the relevant TA, we decided to only calculate the functions and there was no need to consider pre-conditions or post-conditions. For this purpose, we implemented three heuristic functions for each refactoring of Merge Package and Move Method functions.

In the implementation of automated refactoring, we used the widely used ANTLR library in Python and did our work by inheriting the reference Listener class for the Java language grammar. Then we used its walker to parse the code and when entering or leaving the rules that we needed, we did the refactoring process and wrote and replaced the tokens using token_stream_rewriter.

Highlights

- Implementation of the genetic algorithm NSGA II with a focus on the non-random generation of the initial population
- Considering metrics of design patterns as fitness functions
- Using heuristic functions to check if some automated refactoring is feasible or not
- Using the widely used ANTLR module to implement automatic code refactoring (automated refactoring)

Contributions

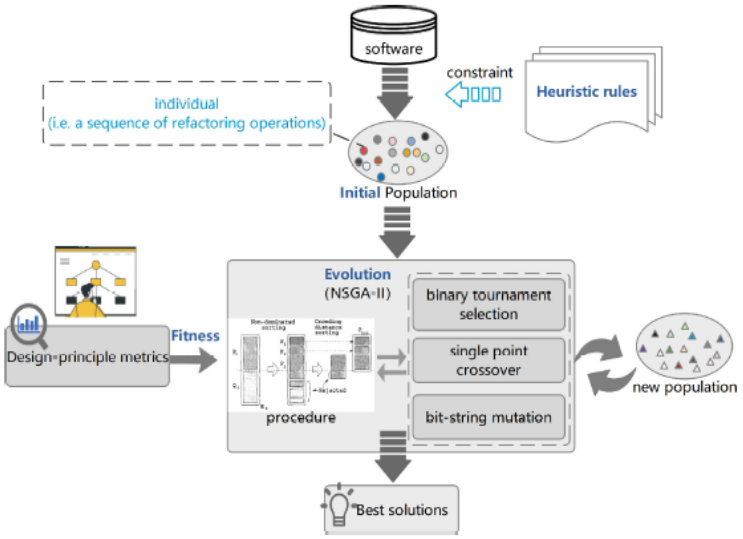
The goals that motivated the implementation of this project are as follows:

1. Previous solutions may violate the design principles, because their fitness functions do not directly reflect and measure the degree of compliance of the software with the design principles.

2. Most approaches start the search process from a completely random initial population, which may lead to suboptimal solutions. Therefore, the accidental generation of the population should be avoided.

Design

The image below shows the overall design of this project. As stated earlier, the purpose of implementing the DEPICTER algorithm is to use the metrics of design patterns as fitness functions and use the limits of heuristic functions to limit and not prevent the random generation of the initial population .



The main parts of the architecture are as follows:

1) Heuristic functions: a set of rules that give us an idea for the initial population generation.

2) Production of the initial population: production of a set of Individuals

3) Fitness functions: using design patterns for fitness functions, so that the compliance of the software with the design patterns can be quantitatively measured.

4) Population evolution using genetic algorithm: using NSGA II algorithm for population evolution and creating a new population using Mutation and Crossover

Implementation

Regarding the relevant paper codes, it is important to mention that, unlike most groups that had paper or dataset codes, we could not find any sources. Even though we e-mailed the authors of the paper several times, we did not receive a response, so we wrote all the codes ourselves from scratch, and the collection of all the codes is located in the GitHub repository at the following address:

https://github.com/Parisa78/AS_G2_Depicter/

Project structure

We have implemented 4 parts of heuristic, refactoring, NSGA and fitness separately. The programming languages used for these 4 sections are as follows:

- Heuristic: part Java and part Python
- Refactoring section: Python
- nsga and fitness section: Java
- Heuristic section: Java

The heuristics folder: contains the merge package heuristic code and the heuristic move method file. The move method file contains heuristic test codes in Java language, final heuristic codes in Java language, and also a Python version implemented for this heuristic.

The refactoring folder: contains the codes related to the automatic refactoring of move method and move class, as well as codes to obtain all the classes and methods of a package.

grammar folder: Java language grammar is located in this folder.

gen folder: Lexer, Parser, Listener, Visitor and all files generated using ANTLR are inside this section.

src folder: a series of files that we used to test the program are located in this section.

NSGA-II-fitness-refactoring folder: genetic algorithm is implemented in this folder.

Part I: Heuristic functions

In this section, two heuristic methods are implemented:

- Merge package:

  In this method, we extract the number of classes of each package, then we select two packages whose number of classes is less than the average number of classes of all packages. To do this, we saved all the packages that satisfy this condition in a list and returned the first two. We merge these two together: we add all the classes and code in the first package to the second package, then we delete the first package.

- Move method:

  The implementation of move method is that at first two classes like c1 and c2 are selected. The first condition is that the classes must be from the same package. One of the conditions for choosing class c1 is that the number of methods in class c1 should not be less than 2. We randomly select a method from methods in c1 to move to c2 that have the following condition: The dependencies of the said method with the methods of class c1 should be less than its dependencies with the methods of class c2.

  Javaparser library is used to implement this heuristic function. Obtaining the dependency between the methods of a class is done in three steps:

1. Returning all methods that are called in a class.
2. Returning the names of the methods of a class
3. Using the output of the previous two steps, the number of times a method has been called in the class is obtained.

Considering the output of the third step, we can determine which method will be transferred to the second class.

MethodDeclaration, CompilationUnit, and MethodCallExpr classes of the javaparser library have been used to implement all three of these steps.

The challenge we are facing in the heuristics section is how to get the appropriate data to carry out each refactor after the heuristic is done by the genetic algorithm? It was decided that when heuristics are run on a package, each package that is suitable for that heuristic is stored separately according to the conditions.

Part II: Automated Refactoring

The refactoring part uses the move method and move class heuristic sections, which are written in Python and to extract information from the input Java code, antlr4 tool and its Listener are used and it works correctly (it considers the packages).

Also, a Java language code of various refactorings has been written, which covers the refactoring of classes and methods, etc., and does not only include the refactoring of packages.

In the part of automatic code refactoring, we have implemented two refactorings, move method and move class.

1) Move class refactoring: In this method, by taking the source package, the destination package and the class to be transferred, we transfer the class

from one package to another package. For its preconditions and post-conditions, we must pay attention to the following:

a) The relevant class must exist in the package in question.

b) In case of class being moved, wherever we have import of that class in other classes, they must be imported from the new package from now on.

2) Move method refactoring: In this method, by taking the source class, the destination class and the method that is to be moved, we move the method from one class to another class. We must pay attention to the following preconditions and post-conditions:

a) The relevant method and source class really exist.

b) The corresponding method is defined inside the source class and not outside it.

c) There may be fields (attributes) that the corresponding method uses. In this case, if the method is transferred to another class, the corresponding field must also be transferred. Otherwise, the method will not work. For this, we will create a graph using the networkx module and find and save the methods and fields related to them.

Part III: NSGA II Algorithm

The steps of this algorithm are as follows: at first, the source code of the relevant software and heuristic functions are taken as input. An initial population is then attempted to be generated and evaluated by focusing on fitness functions that are metrics of design patterns. Then, the population is propagated using crossover or mutation. Then the suitable population is selected and transferred to the next generation.

In the genetic algorithm, the chromosomes of each gene is one of the refactorings. To perform the mutation operation, one of the chromosomes is randomly selected and replaced by another refactor operation .

In order for the crossover operation not to face problems, we consider a certain number for the number of chromosomes. The questions that were raised for this section:

1. Mutation and crossover operations may disrupt the refactoring process, in this case, what should be done?
2. Should the new generation be chosen with memory or without memory?

Answer to question 1: If during the refactoring operation we reach one of the refactorings that cannot be implemented due to mutation or crossover or applying previous refactors, we skip that refactoring. (According to the guidelines, we can act in such a way that if more than 15% of one of the genes is not applicable, it will be deleted.)

Answer to the second question: The selection of the new generation is done by memory. In this way, after producing the children and measuring their fitness, the children are combined with the parents, and after sorting according to their fitness, the best ones are selected for the new generation.

In this part, we used a different dataset that included all 4 datasets in block form.

The implementation of this part has been done in MOOptimization class and in Java language.

Fitness

In this part, according to the steps we have taken and the explanations given by the teacher in the class, we decided to use different fitnesses for the evaluation. So we read different articles and among them we used the following metrics for fitness:

Design property	Metric	Description
Design size	DSC	Design size in classes
Complexity	NOM	Number of methods
Coupling	DCC	Direct class coupling
Polymorphism	NOP	Number of polymorphic methods
Hierarchies	NOH	Number of hierarchies
Cohesion	CAM	Cohesion among methods in class
Abstraction	ANA	Average number of ancestors
Encapsulation	DAM	Data access metric
Composition	MOA	Measure of aggregation
Inheritance	MFA	Measure of functional abstraction
Messaging	CIS	Class interface size

To implement fitness, we used the formulas used in the article of group 9. which are as follows:

Quality attribute	Definition Computation
Reusability	A design with low coupling and high cohesion is easily reused by other designs. $0.25 \times \text{Coupling} + 0.25 \times \text{Cohesion} + 0.5 \times \text{Messaging} + 0.5 \times \text{Design size}$
Flexibility	The degree of allowance of changes in the design $0.25 \times \text{Encapsulation} - 0.25 \times \text{Coupling} + 0.5 \times \text{Composition} + 0.5 \times \text{Polymorphism}$
Understandability	The degree of understanding and the easiness of learning the design implementation details. $0.33 \times \text{Abstraction} + 0.33 \times \text{Encapsulation} - 0.33 \times \text{Coupling} + 0.33 \times \text{Cohesion} - 0.33 \times \text{Polymorphism} - 0.33 \times \text{Complexity} - 0.33 \times \text{Design size}$
Functionality	Classes with given functions that are publically stated in interfaces to be used by others. $0.12 \times \text{Cohesion} + 0.22 \times \text{Polymorphism} + 0.22 \times \text{Messaging} + 0.22 \times \text{Design Size} + 0.22 \times \text{Hierarchies}$
Extendibility	Measurement of design's allowance to incorporate new functional requirements. $0.5 \times \text{Abstraction} - 0.5 \times \text{Coupling} + 0.5 \times \text{Inheritance} + 0.5 \times \text{Polymorphism}$
Effectiveness	Design efficiency in fulfilling the required functionality. $0.2 \times \text{Abstarction} + 0.2 \times \text{Encapsulation} + 0.2 \times \text{Composition} + 0.2 \times \text{Inheritance} + 0.2 \times \text{Polymorphism}$

The reason for not using the fitnesses that were in the paper is that the requested fitnesses were not reasonable and could not be implemented. Because they did not explain the numerical criteria for measuring dependency. We used Java language to implement this section. This implementation has progressed with the help of Group 9.

A brief description of the entered codes for this part:

In the Metrics section, we implemented some of the metrics stated in the evaluation section. We called these implementations in the Metrics class.

Finally, in the solution class, we obtained the considered fitnesses.

Evaluation

Because this research did not have ready sources and we implemented everything ourselves. We decided to move forward with this project in several parts and to implement a few limited examples for different and diverse parts of each part. This research includes 1. heuristics 2. refactoring 3. nsga-ii and 4. fitness.

Each of these 4 parts work separately and return a result to us, but because each part had its own requirements, we used different languages and different techniques to implement them. These projects are still not connected and we don't have results for the whole project.

- Heuristics:

    a. merge package: First, we extracted the classes of each package:

```
C:\Users\Alaie\AppData\Local\Programs\Python\Python38\python.exe G:/DEPICTER/main.py
biz.ganttproject.core => 109
biz.ganttproject.core.calendar => 9
biz.ganttproject.core.calendar.walker => 2
biz.ganttproject.core.chart.canvas => 13
biz.ganttproject.core.chart.grid => 11
biz.ganttproject.core.chart.render => 17
biz.ganttproject.core.chart.scene => 13
biz.ganttproject.core.chart.scene.gantt => 9
biz.ganttproject.core.chart.text => 8
biz.ganttproject.core.model.task => 1
biz.ganttproject.core.option => 18
biz.ganttproject.core.table => 2
biz.ganttproject.core.time => 16
biz.ganttproject.core.time.impl => 5
org.w3c.util => 2
```

Then we made a list of classes that contained the described rule. The rule was to select packages whose number of classes is less than the average number of classes of all packages.

[['biz.ganttproject.core.calendar', 9],

['biz.ganttproject.core.calendar.walker', 2],

['biz.ganttproject.core.chart.canvas', 13],

['biz.ganttproject.core.chart.grid', 11],

['biz.ganttproject.core.chart.scene', 13],

['biz.ganttproject.core.chart.scene.gantt', 9],

['biz.ganttproject.core.chart.text', 8],

['biz.ganttproject.core.model.task', 1],

['biz.ganttproject.core.table', 2],

['biz.ganttproject.core.time.impl', 5],

['org.w3c.util', 2]]

Finally, we chose two classes:

```
Less than average p1 & p2:
[['biz.ganttproject.core.calendar', 9], ['biz.ganttproject.core.calendar.walker', 2]]
```

b. Move method:

The implementation method of move method is that two classes like c1 and c2 are selected at first. The first condition is that the number of classes in the package is not less than two. The condition for choosing class c1 is that the number of methods in class c1 should not be less than 2. We randomly select a method from methods that have conditions in c1. The relation between the methods of class c1 should be less than the relation between the methods of class c2.

Javaparser library is used to implement this heuristic. Obtaining the relationship between the methods of a class is done in three steps:

b. Returning all the fields that exist in a class.

c. Returning the fields of a method.

d. Using the output of the previous two steps, we get the number of times the field of a method has been used in the class.

Considering the output of the third step, we can determine which method will be transferred to the second class.To implement all three of these steps, FieldDeclaration, CompilationUnit, MethodCallExpr, and MethodDeclaration classes from the javaparser library have been used.

The challenge we are facing in the heuristics section is how to get the appropriate data to carry out each refactor after the heuristic is done by the genetic algorithm? It was decided that when the heuristics were run on the package, each package would be stored separately according to the conditions for which the heuristic is suitable.

- Refactoring: With a quick look at the images below, you can see that the refactoring has been implemented correctly.

  1) Move method: As an example, we have moved the printG class from class A to class B.

  Class A image before refactoring process:

```
/* Before refactoring (Original version) */
public class A
{
    public int f; /* printF , printF, */
    public string g; /* printF , printG, */
    public string h; /* printH */

    // Method 1
    void printF(int i)
    {
        this.f = i * this.f;
    }

    // Method 2
    void printF(float i){
        this.f = (int) (i * this.f);
        this.g = (int) (i * this.g);
    }

    // Method 3
    void printG(){
        print(this.g);
    }

    // Method 4
    void printH(){
        print(this.h);
    }
}
```

Class A and B image after refactoring process:

As can be seen, in addition to the printG method, the g field has also been moved to make the class B meaningful.

```
public class A
{
    public int f; /* printF , printF, */
    public string h; /* printH */

    // Method 1
    void printF(int i)
    {
        this.f = i * this.f;
    }

    // Method 2
    void printF(float i){
        this.f = (int) (i * this.f);
        this.g = (int) (i * this.g);
    }

    // Method 3


    // Method 4
    void printH(){
        print(this.h);
    }
}

class B {
    public string g;


    // Method moved to class B
    void printG(){
        print(this.g);
    }
}
```

2) Move class: As an example, we have moved class A from package a.aa to package c.ccc.

Class A image before refactoring:

```
package a.aa;
import static a.aa.A;
import vddf.dfdf.A;

class A
{
    class B {
    }

    public int f, c, a; /* printF , printF, */
    public int g; /* printF, printG */
    public string h; /* printH */

    // Method 1
    void printF(int i)
    {
        this.f = i * this.f;
    }

    // Method 2
    void printF(float i){
        this.f = (int) (i * this.f);
        this.g = (int) (i * this.g);
    }

    // Method 3
    void printG(){
        print(this.g);
    }

    // Method 4
    void printH(){
        print(this.h);
    }
}
```

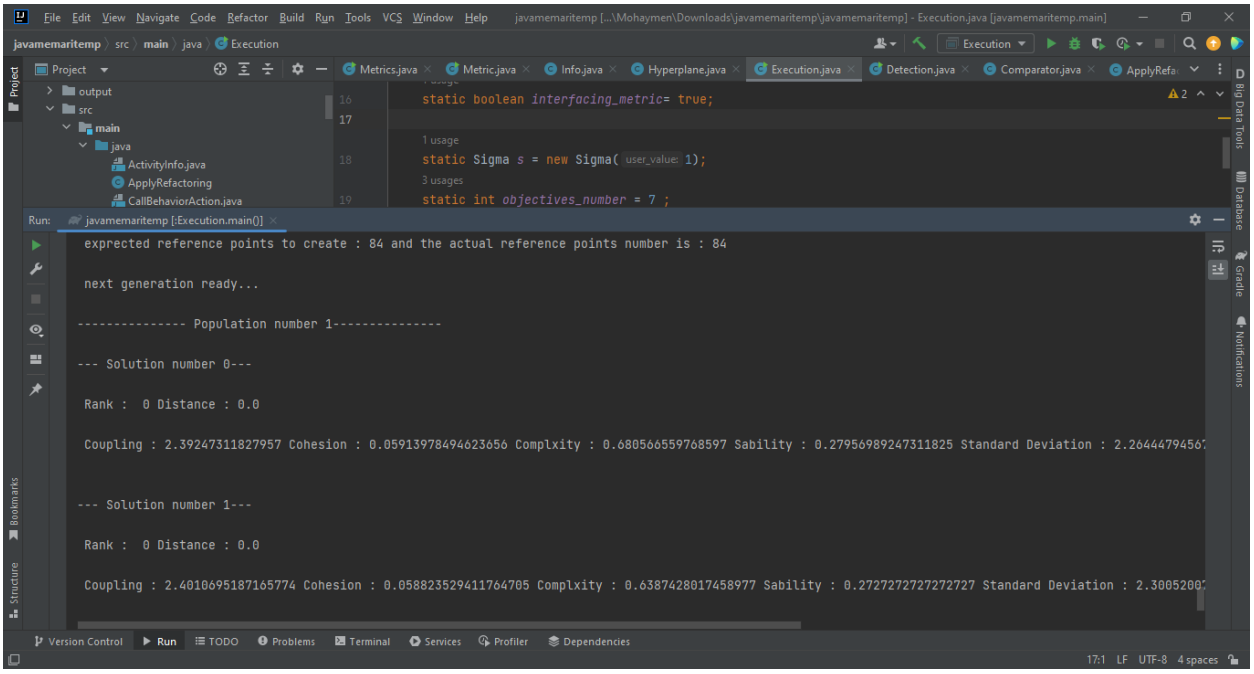
Class A and B image after refactoring:

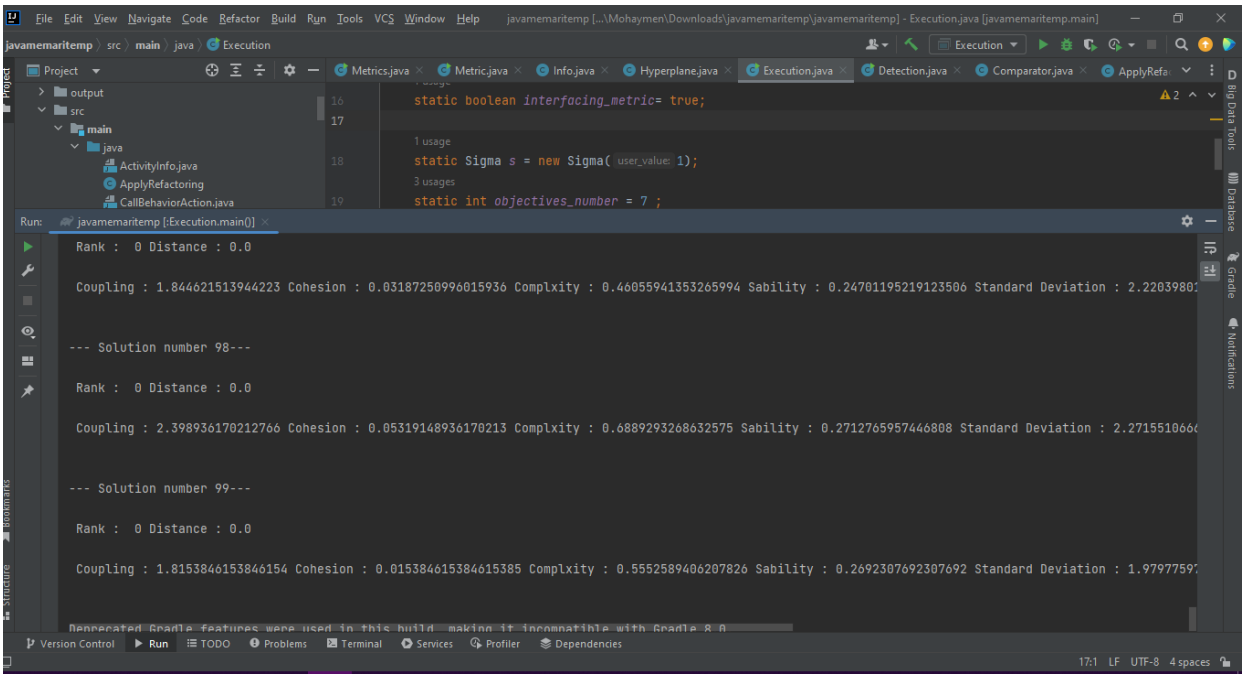
```
package c;

// Class "A" moved here from package a.aa
class A
{
    public int f, c, a; /* printF , printF, */
    public int g; /* printF, printG */
    public string h; /* printH */

    // Method 1
    void printF(int i)
    {
        this.f = i * this.f;
    }

    // Method 2
    void printF(float i){
        this.f = (int) (i * this.f);
        this.g = (int) (i * this.g);
    }

    // Method 3
    void printG(){
        print(this.g);
    }

    // Method 4
    void printH(){
        print(this.h);
    }
}
```

```
package c;

// Class "B" moved here from package a.aa
class B {

}
```

- Fitness:

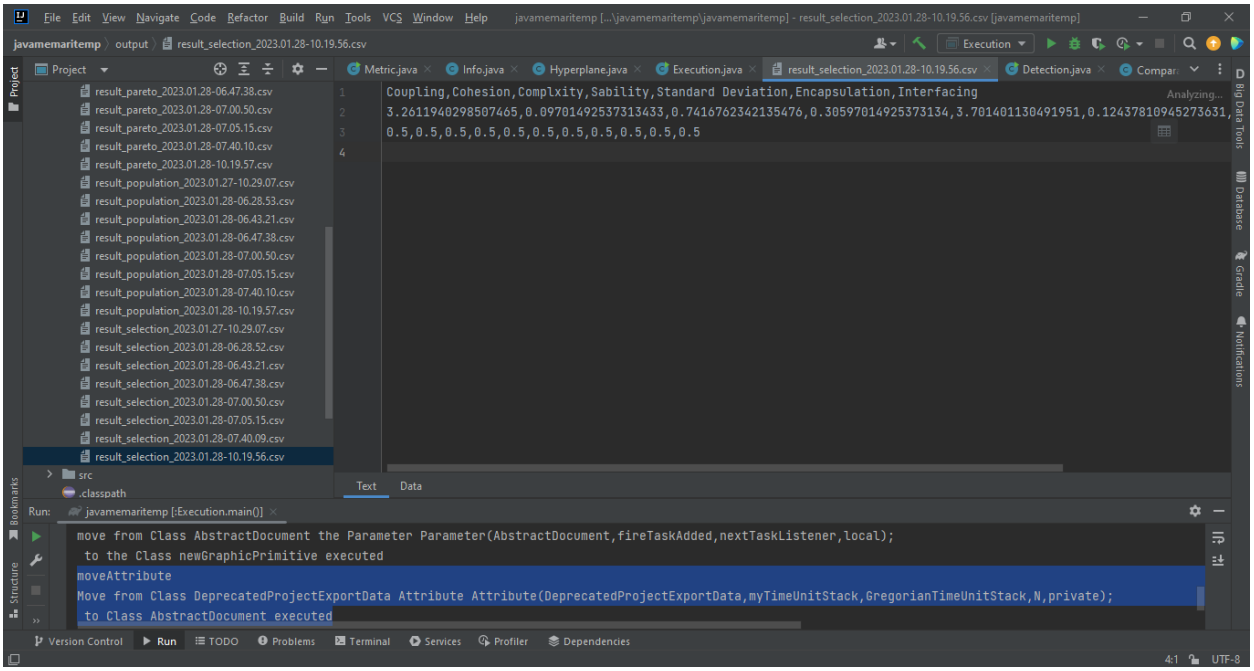
The results obtained in this section:



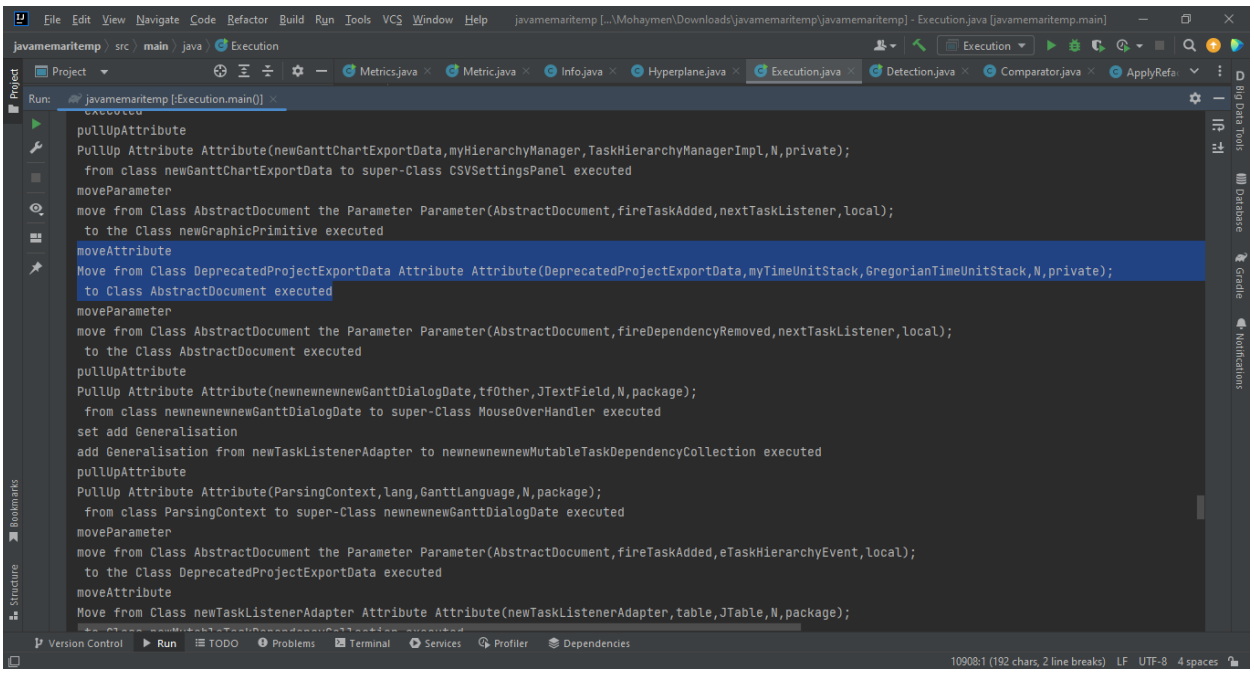


We have printed the different solutions that have been obtained and analyzed the results of each one.

Finally, we saved the best in a csv file.



Also, in this code, refactorings are also shown in the results:



For example, in 10 randomly selected blocks, nearly 100 or even more refactorings were done, and we checked the results and saw that the results were applied well (only the empty files were not deleted).

Division of labor

Division of work and contributions of people in each part of the projects:

- Parisa Alaei: refactoring, heuristics (merge package), nsga_ii, fitness, coordinating team members and project management
- Danial Bazmande: refactorings of move method and move class, work on codart, heuristics (merge package)
- Fateme Ranjbar: NSGA II algorithm, heuristic move method

Problems and challenges

We encountered many problems for implementation and because we could not find a source on the Internet to help us with part of the code. We emailed the authors of the project to send us the source code of the project or help us to write the code, but we did not receive a response.

In the next step, we got help from Mr. Zakari's CodART (we also faced many challenges to run and adjust its configs).

Despite many problems, we were able to implement the codes by ourselves without reading sources that could help us to implement them, although for the nsga code we got help from a Java code for which there was no documentation, and understanding the code itself was a challenge.

To implement heuristics in Python, javalang, javac-parser, and javaclass libraries have been tried, but they did not succeed. One of the reasons for not using the mentioned libraries is that they are old and not updated for several years. Finally, the javaparser library for the Java language has been used to implement heuristics.

For the refactoring section, the rope library was studied.

Conclusion and future work

Considering the stages of this project and its load for implementation and comparison of this project with related projects as well as the reviews of the dataset used in this research, we had doubts about the authenticity of the results of this project.

Next, it is good to integrate these few mini-projects so that we can compare the results obtained with the main project.

Resources

[1] Y. Zhao, Y. Yang, Y. Zhou and Z. Ding, "DEPICTER: A Design-Principle Guided and Heuristic-Rule Constrained Software Refactoring Approach," in IEEE Transactions on Reliability, vol. 71, no. 2, pp. 698-715, June 2022, doi: 10.1109/TR.2022.3159851.

[2] Zakeri, M. (2022). CodART, IUST Reverse Engineering Lab: A refactoring engine with the ability to perform many-objective program transformation and optimization.

[3] G. Booch, R. A. Maksimchuk, M.W. Engle, B. J.Young, J. Connallen, and K. A. Houston, “Object-oriented analysis and design with applications,” ACM SIGSOFT Softw. Eng. Notes, vol. 33, no. 5, pp. 29–29, 2008.

[4] C. C. Venters et al., “Software sustainability: Research and practice from a software architecture viewpoint,” J. Syst. Softw., vol. 138, pp. 174–188,2018

[5] I. Candela, G. Bavota, B. Russo, and R. Oliveto, “Using cohesion and coupling for software remodularization: Is it enough?,” ACMTrans. Softw. Eng. Methodol., vol. 25, no. 3, pp. 1–28, 2016.

[6] B. S. Mitchell and S. Mancoridis, “On the automatic modularization of software systems using the bunch tool,” IEEE Trans. Softw. Eng., vol. 32, no. 3, pp. 193–208, Mar. 2006.

[7] Mansoor, U., Kessentini, M., Wimmer, M. et al. Multi-view refactoring of class and activity diagrams using a multi-objective evolutionary algorithm. Software Qual J 25, 473–501 (2017). https://doi.org/10.1007/s11219-015-9284-4