گزارش پروژه پایانی درس معماری‌های نرم‌افزار

نام کامل مقاله:

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

شماره گروه: G2

اعضای گروه:

دانیال بازمانده ۹۷۵۲۱۱۳۵	danibazi9@gmail.com
پریسا علائی ۴۰۱۷۲۳۲۸۳	amy78rose@gmail.com
فاطمه رنجبر ۴۰۱۷۲۳۱۴۸	fatran95@gmail.com

فهرست مطالب

مقدمه

همانطور که می‌دانیم، هرچه نرم‌افزاری از الگوهای طراحی مناسب‌تری استفاده کند، مقیاس‌پذیری، قابلیت خواندن و نگهداری بالاتری دارد. کیفیت طراحی نرم‌افزار در طول چرخه تکامل نرم‌افزار تحت تاثیر قرار می‌گیرد. این مهم می‌تواند به دلیل اضافه شدن ویژگی‌های جدید و یا باگ‌های احتمالی آینده باشد. این موضوع، باعث شد تا توسعه‌دهندگان به بازنمایی کدها روی بیاورند.

بازنمایی کدها (Software Refactoring) به معنی بهبود دادن ساختار و معماری داخلی نرم‌افزار، بدون تغییر در عملکرد بیرونی و توابع آن است که به بهبود قابلیت نگهداری نرم‌افزار منجر می‌شود. اما عملیات بازنمایی خودکار کدها یک کار بسیار چالش‌برانگیز است. زیرا نیاز به یک دید جامع از کل سیستم نرم‌افزاری دارد. برای این منظور، مطالعات اخیر الگوریتم‌های مبتنی بر جستجو را برای تسهیل بازسازی نرم‌افزار معرفی کردند. با این حال، آن‌ها همچنان محدودیت‌های عمده زیر را دارند:

۱. راه‌حل‌های جستجو شده ممکن است اصول طراحی را نقض کنند، زیرا عملکردهای تناسب آنها (Fitness Functions) مستقیماً میزان انطباق نرم‌افزار با اصول طراحی را منعکس و اندازه‌گیری نمی‌کند.

۲. اکثر رویکردها فرآیند جستجو را از یک جمعیت اولیه کاملاً تصادفی شروع می‌کنند که ممکن است به راه‌حل‌های غیر بهینه منجر شود.

در این مقاله، هدف ما توسعه رویکرد بازسازی مبتنی بر جستجوی موثر برای توصیه فعالیت‌های بازسازی بهتر برای توسعه‌دهندگان است که می‌تواند درجه انطباق نرم‌افزار با اصول طراحی و همچنین کیفیت طراحی نرم‌افزار را بهبود بخشد.

در این تحقیق ما به بررسی و پیاده‌سازی DEPICTER پرداختیم. برای توسعه و افزایش جمعیت از الگوریتم ژنتیک NSGA II استفاده کرده‌ایم که از متریک‌ها و الگوهای طراحی به عنوان توابع Fitness استفاده می‌کند. علاوه بر این، DEPICTER با کمک گرفتن از قوانین اکتشافی برای بهبود کیفیت جمعیت اولیه برای تکامل عمومی بعدی استفاده می‌کند.

بنابراین به طور کلی، بخش‌های اصلی این پروژه که در پیاده‌سازی آن‌ها پرداخته شده است، شامل سه قسمت زیر می‌باشد:

۱. پیاده‌سازی الگوریتم ژنتیک NSGA II

۲. پیاده‌سازی توابع هیوریستیک

۳. پیاده‌سازی عملیات بازنمایی خودکار کدها (Automated Refactoring)

برای پیاده‌سازی الگوریتم ژنتیک از زبان جاوا بهره گرفتیم. به همین منظور، اقدام به تعریف جمعیت اولیه و تکامل جمعیت با استفاده از Fitness Function ها کردیم. لازم به ذکر است که به دلیل اینکه پیاده‌سازی توابع تناسب برای مقاله‌ی ما فرایندی بسیار دشوار بود (به دلیل اینکه باید معیاری عددی برای محاسبه مقدار وابستگی بین کلاس‌ها و متودها پیدا می‌کردیم) و با هماهنگی با استاد که دست ما را برای پیاده‌سازی باز گذاشته بودند، از توابع تناسب برای مقاله دیگری که مشابه با مقاله ما از این الگوریتم استفاده می‌کرد، بهره گرفتیم.

برای قسمت توابع هیوریستیک، با پرسش و تاییدیه‌ای که از تی‌ای مربوطه گرفتیم، قرار شد فقط اقدام به محاسبه‌ی توابع کنیم و نیاز به درنظر گرفتن پیش‌شرط‌ها و یا پس‌شرط‌ها نبود. به همین منظور، اقدام به پیاده‌سازی سه تابع هیوریستیک برای هر یک از بازنمایی‌های Merge Package و Move Method کردیم.

در قسمت پیاده‌سازی بازنمایی خودکار کدها از کتابخانه بسیار پرکاربرد ANTLR در زبان پایتون استفاده کردیم و کار خود را به این صورت انجام دادیم که با ارث‌بری از کلاس Listener مرجع برای گرامر زبان جاوا، پیاده‌سازی خود را انجام دادیم. سپس با استفاده از walker کد را parse می‌کردیم و در هنگام ورود و یا خروج از قواعدی که نیاز به انجام کاری داشتیم، فرایند ریفکتورینگ را انجام می‌دادیم و اقدام به نوشتن و جایگزینی توکن‌ها با استفاده از token_stream_rewriter می‌کردیم.

هایلایت‌ها

- پیاده‌سازی الگوریتم ژنتیک NSGA II با تمرکز بر عدم تولید جمعیت اولیه به صورت تصادفی
- در نظر گرفتن متریک‌های الگوهای طراحی به عنوان توابع تناسب
- استفاده از توابع هیوریستیک برای بررسی انجام‌پذیر بودن یا نبودن تعدادی از بازنمایی خودکار کدها
- استفاده از ماژول پرکابرد ANTLR برای پیاده‌سازی بازنمایی خودکار کدها

مشارکت‌ها (Contributions)

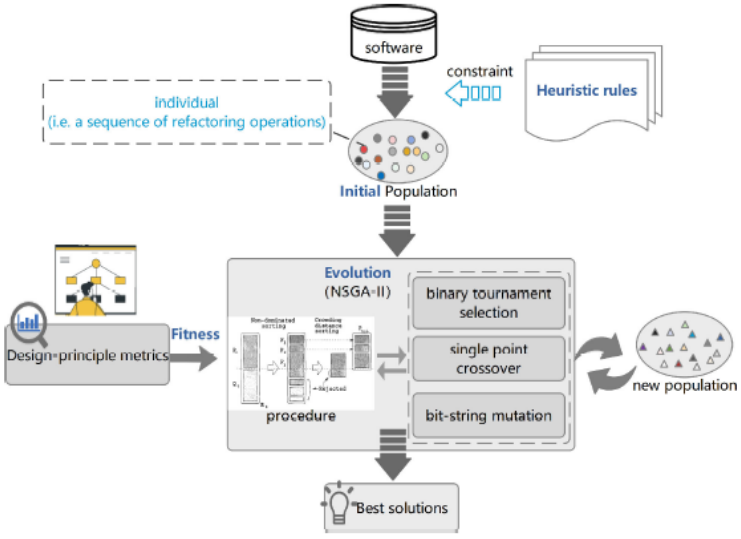
اهدافی که منجر به ایجاد انگیزه برای پیاده‌سازی این پروژه شد، به شرح زیر هستند:

۱. راه‌حل‌های پیشین ممکن است اصول طراحی را نقض کنند، زیرا عملکردهای تناسب آنها (Fitness Functions) مستقیماً میزان انطباق نرم‌افزار با اصول طراحی را منعکس و اندازه‌گیری نمی‌کند.

۲. اکثر رویکردها فرآیند جستجو را از یک جمعیت اولیه کاملاً تصادفی شروع می‌کنند که ممکن است به راه‌حل‌های غیر بهینه منجر شود. بنابراین باید از تصادفی تولید شدن جمعیت جلوگیری شود.

طراحی (Design)

تصویر زیر، شمای کلی طراحی این پروژه را مشخص می‌کند. همانطور که پیش‌تر بیان شد، هدف پیاده‌سازی الگوریتم DEPICTER با رویکرد استفاده از متریک‌های الگوهای طراحی به عنوان Fitness Functions و استفاده از محدودیت‌های توابع هیوریستیک برای محدودسازی و عدم جلوگیری از تولید تصادفی جمعیت اولیه است.



قسمت‌های اصلی معماری به شرح زیر هستند:

۱) توابع هیوریستیک: مجموعه‌ای از قواعد که به ما برای تولید اولیه جمعیت ایده می‌دهند.

۲) تولید جمعیت اولیه: تولید مجموعه‌ای از Individual ها

۳) توابع تناسب: استفاده از الگوهای طراحی برای Fitness Function ها، به گونه ای که میزان انطباق نرم‌افزار با الگوهای طراحی به صورت کمی سنجیده شود.

۴) تکامل جمعیت با استفاده از الگوریتم ژنتیک: استفاده از الگوریتم NSGA II برای تکامل جمعیت و به وجود آوردن جمعیت جدید با استفاده از Mutation و Crossover

پیاده‌سازی (Implementation)

در مورد کدهای مقاله مربوطه، ذکر این نکته ضروری است که برخلاف اکثر گروه‌ها که کدهای مقاله و یا دیتاست را در اختیار داشتند، ما هیچ منبعی نتوانستیم پیدا کنیم. حتی علی رغم اینکه چندین بار به نویسندگان مقاله ایمیل زدیم، اما پاسخی دریافت نکردیم و لذا تمامی کدها را از ابتدا خودمان نوشتیم که مجموعه تمامی کدها در ریپازیتوری گیت‌هاب به آدرس زیر قرار دارد:

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

ساختار پروژه

ما ۴ بخش هیوریستیک، refactoring،nsga و fitness را به صورت جداگانه پیاده سازی کرده‌ایم. زبان‌های برنامه نویسی که برای این ۴ بخش استفاده شده به این صورت است:

- هیوریستیک: بخشی جاوا و بخشی پایتون
- بخش Refactoring: پایتون
- بخش nsga و fitness: جاوا
- بخش هیوریستیک: جاوا

فولدر heuristics: شامل کد هیوریستیک merge packeg و فایل move method هیوریستیک است. فایل move method شامل کدهای تست هیوریستیک به زبان جاوا، کدهای نهایی هیوریستیک به زبان جاوا و کدهای پایتون برای این هیوریستیک است.

فولدر refactoring: شامل کدهای مربوط به بازنمایی خودکار move method و move class و هم‌چنین کدهایی برای به دست آوردن تمامی کلاسها و متودهای یک پکیج است.

فولدر grammar: گرامر زبان جاوا داخل این فولدر قرار دارد.

فولدر gen: لکسر، پارسر، لیستنر، ویزیتور و تمام فایل‌هایی که با استفاده از انتلر تولید شده اند، داخل این بخش قرار دارند.

فولدر src: یکسری فایل‌ها که برای تست برنامه از آنها استفاده کردیم، در این بخش قرار دارد.

فولدر NSGA-II-fitness-refactoring: الگوریتم ژنتیک در این فولدر پیاده‌سازی شده است.

قسمت اول: توابع هیوریستیک

در این بخش دو روش هیوریستیک پیاده‌سازی شده است:

- روش ادغام پکیج‌ها یا Merge package:

در این روش تعداد کلاس هر پکیج را استخراج می‌کنیم، سپس دو پکیج را که تعداد کلاس‌هایشان کمتر از میانگین تعداد کلاس‌های همه پکیج‌هاست انتخاب می‌کنیم. برای این کار ما تمام پکیج هایی که شامل این ویژگی می‌شوند را در یک لیست ذخیره کردیم و دو تای اول را برگرداندیم. این دو را با هم ادغام می‌کنیم: تمامی کلاس‌ها و کدهای درون پکیج اول را به پکیج دوم اضافه می‌کنیم سپس پکیج اول را حذف می‌کنیم.

- روش انتقال متد یا Move method:

روش پیاده سازی move method به اینصورت است که در ابتدا دو کلاس مانند c۱ و c۲ انتخاب می شود. شرط اول این است که تعداد کلاس ها در بسته از دو کمتر نباشد. شرط انتخاب کلاس c۱ این است که تعداد متدهای داخل کلاس c۱ نباید کمتر از ۲ باشد. به طور تصادفی یک متد را از متدهایی انتخاب می کنیم که شرایطی را در c۱ دارند. ارتباط بین متدهای کلاس c۱ کمتر از ارتباط بین متدهای کلاس c۲ باشد.

برای پیاده سازی این هیوریستیک از کتابخانه javaparser استفاده شده است. بدست آوردن رابطه بین متدهای یک کلاس در سه مرحله انجام شده است:

۱. برگرداندن تمام فیلدهای که در یک کلاس وجود دارد.

۲. برگرداندن فیلدهای یک متدهای.

۳. با استفاده از خروجی دو مرحله قبل، تعداد دفعاتی که فیلد یک متد در کلاس استفاده شده است را بدست میاوریم.

با در نظر گرفتن خروجی مرحله سوم می توانیم تعیین کنیم که کدام متد به کلاس دوم انتقال پیدا کند.

برای پیاده سازی هر سه این مراحل از کلاس های FieldDeclaration و CompilationUnit و MethodCallExpr و MethodDeclaration ازکتابخانه javaparser استفاده شده است.

چالشی که در قسمت هیوریستیک‌ها با آن مواجه هستیم، این است که بعد از انجام هیوریستیک داده های مناسب برای انجام هر ریفکتور چگونه در اختیار الگوریتم ژنتیک قرار بگیرد؟. تصمیم بر آن شد که وقتی هیوریستیک‌ها روی بسته را اجرا شد، طبق شروط هر بسته ای که برای انجام آن هیوریستیک مناسب است، بصورت جداگانه ذخیره شود.

قسمت دوم: بازنمایی خودکار کدها

قسمت refactoring از بخش‌های هیوریستیک move method و move class استفاده می‌کند که به زبان پایتون نوشته شده و برای استخراج اطلاعات از کد جاوای ورودی، از ابزار antlr۴ و Listener آن کمک گرفته شده‌است و به درستی و با در نظر گرفتن پکیج ها کار می‌کند.

همینطور کدی به زبان جاوا از refactoringهای مختلف زده شده است کهrefactoring کلاس‌ها و متدها و... را پوشش می‌دهد و تنها ریفکتورینگ مربوط به پکیج‌ها را ندارد.

در قسمت بازنمایی خودکار کدها، اقدام به پیاده‌سازی دو ریفکتورینگ move method و move class کرده‌ایم.

۱) ریفکتورینگ move class: در این روش، با گرفتن پکیج مبدا، پکیج مقصد و کلاسی که قرار است انتقال پیدا کند، کلاس را از یک پکیج به داخل پکیج دیگر منتقل می‌کنیم. به عنوان پیش‌شرط‌ها و پس‌شرط‌ها باید موارد زیر را دقت داشته باشیم:

a) کلاس مربوطه داخل پکیج مدنظر وجود داشته باشد.

b) در صورت انتقال کلاس، هر جایی که در کلاس‌های دیگر از آن کلاس import داشته باشیم، باید از این پس از پکیج جدید import شوند.

۲) ریفکتورینگ move method: در این روش، با گرفتن کلاس مبدا، کلاس مقصد و متودی که قرار است انتقال پیدا کند، متود را از یک کلاس به داخل کلاس دیگر منتقل می‌کنیم. به عنوان پیش‌شرط‌ها و پس‌شرط‌ها باید موارد زیر را دقت داشته باشیم:

a) متود مربوطه و کلاس مبدا واقعا وجود داشته باشند.

b) متود مربوطه داخل کلاس مبدا تعریف شده باشد و نه خارج آن.

c) ممکن است فیلدها (attribute) هایی موجود باشند که متود مربوطه از آن استفاده می‌کند. در این صورت، در صورت انتقال متود به کلاس دیگر، باید حتما فیلد مربوطه نیز منتقل شود. وگرنه متود کار نخواهد کرد. برای این کار، اقدام به تشکیل گرافی با استفاده از ماژول networkx می‌کنیم و متودها و فیلدهای مرتبط با آنها را پیدا کرده و ذخیره می‌کنیم.

قسمت سوم: الگوریتم NSGA II

مراحل این الگوریتم به این صورت است که در ابتدا source code نرم‌افزار مربوطه و توابع هیوریستیک به عنوان ورودی گرفته می‌شود. سپس تلاش می‌شود یک جمعیت اولیه با تمرکز بر توابع تناسب که متریک‌های الگوهای طراحی هستند، تولید و ارزیابی شود. سپس با استفاده از crossover و یا mutation تکثیر جمعیت انجام می‌شود. سپس جمعیت مناسب انتخاب شده و به نسل بعدی منتقل می‌شوند.

در الگوریتم ژنتیک کروموزوم های هر ژن یکی از ریفکتورینگ‌ها است. برای اجرای عملیات جهش یکی از کروموزوم‌ها به تصادفی انتخاب می‌شود و با یکی دیگر از عملیات refactor جایگزین می‌شود.

برای اینکه عملیات crossover با مشکل مواجه نشود، یک عدد معین برای تعداد کروموزوم‌ها در نظر میگیریم. سوالاتی که برای این بخش مطرح بود:

۱. عملیات جهش و crossover ممکن است باعث اخلال در روند ریفکتورینگ شود، در اینصورت چه اقدامی باید صورت گیرد؟

۲. انتخاب نسل جدید با حافظه باشد یا بی‌حافظه؟

پاسخ سوال ۱: اگر در حین عملیات ریفکتورینگ به یکی از ریفکتورینگ‌ها رسیدیم که به واسطه جهش یا crossover یا اعمال ریفکتورهای قبلی قابل اجرا نیست، آن ریفکتور را skip می‌کنیم. (طبق راهنمایی‌ها می‌توانیم به این صورت عمل کنیم که اگر یکی از ژن‌ها بیشتر از ۱۵ درصد آن قابل اجرا نبود، حذف شود.)

پاسخ سوال دوم: انتخاب نسل جدید به صورت با حافظه انجام می‌شود. به اینصورت که بعد از تولید فرزندان و اندازه‌گیری fitness آنها، فرزندان با والدین ترکیب شده و بعد از مرتب‌سازی بر اساس fitness ها بهترین ها برای نسل جدید انتخاب می‌شوند.

در این قسمت ما از دیتاست متفاوتی که شامل هر ۴ دیتاست به صورت بلوک میشد، استفاده کردیم.

پیاده سازی این قسمت در کلاس MOOptimization و با زبان جاوا صورت گرفته است.

قسمت فیتنس

در این قسمت با توجه به مراحلی که پیش آمده بودیم و توضیحاتی که استاد سر کلاس دادند، با همفکری و مشورت تصمیم را بر این گذاشتیم که از فیتنس‌های متفاوتی برای ارزیابی استفاده کنیم. پس مقاله‌های متفاوت را خواندیم و از بین آن‌ها از متریک‌های زیر برای فیتنس‌ها استفاده کردیم:

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

برای پیاده سازی فیتنس‌ها از فرمول‌های استفاده شده در مقاله گروه ۹، استفاده کردیم. که به صورت زیر هستند:

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. $0.2 \times \text{Abstarction} + 0.2 \times \text{Encapsulation} + 0.2 \times \text{Composition} + 0.2 \times \text{Inheritance} + 0.2 \times \text{Polymorphism}$

دلیل استفاده نکردن از فیتنس‌هایی که درون مقاله بود، آن است که فیتنس های خواسته شده منطقی نبودند و به نظر قابل پیاده‌سازی نبودند. چون معیار عددی برای اندازه‌گیری وابستگی یا Dependency توضیح نداده بود.

برای پیاده‌سازی این بخش از زبان جاوا استفاده کردیم. این پیاده سازی با کمک گروه ۹ پیش رفته است.

توضیحات کوتاهی راجب به کدهای زده شده برای این قسمت:

در قسمت Metric ها از متریک‌هایی که در قسمت ارزیابی بیان شده است را پیاده سازی کردیم . در کلاس Metrics این پیاده سازی‌ها را فراخوان کردیم.

در نهایت در کلاس solution، فیتنس‌های درنظر گرفته شده را به دست آوردیم.

ارزیابی

این تحقیق چون سورس‌های آماده نداشت و ما همه چیز را خودمان پیاده‌سازی کردیم. تصمیم گرفتیم که این پروژه را به صورت چند قسمتی جلو ببریم و برای بخش های متفاوت و متنوع هر قسمت چند نمونه محدود را پیاده سازی کنیم. این تحقیق شامل ۱.هیوریستیک ۲. ریفکتورینگ ۳.nsga-ii و ۴. فیتنس است.

این ۴ قسمت هرکدام جداگانه کار می‌کنند و نتیجه ای را به ما برمیگردانند ولی به دلیل آن که هر قسمت نیازمندی‌های خودش را لازم داشت، در پیاده سازی آن ها از زبان‌های متفاوت و تکنیک‌های متفاوت استفاده کردیم. این پروژه‌ها هنوز به هم وصل نیستند و نتیجه‌ای برای کل پروژه نداریم.

- هیوریستیک:

  a. merge package: ابتدا تعداد کلاس های هر پکیج را در آوردیم:

```
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
```

سپس لیستی از کلاس‌ها که شامل قانون توضیح داده شده بود درآوردیم. قانون این بود که پکیج‌هایی انتخاب شوند که تعداد کلاس‌های آنها از میانگین تعداد کل کلاسهای تمامی پکیج‌ها کمتر باشد.

[['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]]

در نهایت دو تا از کلاس ها را انتخاب کردیم:

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

a. move method: **تکمیل شود!**

- ریفکتورینگ: با نگاهی کوتاه به تصاویر زیر می‌توان متوجه شد که ریفکتورینگ به درستی پیاده‌سازی شده است.

۱) Move method: به عنوان نمونه، اقدام به جابجایی کلاس printG از کلاس A به کلاس B کرده‌ایم.

تصویر کلاس A قبل از فرایند بازنمایی:

```
/* 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);
    }
}
```

تصویر کلاس A و B بعد از فرایند بازنمایی:

همانطور که قابل مشاهده است، علاوه بر متود printG فیلد g نیز منتقل شده است تا کلاس B با معنا باشد.

```
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);
    }
}
```

۲) Move class: به عنوان نمونه، اقدام به جابجایی کلاس A از پکیج a.aa به پکیج c.ccc کرده‌ایم.

تصویر کلاس A قبل از بازنمایی:

```
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);
    }
}
```

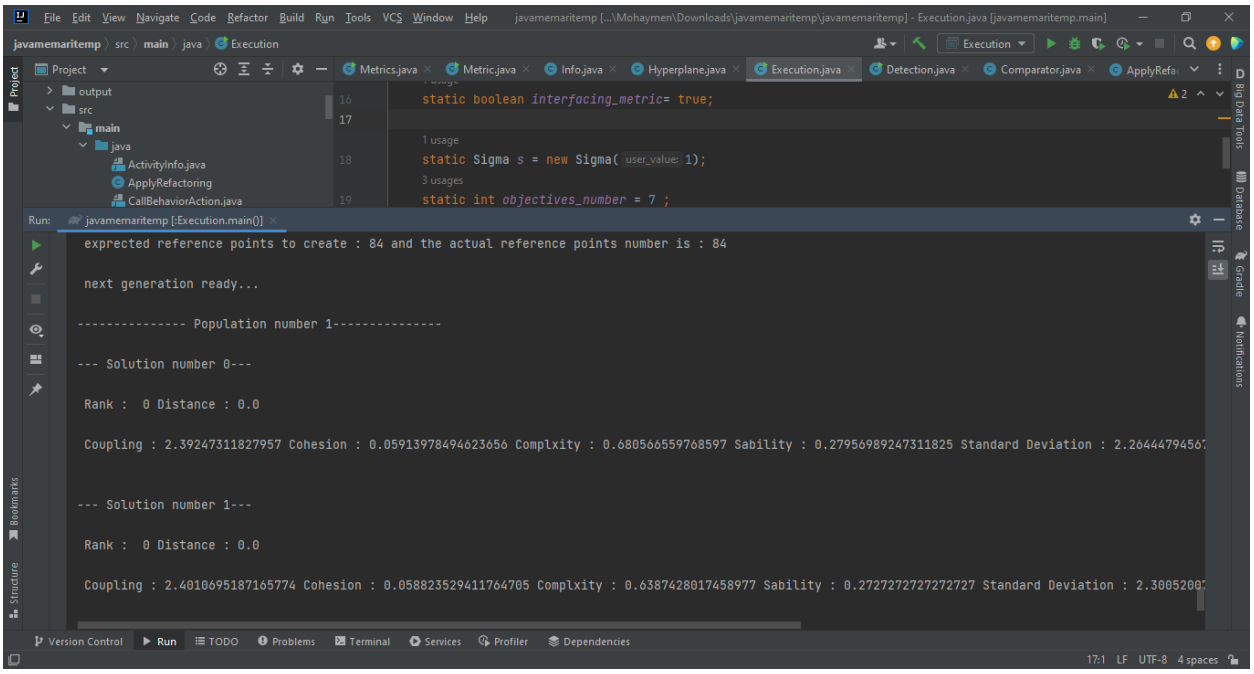
تصویر کلاس A و B بعد از بازنمایی:

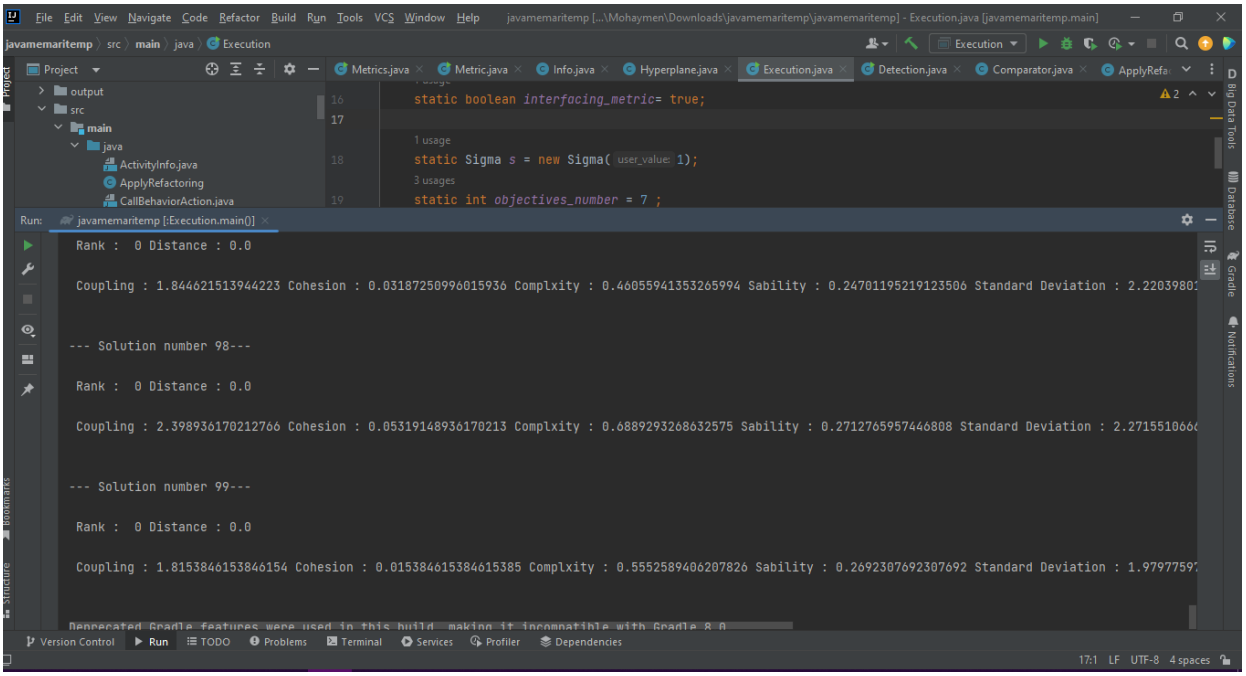
```
package c;

// 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 {

}
```

- فیتنس:

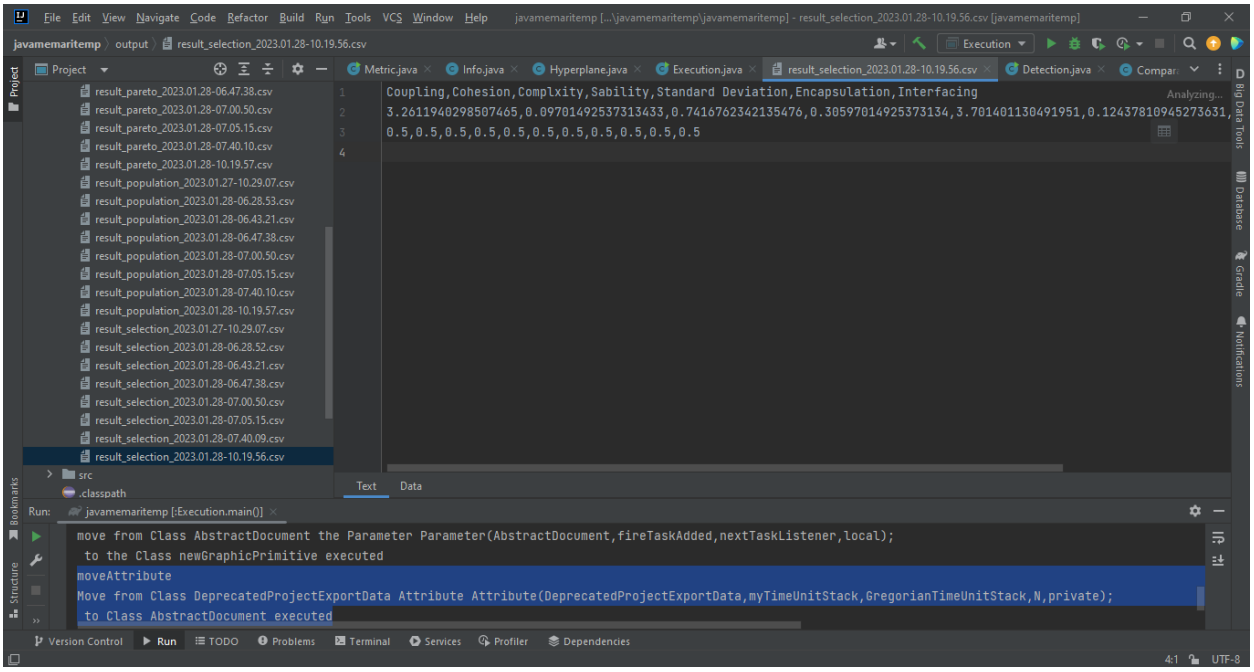
نتایج به دست آمده در این قسمت:



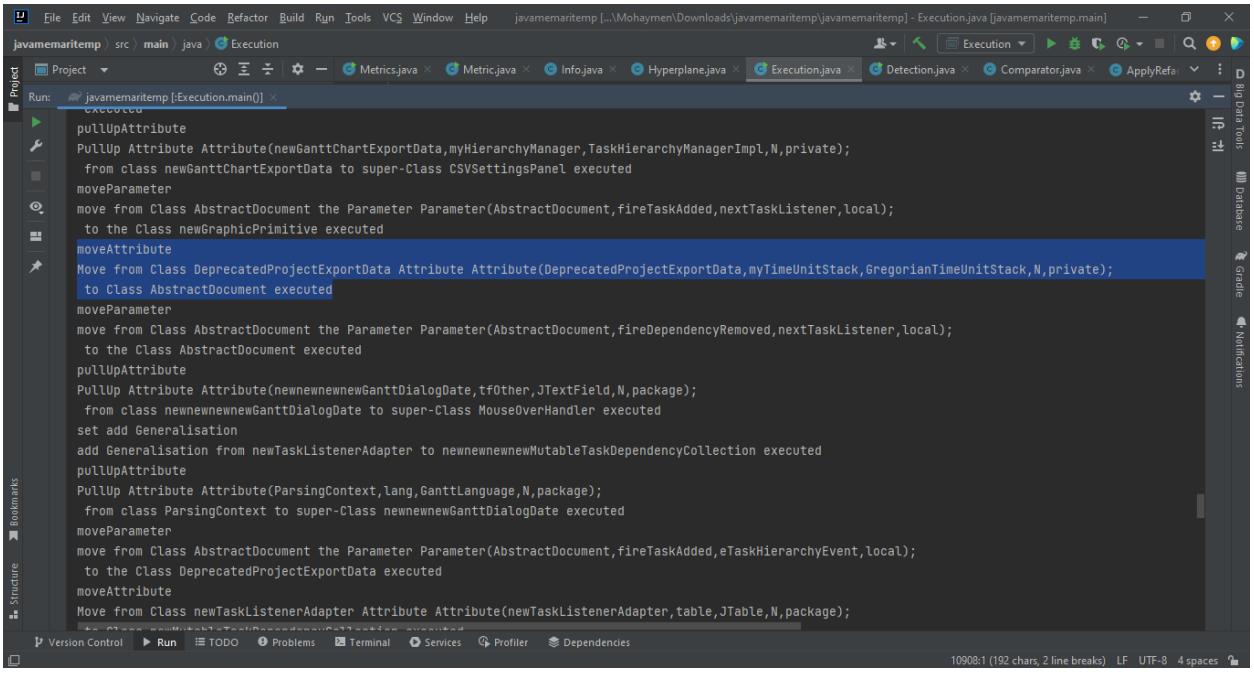


در این جا solution های متفاوتی که بدست آمده است را پرینت کردیم و نتایج هریک را شنان دادیم.

در نهایت بهترین را در یک فایل csv ذخیره کردیم.



همچنین در این کد refactoring ها نیز در نتایج نشان داده شده‌اند:



برای مثال در ۱۰ بلوکی که به صورت تصادفی انتخاب کردیم نزدیک به ۱۰۰ و حتی بیشتر ریفکتورینگ انجام شد و نتایج را ما چک کردیم و دیدیم که نتایج به خوبی اعمال شده است(فقط فایل های خالی حذف نشده بودند).

تقسیم کار

تقسیم کار و مشارکت‌های افراد در هر قسمت از پروژها:

- پریسا علایی: ریفکتورینگ، هیوریستیک (مرج پکیج)،nsga_ii، فیتنس، هماهنگ کردن اعضای تیم و مدیریت پروژه
- دانیال بازمانده: ریفکتورینگ‌های move method و move class، کار بر روی codart، هیوریستیک (مرج پکیج)
- فاطمه رنجبر: الگوریتم NSGA II، هیوریستیک move method

مشکلات و چالش‌ها

برای پیاده‌سازی به مشکلات زیادی خوردیم و چون منبعی در اینترنت پیدا نکردیم تا برای بخشی از کد یاریمان کنند. به نویسندگان پروژه ایمیل زدیم تا سورس کد پروژه را برای ما بفرستند یا برای زدن کد ما را یاری کنند اما پاسخی دریافت نکردیم.

در قدم بعدی از CodART آقای ذاکری کمک گرفتیم (برای اجرا و تنظیم configهای آن نیز با چالش‌های بسیاری رو به رو شدیم).

با وجود مشکلات فراوان توانستیم کدها را بدون مطالعه منابعی که بتوانند برای پیاده سازی یاریمان کنند خودمان پیاده‌سازی کنیم، اگرچه برای کد nsga از یک کد جاوا کمک گرفتیم که همان هم داکیومنتی برایش وجود نداشت و فهم کد خود یک چالش بود.

برای پیاده سازی هیوریستیک‌ها به زبان پایتون، کتابخانه های javalang و javac-parser و javaclass امتحان شده است که به نتیجه نرسید. یکی از علت های استفاده نکردن از کتابخانه های ذکر شده قدیمی بودن

و آپدیت نشدن این کتبخانه ها برای چندین سال است. در نهایت برای پیاده سازی هیوریستیک ها از کتابخانه javaparser برای زبان جاوا استفاده شده است.

در بخش ریفکتورینگ کتابخانه rope مورد مطالعه قرار گرفت.

نتیجه‌گیری و کارهای آتی

با توجه به مراحل این پروژه و سنگینی آن برای پیاده سازی و مقایسه این پروژه با پروژه های مرتبط و همچنین بررسی دیتاست استفاده شده در این تحقیق، شکی در ما بر واقعی بودن نتایج این پروژه بوجود آمد.

در ادامه خوب است که این چند مینی پروژه را یکپارچه‌سازی کنیم تا بتوانیم نتایج بدست آمده را با پروژه اصلی مقایسه کنیم.

منابع و مراجع

[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