دانشکده مهندسی کامپیوتر

طراحی و پیاده‌سازی ابزاری برای بازآرایی خودکار کد منبع جهت بهبود کیفیت نرم‌افزار

پایان‌نامه یا رساله برای دریافت درجه کارشناسی

در رشته مهندسی کامپیوتر

دانشجو:

سید علی آیتی

استاد راهنما:

دکتر سعید پارسا

بهمن ۱۴۰۰


بسم الله الرحمن الرحيم

بسمه‌تعالی

«فرم تصویب فاز صفر پروژه پایانی»

نام و نام خانوادگی: سید علی آیتی	**شماره دانشجویی: ۹۶۵۲۱۰۵۶**

عنوان پروژه:

طراحی و پیاده‌سازی ابزاری برای بازآرایی خودکار کد منبع جهت بهبود کیفیت نرم‌افزار

«مشخصات پروژه»

۱- مقدمه:

با بهبود خودکار طراحی برنامه‌های شی‌گرا بدون تغییر در رفتار آنها، می‌توان هزینه بالای نگهداری نرم‌افزار را کاهش داد. روش مدنظر برای بهبود صفات کیفیت نرم‌افزار، الگوریتم‌های مبتنی بر جست‌وجو مانند ژنتیک است. این الگوریتم با یک فضای حالت تقریباً تصادفی از مجموعه بازآرایی‌های موجود شروع می‌کند و پس از اعمال هر بازآرایی معیارهای فرمول شده کیفیت نرم‌افزار را اندازه‌گیری می‌کند. در هر نسل راهکارهای ضعیف از جمعیت تولید شده حذف می‌شوند. این عمل چندین بار تکرار می‌شود و در فرایند جست‌وجو راه‌حل‌های جدیدی ساخته و به جامعه آماری اضافه می‌شوند. در نهایت تنها بهترین راهکارها باقی خواهند ماند که باعث بیشترین میزان بهبود در کیفیت نرم‌افزار می‌شوند. طراحی و پیاده‌سازی بازآرایی‌ها با تحلیل درخت تجزیه کد منبع و پیمایش آن صورت خواهد گرفت.

۲- مراحل انجام پروژه:

۱. مشخص‌کردن بازسازی‌های موردنیاز (پیش‌شرط‌ها و پس شرط‌های آن)
۲. پیاده‌سازی کامل بازسازی‌ها
۳. پیاده‌سازی الگوریتم ژنتیک
۴. به‌دست‌آوردن سنجه‌های کیفیت نرم‌افزار
۵. اجرای الگوریتم و بررسی نتایج

تأییدیهٔ صحت و اصالت نتایج

باسمه‌تعالی

اینجانب سید علی آیتی به شماره دانشجویی ۹۶۵۲۱۰۵۶ دانشجوی رشته مهندسی کامپیوتر مقطع تحصیلی کارشناسی تأیید می‌نمایم که کلیهٔ نتایج این پایان‌نامه/رساله حاصل کار اینجانب و بدون هرگونه دخل و تصرف است و موارد نسخه‌برداری شده از آثار دیگران را با ذکر کامل مشخصات منبع ذکر کرده‌ام. در صورت اثبات خلاف مندرجات فوق، به تشخیص دانشگاه مطابق با ضوابط و مقررات حاکم (قانون حمایت از حقوق مؤلفان و مصنفان و قانون ترجمه و تکثیر کتب و نشریات و آثار صوتی، ضوابط و مقررات آموزشی، پژوهشی و انضباطی ...) با اینجانب رفتار خواهد شد و حق هرگونه اعتراض در خصوص احقاق حقوق مکتسب و تشخیص و تعیین تخلف و مجازات را از خویش سلب می‌نمایم. در ضمن، مسئولیت هرگونه پاسخگویی به اشخاص اعم از حقیقی و حقوقی و مراجع ذی‌صلاح (اعم از اداری و قضایی) به عهدهٔ اینجانب خواهد بود و دانشگاه هیچ‌گونه مسئولیتی دراین‌خصوص نخواهد داشت.

نام و نام خانوادگی:

سید علی آیتی

امضا و تاریخ:

بهمن ۱۴۰۰

مجوز بهره‌برداری از پایان‌نامه

بهره‌برداری از این پایان‌نامه در چهارچوب مقررات کتابخانه و باتوجه‌به محدودیتی که توسط استاد راهنما به شرح زیر تعیین می‌شود، بلامانع است:

☐ بهره‌برداری از این پایان‌نامه/ رساله برای همگان بلامانع است.

☑ بهره‌برداری از این پایان‌نامه/ رساله با اخذ مجوز از استاد راهنما، بلامانع است.

☐ بهره‌برداری از این پایان‌نامه/ رساله تا تاریخ ..... ممنوع است.

نام استاد یا اساتید راهنما:

دکتر سعید پارسا

تاریخ:

بهمن ۱۴۰۰

امضا:

تقدیم به

آنان که ناتوان شدند تا ما به توانایی برسیم...

موهایشان سپید شد تا ما روسفید شویم...

و عاشقانه سوختند تا گرمابخش وجود ما و روشنگر راهمان باشند...

پدرم

مادرم

و استادانم

تشکر و قدردانی

از استاد گرامیم جناب آقای دکتر سعید پارسا بسیار سپاسگزارم چرا که بدون راهنمایی‌های ایشان به اتمام رساندن این کار بسیار مشکل می‌نمود.

از جناب آقای مرتضی ذاکری، دانشجوی دکتری مهندسی نرم‌افزار در دانشگاه علم و صنعت، به دلیل یاری‌ها و راهنمایی‌های بی چشم داشت ایشان که بسیاری از سختی‌ها را برایم آسان‌تر نمودند، قدردانی می‌کنم.

چکیده

مهندسی نرم‌افزار مبتنی بر جست‌وجو، حوزه‌ای از تحقیقات است که از بازسازی، سنجه‌های نرم‌افزار و الگوریتم‌های بهینه‌سازی مبتنی بر جست‌وجو برای خودکارسازی فرایند نگهداری نرم‌افزار استفاده می‌کند. هدف اصلی بازسازی، بهبود ساختار و کیفیت نرم‌افزار بدون تغییر کردن عملکرد آن است. برای پیدا کردن توالی مناسب از بازسازی‌ها، می‌توان الگوریتم‌های بهینه‌سازی مبتنی بر جست‌وجو را باتکیه‌بر سنجه‌های کیفیتی نرم‌افزار، تطبیق داد. پژوهش انجام شده در این پایان‌نامه باهدف بررسی حوزه تحقیقاتی مهندسی نرم‌افزار مبتنی بر جست‌وجو و آزمایش روش‌هایی برای خودکارسازی مجدد نرم‌افزار با استفاده از الگوریتم‌های بهینه‌سازی است.

در وضعیت فعلی مهندسی نرم‌افزار مبتنی بر جست‌وجو، شکاف‌هایی شناسایی می‌شود و این خاص نیازنمند مطالعه و پژوهش بیشتری است. به عبارت دیگر، نیاز به بررسی بیشتر روش‌های بهینه‌سازی چندهدفه و همچنین آزمایش با معیارهای مختلف برای اندازه‌گیری کیفیت نرم‌افزار، مشهود است. به‌منظور آزمایش روش‌های مختلف برای بهینه‌سازی نرم‌افزار برای بهبود کیفیت، یک ابزار بازسازی خودکار توسعه داده شده است. با استفاده از این ابزار، کد منبع نرم‌افزار بررسی شده و به‌عنوان ورودی سامانه برای ارزیابی و سپس بهبود کیفیت نرم‌افزار مورداستفاده قرار می‌گیرد. از یک الگوریتم بهینه‌سازی چندهدفه استفاده می‌شود تا تمامی اهداف کیفیتی نرم‌افزار بهبود یابد. با استفاده از ابزار تعمیر و نگهداری خودکار و رویکردهای زیربنایی، روشی برای خودکارسازی فرایند بازسازی ارائه شده است. شش حوزه مختلف بااهمیت به‌عنوان اهداف برای بازسازی خودکار بررسی شده است.

این ابزار بر روی سه پروژه متن‌باز جاوا (JSON, jVLT, jOpenChart) آزمایش و ارزیابی شده است و به طور میانگین ۵ درصد کیفیت نرم افزارها را بهبود داد. ابزار توسعه داده شده در پروژه JSON توانست کیفیت نرم‌افزار را حدود ۱۰ درصد افزایش دهد. همچنین با اجراها و آزمایش‌های بیشتر متوجه شدیم اندازه جمعیت و اندازه توالی پاسخ (متناسب با اندازه نرم‌افزار ورودی)، با کیفیت نرم‌افزار رابطه مستقیم دارد. در نهایت ارزیابی کلی سامانه توسعه داده شده بررسی می‌شود.

واژه‌های کلیدی:

مهندسی نرم‌افزار مبتنی بر جست‌وجو، بازآرایی خودکار کد منبع، الگوریتم تکاملی چندهدفه، سنجه‌های کیفیت کد نرم‌افزار

فهرست مطالب

فهرست اشکال

فصل ۱ مقدمه

یک محصول نرم‌افزاری اغلب برای رسیدن به عملکردهای مختلف، همواره در حال تکامل است. این تحولات ممکن است طراحی نرم‌افزار را پیچیده‌تر و متفاوت‌تر از نمونه اصلی کند و کیفیت نرم‌افزار را کاهش دهد. از این نظر، تلاش قابل‌توجهی به مرحله تعمیر و نگهداری نرم‌افزار اختصاص داده شده است، جایی که می‌توان از بازسازی‌های مختلف نرم‌افزاری استفاده کرد. این کار برای بهبود کیفیت نرم‌افزار، از طریق بهبود برخی ویژگی‌های کیفی، مانند قابل درک بودن، قابلیت نگهداری، توسعه پذیری و عملکرد برنامه استفاده می‌شود. علاوه بر این، بازسازی می‌تواند در مراحل اولیه مهندسی نرم‌افزار مانند توسعه نرم‌افزار، طراحی و مهندسی مجدد[۱]نیز مورداستفاده قرار گیرد.

اصطلاح بازسازی در سال ۱۹۹۰ توسط Opdyke و Johnson معرفی شد[۱, ۲]. آنها مجموعه‌ای از بازسازی‌های مختلف را پیشنهاد کردند تا در برنامه‌های C++ اعمال شوند. هر یک از این اصلاح ساختارها، بازسازی نامیده شد. این اصطلاح توسط Fowler، پس از انتشار کتابش، رایج شد. از آن زمان، این واژه برای نشان‌دادن کل فرایند تغییر یک مصنوع نرم‌افزاری نیز مورداستفاده قرار گرفت و معانی و تعاریف متفاوتی به دست آورد. در این مقاله، ما تعریف بازسازی را بر اساس کتاب Fowler، در نظر گرفتیم.

بر اساس تعریف فاولر، بازسازی عبارت است از: «تغییر دادن ساختار داخلی نرم‌افزار به‌منظور درک آسان‌تر و راحت‌تر کردن اصلاح آن بدون تغییر رفتار قابل‌مشاهده توسط نرم‌افزار.»[۲] به بیان ساده‌تر، بازسازی‌ها عملیات نسبتاً ساده‌ای هستند که برای تغییر یک بخش از کد نرم‌افزار انجام می‌شوند، مانند جابه‌جایی یک تابع، جابه‌جایی یک صفت و استخراج یک کلاس.

بازسازی در اصل در زمینه نرم‌افزارهای شی‌گرا پیشنهاد شد، زیرا برخی از سناریوهای بازسازی تنها در این زمینه وجود دارد. اما امروزه، بازسازی نرم‌افزار در زمینه‌های مختلف، مانند نرم‌افزارهای جنبه‌گرا، خط تولید نرم‌افزار، و در مصنوعات متمایز از قبیل کد، مدل‌ها، مستندات، نیازمندی‌ها و غیره اعمال شده است.

یافتن یک توالی خوب از بازسازی‌ها برای اعمال در یک پروژه نرم‌افزاری، یک کار سخت در نظر گرفته می‌شود؛ زیرا طیف گسترده‌ای از بازسازی‌ها وجود دارد و توالی ایده‌آل با ویژگی‌های کیفی مختلف که باید بهبود یابد مرتبط است. در واقع، این یک مشکل بهینه‌سازی است که می‌تواند با شیوه‌های جست‌وجو در

[۱] Re-engineering

زمینه معروف به مهندسی نرم‌افزار مبتنی بر جست‌وجو حل شود. الگوریتم‌های جست‌وجو اجازه می‌دهد تا چندین معیار برای محاسبه کیفیت راه‌حل اضافه شود. این یکی از عواملی است که استفاده از شیوه‌های جست‌وجو برای بازسازی نرم‌افزار را بسیار جذاب می‌کند. علاوه بر این، این الگوریتم‌ها می‌توانند به طور خودکار در یک فضای بزرگ راه‌حل‌هایی را بیابند که ممکن است یک مهندس نرم‌افزار نتواند به آن فکر کند.

۱-۱ تعریف مسئله

بر اساس تعریف مطرح شده از بازسازی، فرایند بازسازی نرم‌افزار در طی چند مرحله انجام می‌شود. این مراحل عبارت‌اند از:

۱. **تشخیص بازسازی و مکان و محل اعمال آن:** ابتدا لازم است مصنوع موردنظر برای بازسازی مشخص شود. به‌عنوان نمونه مصنوع موردنظر می‌تواند کد منبع یک پروژه نرم‌افزاری و یا نسخه مدل شده آن باشد. پس از آن لازم است عناصری که باید بازسازی شوند شناسایی شوند. از این نظر، راه‌های مختلفی برای شناسایی فرصت‌های بازسازی[۱] وجود دارد. یکی از آنها وجود بوی بد است. بوی بد استعاره‌ای برای توصیف الگوهای نرم‌افزاری است که ممکن است با طراحی بد و برنامه‌نویسی بد همراه باشد. همچنین فرصت دیگر برای بازسازی زمانی است که تلاش معناداری برای نگهداری و درک یک نرم‌افزار صرف شود. علاوه بر این، وظیفه شناسایی می‌تواند به‌شدت به دامنه برنامه وابسته باشد.

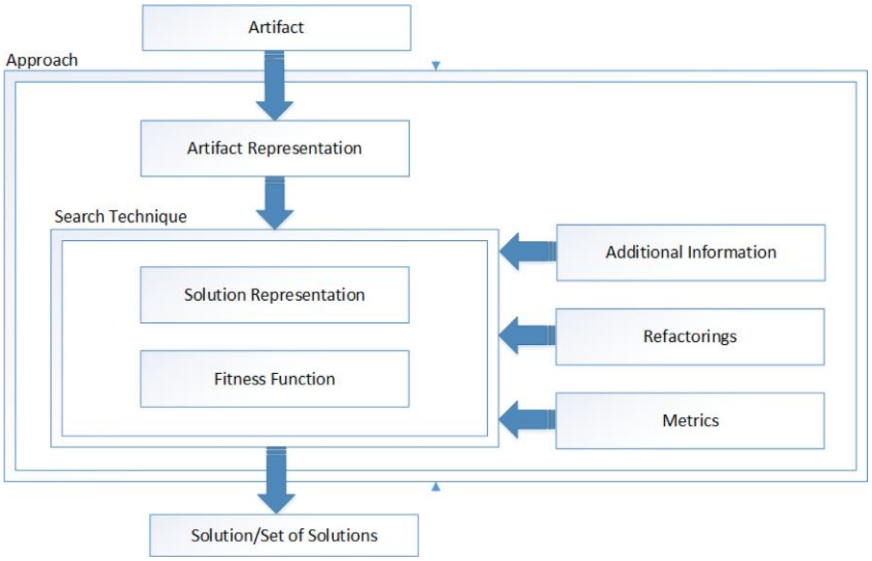
۲. **اعمال بازسازی به صورتی که رفتار برنامه تغییر نکند:** این مرحله به استفاده از روش‌هایی برای تضمین حفظ رفتار نرم‌افزار پس از اعمال بازسازی اشاره دارد. تعریف اصلی حفظ رفتار بیان می‌کند که مقادیر خروجی باید قبل و بعد از یک بازسازی، هنگام استفاده از مجموعه ورودی‌های یکسان، یکسان باشند. علاوه بر این، بسته به دامنه، جنبه‌های دیگری مانند زمان اجرا، محدودیت‌های حافظه و مصرف انرژی می‌تواند برای اطمینان از حفظ رفتار استفاده شود.

۳. **مقایسه سنجه‌های کیفیت با نسخه قابل از اعمال بازسازی و حفظ آن در صورت**

[۱] Refactoring Opportunities

بهبود کیفیت نرم‌افزار: این وظیفه به ارزیابی تأثیر بازسازی‌های اعمال شده بر ویژگی‌های کیفیت نرم‌افزار اشاره دارد. به‌عنوان مثال، می‌توان میزان تأثیر نتیجه حاصل بر قابلیت درک نرم‌افزار از دیدگاه کاربر را با تجزیه‌وتحلیل دستی مقایسه کرد. علاوه بر این، می‌توان با استفاده از برخی فنون، مانند سنجه‌های نرم‌افزاری جنبه‌های مختلف کیفیت نرم‌افزار را، ارزیابی کرد.

زمینه بازسازی مبتنی بر جست‌وجو (SBR)، به اعمال الگوریتم‌های جست‌وجو در بازسازی‌های مختلف نرم‌افزار اختصاص داده شده است. روش‌های موجود معمولاً از شیوه‌های جست‌وجو برای پیشنهاد یا اعمال بازسازی در پروژه‌های نرم‌افزاری استفاده می‌کنند. شکل ۱-۱یک نمای کلی از چنین رویکردی را نشان می‌دهد[۳].



شکل ۱-۱ نگاهی کلی بر بازسازی مبتنی بر جست‌وجو[۳]

همان‌طور که مشاهده می‌شود، ورودی برنامه یک پروژه نرم‌افزاری است که باید بهبود یابد. در بسیاری از پیاده‌سازی‌های انجام شده تا کنون، برنامه به یک مدل تبدیل می‌شود تا توسط شیوه‌های جست‌وجو قابل‌استفاده شود. این مدل می‌تواند کد منبع پروژه به‌صورت مستقیم یا یک نمایش انتزاعی‌تر، مانند نمودارهای کلاسی یا UML[۲]باشد. الگوریتم جست‌وجو باید به‌عنوان ورودی، بازسازی‌های اولیه، سنجه‌ها، معیارها و هر اطلاعات اضافی دیگر برای هدایت فرایند را دریافت کند. به‌عنوان خروجی، روش جست‌وجو

[۱] Search Based Refactoring

[۲] Unified Modeling Language

یک راه‌حل (توالی از بازسازی ها) را برای مسئله برمی‌گرداند.

۱-۲ هدف از انجام پروژه

یکی از معیارهای کیفیت یک نرم‌افزار شی‌گرا، میزان مشکلاتی است که در اجرای برنامه، تعمیر و نگهداری آن وجود دارد. درواقع، هدف رویکرد شی‌گرا تولید طرح‌ها و نرم‌افزارهای قابل‌فهم و منظم، به‌منظور به حداقل رساندن پیچیدگی‌های شناختی در روند برنامه‌نویسی است. با این حال، ممکن است با طرح‌هایی مواجه شویم که در اثر اضافه‌شدن مکرر کد در طول توسعه ضعیف شده‌اند یا درگذشته به‌درستی نگهداری نشده‌اند. چنین طرح‌هایی می‌توانند به بازسازی قابل‌توجهی نیاز داشته باشند تا قابلیت نگهداری آنها تا حد قابل قبولی افزایش یابد و در نتیجه هزینه انجام وظایف تعمیر و نگهداری افزایش می‌یابد.

راه‌حل ایده‌آل برای این مشکل، خودکارسازی بخشی از مرحله بازسازی با استفاده از یک ابزار خودکار بهبود طراحی خواهد بود. چنین ابزاری جدول نماد های برنامه را به همراه کد منبع آن، به‌عنوان ورودی دریافت می‌کند و خروجی این ابزار توالی از بازسازی‌های مناسب و اعمال آن در راستای بهبود کیفیت کد است.

هدف اصلی از انجام این پروژه، پیاده‌سازی چنین ابزاری است، اما ما دید خود را فراتر در نظر گرفتیم و هدف خود را طراحی و توسعه یک ابزار جامع و کامل در حوزه تحلیل و بهبود کیفیت کد، به نام CodART [۴] را در نظر گرفتیم؛ بنابراین، می‌توان گفت هدف از انجام این پروژه، مشارکت در طراحی و توسعه ابزار CodART و پیاده‌سازی بخش‌هایی از آن است.

این ابزار باید شامل بخش‌های مختلفی باشد که از یک ابزار جامع تحلیل کد انتظار می‌رود؛ مانند انواع سنجه‌های نرم‌افزاری، جدول نمادها، پیاده‌سازی خودکار بازسازی‌های مختلف و الگوریتم‌های جست‌وجو تکاملی مختلف.

اگر بخواهیم این هدف نسبتاً وسیع را به چند هدف کوچک‌تر تقسیم کنیم، داریم:

- پیاده‌سازی سنجه‌های نرم‌افزاری مرتبط باکیفیت کد.
- پیاده‌سازی خودکار بازسازی‌های نرم‌افزاری مختلف.
- پیاده‌سازی الگوریتم جست‌وجوی تکاملی، جهت رسیدن به توالی‌ای از بازسازی‌های مناسب.

۱-۳ نوآوری

ابزار CodART از بازسازی‌های مختلف برای بهبود پروژه‌های جاوا با استفاده از سنجه‌های کیفیتی نرم‌افزار برای هدایت الگوریتم جست‌وجو استفاده می‌کند. بسیاری از ابزارهای موجود دیگر دارای انتخاب محدودی از بازسازی‌ها یا و تعداد پایینی از هدف‌ها (برای الگوریتم جست‌وجو) هستند. تلاش‌های انجام شده در این پروژه در راستای تجهیز ابزار CodART با طیف وسیعی از بازسازی‌ها و معیارهای مختلف برای گرفتن بهترین نتیجه انجام شده است.

لازم به ذکر است که این ابزار، توانایی استفاده از رویکرد چندهدفه (۶ تا ۸ هدف) را با توانایی اجرای عملی بازسازی‌ها بر روی کد منبع را ترکیب می‌کند. این ابزار به پیش‌شرط‌ها و پس شرط‌های هر بازسازی توجه می‌کند و تنها بازسازی‌های معتبر در دامنه پروژه ورودی را اجرا می‌کند تا از بروز خطاهای نوشتاری و منطقی جلوگیری شود.

۱-۴ مروری بر مباحث

در فصل بعدی (فصل ۲، ادبیات موضوع) به توضیحات مفاهیم و ابزارهای به‌کاررفته در این پروژه پرداخته خواهد شود. مراحل خودکارسازی بازسازی‌ها به تفسیر بیان می‌شود و مولد پارسر ANTLR [۵] که در بازسازی خودکار نقشی اساسی را ایفا می‌کند، معرفی می‌شود و نحوه استفاده از آن شرح داده خواهد شد. علاوه بر این موارد، مفاهیم اصلی فرایند تکاملی و هدف‌های آن، سنجه‌های کیفیتی QMOOD [۶] و ابزار تحلیل ایستای Understand [۷] شرح و بسط داده می‌شوند.

در فصل سوم (کارهای مرتبط)، کارهای مرتبط بررسی و مقایسه می‌شوند و خواهیم دید حوزه «مهندسی نرم‌افزار مبتنی بر جست‌وجو» توجه محققان را به خود جلب کرده است. در این فصل، خلاصه‌ای از کارهای محققانی مانند Mkaouer، Ouni و Mohan بررسی و مقایسه می‌شود.

در فصل چهارم (روش پیشنهادی و پیاده‌سازی)، ساختار کامل پروژه CodART و نحوه عملکرد بخش‌های مختلف آن شرح داده می‌شود. همچنین نحوه پیاده‌سازی بخش‌های مختلف مانند پیاده‌سازی سنجه‌ها، فرایند تکاملی و پیدا کردن مراجع موجودیت ها در سطح نرم‌افزار به همراه جزئیات و مستندات فنی مطرح شده است.

در فصل پنجم (ارزیابی و نتیجه‌گیری)، ابزار توسعه داده شده (CodART) بر روی پروژه‌های متن‌باز و

در دسترس جاوا اجرا و آزمایش می‌شود. نتایج آزمایش‌ها با هم مقایسه می‌شوند و عملکرد سامانه مورد بحث قرار خواهد گرفت. برای ارزیابی از سه پروژه متن‌باز JSON [۸]، jOpenChart [۹] و jVLT [۱۰] استفاده شده است.

در فصل ششم (کارهای آتی)، مقدمات پروژه Open Understand [۱۱] مطرح شده است که در صورت کامل شدن پتانسیل جایگزین شدن با ابزار قدرتمند Understand را دارد. همچنین راه‌ها و راهکارهایی برای بهبود عملکرد پروژه و پیشنهاداتی برای افزودن بخش‌های جدید به پروژه ارائه خواهد شد.

در فصل هفتم، پیوست‌ها، جزئیات فنی و مستندات کامل سه بازسازی به نسبت دشوار به تفسیر بیان شده است. برای هر بازسازی، ابتدا مفاهیم اولیه شرح داده می‌شود و سپس شبه کد و نحوه کارکرد به‌صورت کلی مورد بحث قرار می‌گیرد و در ادامه کدهای نوشته شده به‌صورت جزئی‌تر به همراه ترسیم درخت تجزیه شرح داده می‌شود. همچنین در این فصل جزئیات و توضیحات مربوط به فرایند تکاملی چندهدفه NSGA-III [۱۲] بیان شده است.

فصل ۲ ادبیات موضوع

۲-۱ مقدمه

در بازسازی خودکار، ورودی‌ها و پیش‌شرط‌ها از اهمیت بالایی برخوردارند. هر بازسازی در یک فایل پایتون پیاده‌سازی شده است و در این فصل ورودی‌ها و پیش‌شرط‌های آنها به طور خلاصه مطرح می‌شود. برای پیاده‌سازی خودکار بازسازی‌ها، از ابزار ANTLR که یک مولد پارسر است استفاده می‌شود. در این فصل این ابزار به طور کامل معرفی و نحوه استفاده از آن، مفصل توضیح داده خواهد شد.

همان‌طور که پیش‌تر گفته شد، برای پیدا کردن ترتیبی از بازسازی‌های بهینه به فرایند تکاملی (مانند الگوریتم ژنتیک) نیازمندیم. در ادامه این فصل بخش‌ها و مفاهیم یک الگوریتم ژنتیک شرح داده می‌شود.

برای راهبرد و هدایت فرایند تکاملی به مجموعه از اهداف مبتنی بر سنجه‌های کیفیت نرم‌افزار نیاز است. در این فصل سنجه‌های نرم‌افزار و اهداف به‌دست‌آمده از آن نیز شرح داده می‌شود. برای محاسبه این سنجه‌ها و نیز برای بخشی از پیاده‌سازی بازسازی‌های خودکار، به جدول نمادها نیاز است که در این فصل به بررسی آن خواهیم پرداخت.

۲-۲ بازسازی خودکار کد

در برنامه‌نویسی و طراحی نرم‌افزار، به فرایند ساختاردهی مجدد به متن کد، بدون آنکه رفتار آن را تغییر دهد، بازسازی گفته می‌شود. هدف اصلی این کار، بهبود طراحی، ساختار و کیفیت نرم‌افزار است. به عبارتی دیگر، می‌توان بازسازی کد را راهی برای منظم و تمیزکردن ساختار کد دانست. گاهی نیز برای جلوگیری از بروز خطاهای نرم‌افزاری این کار انجام می‌شود.

در بازسازی خودکار کد، هدف آن است که به‌صورت کاملاً خودکار و برنامه‌ریزی‌شده، محل‌هایی از کد منبع که نیاز به بازسازی دارند شناسایی شوند، نوع بازسازی موردنیاز تشخیص داده شود و در نهایت بازسازی مربوطه بر روی کد اجرا شود، بدون آنکه رفتار منطقی کد تغییر کند.

یکی از راه‌های تشخیص اینکه چه قسمتی از یک پروژه نرم‌افزاری به بازسازی نیاز دارد، توسط بوی

کد[1]صورت می‌گیرد. در برنامه‌نویسی نرم‌افزار، بوی کد هر مشخصه‌ای در کد منبع یک برنامه است که احتمالاً مشکل عمیق‌تری را نشان می‌دهد. به‌عنوان مثال، طبق قوانین کد تمیز و طراحی نرم‌افزار بر همگان واضح است که یک تابع نباید زیاد طولانی باشد؛ بنابراین بوی کد معروفی به نام تابع بلند[2]این توابع را مشخص می‌کند و احتمالاً آن توابع باید به توابع کوچک‌تری شکسته شوند.

بوی کد به‌تنهایی برای تشخیص و اعمال بازسازی کافی نیست و پس از تشخیص آن، الگوریتم‌ها و تحلیل‌های پیچیده‌ای نیاز است تا بتوانیم بازسازی را به بهترین شکل ممکن اعمال کنیم؛ بنابراین در این پروژه بر روی اعمال بازسازی خودکار تمرکز شده است و برای تشخیص محل‌های بازیابی از ابزارهای مخصوص این کار مانند JDeodorant که یک افزونه برای ویرایشگر کد eclipse است، استفاده شده است[۱۳].

اعمال بازسازی خودکار، توسط تحلیل و بررسی درخت تجزیه قابل انجام است که در ادامه این فصل به شرح آن پرداخته شده است. به‌طورکلی، مراحل انجام یک بازسازی به‌صورت خودکار به‌صورت زیر است:

۱. شناسایی بازسازی و اطلاعات موردنیاز جهت بازسازی خودکار
۲. بررسی پیش‌شرط‌های بازسازی
۳. انجام بازسازی و اعمال تغییرات بر روی متن کد

در این پروژه در مجموع ۱۷ بازسازی پیاده‌سازی شده است که در جدول ۱-۲ جزئیات هر بازسازی مشخص شده است. در پیوست‌ها، توضیحات و جزئیات کامل سه بازسازی انتقال کلاس (پیوست الف)، استخراج کلاس (پیوست ب) و استخراج تابع (پیوست پ) که نسبت به سایر بازسازی‌ها پیچیده‌تر هستند، به‌صورت کامل مستند شده است.

توجه: همه بازسازی‌ها پیش‌شرط «معتبر بودن داده‌های ورودی» را دارند.

جدول ۲-۱ اطلاعت بازسازی‌های پیاده‌سازی شده

نام بازسازی	سطح انجام	ورودی‌ها	پیش‌شرط‌ها	توضیحات
ایستا کردن صفت	صفت	نام کلاس نام صفت	-	تبدیل یک صفت غیرایستا به ایستا

[1] Code Smell

[2] Long Method

غیرایستا کردن صفت	صفت	نام کلاس نام صفت	-	تبدیل یک صفت ایستا به غیرایستا
افزایش سطح صفت	صفت	نام بسته نام کلاس نام صفت	خصوصی بودن صفت	تبدیل یک صفت خصوصی به یک صفت عمومی
کاهش سطح صفت	صفت	نام بسته نام کلاس نام صفت	عمومی بودن صفت بدون استفاده خارجی بودن صفت	تبدیل یک صفت عمومی به یک صفت خصوصی
انتقال صفت	صفت	نام بسته مبدأ نام کلاس مبدأ نام صفت نام بسته مقصد نام کلاس مقصد	متفاوت بودن مبدأ و مقصد عدم وجود وابستگی دایره‌ای	انتقال یک صفت از یک کلاس به کلاسی دیگر
انتقال صفت به کلاس (های) فرزند	صفت	نام بسته مبدأ نام کلاس مبدأ نام صفت لیستی از نام کلاس‌های فرزند جهت انتقال صفت	عدو وجود صفت در کلاس فرزند بدون استفاده داخلی بودن صفت	انتقال صفت از کلاس پدر به کلاس فرزند (ها)
انتقال صفت به کلاس پدر	صفت	نام بسته نام کلاس فرزند نام صفت	عدم وجود صفت در کلاس پدر	انتقال صفت از کلاس (های) فرزند به کلاس پدر
ایستا کردن تابع	تابع	نام کلاس نام تابع	-	تبدیل یک تابع غیرایستا به ایستا
غیرایستا کردن تابع	تابع	نام کلاس نام تابع	-	تبدیل یک تابع ایستا به غیرایستا

افزایش سطح تابع	تابع	نام بسته نام کلاس نام تابع	خصوصی بودن تابع	تبدیل یک تابع خصوصی به تابع عمومی
کاهش سطح تابع	تابع	نام بسته نام کلاس نام تابع	عمومی بدون تابع بدون استفاده خارجی بودن تابع	تبدیل یک تابع عمومی به تابع خصوصی
انتقال تابع	تابع	نام بسته مبدأ نام کلاس مبدأ نام تابع نام بسته مقصد نام کلاس مقصد	بررسی ایستا یا غیرایستا بودن تابع متفاوت بودن مبدأ و مقصد عدم وجود وابستگی دایره‌ای عدم وجود تابع در سلسله مراتب ارث‌بری یا پیاده‌سازی	انتقال یک تابع از یک کلاس به کلاسی دیگر
انتقال تابع به کلاس (های) فرزند	تابع	نام بسته نام کلاس نام تابع لیستی از نام کلاس‌های فرزند جهت انتقال تابع	عدم وجود تابع در کلاس فرزند عدم داشتن وابستگی‌های داخلی غیرعمومی	انتقال یک تابع از کلاس پدر به کلاس (های) فرزند
انتقال تابع به کلاس پدر	تابع	نام تابع لیست نام کلاس‌های فرزند	حداقل ۲ عضو داشتن لیست فرزندان یکسان بودن محتویات توابع نداشتن وابستگی‌های داخلی	انتقال یک تابع از کلاس (های) فرزند به کلاس پدر

انتقال تابع سازنده به کلاس پدر	تابع	نام بسته نام کلاس پدر لیست نام کلاس‌های فرزند	حداقل ۲ عضو داشتن لیست فرزندان در یک بسته بودن کلاس پدر و فرزندان	انتقال مشترکات تابع سازنده بین فرزندان به کلاس پدر
استخراج کلاس	کلاس	مسیر فایل نام کلاس لیستی از توابع جهت انتقال لیستی از صفات جهت انتقال	-	شکستن یک کلاس بلند به دو کلاس کوچک‌تر با استفاده از انتقال صفات و توابع
انتقال کلاس	کلاس	نام بسته مبدأ نام کلاس نام بسته مقصد	متفاوت بودن مبدأ و مقصد متفاوت بودن بسته مبدأ یا مقصد با بسته پیش‌فرض جاوا عدم وجود کلاس با نام مشابه در بسته مقصد	انتقال یک کلاس از بسته‌ای به دسته دیگر

۲-۳ الگوریتم ژنتیک

در سطح یک پروژه نرم‌افزاری، بازسازی‌های زیادی با توالی‌های مختلف قابل انجام است که باعث می‌شود یک فضای حالت وسیعی به وجود آید. از طرفی دیگر هدف تمام توسعه دهندگان و صاحبان نرم‌افزار بهبود کیفیت نرم‌افزار در جهت‌های مختلف است. از این روی برای جست‌وجو بهینه با داشتن چندین هدف، می‌بایست از الگوریتم ژنتیک چندهدفه استفاده شود.

اصول کاری الگوریتم ژنتیک، در ساختار الگوریتمی زیر و شبه کد آن در شکل ۲-۱ نمایش داده شده است [۳۳]. مهم‌ترین گام‌های لازم برای پیاده‌سازی الگوریتم ژنتیک و انواع مختلف آن عبارت‌اند از:

- تولید جمعیت (اولیه) از جواب‌های یک مسئله

- مشخص‌کردن تابع هدف
- تابع برازندگی[۱]
- به‌کارگرفتن عملگرهای ژنتیک (تقاطع[۲] و جهش[۳]) جهت ایجاد تغییرات در جمعیت جواب‌های مسئله

اصول کاری الگوریتم ژنتیک به شرح زیر می‌باشد:

- فرموله کردن جمعیت ابتدایی متشکل از جواب‌های مسئله
- مقداردهی اولیه و تصادفی جمعیت ابتدایی متشکل از جواب‌های مسئله
- حلقه تکرار تا زمانی که شرط توقف ارضا شود:
  - ارزیابی تابع هدف مسئله
  - پیدا کردن تابع برازندگی مناسب
  - انجام عملیات روی جمعیت متشکل از جواب‌های مسئله با استفاده از عملگرهای ژنتیک
    - عملگر تولیدمثل
    - عملگر ترکیب یا آمیزش
    - عملگر جهش

[۱] Fitness Function

[۲] Crossover

[۳] Mutation

Algorithm 1. Genetic Algorithm Pseudocode

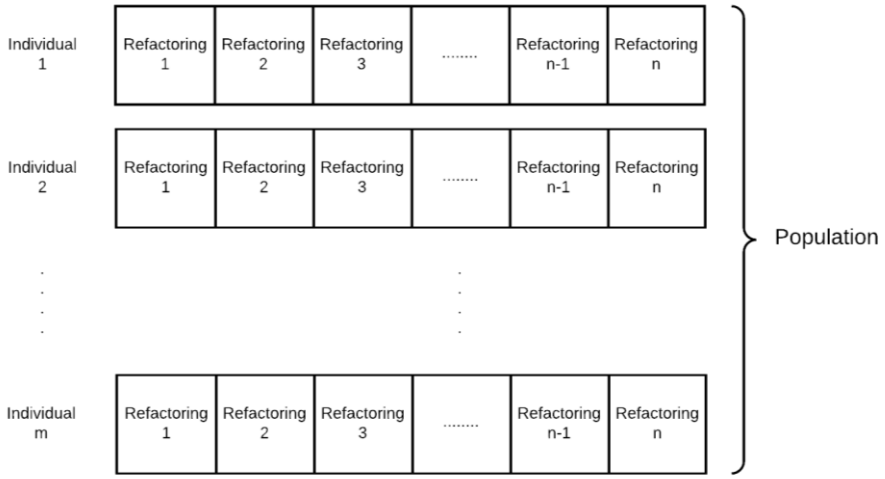
Input: maxIteration
Input: maxFitness
Input: crossoverProbability
Input: mutationProbability
Output: bestSolution

```
// Initialize required parameters
t ← 0
currentFitness ← 0
P(t = 0) = generateInitialPopulation()
// Repeate until termination conditions are satisfied
while t < maxIteration do
    P(t + 1) ← selectBest(P(t))
    P(t + 1) ← doCrossover(crossoverProbability, P(t+1))
    P(t + 1) ← doMutation(mutationProbability, P(t+1)
    currentFitness ← CalculateFitness(P(t + 1)) // Get fitness score for the new generated population
    if currentFitness ≥ maxFitness then
        bestSolution ← P(t + 1)
        break
    end if
    t ← t + 1
end while
```

شکل ۱-۲ شبه کد الگوریتم ژنتیک

۲-۳-۲- جمعیت اولیه

جمعیت اولیه عبارت است از مجموعه‌ای از جواب‌های اولیه مسئله که فرایند تکاملی از آن آغاز می‌شود. جواب مسئله این پایان‌نامه، توالی از بازسازی‌های ممکن جهت بهبود کیفیت نرم‌افزار است؛ بنابراین جمعیت اولیه برای مسئله بهبود کیفیت نرم‌افزار عبارت است از مجموعه‌ای از توالی‌های بازسازی‌های مختلف ممکن در یک پروژه نرم‌افزاری. در شکل ۲-۲ ساختار جمعیت اولیه ترسیم شده است.



شکل ۲-۲ ساختار جمعیت اولیه الگوریتم ژنتیک

همان‌طور که مشاهده می‌شود، این ساختار یک آرایه دوبعدی است. به عبارت دیگر هر عضو این جمعیت، یک توالی از بازسازی‌ها است؛ بنابراین هر بازسازی، یک ژن (بخشی از جواب مسئله) و هر توالی یک کروموزوم (جواب مسئله) است. در این پروژه مقدار n عددی ثابت در نظر گرفته نشده است و هنگام ایجاد هر ژن یک عدد تصادفی در یک محدوده از پیش تعریف شده انتخاب می‌شود.

۲-۳-۳- عملگر آمیزش یا ترکیب

از عملگر ترکیب یا آمیزش، برای بازترکیب دو رشته یا کروموزوم استفاده می‌شود. این کار، باهدف تولید رشته‌ها یا کروموزوم‌های بهتر انجام می‌شود. در عملیات ترکیب در الگوریتم ژنتیک، مواد ژنتیکی دو کروموزوم موجود در جمعیت نسل فعلی، کروموزوم‌های جدیدی در نسل‌های آینده می‌شود. به عبارت دیگر، فرایند بازترکیب، ژن‌های موجود در دو کروموزوم را ترکیب و کروموزوم‌های جدیدی تولید می‌کند.

چنین فرایندی به‌صورت تکراری و در تمامی نسل‌های یک الگوریتم ژنتیک انجام خواهد شد. در فرایند تولیدمثل، معمولاً تعداد کپی‌های ایجاد شده از کروموزوم‌هایی که برازندگی بالایی دارند، بیشتر از دیگر

کروموزوم‌ها خواهد بود. در پایان فرایند تولیدمثل، مخزن جفت‌گیری[1]تشکیل می‌شود و تمامی کپی‌های تولید شده در آن قرار می‌گیرد.

همان‌طور که پیش از این نیز اشاره شد، در مرحله تولیدمثل، رشته‌ها یا کروموزوم‌های جدیدی با مقادیر متغیر متفاوت از جمعیت اصلی، در جمعیت تشکیل نمی‌شوند. در این مرحله (پس از عملیات حاصل از عملگر ترکیب)، رشته‌ها یا کروموزوم‌های جدیدی از طریق تبادل اطلاعات ژنی میان رشته‌ها یا کروموزوم‌های موجود در مخزن جفت‌گیری تشکیل می‌شوند.

به دو کروموزوم یا رشته‌ای که در عملیات ترکیب یا آمیزش مشارکت می‌کنند، کروموزوم‌های والد گفته می‌شود. همچنین، کروموزوم‌هایی که در اثر فرایند ترکیب یا آمیزش تولید می‌شوند، کروموزوم‌های فرزند نامیده می‌شوند.

بنابراین، یک نتیجه‌گیری ممکن از عملیات ترکیب می‌تواند این‌گونه باشد که ترکیب زیررشته‌های خوب (منظور، مجموعه‌ای از ژن‌های خوب در کروموزوم‌های والد) از رشته‌ها یا کروموزوم‌های والدین با یکدیگر، می‌تواند منجر به تولید رشته‌ها یا کروموزوم‌های فرزند خوب شوند. چنین برداشتی، زمانی منطقی و صحیح به شمار می‌آید که عملیات ترکیب زیررشته‌ها، به‌صورت مشخص و روی ژن‌هایی از رشته‌های والدین صورت بگیرد که ترکیب آن‌ها، سبب تولید رشته یا کروموزوم فرزند خوب می‌شود (ترکیب این زیر رشته‌ها، به‌صورت احتمالی انجام می‌شود).

در صورتی که عملیات انتخاب زیررشته‌ها و ترکیب آن‌ها، بر اساس فرایندهای تصادفی انجام شود، هیچ تضمینی وجود ندارد که فرزندهای حاصل، زیررشته‌های خوب والدین خود را به ارث ببرند. در این حالت، خوب بودن یا خوب نبودن فرزندان، به طور مستقیم، به ژن‌هایی در رشته‌های والدین بستگی دارد که در عملیات ترکیب و تولید فرزندان مشارکت دارند.

با این حال، چنین منطقی در الگوریتم ژنتیک نگران‌کننده نیست؛ زیرا در صورتی که کروموزوم‌های فرزند خوبی توسط عملیات ترکیب یا آمیزش تولید شوند، در نسل‌های بعدی، با احتمال بیشتری انتخاب می‌شوند و کپی‌های بیشتری از آن‌ها تولید می‌شود. کپی‌های تولید شده نیز در مخزن جفت‌گیری قرار می‌گیرند تا در مراحل بعدی مورد دست‌کاری ژنی قرار بگیرند.

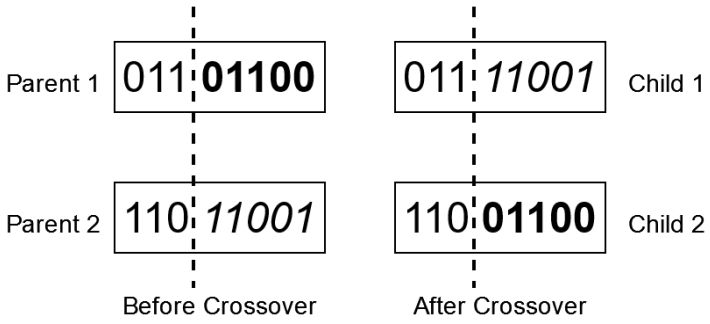
[1] Mating Pool

طبیعت تصادفی عملگرهای ژنتیک (نظیر عملگر ترکیب) ممکن است اثر مخرب[۱] یا سودمند[۲] در کیفیت کروموزوم‌ها یا همان جواب‌های مسئله داشته باشد. در صورتی که کروموزوم‌های فرزند خوبی در نتیجه ترکیب تولید شوند، در جمعیت نسل‌های بعدی، کروموزوم‌های خوبی مشارکت خواهند کرد و برعکس؛ بنابراین، برای حفظ برخی از رشته‌ها یا کروموزوم‌های خوبی که در مخزن وجود دارند، تمامی رشته‌ها یا کروموزوم‌های موجود در مخزن، توسط عملگر ترکیب دست‌کاری نخواهند شد. برای چنین کاری، از مفهومی به نام احتمال ترکیب یا p_c استفاده می‌شود.

وقتی که در الگوریتم ژنتیک، پارامتر احتمال ترکیب تعریف می‌شود، یعنی تنها p_c درصد از رشته‌ها یا کروموزوم‌های موجود در جمعیت، توسط عملگر ترکیب دست‌کاری می‌شوند. به عبارت دیگر، $1 - p_c$ درصد از رشته‌ها یا کروموزوم‌های موجود در جمعیت، به همان شکل اصلی خودشان در جمعیت نسل فعلی باقی خواهند ماند.

تاکنون، عملگرهای ترکیب متعددی برای الگوریتم ژنتیک توسعه داده شده‌اند. روش‌های تک نقطه‌ای و دو نقطه‌ای، از جمله مهم‌ترین عملگرهای ترکیب در الگوریتم ژنتیک محسوب می‌شوند. در غالب عملگرهای ترکیب توسعه داده شده برای الگوریتم ژنتیک، دو رشته یا کروموزوم به طور تصادفی از مخزن انتخاب می‌شوند و بخش‌هایی از رشته‌های این دو کروموزوم با یکدیگر ترکیب می‌شوند تا رشته‌ها یا کروموزوم‌های جدیدی پدید آیند.

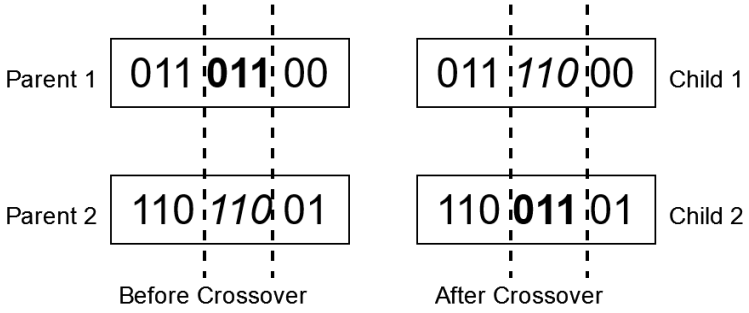
در شکل ۲-۳ و در شکل ۲-۴ به ترتیب عملگر ترکیب یک نقطه‌ای و دو نقطه‌ای نشان داده شده است.



شکل ۳-۲ عملگر ترکیب یک نقطه‌ای

[۱] Detrimental

[۲] Beneficial



شکل ۴-۲ عملگر ترکیب دو نقطه‌ای

در این پروژه، ترکیب تک نقطه‌ای به‌عنوان عملگر ترکیب انتخاب شده است.

۲-۳-۴- عملگر جهش

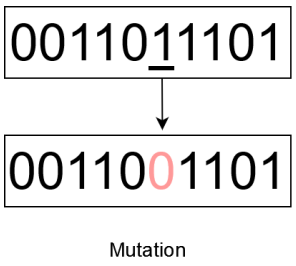
عملگر جهش یکی از مهم‌ترین فرایندهای تکاملی برای رسیدن به جواب بهینه در الگوریتم ژنتیک محسوب می‌شود. در عملگر جهش، به شکل تصادفی، اطلاعات جدیدی به فرایند جست‌وجو در الگوریتم ژنتیک اضافه می‌شود. چنین ویژگی مهمی به الگوریتم ژنتیک کمک می‌کند تا از قرارگرفتن در دام بهینه محلی فرار کند.

زمانی که در نسل‌های متوالی، از عملیات ترکیب و تولیدمثل به‌دفعات روی رشته‌ها یا کروموزوم‌ها استفاده می‌شود، جمعیت کروموزوم‌ها یا جواب‌های کاندید به همگن شدن گرایش پیدا می‌کند. عملگر جهش به الگوریتم ژنتیک کمک می‌کند تا تنوع در جمعیت کروموزوم‌ها یا جواب‌های کاندید افزایش پیدا کند.

عملگر جهش ممکن است منجر به تغییرات عمده در کروموزوم‌های فرزندان تولید شده شود و سبب شود که کروموزوم‌ها یا رشته‌های فرزند تولید شده، ژن‌های کاملاً متفاوتی نسبت به کروموزوم یا رشته والدین داشته باشند.

به بیان ساده‌تر، عملگر جهش، فرایندی تصادفی برای به‌هم‌ریختن و ایجاد اختلال در اطلاعات ژنتیکی محسوب می‌شود. بر خلاف عملگر ترکیب، عملگر جهش در سطح ژن کار می‌کند؛ یعنی، زمانی که ژن‌ها از رشته یا کروموزوم فعلی در رشته یا کروموزوم جدید کپی می‌شوند، این احتمال وجود دارد که هرکدام از این ژن‌ها جهش پیدا کنند. این احتمال معمولاً مقدار بسیار کوچکی است که به آن احتمال جهش یا p_m

گفته می‌شود.



شکل ۵-۲ مثالی از عملگر جهش

در شکل ۲-۵ مثالی ساده از عملگر جهش نشان داده شده است. بسته به تعریف مسئله، نحوه اعمال و پیاده‌سازی عملگر جهش متفاوت است. در این پروژه هر ژن (بازسازی)، تحت احتمال جهش به‌صورت تصادفی انتخاب می‌شود، و با یک بازسازی جدید جایگزین می‌شود.

۲-۳-۵- الگوریتم NSGA-III

در این پروژه، از الگوریتم تکاملی چندهدفه NSGA-III استفاده شده است. الگوریتم NSGA-III یک الگوریتم جدید مبتنی بر الگوریتم ژنتیک مرتب‌سازی نامغلوب است که در سال ۲۰۱۴ توسط پروفسور دب و همکارانش ارائه شده است. این الگوریتم مانند سایر الگوریتم‌های تکاملی از جمعیت اولیه و عملگرهای تقاطع و جهش برای تولید فرزندان استفاده می‌کند.

تفاوت این الگوریتم با سایر الگوریتم‌های تکاملی در روش مرتب‌سازی نا مغلوب و نقاط مرجع بر روی یک صفحه موسوم به صفحه بالایی است. شرح این دو موضوع خارج از مباحث این گزارش است و به خوانندگان واگذار می‌شود. (پیوست ت)

۲-۴ شرحی بر ابزار ANTLR

تحلیل نحوی یا تجزیه، فاز دوم کامپایلر است. یک تحلیل‌گر واژه‌های می‌تواند توکن‌ها را با کمک عبارت‌های منظم و قواعد الگو شناسایی کند، اما نمی‌تواند ساختار یک جمله مفروض را به دلیل

محدودیت‌های عبارت‌های منظم بررسی کند. عبارت‌های منظم نمی‌توانند از توکن‌های متعادل‌سازی مانند پرانتز استفاده کنند؛ بنابراین، این فاز از گرامر مستقل از متن استفاده می‌کند که به‌عنوان اتوماتای پشته‌ای نیز شناخته می‌شود.

شکل ۶-۲ گرامر منظم و مستقل از متن

در شکل ۲-۶ مشخص است که گرامر منظم نیز بخشی از گرامر مستقل از متن است؛ اما برخی مشکلات وجود دارند که خارج از حوزه گرامر منظم است. گرامر مستقل از متن، ابزاری مفیدی در توصیف ساختار زبان‌های برنامه‌نویسی محسوب می‌شود.

۲-۴-۲- گرامر مستقل از متن

یک گرامر مستقل از متن از چهار مؤلفه تشکیل شده است:

- مجموعه‌ای از حالت‌های غیر پایانی (V). حالت‌های غیر پایانی مجموعه‌ای از رشته‌ها را نشان می‌دهند که به تعریف زبان تولید شده از سوی گرامر کمک می‌کنند.
- مجموعه‌ای از توکن‌ها که به نام نمادهای پایانی یا Σ نامیده می‌شوند. نمادهای پایانی نمادهای پایه‌ای هستند که رشته‌ها بر مبنای آن‌ها تشکیل می‌یابند.
- مجموعه‌ای از ترکیب‌ها (P). ترکیب‌های گرامر روشی که نمادهای پایانی و غیر پایانی را می‌توان برای تشکیل رشته‌ها ترکیب کرد، تعیین می‌کنند. هر ترکیب شامل یک نماد غیر پایانی است که سمت چپ ترکیب نامیده می‌شود و یک پیکان است و یک توالی از توکن‌ها و/یا نمادهای پایانی است که سمت راست ترکیب است.

به‌عنوان مثال گرامر زیر را در نظر بگیرید:

$G = (V, \Sigma, P, S)$

$V = \{Q, Z, N\}$

$\Sigma = \{0, 1\}$

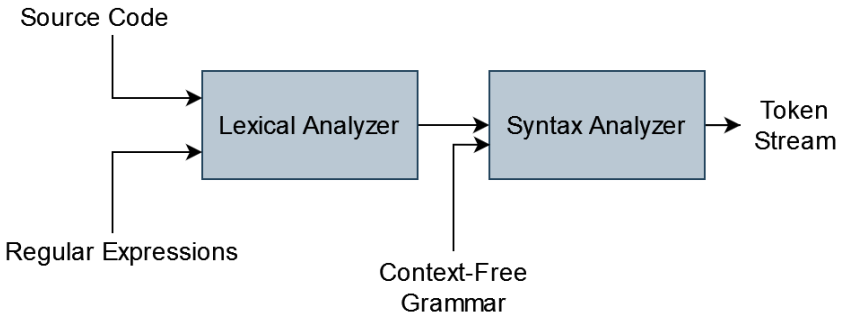
$P = \{Q \rightarrow Z \mid Q \rightarrow N \mid Q \rightarrow \varepsilon \mid Z \rightarrow 0Q0 \mid N \rightarrow 1Q1\}$

$S = \{Q\}$

این گرامر، زبان پالیندوم[1](مجموعه کلماتی که از هر دو طرف یکسان خوانده و نوشته می‌شوند) را توصیف می‌کند: مثلاً ۱۰۰۱, ۱۱۱۰۰۱۱۱, ۰۰۱۰۰, ۱۰۱۰۱۰۱, ۱۱۱۱۱, و غیره.

۲-۴-۳- تحلیلگر نحوی

یک تحلیل‌گر نحوی یا تجزیه‌کننده، ورودی را از تحلیل‌گر واژه‌ای به شکل جریان‌هایی از توکن می‌گیرد. سپس تجزیه‌کننده کد منبع، جریان توکن را بر اساس قواعد ترکیب آنالیز می‌کند تا خطاهای کد را بیابد. خروجی این فاز درخت تجزیه است. در شکل ۲-۷ مراحل تولید درخت تجزیه (مجموعه از توکن‌های ساختار یافته) نشان داده شده است.



شکل ۲-۷ مراحل تشکیل درخت تجزیه

اشتقاق اساساً یک توالی از قواعد ترکیب در جهت دریافت رشته ورودی است. در طی تجزیه ما دو تصمیم برای برخی شکل‌های جمله ورودی می‌گیریم:

[1] Palindrome

- تصمیم در مورد نماد غیر پایانی که باید تعویض شود
- تصمیم در مورد قواعد ترکیبی که به‌وسیله آن نماد غیر پایانی تعویض می‌شود.

برای تصمیم‌گیری در مورد این که نماد غیر پایانی باید با قواعد ترکیب جایگزین شود یا نه دو گزینه داریم: اشتقاق چپ‌ترین و اشتقاق راست‌ترین.

اشتقاق چپ‌ترین

اگر صورت جمله ورودی اسکن شده و از چپ به راست تعویض شود، به نام اشتقاق چپ‌ترین نامیده می‌شود. صورت جمله‌ای اشتقاق یافته به روش اشتقاق چپ‌ترین نیز به نام صورت جمله‌ای چپ نامیده می‌شود.

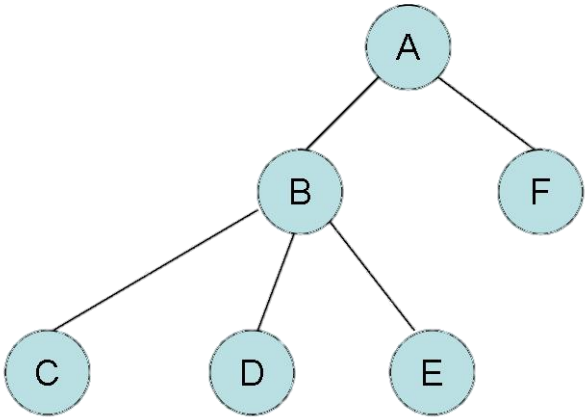
اشتقاق راست‌ترین

اگر یک ورودی را از سمت راست به چپ اسکن کرده و با قواعد ترکیبی جایگزین کنیم، به نام اشتقاق راست‌ترین نامیده می‌شود. صورت جمله‌ای اشتقاق یافته از اشتقاق راست‌ترین، به نام صورت جمله‌ای راست شناخته می‌شود.

۲-۴-۴- درخت تجزیه

همان‌طور که نام این مفهوم پیداست، درخت تجزیه یک ساختار درختی با مجموعه‌ای از گره‌های متصل به هم دارد. در واقع درخت یک گراف همبند بدون دور و بدون جهت است. هر گره در درخت دارای تعدادی (صفر یا بیشتر) گره فرزند دارد که در زیر آن قرار می‌گیرند.[1]به گره که فرزند دارد گره پدر آن فرزند گفته می‌شود. بالاترین گره درخت که هیچ پدری ندارد، ریشه نام دارد. معمولاً عملیات‌های روی درخت از این گره شروع می‌شوند. سایر گره‌ها با دنبال‌کردن یال‌ها از گره ریشه قابل‌دسترسی‌اند. پایین‌ترین گره‌های یک درخت گره‌های برگ نام دارند. این گره‌ها هیچ فرزندی ندارند. در شکل ۲-۸ تصویر داده ساختار درخت ترسیم شده است.

[1] به طور قرار دادی درخت به سمت پایین رشد می‌کند، بر خلاف آنچه در طبیعت می‌بینیم.



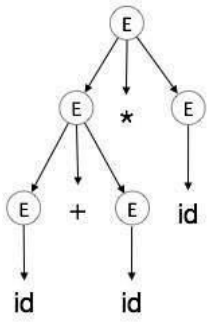
شکل ۸-۲ داده ساختار درخت

در شکل ۲-۸ گره A گره ریشه و گره‌های C, D, E گره‌های برگ هستند. درخت تجزیه (درخت اشتقاق نیز نامیده می‌شود) یک بازسازی از اشتقاق محسوب می‌شود. بدین ترتیب مشاهده شیوه اشتقاق رشته‌ها از نماد آغازین آسان‌تر است. نماد آغازین به ریشه درخت تجزیه تبدیل می‌شود. در یک درخت تجزیه داریم:

- همه گره‌های برگ، پایانی هستند.
- همه گره‌های داخلی، غیر پایانی هستند.
- پیمایش میان ترتیبی، همان رشته ورودی اولیه را به دست می‌دهد.

به‌عنوان مثال درخت تجزیه عبارت a + b * c با گرامر زیر درشکل ۲-۹ ترسیم شده است.

$E \rightarrow E * E$
$E \rightarrow E + E * E$
$E \rightarrow id + E * E$
$E \rightarrow id + id * E$
$E \rightarrow id + id * id$



شکل ۹-۲ نمونه‌ای از درخت تجزیه

۲-۴-۵- معرفی ابزار ANTLR

ابزار ANTLR یک مولد پارسر است که توسط گرامری که دارد به ما می‌کند پارسرهای اختصاصی خود را تولید کنیم. در واقع پارسر یک متن خام را دریافت می‌کند و آن را به یک ساختار منظم (درخت تجزیه) تبدیل می‌کند. مراحل این کار عبارت است از:

۱. تعریف کردن گرامر لغوی و نحوی
۲. صدا زدن ANTLR و تولید تحلیلگر نحوی و لغوی در زبان برنامه‌نویسی مدنظر (پایتون، جاوا، سی شارپ)
۳. استفاده از کدهای تولید شده جهت ساخت و پیمایش درخت

۲-۴-۶- نصب ابزار ANTLR

قبل از نصب ANTLR، نیاز است تا جاوا بر روی دستگاه نصب باشد. برای اطمینان از نصب بودن جاوا، می‌توان دستور زیر را در محیط خط فرمان وارد کرد:

```
java –version
```

خروجی دستور می‌بایست مانند زیر باشد:

```
java version "1.8.0_311"
Java(TM) SE Runtime Environment (build 1.8.0_311-b11)
Java HotSpot(TM) 64-Bit Server VM (build 25.311-b11, mixed mode)
```

اگر پیام زیر دریافت شد، می‌بایست جاوا را بر روی سیستم‌عامل نصب کرد:

‘java’ is not recognized as an internal or external command, operable program or batch file.

برای نصب جاوا، می‌بایست ابتدا ابزار JDK[۱] را دانلود و نصب کرد. پس از نصب باید یک متغیر محیطی با نام JAVA_HOME ایجاد کرد و مقدار آن را باید آدرس محل نصب JDK قرار داد. برای نصب ابزار ANTLR، ابتدا باید فایل کتابخانه جاوا را با استفاده لینک زیر دانلود کرد و آن داخل یک پوشه مانند C:/Javalib ذخیره کرد.

https://www.antlr.org/download/antlr-4.9.2-complete.jar

سپس آدرس پوشه مذکور را باید تحت عنوان CLASSPATH در متغیرهای محیطی ذخیره کرد.

[۱] Java Development Kit

می‌توان این کار را با استفاده از خط فرمان به‌صورت موقتی انجام داد.

```
SET CLASSPATH = .;C:\Javalib\antlr4-complete.jar;%CLASSPATH%
```

در پوشه‌ای جدید یا همان پوشه می‌بایست دستورات لازم برای اجرای ANTLR را تعریف کرد. بدین منظور دو فایل با نام‌های antlr4.bat و grun.bat ایجاد می‌کنیم و محتویات آن را به‌صورت زیر قرار می‌دهیم.

antlr4.bat:

```
java org.antlr.v4.Tool %*
```

grun.bat:

```
java org.antlr.v4.gui.TestRig %*
```

آدرس محل ذخیره این دو فایل را باید به متغیر محیطی PATH اضافه کرد. در صورت اجرای کامل این مراحل با اجرای مجدد خط فرمان و وارد کردن دستور antlr4 خروجی زیر نمایش داده خواهد شد.

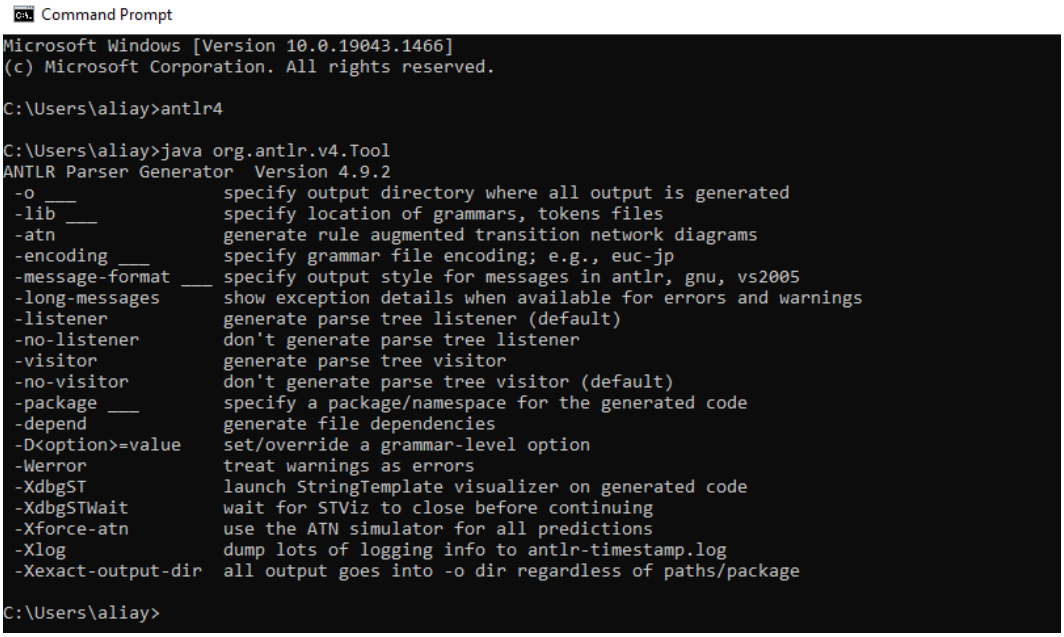
```
ANTLR Parser Generator  Version 4.9.2….
```

در شکل ۲-۱۰ و شکل ۲-۱۱ به ترتیب خلاصه مراحل نصب و نمونه خروجی خط فرمان مشاهده می‌شود.

Windows

1. Download https://www.antlr.org/download/antlr-4.9.2-complete.jar.
2. Add antlr4-complete.jar to CLASSPATH, either:
    1. Permanently: Using System Properties dialog > Environment variables > Create or append to CLASSPATH variable
    2. Temporarily, at command line:
       `SET CLASSPATH=.;C:\Javalib\antlr4-complete.jar;%CLASSPATH%`
3. Create batch commands for ANTLR Tool, TestRig in dir in PATH
   `antlr4.bat: java org.antlr.v4.Tool %*`
   `grun.bat:   java org.antlr.v4.gui.TestRig %*`

شکل ۲-۱۰ خلاصه مراحل نصب ابزار ANTLR



شکل ۲-۱۱ نمونه خروجی از دستور antlr4 پس از نصب موفقیت‌آمیز

۲-۴-۷- نمایش درخت تجزیه

در این بخش نمایش درخت تجزیه به دو روش استفاده از خط فرمان و استفاده از پلاگین نرم‌افزار PyCharm توضیح داده می‌شود. در هر دو روش برای تولید درخت تجزیه به یک گرامر مستقل از متن نیاز داریم. در این مثال گرامر عبارات در نظر گرفته شده است. برای این کار می‌بایست یک فایل با پسوند g4 و نام دلخواه به‌عنوان مثال Expr.g4 ایجاد کرد و محتویات آن را به‌صورت زیر قرار داد.

```
grammar Expr;
prog:   (expr NEWLINE)* ;
expr:   expr ('*'|'/') expr
    |   expr ('+'|'-') expr
    |   INT
    |   '(' expr ')'
    ;
NEWLINE : [\r\n]+ ;
INT     : [0-9]+ ;
```

استفاده از خط فرمان

در محلی که فایل مذکور ایجاد شده است خط فرمان را اجرا می‌کنیم و دستورات زیر را به ترتیب اجرا می‌کنیم.

```
antlr4 Expr.g4
```

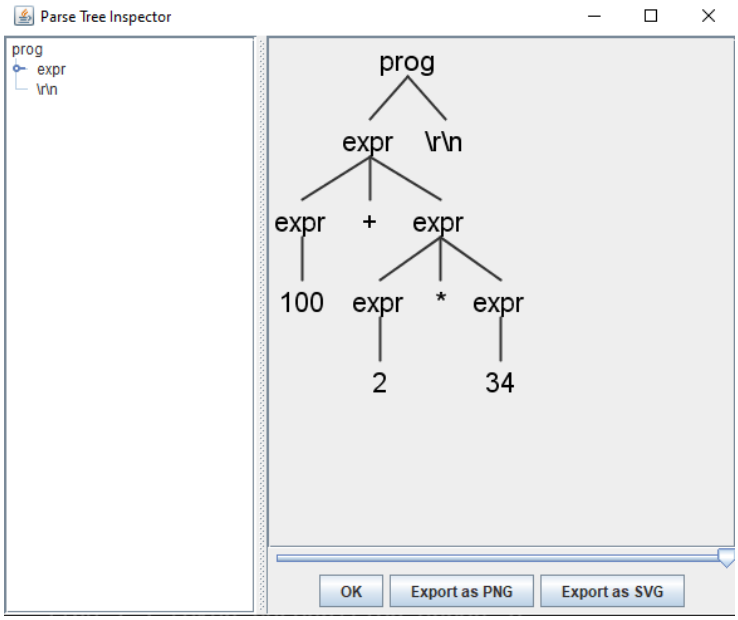
با اجرای دستور فوق مشاهده می‌شود فایل‌های تحلیلگر نحوی و لغوی در زبان جاوا (زبان پیش‌فرض) تولید می‌شود. پس از تولید این فایل‌ها، باید آنها را کامپایل کنیم:

```
javac Expr*.java
```

پس از کامپایل شدن می‌توانیم توسط دستور grun درخت تجزیه برای ورودی دریافت شده نشان داد.

```
grun Expr prog -gui
100+2*34
^D
```

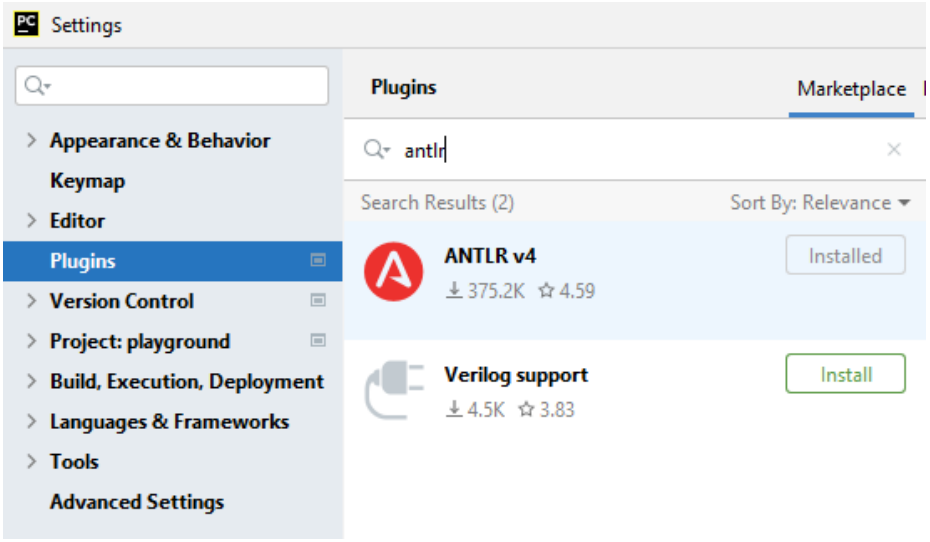
پس از اجرای دستور فوق درخت تجزیه گرامر عبارات برای رشته 100+2*34 مانند شکل ۲-۱۲ نشان داده می‌شود.



شکل ۲-۱۲ نمونه خروجی از دستور grun برای ترسیم درخت تجزیه

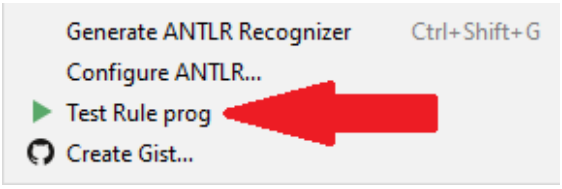
استفاده از افزونه PyCharm

برای نصب افزونه کافی است از زبانه file وارد settings شویم. سپس در بخش plugins عبارت ANTLR را جست‌وجو می‌کنیم (شکل ۲-۱۳) و افزونه را نصب می‌کنیم.



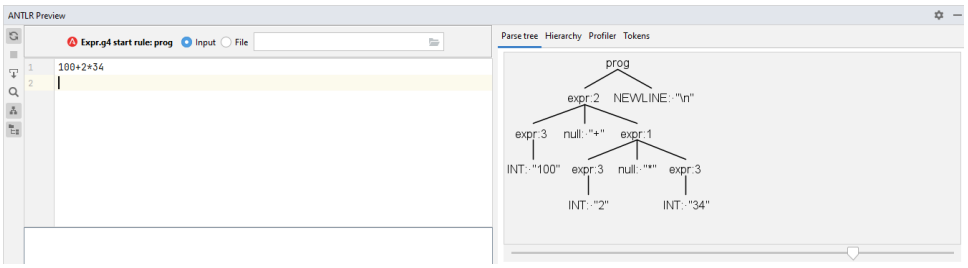
شکل ۲-۱۳ نصب افزونه ابزار ANTLR

پس از نصب افزونه، فایل گرامر را باز کرده و بر روی قانون آغازین که در این مثال prog است کلیک راست کنید و همانند شکل ۲-۱۴ گزینه Test Rule را انتخاب کنید.



شکل ۲-۱۴ ایجاد درخت تجزیه با استفاده از پلاگین

سپس در پایین صفحه پنجره‌ای با عنوان ANTLR Preview باز می‌شود که در قسمت سمت چپ آن می‌توان ورودی را مشخص کرد و در قسمت سمت راست آن هم‌زمان درخت تجزیه ترسیم می‌شود. (شکل ۲-۱۵)

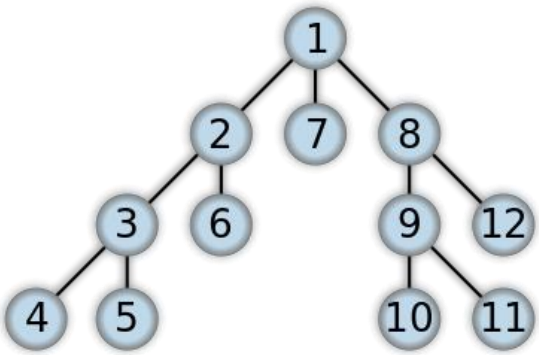


شکل ۲-۱۵ نمونه‌ای از درخت تجزیه ترسیم شده توسط افزونه ANTLR

۲-۴-۸- پیمایش درخت تجزیه

نمایش دادن و مشاهده کردن درخت تجزیه برای کسب اطلاعات در مورد ساختار گرامری زبان موردنظر مفید است، اما جهت کسب اطلاعات بیشتر و یا اصلاح کد لازم است تا درخت را پیمایش کنیم. ابزار ANTLR این امکان را به برنامه‌نویسان می‌دهد تا درخت تجزیه را توسط الگوریتم جست‌وجوی عمق اول پیمایش کنند.

الگوریتم از یک ریشه درخت شروع می‌کند و در هر مرحله همسایه‌های رأس جاری را از طریق یال‌های خروجی رأس جاری به ترتیب بررسی کرده و به‌محض روبه‌رو شدن با همسایه‌ای که قبلاً دیده نشده باشد، به‌صورت بازگشتی برای آن رأس به‌عنوان رأس جاری اجرا می‌شود. در صورتی که همه همسایه‌ها قبلاً دیده شده باشند، الگوریتم عقب‌گرد می‌کند و اجرای الگوریتم برای رأسی که از آن به رأس جاری رسیده‌ایم، ادامه می‌یابد. به عبارتی الگوریتم تا آنجا که ممکن است، به عمق بیشتر می‌رود و در مواجهه با بن‌بست عقب‌گرد می‌کند. این فرایند تا زمانی که همه رأس‌های قابل دستیابی از ریشه دیده شوند ادامه می‌یابد. این فرایند در شکل ۲-۱۶ برای یک درخت ساده نشان داده شده است.



شکل ۲-۱۶ ترتیب پیمایش رئوس در جست‌وجوی عمق اول

به‌منظور پیاده‌سازی این الگوریتم، ابزار ANTLR مفهومی به نام شنونده یا مستمع[۱] معرفی می‌کند. با ارث‌بری کردن از کلاس مستمع تولید شده توسط ANTLR می‌توان پیمایش عمق اول درخت را به طور دلخواه کنترل کرد. برای پیاده‌سازی ابتدا باید گرامر زبان جاوا را داشته باشیم. برای این کار می‌توان فایل‌های

[۱] Listener

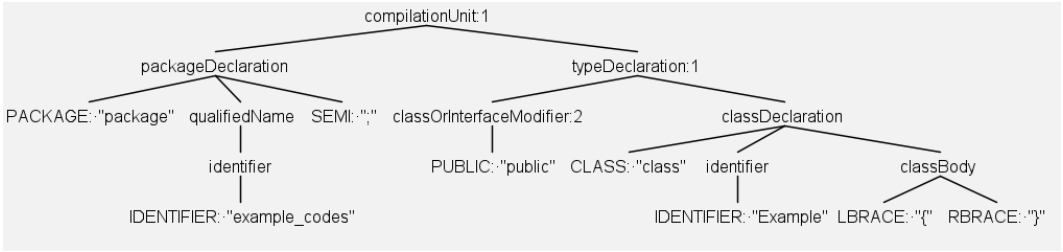
JavaLexer.g4 و JavaParser.g4 را از لینک زیر دریافت کرد.

https://github.com/antlr/grammars-v4/tree/master/java/java

به‌عنوان مثال کد زیر در فایلی با نام Example.java کنار فایل گرامر قرار داده شده است. با استفاده از گرامر پارسر می‌توان درخت تجزیه برای این کد را مشاهده کرد.

```
public class Example {
    public Example(){
        System.out.println("Constructor called...");
        this.testMethod();
    }

    public void testMethod(){
        System.out.println("testMethod called...");
    }
}
```

درخت کد فوق مانند شکل ۲-۱۷ خواهد بود.



شکل ۲-۱۷ درخت تجزیه برای قطعه کدی ساده در زبان جاوا

سپس با استفاده دستورهای زیر، ابزار ANTLR تحلیلگرها و کلاس شنوده را برای زبان پایتون تولید می‌کند.

```
antlr4 -Dlanguage=Python3 JavaLexer.g4
antlr4 -Dlanguage=Python3 JavaParser.g4
```

با مشخص‌کردن مقدار Dlanguage می‌توان زبان مقصد را مشخص کرد. به‌صورت پیش‌فرض زبان مقصد جاوا است. با نوشتن کد زیر در زبان پایتون می‌توان درخت فوق را با الگوریتم DFS پیمایش کرد. اجرا این کد در زبان پایتون نیازمند کتابخانه antlr4 است. با اجرای دستور زیر در خط فرمان می‌توان این کتابخانه را نصب کرد.

```
pip install antlr4-python3-runtime
```

در همان محلی که کلاس مستمع ANTLR تولید شده است فایلی با نام main.py ایجاد کنید و کد زیر را در آن قرار دهید.

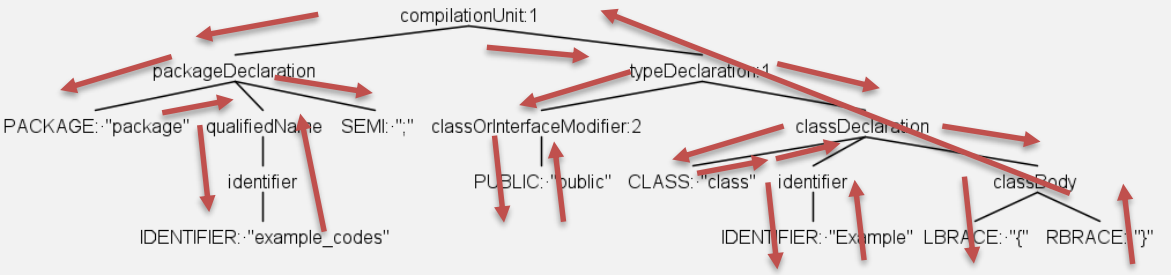
```
from antlr4 import *

from JavaLexer import JavaLexer
from JavaParser import JavaParser
from JavaParserListener import JavaParserListener

class MyListener(JavaParserListener):
    def enterPackageDeclaration(self, ctx):
        print("Entered package declaration...")

    def exitPackageDeclaration(self, ctx):
        print("Exited package declaration...")

    def enterClassDeclaration(self, ctx):
        print("Entered class declaration...")

    def exitClassDeclaration(self, ctx):
        print("Exited class declaration...")

if __name__ == "__main__":
    file_stream = FileStream("./Example.java")
    lexer = JavaLexer(file_stream)
    token_stream = CommonTokenStream(lexer)
    parser = JavaParser(token_stream)
    tree = parser.compilationUnit()
    listener = MyListener()
    walker = ParseTreeWalker()
    walker.walk(listener, tree)
```

همان‌طور که مشاهده می‌شود به از هر قانون در گرامر جاوا یک تابع enter و یک تابع exit دارد. تابع enter هنگامی صدا زده می‌شود که پیمایشگر وارد آن قانون شده باشد و هنگام خروج از آن تابع exit آن قانون صدا زده می‌شود؛ بنابراین باتوجه‌به شکل درخت تجزیه (شکل ۲-۱۸) خروجی کد فوق به‌صورت زیر خواهد بود:

```
Entered package declaration...
Exited package declaration...
Entered class declaration...
Exited class declaration...
```

دلیل این خروجی این است که بر اساس پیمایش عمق اول درخت ابتدا وارد قانون

packageDeclaration می‌شویم و قبل از ورود به classDeclaration از آن خارج می‌شویم. برای درک بهتر این موضوع، شکل ۲-۱۸ نحوه پیمایش نمایش داده شده است. مشاهده می‌شود که پیمایش گر به هر قانون هم وارد می‌شود و هم از آن خارج می‌شود تا زمانی که دوباره به ریشه درخت برسد.



شکل ۲-۱۸ پیمایش DFS درخت تجزیه

۲-۵ سنجه‌های نرم‌افزاری

سنجه نرم‌افزاری یک معیار کمی و قابل محاسبه است که برای اندازه‌گیری ویژگی خاصی از نرم‌افزار استفاده می‌شود. از آنجایی که اندازه‌گیری‌های کمی در همه علوم ضروری است، تلاش مستمری توسط متخصصان علوم کامپیوتر و نظریه‌پردازان برای ارائه رویکردهای مشابه برای توسعه نرم‌افزار وجود دارد. هدف دستیابی به‌اندازه‌گیری‌های عینی، قابل تکرار و کمی است که ممکن است کاربردهای ارزشمند متعددی در برنامه‌ریزی زمان‌بندی و بودجه، تخمین هزینه، تضمین کیفیت، آزمایش، اشکال‌زدایی نرم‌افزار و بهینه‌سازی عملکرد نرم‌افزار داشته باشد.

هر سنجه نرم‌افزاری به‌تنهایی کاربرد خاصی ندارد و معمولاً از ترکیبی از سنجه‌های مختلف استفاده می‌شود. در الگوریتم تکاملی چندهدفه از ترکیبی از سنجه‌های نرم‌افزاری QMOOD تحت عنوان هدف استفاده شده است. QMOOD لیستی از سنجه‌های نرم‌افزار مربوط به کیفیت طراحی و ساختار یک نرم‌افزار است که در جدول ۲-۲ شرح داده شده‌اند.

جدول ۲-۲ لیست سنجه‌های نرم‌افزاری QMOOD

نشانه اختصاری	نام کامل	خصیصه مرتبط با طراحی نرم‌افزار	توضیحات
DSC	Design Size in Classes	Design Size	تعداد کل کلاس‌های یک پروژه.
NOH	Number of Hierarchies	Hierarchies	تعداد کلاس‌هایی که در سلسله‌مراتب ارث‌بری هستند.
ANA	Average Number of Ancestors	Abstraction	متوسط تعداد اجداد هر کلاس.
DAM	Data Access Metric	Encapsulation	نسبت تعداد صفت‌های خصوصی یک کلاس به تعداد کل صفت‌های آن کلاس.
DCC	Direct Class Coupling	Coupling	تعداد صفت‌ها و ورودی توابعی که از جنس کلاس‌های تعریف شده

در پروژه هستند.			
نسبت تعداد انواع ورودی مشترک بین تابع‌های یک کلاس به کل تعداد انواع ورودی‌های توابع.	Cohesion	Cohesion Among Methods of Class	CAM
تعداد متغیرهای تعریف شده در کلاس که از جنس کلاس‌های تعریف شده در پروژه هستند.	Composition	Measure of Aggregation	MOA
نسبت تعداد متدهای به ارث رسیده به کلاس (به‌صورت مستقیم در کلاس تعریف نشده‌اند) به تعداد متدهایی که توسط متدهای عضو آن کلاس قابل‌دسترسی هستند.	Inheritance	Measure of Functional Abstraction	MFA
تعداد تابع‌هایی که می‌توانند به ارث گذاشته شوند که برابر است با تعداد توابع دسترسی عضوها منهای تعداد توابع خصوصی و نهایی.	Polymorphism	Number of Polymorphic Methods	NOP
تعداد توابع عمومی یک کلاس.	Messaging	Class Interface Size	CIS
تعداد کل توابع یک کلاس.	Complexity	Number of Methods	NOM

در محاسبه صفات کیفیت پروژه، مقدار سنجه‌های طراحی که در سطح کلاس تعریف می‌شوند، برابر با میانگین حسابی روی کلاس‌ها، در نظر گرفته می‌شود؛ یعنی، محاسبه و جمع مقادیر سنجه موردنظر برای تمام کلاس‌ها تقسیم بر تعداد کلاس‌ها. از بین سنجه‌های طراحی معرفی شده در جدول فوق، تنها سنجه‌های DSC, NOH, ANA در سطح پروژه هستند و سایر سنجه‌ها در سطح کلاس تعریف شده‌اند.

از ترکیبی از سنجه‌های معرفی شده، هدف‌های بهبود کیفیت نرم‌افزار به دست می‌آید. این هدف در جدول ۳-۲ فرمول‌نویسی شده‌اند. هدف‌های مطرح شده، هدف‌های مثبتی هستند و مقدار آن بیشتر باشد، نرم‌افزار کیفیت بالاتری خواهد داشت.

جدول ۲-۳ لیست معیارهای کیفیتی نرم‌افزار

نام معیار	فرمول محاسبه معیار
Reusability	-0.25*Coupling + 0.25*Cohesion + 0.5*Messaging + 0.5*Design Size
Flexibility	0.25*Encapsulation - 0.25*Coupling + 0.5*Composition + 0.5*Polymorphism
Understandability	-0.33*Abstraction + 0.33*Encapsulation - 0.33*Coupling + 0.33*Cohesion - 0.33*Polymorphism - 0.33*Complexity - 0.33*Complexity - 0.33*Design Size
Functionality	0.12*Cohesion + 0.22*Polymorphism + 0.22*Messaging + 0.22 * Design Size + 0.22*Hierarchies
Extendibility	0.5*Abstraction – 0.5*Coupling + 0.5*Inheritance + 0.5*Polymorphism
Effectiveness	0.2*Abstraction + 0.2*Encapsulation + 0.2*Composition + 0.2*Inheritance + 0.2*Polymorphism

۲-۶ جدول نمادها

ابزار Understand تولید شده توسط شرکت SciTools یکی از قوی‌ترین و سریع‌ترین ابزارهای تحلیل ایستای کد است که از زبان‌های مختلف برنامه‌نویسی مانند جاوا، پایتون، سی‌شارپ و غیره پشتیبانی می‌کند. از ویژگی‌های این ابزار می‌توان به موارد زیر اشاره کرد:

- محاسبه مشخصه‌های کیفیتی نرم‌افزار
- ترسیم نمودارهای مختلف مانند UML
- ذخیره تحلیل ایستا در پایگاه‌داده مخصوص به خود
- داشتن رابط برنامه‌نویسی به زبان‌های پایتون و پرل

همان‌طور که پیش‌تر بیان شد، پس از اعمال برخی از بازسازی‌ها مانند انتقال تابع و بازسازی‌های مشابه آن، نیاز است تا در کل کد پروژه هر جا اشاره‌ای به موجودیت بازسازی شده به طرز مناسبی تغییر کند تا خطاهای کامپایل جلوگیری شود. استفاده از این ابزار در یافتن مراجع موجودیت‌ها تصمیمی معقول

است، چرا که این ابزار بسیار سریع و دقیق است. در ادامه این بخش با رابط برنامه‌نویسی پایتون این ابزار جهت یافتن مراجع موجودیت‌های مختلف آشنا خواهیم شد.

۲-۶-۱- مشخص‌کردن مراجع موجودیت‌ها با رابط برنامه‌نویسی

باتوجه‌به مستندات این ابزار، به تعریف‌هایی مانند فایل، کلاس، متغیر و تابع موجودیت[۱] گفته می‌شود و به هر قسمتی از کد که از موجودیت‌ها استفاده شود مرجع[۲] گفته می‌شود. به‌عنوان مثال قطعه کد زیر را در نظر بگیرید. این کد در فایلی به نام Adder.java در یک پوشه با نام src ذخیره شده است.

```
public class Adder {
    int a, b, result;
    public Adder(int a, int b){
        this.a = a;
        this.b = b;
    }

    public int doAdd(){
        result = a + b;
        return result;
    }
}
```

ابتدا با استفاده خط فرمان و با فرمان زیر، به ابزار Understand می‌گوییم تا تحلیل خود را روی متن کد انجام دهد.

```
und create -db undserstand.und -languages java add .\Adder.java analyze --all
```

به کمک این دستور، ابزار تحلیل را انجام می‌دهد و نتایج را درون پایگاه‌داده خود ذخیره می‌کند. پس از اجرا در داخل پوشه src یک پوشه با نام understand.und ایجاد می‌شود که اطلاعات مربوط به پایگاه‌داده این پروژه در آن قرار دارد. خروجی دستور فوق به‌صورت زیر است:

[۱] Entity

[۲] Reference

```
File: C:\Users\aliay\Desktop\playground\src\Adder.java has been added.
Files added: 1
 Java
  Analyze Pass1
   C:\Users\aliay\Desktop\playground\src\Adder.java
  Analyze Pass2
   C:\Users\aliay\Desktop\playground\src\Adder.java
Analyze Completed (Errors:0 Warnings:0)
```

حال با استفاده از رابط برنامه‌نویسی پایتون ارائه شده توسط این ابزار می‌توان به موجودیت‌ها و اشاره‌های آن دسترسی داشت. به‌عنوان مثال قطعه کد زیر لیستی از موجودیت‌ها را بر روی خط فرمان چاپ می‌کند.

```python
try:
    import understand as und
except ImportError:
    print("Can not import understand")

db = und.open("C:/Users/aliay/Desktop/playground/src/undserstand.und")

for entity in db.ents():
    print(entity, f"[{entity.kind()}]")
```

قطعه کد فوق موجودیت‌ها را به همراه نوع آن بر روی خط فرمان چاپ می‌کند. نتیجه اجرا به‌صورت زیر است:

```
Adder.java [File]
(Unnamed_Package) [Package]
Adder [Public Class]
Adder [Public Constructor]
a [Parameter]
b [Parameter]
doAdd [Public Method]
a [Variable]
b [Variable]
result [Variable]
Object [Unresolved Type]
```

همان‌طور که مشاهده می‌شود، از ابتدای فایل انواع موجودیت‌ها شامل فایل، بسته، متغیر و پارامتر را تشخیص می‌دهد. اما این کار با ابزار ANTLR نیز قابل انجام است. قدرت ابزار Understand در پیاده کردن مراجع هر موجودیت نهفته است. به‌عنوان مثال تنها با افزودن یک حلقه for دیگر می‌توان به لیست مراجع

هر موجودیت دست پیدا کرد.

```
for entity in db.ents():
    print(entity, f"[{entity.kind()}]")
    for reference in entity.refs():
        print("\t", reference)
```

در بخش زیر، قسمتی از نتیجه کد فوق آورده شده است.

```
a [Parameter]
 Define Adder Adder.java(3)
 Use Adder Adder.java(4)
```

مراجع به طور پیش‌فرض به‌صورت زیر چاپ می‌شوند:

```
[Kind] [Scope] [File Name] [Line]
```

به‌عنوان مثال می‌بینیم که این ابزار برای موجودیت پارامتر a دو مرجع پیدا کرده است.

۱. جایی که این پارامتر تعریف[۱]شده است.

۲. جایی که این پارامتر استفاده[۲]شده است.

سپس، نام محل رخدادن مرجع آورده شده است. هر دو این مرجع در تابع سازنده (Adder) رخداده است و پس از آن نام فایل و شماره خطی که مرجع در آن رخداده چاپ می‌شود.

۲-۷ نتیجه‌گیری

تعداد کمی از ابزارها، بازسازی واقعی را روی خود کد اعمال می‌کنند، بنابراین خودکار بودن، آنها را از راه‌حل‌های بازسازی به‌صورت دستی بی‌نیاز می‌کند. همچنین، اگرچه انتخاب ابزارها به‌طورکلی شامل تعداد بی‌شماری از گزینه‌های ممکن برای بازسازی مجدد و تعداد زیادی از بازسازی‌ها، معیارها و شیوه‌های جست‌وجو برای استفاده است، گزینه‌های موجود در بسیاری از خود ابزارها محدود هستند. در ابزار CodART سعی شده است این محدودیت‌ها بر طرف شود.

یکی از اجزای اصلی نگهداری و مهندسی نرم‌افزار مبتنی بر جست‌وجو، معیارهایی است که برای اندازه‌گیری کیفیت یک برنامه استفاده می‌شود. باتوجه‌به ماهیت بسیار ذهنی کیفیت یک سیستم نرم‌افزاری، معیارها می‌توانند تأثیر زیادی بر سودمندی فرایند بهینه‌سازی داشته باشند، بسته به اینکه با چه دقتی

[۱] Define

[۲] Use

کیفیت را در چشم کاربر به تصویر می‌کشند. معیارهای صریح برای هدایت بهینه‌سازی یک راه‌حل موردنیاز است، اما دیدگاه یک توسعه‌دهنده نسبت به کیفیت ممکن است با دیدگاه دیگری متفاوت باشد. دیدیم که معیارهای QMOOD به طور کمی و کیفی کیفیت نرم‌افزار را می‌تواند تعریف کند.

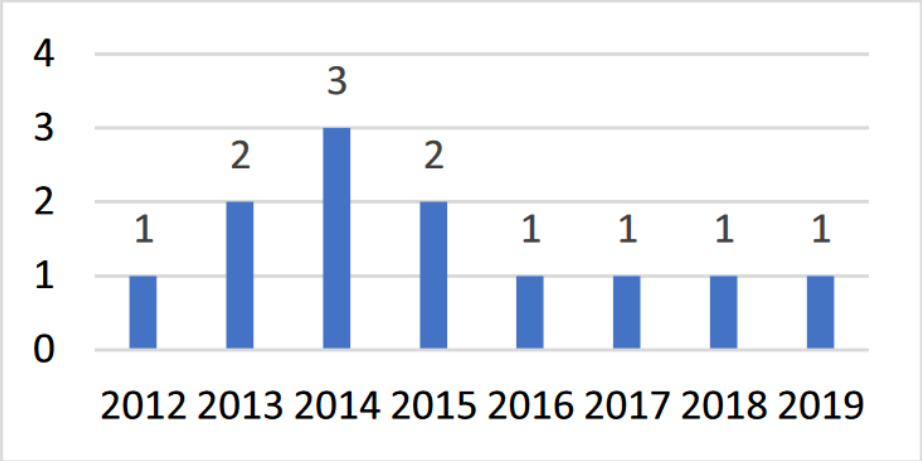
از میان شیوه‌های جست‌وجوی مختلف مورداستفاده برای رسیدگی به تعمیر و نگهداری نرم‌افزار، بخش بزرگی از نمونه‌های مشابه از فرایندهای تکاملی استفاده می‌کنند. در میان این مطالعات، بسیاری از کارهای اخیر (و همین‌طور کار پیش رو) به رویکردهای چندهدفه نگاه کرده‌اند. این روش‌ها باتوجه‌به جنبه‌های متعددی که در نظر گرفته می‌شوند، مسئله را حل می‌کنند. اما هنوز به تحلیل و بررسی بیشتر از این فنون برای کشف پتانسیل استفاده از آنها و استخراج راه‌هایی برای عملی‌تر کردن این رویکرد برای استفاده در محیط توسعه نرم‌افزار موردنیاز است.

فصل ۳ کارهای مرتبط

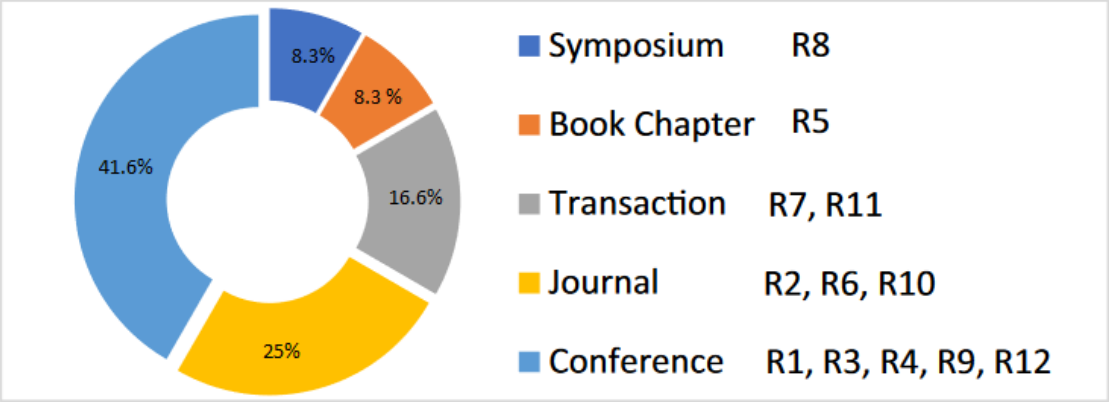
۳-۱ مقدمه

هدف اصلی شیوه‌های بازسازی مبتنی بر جست‌وجو، کشف توالی‌های بازسازی است که منجر به حداکثر بهبود کیفیت نرم‌افزار می‌شود[۳]. در بازسازی مبتنی بر جست‌وجو، جواب مسئله شامل توالی از بازسازی‌های تصادفی انتخاب شده است که بر روی یک پروژه نرم‌افزاری خاص اعمال می‌شود. یک یا چند سنجه کیفیت نرم‌افزار، اغلب به‌عنوان تابع برازندگی برای هدایت فرایند جست‌وجو به سمت تولید توالی بازسازی موردنظر استفاده می‌شود. اکثر روش‌های بازسازی مبتنی بر جست‌وجوی موجود، به‌صورت مستقیم بر روی کد منبع کار نمی‌کنند. بلکه پروژه را مدل سازی می‌کنند و فرضا بازسازی‌ها را انجام می‌دهند.

رویکردهای چندهدفه، برای توصیه راه‌حل‌های بازسازی به کار گرفته شده‌اند و شناسه‌های منحصربه‌فرد تخصیص داده شده به این آثار موجود در جدول ۱-۳ فهرست شده‌اند. توزیع مطالعات صورت‌گرفته در این حوزه که شکل ۳-۱ بیانگر آن است، نشان‌دهنده این است که این زمینه همچنان توجه محققین را به خود جلب می‌کند. علاوه بر این، اکثر این مطالعات در مقالات مجلات یا کنفرانس منتشر شده است (شکل ۳-۲). شرح مختصری از برخی از رویکردهای چندهدفه بازسازی در ادامه این فصل آورده شده است[۱۴].



شکل ۱-۳ توزیع مطالعات اخیر بر اساس سال انتشار



شکل ۲-۳ توزیع مطالعات اخیر بر اساس نوع انتشار

جدول ۳-۱ مطالعات اخیر و شناسه های منحصر به فرد

شناسه	مرجع	شناسه	مرجع
R1	Ouni et al. [16]	R7	Ouni et al. [15]
R2	Ouni et al. [18]	R8	Mkaouer et al. [17]
R3	Ouni et al. [20]	R9	Mkaouer et al. [19]
R4	Ouni et al. [22]	R10	Mkaouer et al. [21]
R5	Ouni et al. [24]	R11	Alizadeh et al. [23]
R6	Ouni et al. [26]	R12	Wang et al. [25]

۳-۲ مطالعات Ouni و همکارانش

آقای Ouni و همکارانش یک رویکرد بازسازی چندهدفه برای اولویت‌بندی فرصت‌های بازسازی نرم‌افزار پیشنهاد کرد. شیوه پیشنهادی از NSGA-II برای به‌دست‌آوردن بهترین موازنه بین دو هدف متناقض از جمله کیفیت نرم‌افزار و تلاش اصلاح کد[۱]استفاده می‌کند. (R2) این روش بر روی شش مجموعه داده ارزیابی شده و موفق به رفع اکثر ناهنجاری‌های طراحی شناسایی شده با حداقل تلاش شد. اونی و همکارانش سپس رویکرد بهینه‌سازی چندهدفه دیگری را برای به‌دست‌آوردن بهترین سازش بین انسجام معنایی و کیفیت نرم‌افزار با استفاده از الگوریتم بهینه‌سازی مشابه پیش‌بینی کرد. (R1) انسجام معنایی با استفاده از

[۱] Code Modification Error

دو روش اکتشافی یعنی جفت ساختاری[1] و شباهت واژگانی[2] به دست میآید. راه‌حل‌های ارائه‌شده توسط رویکرد پیشنهادی به استخراج بازسازی‌های معنادارتر کمک می‌کنند که نقص‌های طراحی را حل می‌کنند و در عین حال انسجام معنایی را نیز حفظ می‌کنند.

بعداً در سال ۲۰۱۳، نویسندگان این کار را با ادغام تغییرات کد ضبط شده به‌عنوان یک هدف اضافی برای ارائه راه‌حل‌های بازسازی جدید گسترش دادند (R3). پس از آن، این رویکرد با افزودن یک هدف دیگر به نام تلاش اصلاح کد بیشتر مورد بررسی قرار گرفت (R5). علاوه بر این، هدف انسجام معنایی با ترکیب معیارهای جدید از جمله شباهت مبتنی بر اجرا[3] وابستگی مبتنی بر انسجام و وراثت ویژگی اصلاح شد. در سال ۲۰۱۶، این کار به دو بوی کد دیگر گسترش یافت و بر روی یک سیستم نرم‌افزار صنعتی اضافی مورد ارزیابی قرار گرفت (R7).

علاوه بر این، Ouni و همکاران، همچنین پتانسیل استفاده از تاریخچه توسعه، همراه باکیفیت نرم‌افزار و انسجام معنایی را برای اولویت‌بندی فعالیت‌های بازسازی مجدد بررسی کردند. (R4) رویکرد پیشنهادی بر روی دو مجموعه داده منبع‌باز ارزیابی می‌شود. نتایج تجربی، بهبود جزئی را در حفظ معنایی و کیفیت نرم‌افزار (در مقایسه با رویکردهای قبلی که تاریخچه توسعه را در نظر نمی‌گیرند)، نشان داد. با این حال، تاریخچه تغییرات برای بسیاری از سیستم‌های نرم‌افزاری، به‌ویژه سیستم‌هایی که به‌تازگی توسعه‌یافته‌اند، در دسترس نیست. از این رو، برای مقابله با این وضعیت، نویسندگان این اثر را با وام گرفتن تاریخ بازسازی‌های گذشته که در زمینه‌های مشابه، از سیستم‌های نرم‌افزاری مختلف اعمال شده‌اند، گسترش دادند (R6). علاوه بر آن، ارزیابی کیفی و کمی رویکرد پیشنهادی به سه مجموعه داده دیگر، دو معیار جدید و بوی کد اضافی برای به تصویر کشیدن عملکرد و مقیاس‌پذیری رویکرد پیشنهادی گسترش یافت.

۳-۳ مطالعات Mkaouer و همکارانش

آقای Mkaouer و همکارانش رویکردی برای تولید راه‌حل‌های مقاوم‌سازی مجدد ارائه کرد که می‌توانست در برابر ماهیت پویای محیط‌های نرم‌افزاری مقاومت کند (R8). رویکرد پیشنهادی با استفاده از

[1] Structrual Coupling

[2] Vocabulary Similarity

[3] Implementation-based Similarity

NSGA-II اجرا می‌شود که هدف آن به‌دست‌آوردن بهترین موازنه بین دو هدف متضاد یعنی استحکام و کیفیت نرم‌افزار به دنبال حداکثر کردن هم‌زمان این اهداف است. کیفیت نرم‌افزار بر حسب تعداد بوهای بد تعیین شده توسط توالی بازسازی تولید شده تعیین می‌شود و استحکام ترکیبی از شدت بوی بد و اهمیت طبقه حاوی آن بو است. راه‌حل‌های مقاوم‌سازی مجدد، عدم قطعیت‌های مربوط به‌شدت بوی بد و اهمیت طبقه را در نظر می‌گیرند، زیرا مقادیر آنها ممکن است به دلیل دیدگاه‌های در حال تکامل نرم‌افزار تغییر کند. روش اقتباس شده بر روی شش سیستم نرم‌افزاری منبع‌باز جاوا ارزیابی شده و عملکرد آن با الگوریتم ژنتیک تک‌هدفه، MOPSO، جست‌وجوی تصادفی، روش غیر فراابتکاری و سایر رویکردهای بازسازی مبتنی بر جست‌وجو مقایسه شده است. نتایج تجربی نشان می‌دهد که NSGA-II [۲۷] نتایج بسیار رقابتی به دست می‌آورد و از رقبای خود از نظر هشت معیار عملکرد بهتر عمل می‌کند. در سال ۲۰۱۷، این کار با در نظر گرفتن شدت بوی کد و اهمیت کلاس به‌عنوان دو هدف مجزا همراه باکیفیت نرم‌افزار گسترش بیشتری یافت. علاوه بر این، رویکرد پیشنهادی سه بوی کد دیگر را در نظر گرفت و بر روی یک نرم‌افزار صنعتی اضافی و دو سیستم نرم‌افزار منبع‌باز دیگر ارزیابی شد (R10).

علاوه بر این، Mkaouer و همکارانش، ابزاری را پیاده‌سازی کرد که به‌صورت پویا فعالیت‌های بازآفرینی را بر اساس تغییرات کد معرفی‌شده و بازخورد توسعه‌دهندگان تغییر می‌دهد و توصیه می‌کند. ابزار پیشنهادی ابتدا مجموعه‌ای از POS های غیر مسلط را به دست می‌آورد که از NSGA-II به دنبال بهینه‌سازی کیفیت نرم‌افزار، تلاش بازسازی و انسجام معنایی استفاده می‌کنند. بعداً از یک الگوریتم نوآوری برای تجزیه‌وتحلیل و کشف بهترین راه‌حل از مجموعه راه‌حل‌های تولید شده استفاده می‌کند. الگوریتم بکار گرفته شده، متداول‌ترین فعالیت‌های بازسازی را از مجموعه راه‌حل‌ها پیدا می‌کند و پس از اولویت‌بندی آن‌ها را به توسعه‌دهندگان پیشنهاد می‌کند. توسعه‌دهنده می‌تواند هر فعالیت بازسازی پیشنهادی را رد، بپذیرد یا اصلاح کند. این بازخورد کیفیت راه‌حل را با به‌روزرسانی اولویت فعالیت‌های بازسازی پیشنهادی با استفاده از الگوریتم جست‌وجوی محلی افزایش می‌دهد. عملکرد ابزار پیشنهادی بر روی یک سیستم نرم‌افزاری صنعتی و چهار سیستم نرم‌افزار منبع‌باز دیگر ارزیابی می‌شود. این کار بعداً با بهبود مکانیسم تعامل، در نظر گرفتن یک سیستم نرم‌افزاری صنعتی و منبع‌باز اضافی، ارائه نتایج آزمایشی جامع و انجام مقایسه با روش‌های بازسازی پیشرفته‌تر گسترش یافت (R11).

۳-۴ ابزار MultiRefacotor

یکی دیگر از معدود ابزارهایی که بازسازی‌ها بر روی کد منبع اعمال می‌کند و به طور مستقیم آن را تغییر می‌دهد، ابزار MultiRefactor حاصل کار موهان و همکارانش است [۲۸]. ابزار MultiRefactor از بازسازی‌های مختلف برای بهبود پروژه‌های جاوا با استفاده از سنجه‌های نرم‌افزاری برای هدایت جست‌وجو استفاده می‌کند. بسیاری از ابزارهای دیگر موجود، دارای انتخاب محدودی از بازسازی‌ها یا سنجه‌ها برای سنجش کیفیت هستند. ابزار MultiRefactor به طیف وسیعی از بازسازی‌ها و معیارهای مختلف مجهز شده است. MultiRefactor توانایی استفاده از یک رویکرد چندهدفه را با توانایی عملی تر برای بهبود خود کد منبع ترکیب می‌کند و بازآفرینی های اعمال شده را بررسی می‌کند تا تغییرات در کد باتوجه‌به دامنه برنامه معتبر باشد.

رویکرد MultiRefactor از چارچوب RECODER برای تغییر کد منبع در برنامه‌های جاوا استفاده می-کند [۲۹]. چارچوب RECODER مدلی از کد را استخراج می‌کند که می‌تواند برای تجزیه‌وتحلیل و اصلاح کد قبل از اعمال تغییرات و نوشتن در فایل استفاده شود. این ابزار کد منبع جاوا را به‌عنوان ورودی می‌گیرد و کد منبع اصلاح شده را به یک پوشه مشخص خروجی می‌دهد. ورودی باید کاملاً قابل کامپایل باشد و باید با هر فایل کتابخانه‌ای ضروری به‌عنوان فایل فشرده jar همراه باشد. جست‌وجوهای متعدد موجود در ابزار دارای تنظیمات ورودی مختلفی هستند که می‌توانند بر اجرای جست‌وجو تأثیر بگذارند. بازسازی‌ها و معیارهای مورداستفاده نیز می‌توانند مشخص شوند. به این ترتیب، ابزار را می‌توان به روش‌های مختلف پیکربندی کرد تا کار خاصی را که می‌خواهید اجرا کنید مشخص کنید. در صورت تمایل، می‌توان چندین کار را به‌گونه‌ای تنظیم کرد که یکی پس از دیگری اجرا شوند.

ابزار MultiRefactor شامل شش گزینه جست‌وجوی مختلف برای تعمیر و نگهداری خودکار، با سه رویکرد متمایز جست‌وجوی فراابتکاری در دسترس است. برای هر نوع جست‌وجو، مجموعه‌ای از موارد قابل تنظیم وجود دارد.

بازسازی‌های مورداستفاده در این ابزار بیشتر بر اساس لیست فاولر از بازسازی‌ها است [۲]. این ابزار، تمام بررسی‌های معنایی مربوطه را انجام می‌دهد و معتبر بودن یا نبودن آن را برمی‌گرداند تا نشان دهد که آیا به‌عنوان یک بازسازی قابل اجرا است و اینکه آیا کد پس از اعمال می‌تواند کامپایل شود یا خیر. بررسی‌های اعمال شده به بازسازی مجدد بستگی دارد و به‌منظور حذف عناصری که برای آن بازسازی قابل اعمال نیستند، مهم هستند. این بررسی‌ها، و همچنین خود فرایند بازسازی، اطمینان حاصل می‌کند که

بازسازی‌های انتخاب شده حفظ رفتار هستند و این برنامه همچنان پس از اعمال بازسازی روی راه‌حل قابل کامپایل خواهد بود. چارچوب RECODER به ابزار اجازه می‌دهد تا تغییرات را در عنصر موجود در مدل اعمال کند. این ممکن است شامل یک تغییر واحد باشد یا مانند مورد بازسازی‌های پیچیده‌تر، ممکن است شامل تعدادی تغییرات فردی در مدل باشد. تغییرات خاصی که در چارچوب RECODER اعمال می‌شود شامل افزودن یک عنصر به عنصر والد، حذف یک عنصر از عنصر والد، یا جایگزینی یک عنصر با عنصر دیگر در مدل است. خود بازسازی با استفاده از این تغییرات مدل خاص ساخته خواهد شد.

۳-۵ نتیجه‌گیری

همان‌طور که کارها و مطالعات انجام شده در سال‌های اخیر بررسی شد و موهان متذکر شد در اکثر روش‌ها مشکلات زیر وجود دارد:

- استفاده از طیف محدودی از بازسازی‌ها
- عدم اجرا واقعی بازسازی‌ها بر روی کد منبع
- استفاده از فرایندهای تکاملی تک‌هدفه یا چندهدفه با طیف محدودی از اهداف

در ابزار CodART سعی شده است تا بر این موارد غلبه کنیم؛ بنابراین این ابزار از طیف وسیعی از بازسازی‌ها پشتیبانی می‌کند و همچنین به‌گونه‌ای توسعه‌یافته شده است تا بتوان بازسازی‌های بیشتری را توسعه داد و یا بازسازی‌های کنونی را ویرایش و اصلاح کرد.

کارها و مطالعات اخیر به ما ثابت کردند که فرایندهای تکاملی چندهدفه عملکرد بهتری نسبت به تک‌هدفه دارند. علی ای حال، این پروژه از سه فرایند تکاملی مختلف تک‌هدفه، و دو فرایند چندهدفه NSGA-II و NSGA-III پشتیبانی می‌کند. برای هدایت این فرایندها ۶ هدف مختلف در راستای بهبود کیفیت نرم‌افزار توسعه داده شده است [۳۰].

علاوه بر موارد گفته شده، تمرکز اصلی این پروژه پیاده‌سازی و اعمال بازسازی‌ها به‌صورت کاملاً خودکار بر روی کد منبع است. همچنین این پروژه بر خلاف اکثر نمونه‌های موجود، به‌صورت متن‌باز و رایگان تعریف شده است و سعی شده است قسمت‌های مختلف به‌خوبی مستند شود. این کار به سایر توسعه دهندگان و مهندسین نرم‌افزار، امکان توسعه دلخواه و شخصی‌سازی را می‌دهد.

فصل ۴ روش پیشنهادی و پیاده‌سازی

۴-۱ مقدمه

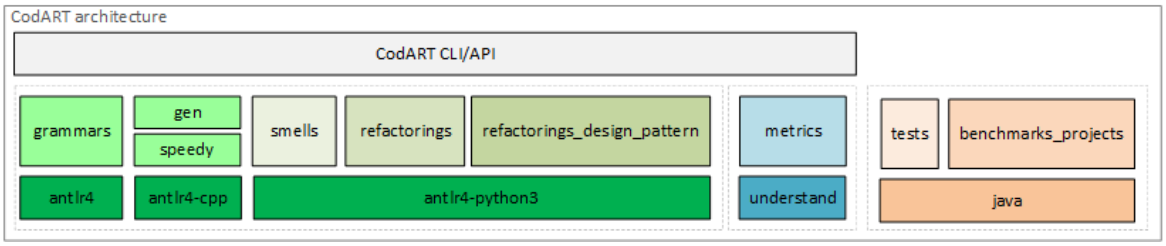
همان‌طور که پیش‌تر شرح داده شد، یک ابزار خودکارسازی بازسازی‌های مختلف از بخش‌های مختلفی تشکیل می‌شود. نام این پروژه CodART انتخاب شده است، زیرا اعتقاد داریم کدنویسی تمیز و اصولی، شاهکار و هنر یک مهندس نرم‌افزار است.

برای حل مسئله تعریف شده در این پروژه (بخش ۱-۱) بخشی از پروژه CodART توسعه داده شده است. مهم‌ترین و وسیع‌ترین بخش این پروژه پیاده سازی خودکار بازسازی های مختلف می‌باشد، که در بسته refactorings این بازسازی ها پیاده سازی شده اند. سپس الگوریتم ژنتیک چند هدفه NSGA-III مبتنی بر معیار های کیفی QMOOD پیاده سازی شدند.

این فصل جزئیات پیاده‌سازی CodART را شرح می‌دهد. این جزئیات شامل معماری CodART، ساختار بسته‌بندی و پوشه بندی پروژه، خودکارسازی بازسازی ها با ابزار ANTLR و توصیه‌های بازسازی از طریق شیوه‌های مهندسی نرم‌افزار مبتنی بر جست‌وجوی چندهدفه است.

۴-۲ معماری پروژه

معماری سطح بالای CodART در شکل ۴-۱ نشان داده شده است. کد منبع از چندین بسته و پوشه پایتون تشکیل شده است. در این بخش، هر جزء را در CodART را توضیح می‌دهیم.



شکل ۴-۱ معماری پروژه

۴-۲-۲- بسته grammars

در این پوشه چهار گرامر مختلف برای زبان برنامه‌نویسی جاوا وجود دارد:

گرامر Java9_v2.g4: این دستور زبان در نسخه اولیه CodART استفاده شد. مشکل اصلی این گرامر این است که تجزیه فایل‌های کد منبع بزرگ به دلیل برخی تصمیمات مورداستفاده در طراحی گرامر بسیار کند انجام می‌شود؛ بنابراین به گرامر سریع JavaParserLabled.g4 روی آورده‌ایم.

گرامر JavaLexer.g4: در گرامر سریع، قوانین لغوی در این فایل مجزا نوشته شده است. این گرامر برای هر دو نسخه از گرامر سریع یعنی JavaParser.g4 و JavaParserLabeled.g4 قابل‌استفاده است.

گرامر JavaParser.g4: این گرامر، نسخه اصلی گرامر سریع جاوا است. این گرامر در حال حاضر در برخی از بازسازی‌ها استفاده می‌شود. در نسخه بعدی، این گرامر با JavaPaseredLabled.g4 جایگزین خواهد شد.

گرامر JavaParserLabeled.g4: این فایل حاوی همان گرامر JavaParsar.g4 است. تنها تفاوت این است که قوانینی که از بسط‌های مختلفی تشکل شده‌اند با یک نام خاص برچسب‌گذاری می‌شوند؛ بنابراین، مولد تجزیه‌کننده ANTLR روش‌های بازدیدکننده و شنونده جداگانه برای هر بسط را ایجاد می‌کند. این دستور زبان توسعه برخی از بازسازی‌ها را تسهیل می‌کند. این گرامر در اکثر بازسازی‌های پروژه CodART استفاده می‌شود.

۴-۲-۳- بسته gen

بسته gen شامل همه کد منبع تولید شده برای تجزیه‌کننده، تحلیل‌گر لغوی، بازدیدکننده و شنونده برای دستور زبان‌های مختلف موجود در فهرست دستور زبان هستند. برای توسعه بازسازی و بوهای کد، بسته gen که حاوی کد منبع تولید شده JavaParserLabled.g4 است، باید استفاده شود. محتوای این بسته به‌صورت خودکار تولید می‌شود و بنابراین نباید به‌صورت دستی آن را تغییر دهید. ماژول‌های این بسته ژن فقط برای وارد کردن و استفاده در ماژول‌های دیگر هستند.

۴-۲-۴- بسته speedy

پیاده‌سازی پایتون برای ANTLR کارایی کمتری نسبت به پیاده‌سازی جاوا یا سی پلاس پلاس دارد. بسته speedy یک تجزیه‌کننده جاوا را با زبان برنامه‌نویسی C++ پیاده‌سازی می‌کند که کارایی و سرعت تجزیه را بهبود می‌بخشد. این بسته از کتابخانه speedy-antlr با برخی تغییرات جزئی استفاده می‌کند. برای استفاده، ابتدا بسته speedy باید روی سیستم کاربر نصب شود. برای نصب، به یک کامپایلر C++ نیاز است. در برنامه آینده CodArt جایگزینی کامل پیاده‌سازی پایتون با سی پلاس پلاس است.

۴-۲-۵- بسته refactorings

بسته refactorings بسته اصلی در پروژه CodART است و شامل چندین فایل پایتون است که عملکردهای هسته CodART را تشکیل می‌دهند. هر ماژول خودکارسازی یک عملیات بازسازی را طبق روش‌های استاندارد پیاده‌سازی می‌کند. ماژول‌ها ممکن است شامل چندین کلاس باشند که از شنونده ANTLR به ارث می برند. بسته‌های فرعی در این بسته حاوی بازسازی‌هایی هستند که در مرحله اولیه توسعه یا نسخه منسوخ شده یک بازسازی موجود هستند. این بسته در حال توسعه و آزمایش است. ماژول موجود در بسته های ریشه می‌تواند برای اهداف آزمایشی استفاده شود. جهت آشنایی با نحوه پیاده‌سازی بازسازی‌ها به پیوست‌ها مراجعه کنید.

۴-۲-۶- بسته refactoring_design_patterns

در این بسته، فایل‌ها و کدهای مربوط به اعمال الگوهای طراحی مختلف بر روی کد منبع قرار دارد. این بسته در حال توسعه و آزمایش است و در نسخه‌های آینده CodART منتشر خواهد شد.

۴-۲-۷- بسته smells

بسته بویایی تشخیص خودکار کد نرم‌افزار و بوهای طراحی مربوط به عملیات بازسازی که توسط

CodART پشتیبانی می‌شود را پیاده‌سازی می‌کند. هر بو مربوط به یک یا چند بازسازی در بسته بازسازی است. این بسته در حال توسعه و آزمایش است و در نسخه‌های آینده CodART منتشر خواهد شد. در حال حاضر از کتابخانه jDeodorant استفاده می‌شود.

۴-۲-۸- بسته metrics

این بسته شامل چندین فایل هستند که محاسبات شناخته شده ترین معیارهای کد منبع را پیاده‌سازی می‌کنند. این معیارها برای تشخیص بوی کد و اندازه‌گیری کیفیت نرم‌افزار استفاده می‌شوند.

۴-۲-۹- بسته sbse

واژه sbse مخفف Search-Based Software Engineering که حوزه تحقیقاتی اصلی این پروژه است، می‌باشد. در این بسته الگوریتم‌های ژنتیک تک‌هدفه و چندهدفه قرار دارند.

۴-۳ نحوه پیدا کردن مرجع‌ها

یکی از ساده‌ترین بازسازی‌ها که در اکثر ویرایشگرهای کد وجود دارد بازسازی تغییر نام[۱]است. در قطعه کد زیر اگر بخواهیم متغیر a را تغییر نام دهیم به number نیاز است تا هرجایی a استفاده شده است به number تغییر پیدا کند. با ترکیب کردن ابزارهای ANTLR و Understand این کار به‌سرعت و بادقت انجام می‌شود.

[۱] Reference

```
public class Adder {
    int a, b, result;
    public Adder(int a, int b){
        this.a = a;
        this.b = b;
    }

    public int doAdd(){
        result = a + b;
        return result;
    }
}
```

بخش پیمایش درخت تجزیه و تغییر نام توسط ANTLR انجام می‌شود و بخش پیدا کردن مراجع توسط Understand صورت می‌گیرد. (به کد زیر دقت کنید.)

```
class RenameVariableListener(JavaParserLabeledListener):
    def __init__(self, current_name: str, new_name: str, lines: list,
rewriter: TokenStreamRewriter):
        self.current_name = current_name
        self.new_name = new_name
        self.lines = lines
        self.rewriter = rewriter

    def enterExpression1(self, ctx:
JavaParserLabeled.Expression1Context):
        if ctx.IDENTIFIER().getText() == self.current_name:
            if ctx.start.line in self.lines:
                self.rewriter.replaceSingleToken(
                    token=ctx.stop,
                    text=self.new_name
                )
                self.lines.remove(ctx.start.line)

    def enterPrimary4(self, ctx: JavaParserLabeled.Primary4Context):
        if ctx.IDENTIFIER().getText() == self.current_name:
            if ctx.start.line in self.lines:
                self.rewriter.replaceSingleToken(
                    token=ctx.stop,
                    text=self.new_name
                )
                self.lines.remove(ctx.start.line)

    def enterVariableDeclaratorId(self, ctx:
JavaParserLabeled.VariableDeclaratorIdContext):
        if ctx.IDENTIFIER().getText() == self.current_name:
            if ctx.start.line in self.lines:
                self.rewriter.replaceSingleToken(
                    token=ctx.stop,
                    text=self.new_name
                )
                self.lines.remove(ctx.start.line)
```

در ادامه این کد داریم:

```
db = und.open("C:/Users/aliay/Desktop/playground/src/undserstand.und")
file_path = "C:/Users/aliay/Desktop/playground/src/Adder.java"
usages = []
variable = db.lookup("a", "Variable")[0]
for reference in variable.refs():
    usages.append(reference.line())
stream = FileStream(file_path)
lexer = JavaLexer(stream)
tokens = CommonTokenStream(lexer)
parser = JavaParserLabeled(tokens)
tree = parser.compilationUnit()
rewriter = TokenStreamRewriter(tokens)
listener = RenameVariableListener(current_name="a", new_name="number",
lines=usages, rewriter=rewriter)
ParseTreeWalker().walk(listener, tree)

with open(file_path, "w") as fp:
    fp.write(listener.rewriter.getDefaultText())
```

این برنامه، با استفاده از Understand خطوط مراجع را در یک آرایه می‌ریزد و آن را به پیمایشگر ANTLR می‌دهد. سپس ANTLR عملیات تغییر نام را انجام می‌دهد. در پایان قطعه کد جاوا به‌صورت زیر، به‌درستی تغییر پیدا می‌کند.

```
public class Adder {
    int number, b, result;
    public Adder(int a, int b){
        this.number = a;
        this.b = b;
    }

    public int doAdd(){
        result = number + b;
        return result;
    }
}
```

۴-۴ پیاده‌سازی سنجه‌های کیفیت نرم‌افزار

همان‌طور که در فصل دوم شرح داده شد از سنجه‌های نرم‌افزاری مربوط به کیفیت نرم‌افزار تحت عنوان QMOOD برای ارزیابی کیفیت کد منبع استفاده شده است. این سنجه‌ها در بسته metrics و فایل qmood.py محاسبه می‌شوند.

همان‌طور که پیش‌تر توضیح داده شد برای سنجه‌هایی که در سطح کلاس هستند، میانگین حسابی گرفته می‌شود. تابع زیر همین موضوع را پیاده‌سازی می‌کند. ورودی آن تابع مربوط به محاسبه سنجه در سطح کلاس است.

```
def get_class_average(self, class_level_metric):
        scores = []
        for ent in self.known_class_entities:
            class_metric = class_level_metric(ent.longname())
            scores.append(class_metric)
        return sum(scores) / len(scores)
```

به‌عنوان مثال سنجه NOM[1] را در نظر بگیرید. برای محاسبه سنجه از کتابخانه Understand استفاده می‌شود. این یک سنجه در سطح کلاس است بنابراین تابعی نوشته شده تا بر روی کلاس ورودی این سنجه را حساب کند.

```
def ClassLevelNOM(self, class_longname):
        """
        NOM - Class Level Number of Methods
        :param class_longname: The longname of a class. For examole:
package_name.ClassName
        :return: Number of methods declared in a class.
        """
        class_entity = self.get_class_entity(class_longname)
        if class_entity:
            return
class_entity.metric(['CountDeclMethod']).get('CountDeclMethod', 0)
        return 0
```

سپس بر روی تمام کلاس‌ها میانگین حسابی گرفته می‌شود.

```
@property
@divide_by_initial_value
def NOM(self):
    """
    NOM - Number of Methods
    :return: AVG(All class's NOM)
    """
    return self.get_class_average(self.ClassLevelNOM)
```

نماد @property باعث می‌شود تا این تابع به‌عنوان یک صفت از کلاس عمل کند و نماد

[1] Number of Methods

@divide_by_initial_value باعث می‌شود مقدار نهایی تقسیم بر مقدار اولیه این سنجه شود تا مقدار نسبی آن به دست آید. داشتن مقدار نسبی در الگوریتم تکاملی برای همگرا شدن مسئله بهتر است. مقدار اولیه برای پروژه‌های مختلف حساب شده‌اند و در داخل فایلی ذخیره شده‌اند. (جدول ۱-۵)

```
def divide_by_initial_value(func):
    def wrapper(*args, **kwargs):
        value = func(*args, **kwargs)
        initial = CURRENT_METRICS.get(func.__name__)
        return round(value / initial, 2)
    return wrapper
```

در هدف‌های الگوریتم تکاملی از سنجه‌های QMOOD استفاده شده است. این محاسبات در داخل بسته sbse و فایل objectives.py پیاده‌سازی شده‌اند. در سازنده کلاس Objectives سنجه‌های QMOOD داخل متغیر ذخیره می‌شوند. برای هر هدف یک تابع نوشته شده است و طبق فرمول‌های جدول ۳-۲ کدنویسی شده است. به‌عنوان مثال برای هدف reusability داریم:

```
@property
    def reusability(self):
        """
        A design with low coupling and high cohesion is easily reused by
other designs.
        :return: reusability score
        """
        self._reusability = -0.25 * self.DCC + 0.25 * self.CAMC + 0.5 *
self.CIS + 0.5 * self.DSC
        return self._reusability
```

۵-۴ پیاده‌سازی الگوریتم‌های تکاملی

برای پیاده‌سازی الگوریتم‌های تکاملی از چهارچوب pymoo استفاده شده است [۳۱]. چهارچوب pymoo این امکان را به برنامه‌نویسان و توسعه دهندگان می‌دهد تا مسائل خود را تعریف کنند و الگوریتم‌های مختلف تکاملی را اجرا کنند. در بسته sbse و فایل search_based_refactoring می‌توانید الگوریتم‌های تکاملی را اجرا و آزمایش کنید.

۴-۵-۱- مقداردهی اولیه

کدهای مربوط به ساختن جمعیت اولیه داخل sbse\initialize.py قرار دارد. در این فایل مقداردهی های مختلف مانند تصادفی، شبه تصادفی و استفاده از خروجی ابزار jDeodorant قرار دارد. هر تابع مسئول مقداردهی یک بازسازی است و خروجی هر تابع به ترتیب، تابع اصلی بازسازی، ورودی‌ها و نام بازسازی است. به‌عنوان مثال کد زیر برای مقداردهی اولیه بازسازی «غیرایستا کردن صفت کلاس» نوشته شده است. در این کد، از میان تمام صفت‌های ایستا، یک صفت تصادفی انتخاب می‌شود.

```
def init_make_field_non_static(self):
        """
        Finds all static fields and randomly chooses one of them
        :return: refactoring main method and its parameters.
        """
        refactoring_main = make_field_non_static.main
        params = {"udb_path": self.udb_path}
        candidates = self._static_variables
        params.update(random.choice(candidates))
        params.pop("source_package")
        return refactoring_main, params, 'Make Field Non-Static'
```

برای سایر بازسازی‌ها نیز به همین ترتیب تابع موردنظر نوشته می‌شود. برای تولید کل جمعیت کافی است در یک حلقه تکرار به‌اندازه جمعیت، توابع مقداردهی اولیه را صدا بزنیم. در کد زیر جمعیت اولیه ساخته می‌شود.

```
def select_random(self):
        """
        Randomly selects a refactoring. If there are no candidates it
tries again!

        Returns:
            main_function: function
            params: dict
            name: str
        """
        initializer = random.choice(self.initializers)
        main_function, params, name = handle_index_error(initializer)()
        if main_function is None:
            return self.select_random()
        else:
            return main_function, params, name

    def generate_population(self):
        population = []
        for _ in progressbar.progressbar(range(self.population_size)):
            individual = []
            individual_size = random.randint(self.lower_band,
self.upper_band)
            for j in range(individual_size):
                individual.append(
                    self.select_random()
                )
            population.append(individual)
        self._und.close()
        logger.debug("Database closed after initialization.")
        return population
```

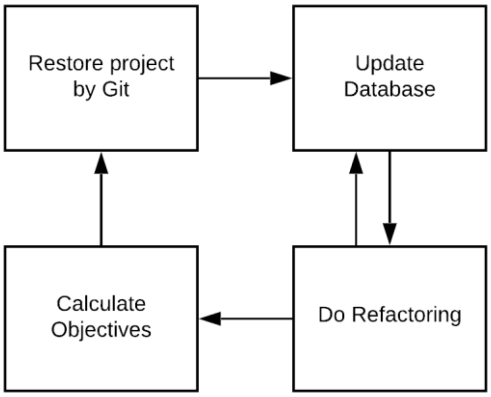
همان‌طور که پیش‌تر توضیح داده شد، ساختار جمعیت یک آرایه دوبعدی (ماتریس) از بازسازی‌ها است؛ بنابراین برای ایجاد آن به دو حلقه تکرار نیاز داریم.

۴-۵-۲- ترتیب اجرای الگوریتم

از آنجایی که این پروژه کد منبع را تغییر می‌دهد بنابراین باید در اجرای بازسازی‌ها ترتیب اجرا را هم در نظر داشت. قبل از اجرای بازسازی با استفاده از ابزار git [۳۲] که مخصوص مدیریت ورژن کد منبع طراحی شده است، حالت پروژه تحت آزمایش را به حالت اولیه بر می‌گردانیم تا تغییراتی که توسط عضو

قبلی جمعیت ایجاد شده خنثی شود. سپس جدول نمادها نیز منطبق با حالت اولیه می‌کنیم. سپس هر ژن (بازسازی) اجرا می‌شود (کد منبع را تغییر می‌دهد) و بنابراین نیاز است تا دوباره جدول نمادها به‌روزرسانی شود. پس از اجرای تمامی بازسازی‌ها، مقادیر هدف (معیارهای کیفیتی QMOOD) محاسبه می‌شود و در یک آرایه ذخیره می‌شود. در نهایت پس اجرای کامل نسل، مقادیر به دست آمده به الگوریتم NSGA-III داده می‌شود.

این ترتیب اجرا در تابع evaluate کلاس ProblemManyObjective پیاده‌سازی شده است. فرایند اجرا در شکل ۴-۲ ترسیم شده است.



شکل ۲-۴ ترتیب اجرای الگوریتم

```
def _evaluate(self, x, out, *args, **kwargs):
        objective_values = []
        for k, individual_ in enumerate(x):
            # Stage 0: Git restore
            logger.debug("Executing git restore.")
            git_restore(config.PROJECT_PATH)
            logger.debug("Updating understand database after git
restore.")
            update_understand_database(config.UDB_PATH)

            # Stage 1: Execute all refactoring operations in the sequence
x
            logger.debug(f"Reached an Individual with size
{len(individual_[0])}")
            for refactoring_operation in individual_[0]:
                refactoring_operation.do_refactoring()
                # Update Understand DB
                logger.debug(f"Updating understand database after
{refactoring_operation.name}.")
                update_understand_database(config.UDB_PATH)
            arr = [0, 1, 2, 3, 4, 5]
            # Stage 2: Computing quality attributes
            qmood = Objectives(udb_path=config.UDB_PATH)
            arr[0] = qmood.reusability
            arr[1] = qmood.understandability
            arr[2] = qmood.flexibility
            arr[3] = qmood.functionality
            arr[4] = qmood.effectiveness
            arr[5] = qmood.extendability
            del qmood
            # Stage 3: Marshal objectives into vector
            ov = [-1 * i for i in arr]
            objective_values.append(ov)
            logger.info(f"Objective values for individual [1]: {ov}")

        # Stage 4: Marshal all objectives into out dictionary
        out['F'] = np.array(objective_values, dtype=float)
```

فصل ۵ ارزیابی

از سه پروژه متن‌باز با اندازه‌های کوچک و بزرگ جهت آزمایش و ارزیابی سامانه CodART استفاده شده است که اطلاعات این سه پروژه در جدول ۱-۵ دیده می‌شود. پروژه‌های استفاده شده، متن‌باز و قابل دانلود هستند. کد منبع پروژه‌های فوق را می‌توان از مرجع داده شده دانلود و استفاده کرد.

جدول ۵-۱ پروژه‌های استفاده شده جهت ارزیابی

نام پروژه	نسخه	تعداد کلاس	مرجع کد منبع
JSON	1.1	26	https://github.com/stleary/JSON-java
JOpenChart	0.94	46	http://jopenchart.sourceforge.net/
JVLT	1.3.2	420	http://jvlt.sourceforge.net/

اجرا و ارزیابی‌ها بر روی یک رایانه سرور به مشخصات لیست شده در جدول ۲-۵ انجام شد. همچنین مقادیر تنظیمات برای فرایند تکاملی هر سه اجرا در جدول ۳-۵ مکتوب شده است. در همه اجراها از فرایند تکاملی چندهدفه NSGA-III با مقدار احتمال جهش برابر با ۰٫۱ و مقدار احتمال ترکیب برابر با ۰٫۸ استفاده شده است.

جدول ۵-۲ مشخصات سرور و زمان اجرا

شماره اجرا	نام پروژه	پردازنده	حافظه	زمان اجرا (ساعت)
۱	JSON	Intel Core i7 2.8 GHz	12 GB	1.13
۲	JSON	Intel Core i7 2.8 GHz	12 GB	31.43
۳	JOpenChart	Intel Xeon 2.4 GHz	6 GB	2.69
۴	JVLT	Intel Core i7 2.8 GHz	12 GB	57.51

جدول ۵-۳ تنظیمات فرایند تکاملی

شماره اجرا	نام پروژه	اندازه جمعیت	کمینه اندازه جواب	بیشینه اندازه جواب	حداکثر تکرار مجاز
۱	JSON	20	10	15	10
۲	JSON	50	10	50	50
۳	JOpenChart	20	15	20	20
۴	JVLT	50	10	50	50

۵-۲ سنجه‌ها و هدف‌های اولیه

همان‌طور که در فصل‌های گذشته بیان شد، از سنجه‌های نرم‌افزاری QMOOD برای محاسبه هدف‌های کیفیتی نرم‌افزار استفاده می‌شود. برای اینکه فرایند تکاملی به همگرایی برسد نیاز است تا این مقادیر به‌صورت نسبی به فرایند داده شود. برای پیاده‌سازی این امر ابتدا سنجه‌ها و اهداف قبل از اجرای سامانه محاسبه می‌شوند و در هر مرحله مقدار جدید محاسبه و بر مقدار اولیه تقسیم می‌شود تا مقدار نسبی آن به دست آید.

در جدول ۵-۴ مقدار اولیه سنجه‌های QMOOD برای هر دو پروژه و در جدول ۵-۵ مقدار اولیه اهداف محاسبه و درج شده است.

جدول ۵-۴ مقادیر اولیه سنجه‌های QMOOD

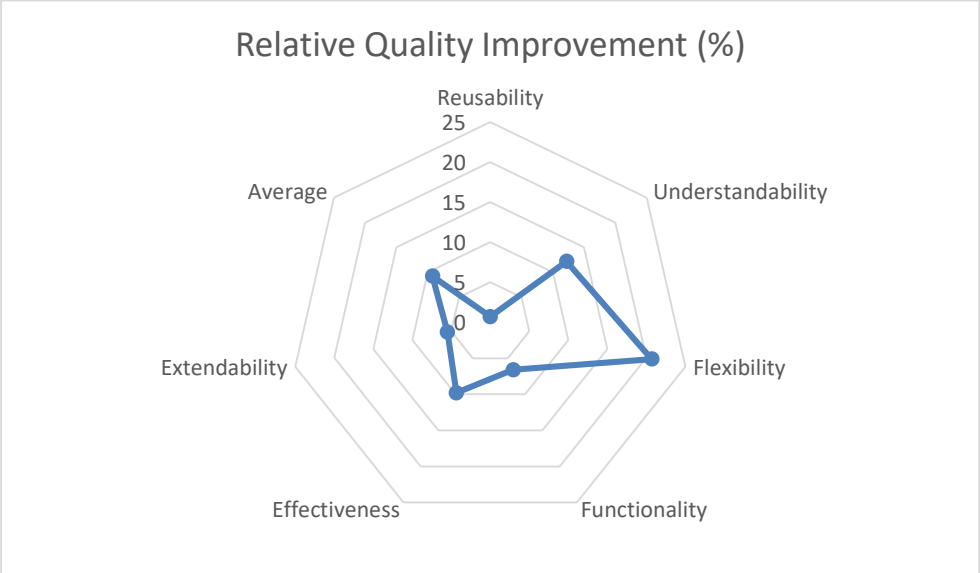
Design metric	DSC	NOH	ANA	MOA	DAM	CAM	CIS	NOM	DCC	MFA	NOP
Design property	Design size	Hierarchies	Abstraction	Composition	Encapsulation	Cohesion	Messaging	Complexity	Coupling	Inheritance	Polymorphism
JSON	26	2	0.5	1.2	0.30	0.84	9.5	10.9	2.86	0.09	6.56
Jvlt	420	17	0.80	2.4	0.61	0.74	4.02	4.89	2.55	0.22	4.13
JOpenChart	46	4	0.69	0.92	0.52	0.57	8.02	8.61	2.35	0.27	8.07

جدول ۵-۵ مقادیر اولیه هدف‌های الگوریتم تکاملی

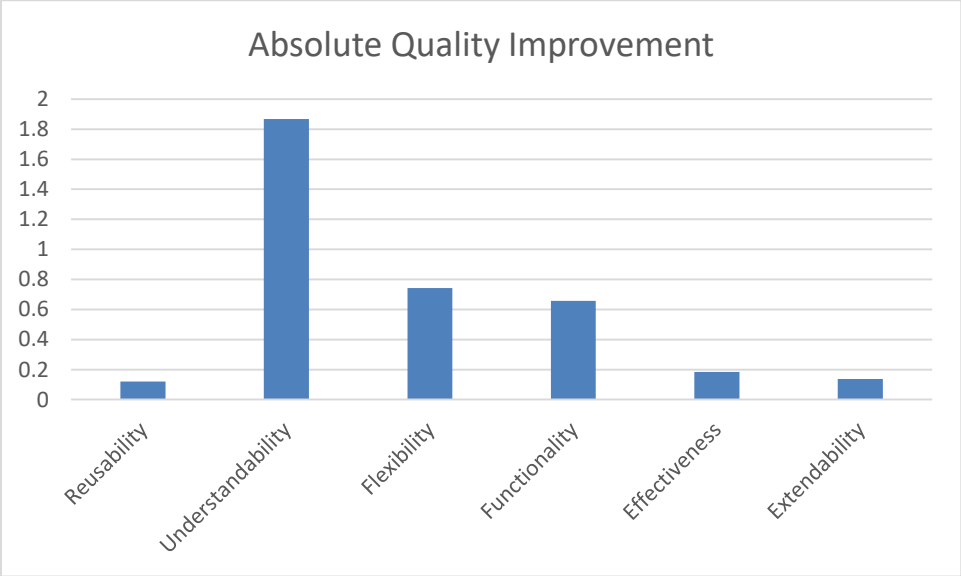
	reusability	flexibility	understandability	functionality	extendability	effectiveness
JSON	17.24	3.59	-15.30	9.95	2.49	1.87
Jvlt	211.55	2.77	-142.24	98.02	1.30	1.63
JOpenChart	26.56	4.04	-21.33	14.61	3.34	2.09

۵-۳ بهبود کیفیت پروژه JSON

از آنجایی که پروژه JSON یک پروژه با اندازه کوچک محسوب می‌شود، ابتدا با اندازه جمعیت ۲۰ و اندازه پاسخ ۱۰ تا ۱۵ که مقادیر کوچکی هستند، اجرا شد. این اجرا حدود ۱ ساعت به طول انجامید. مقادیر نسبی بهبود اهداف در شکل ۵-۱ و مقادیر مطلق اهداف پس از اجرا در شکل ۵-۲ ترسیم شده است.



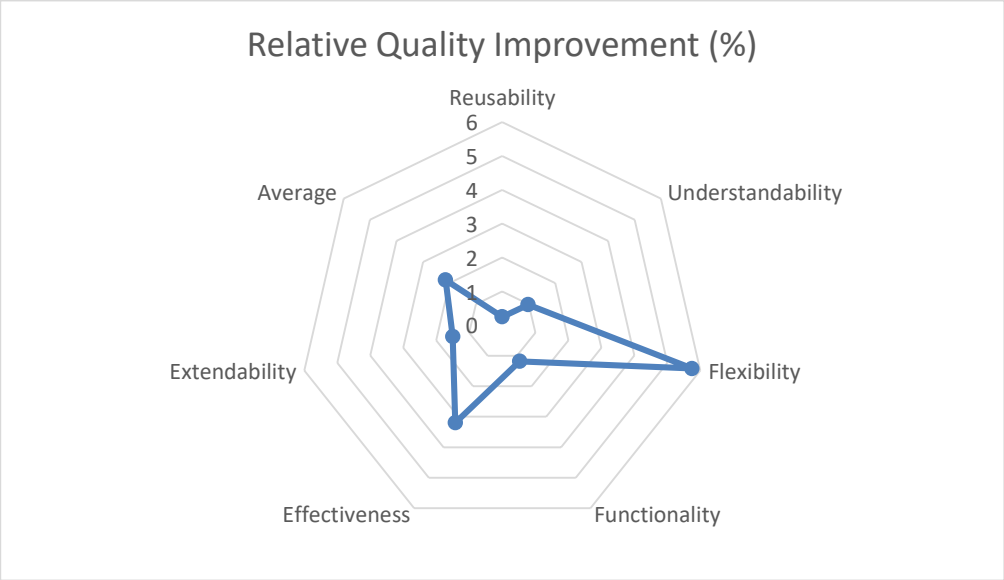
شکل ۵-۱ مقادیر نسبی بهبود اهداف در پروژه JSON



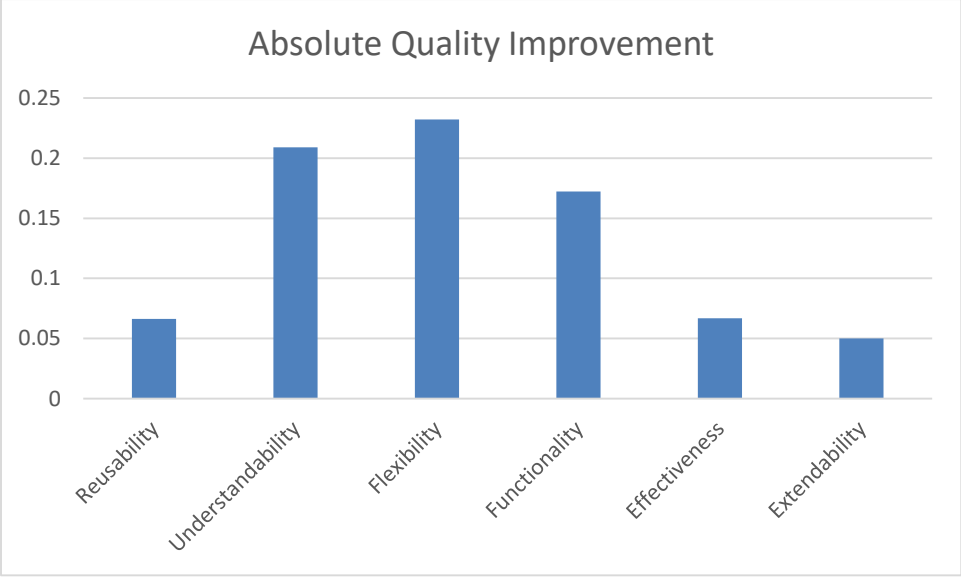
شکل ۵-۲ مقادیر مطلق بهبود اهداف در پروژه JSON

۵-۴ بهبود کیفیت پروژه JOpenChart

پروژه JOpenChart یک پروژه با اندازه کوچک محسوب می‌شود اما این پروژه نسبت به پروژه JSON کمی بزرگ‌تر می‌باشد؛ بنابراین با اندازه جمعیت ۲۰ و اندازه پاسخ ۱۵ تا ۲۰ که مقادیر کوچکی هستند، اجرا شد. این اجرا حدود ۲ ساعت به طول انجامید. مقادیر نسبی بهبود اهداف در شکل ۵-۳ و مقادیر مطلق اهداف پس از اجرا در شکل ۵-۴ ترسیم شده است.



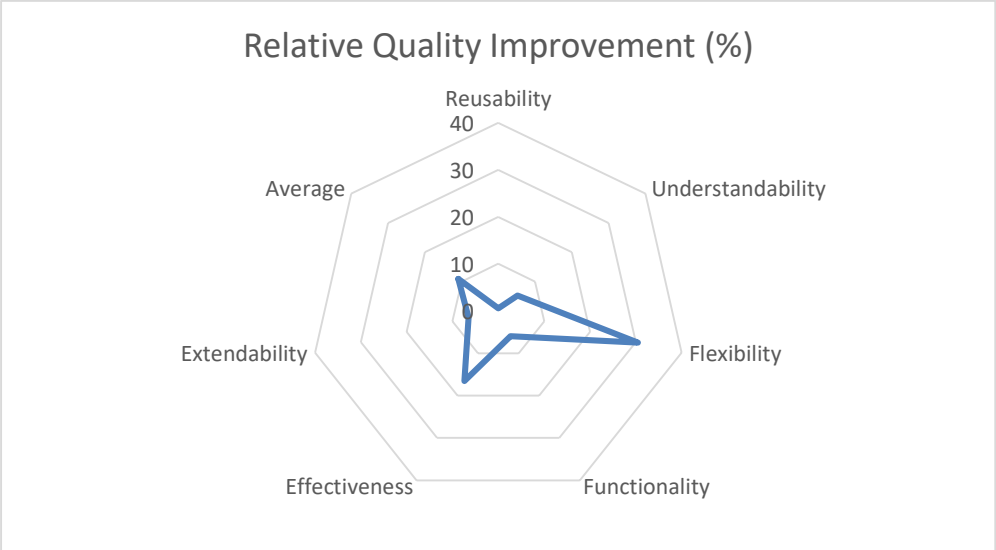
شکل ۵-۳ مقادیر نسبی بهبود اهداف در پروژه JOpenChart



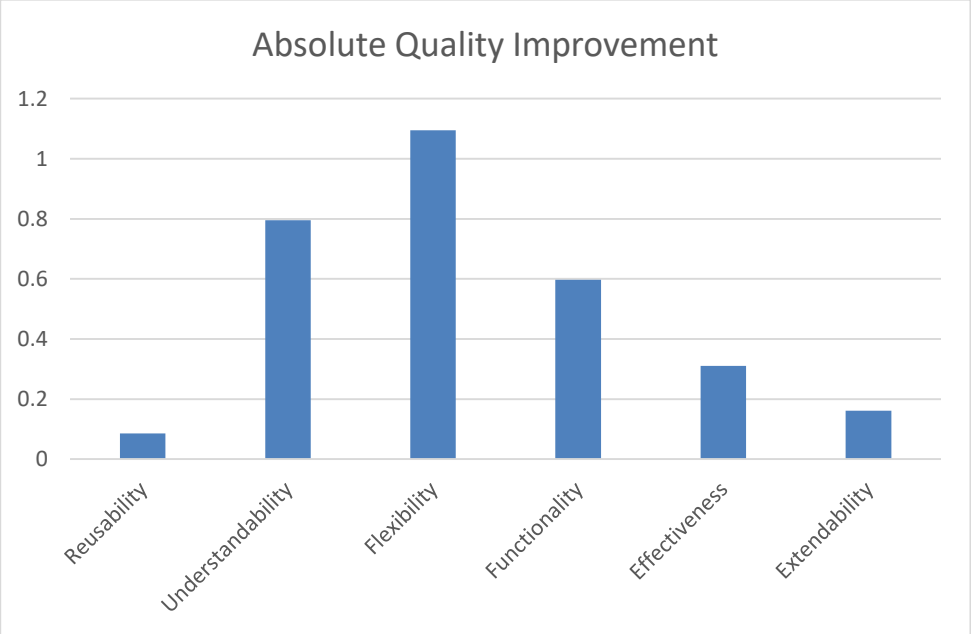
شکل ۵-۴ مقادیر مطلق بهبود اهداف در پروژه JOpenChart

۵-۵ بررسی تاثیر طول توالی پاسخ

در اجرای دوم، همان پروژه JSON اما با اندازه جمعیت ۵۰ و اندازه پاسخ بین ۱۰ تا ۵۰ قرار داده شد. این اجرا حدود ۳۱ ساعت به طول انجامید. مقادیر نسبی بهبود اهداف در شکل ۵-۵ و مقادیر مطلق آن در شکل ۵-۶ درج شده است.

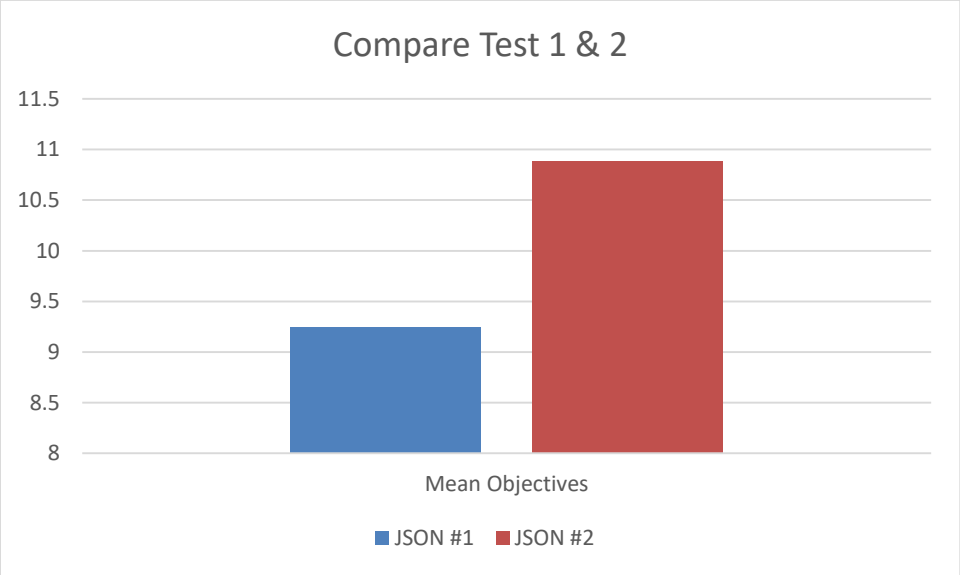


شکل ۵-۵ مقادیر نسبی بهبود اهداف در پروژه JSON اجرای دوم



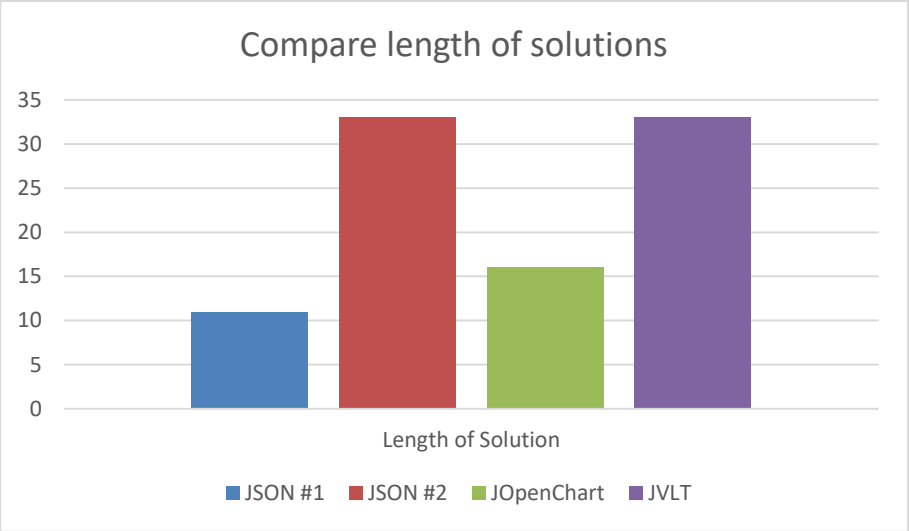
شکل ۵-۶ مقادیر مطلق بهبود اهداف در پروژه JSON اجرای دوم

در اجرای شماره ۱ و شماره ۲، پروژه انتخاب شده جهت آزمایش یکسان بود (JSON) اما اندازه جمعیت و اندازه پاسخ را افزایش دادیم. این امر منجر شد تا به صورت میانگین کیفیت نرم‌افزار بیشتر بهبود یابد. از مقایسه این دو آزمایش می‌توان نتیجه گرفت هر چه طول پاسخ بیشتر باشد و بازسازی های بیشتری اعمال خواهد شد و احتمال بهبود کیفیت افزایش می‌یابد. در شکل ۵-۷ مقدار میانگین اهداف کیفیت مقایسه شده است.



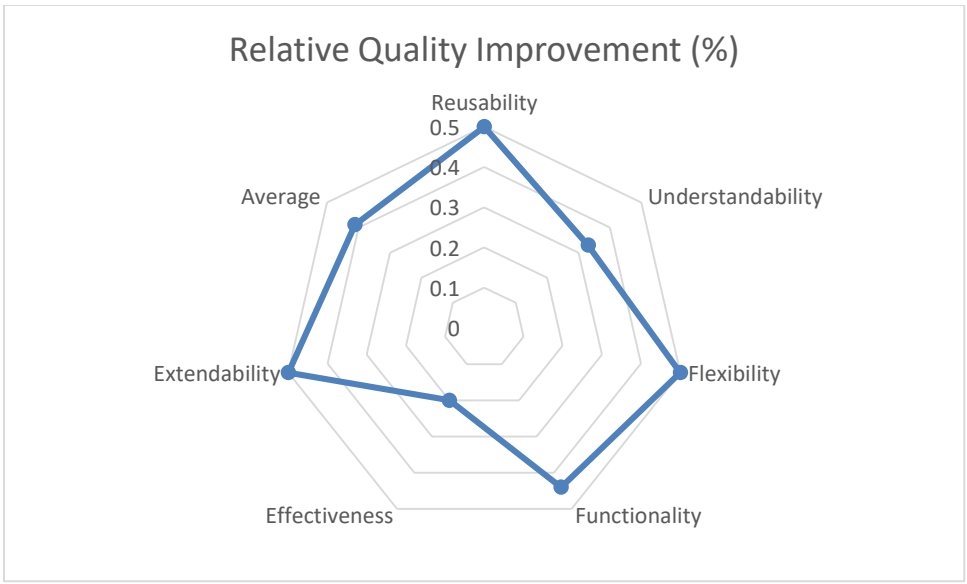
شکل ۵-۷ بهبود میانگین اهداف در اثر افزایش طول پاسخ

در شکل ۵-۸ طول بهترین پاسخ هر اجرا ترسیم شده است. مشاهده می‌شود که طول پاسخ اجرای دوم ۳ برابر طول پاسخ اجرای اول است که نتیجه گرفته شده را تایید می‌کند.

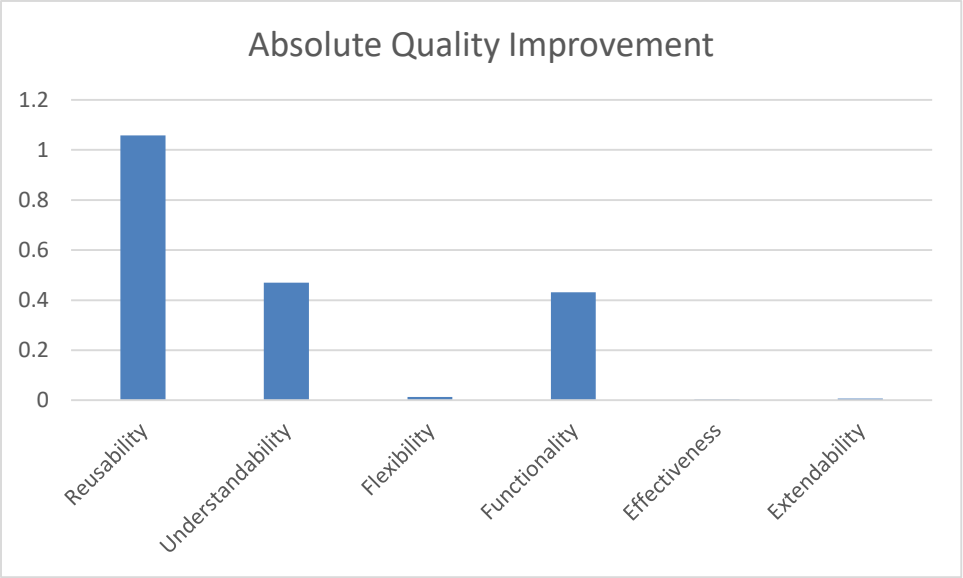


شکل ۸-۵ مقایسه طول پاسخ هر اجرا

پروژه JVLT یک پروژه با اندازه بزرگ محسوب می‌شود، لذا با اندازه جمعیت ۵۰ و اندازه پاسخ ۱۰ تا ۵۰ اجرا شد. مقادیر نسبی بهبود اهداف در شکل ۵-۹ و مقادیر مطلق اهداف پس از اجرا در شکل ۵-۱۰ ترسیم شده است.



شکل ۹-۵ مقادیر نسبی بهبود اهداف در پروژه JVLT



شکل ۱۰-۵ مقادیر مطلق بهبود اهداف در پروژه JVLT

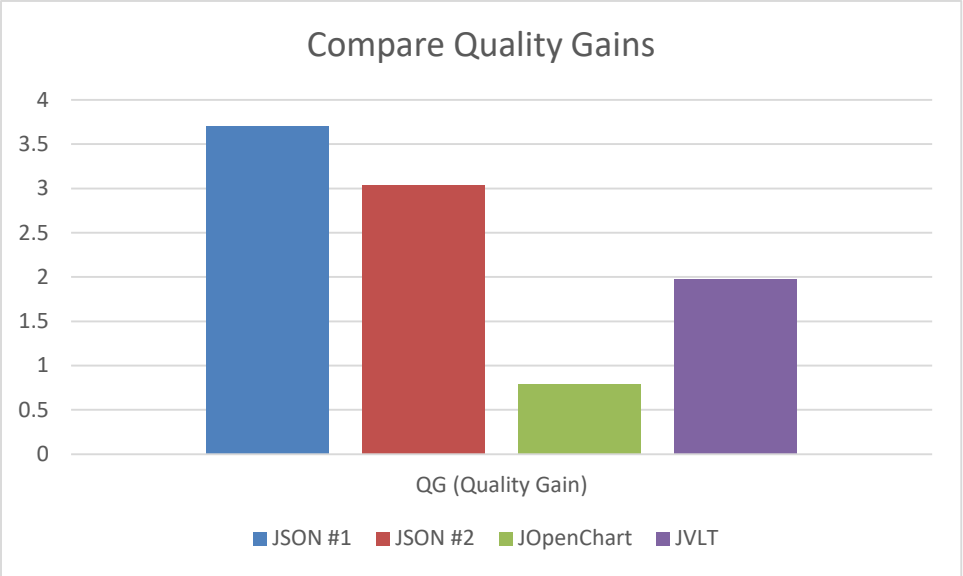
همانطور که مشاهده می‌شود، اگر چه سامانه تلاش کرده است تا کیفیت پروژه را بهبود دهد اما میزان افزایش کیفیت چشمگیر نیست. دلیل این امر این است که طول جمعیت و پاسخ ۵۰ برای این پروژه بزرگ ناکافی بوده است.

۵-۶ عملکرد کلی سامانه

برای ارزیابی عملکرد کلی سامانه از مفهومی تحت عنوان «افزایش کیفیت» استفاده شده است. این مقدار بر اساس فرمول زیر بدست می‌آید:

$$Quality\ Gain\ or\ QG = \sum_{i=1}^{6}(q_i' - q_i)$$

که در آن q_i' برابر است با مقادیر مطلق اهداف پس از اجرای فرایند و q_i برابر است با مقادیر اولیه اهداف قبل از شروع فرایند. نمودار افزایش کیفیت برای ۴ اجرای فوق در شکل ۵-۱۱ رسم شده است.



شکل ۱۱-۵ مقایسه عملگر افزایش کیفیت

هنگامی از بهبود کیفیت یک نرم‌افزار صحبت می‌شود، بدون شک بهترین حالت این است که تمامی جنبه های کیفیتی نرم‌افزار بهبود یابد. همانطور که مشاهده می‌شود و انتظار داشتیم، فرایند تکاملی NSGA-III سعی کرده است تمامی اهداف را با هم بهبود دهد. اما دیدیم بعضی از صفات کیفیتی نرم‌افزار مانند کارایی یا functionality بیشتر و بهتر بهبود یافته است. این امر به این دلیل است که اکثر بازسازی های اجرا شده سعی در تغییر ساختار و کارآیی نرم‌افزار داشته اند.

فصل ۶ نتیجه گیری و کارهای آتی

۶-۱ نتیجه گیری

ما یک راهکار جدید (اعمال مستقیم بازسازی بر روی کد منبع) از مسئله جست و جو بازسازی خودکار را به عنوان یک مسئله بهینه سازی چند هدفه، بر اساس فرایند تکاملی NSGA-III، با استفاده از سنجه‌های کیفیتی QMOOD به عنوان اهداف سامانه به همراه طیف وسیعی از بازسازی ها و حفظ انسجام طراحی پیشنهاد کردیم.

این پایان نامه یکی از اولین کاربردهای دنیای واقعی فرایند NSGA-III را نشان می دهد. این بررسی تجربی اولیه نشان داد که یک تعارض احتمالی بین اهداف QMOOD ممکن است بسته به نوع بازسازی‌های بکار گرفته شده رخ دهد. در این زمینه، برای اثبات صحت این بینش، باید یک تحلیل آماری وسیع با تعداد آزمایش‌های زیاد انجام شود. علاوه بر این، ما هیچ هدف (هایی) را اولویت بندی نکرده ایم و برای همه اهداف وزن برابر و یکسان در نظر گرفتیم تا به بهبود کلی کیفیت کمک کرده باشیم.

این پروژه بر روی سه پروژه متن باز ارزیابی شد و دیدیم که برای گرفتن بهترین عملکرد، می‌بایست اندازه جمعیت اولیه و اندازه توالی پاسخ متناسب با پروژه تنظیم شود. در غیر اینصورت، اگر چه کیفیت پروژه ورودی افزایش می‌یابد اما ممکن میزان افزایش چشم‌گیر نباشد و به عبارتی دیگه الگوریتم جست‌وجو به جواب های اصلی همگرا نشود.

۶-۲ کار های آتی

این مطالعه صرفاً به بازسازی خودکار با فرایند تکاملی NSGA-III و اعمال آن در سطح کد می‌پردازد، اما این ابزار جامع طراحی شده است و گنجایش ارائه دادن قابلیت‌های خیلی بیشتری را دارد. در مطالعات آتی در این زمینه، تغییرات زیر پیشنهاد می‌شود.

- توسعه و استفاده فرایندهای تکاملی متفاوت و اجرا و آزمایش با تنظیمات مختلف.
- توسعه و استفاده از اهداف جدید مانند (ماژولار بودن و تست پذیری).
- توسعه و استفاده از بازسازی‌های بیشتر.
- توسعه انواع بوی کد و شناسایی بهتر فرصت‌های بازسازی.

- توسعه و پیاده‌سازی بازسازی‌های مربوط به الگوهای طراحی.
- بهبود الگوریتم‌های موجود و سعی در افزایش سرعت.

در ادامه این پروژه، برای مستقل شدن آن از ابزار غیر رایگان و خارجی Understand می‌توان آن را مهندسی مجدد کرد و نسخه متن‌باز و رایگان آن را توسعه داد. در ادامه شرحی بر این کار گفته شده است.

۶-۳ آشنایی با پروژه Open Understand

همان‌طور که پیش‌تر بیان شد، برای پیدا کردن مراجع و استفاده‌های موجودیت‌های مختلف از رابط برنامه‌نویسی ابزار Understand استفاده شده است. متأسفانه کد منبع این پروژه در دسترس عموم نیست و تغییر، سفارشی‌سازی و استفاده مجدد در فعالیت‌ها و محیط‌های جدید که در تحقیقات دانشگاهی و صنعتی ظاهر می‌شود را دشوار می‌کند. همین‌طور استفاده از محیط نرم‌افزار و رابط برنامه‌نویسی آن نیازمند تهیه و خرید مجوز استفاده است؛ بنابراین، در راستای ادامه این پروژه قصد مهندسی معکوس و ارائه متن‌باز این سامانه نرم‌افزاری را داریم که نام آن را Open Understand قرار دادیم.

با بررسی و تحلیل داده‌های جمع‌آوری شده توسط Understand متوجه شدیم اکثر داده‌ها در دو مفهوم موجودیت و مرجع ذخیره می‌شوند.

- **موجودیت:** هر چیزی در کد است که Understand اطلاعات مربوط به آن را می‌گیرد: به‌عنوان مثال، یک فایل، یک کلاس، یک متغیر و یک تابع .
- **مرجع:** مکان خاصی که یک موجودیت در کد ظاهر می‌شود. یک مرجع همیشه به‌عنوان رابطه بین دو موجودیت تعریف می‌شود.

جزئیات بیشتر این پروژه در پیوست ث درج شده است.

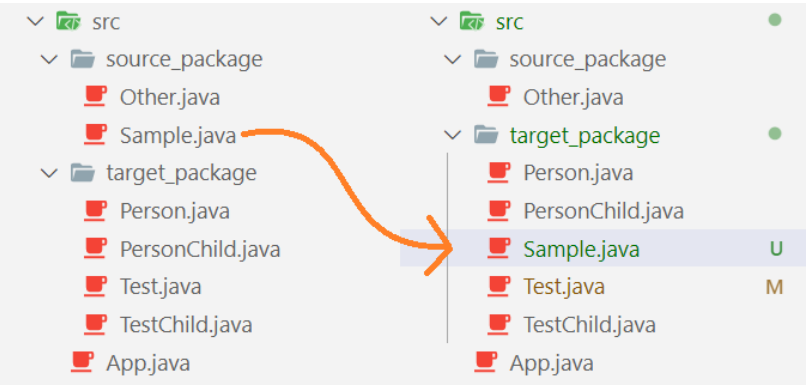
پیوست‌الف: بازسازی انتقال کلاس

تعریف: بازآرایی انتقال کلاس به انتقال یک کلاس از یک بسته به بسته‌ای دیگر گفته می‌شود.

مشکل: ممکن است به دلیل گسترده شدن کدهای برنامه و عدم دقت کدنویس، کلاسی در یک بسته‌ای تعریف شود که استفاده عملی زیادی در آن بسته ندارد. در این صورت بهتر است آن کلاس به بسته‌ای انتقال یابد که بیشترین استفاده را دارد. همچنین طبق اصول کدنویسی تمیز بهتر است یک بسته بیشتر از ده کلاس را شامل نشود. در صورت بزرگ‌شدن یک بسته بهتر است برخی کلاس‌ها را به یک بسته دیگر یا یک بسته جدید (در صورت نیاز) انتقال داد.

راه‌حل: حذف کلاس از بسته مبدأ و انتقال آن به بسته مقصد.

دلیل: کلاس‌ها اغلب در بسته‌هایی نزدیک به محل استفاده آن ایجاد می‌شوند، این موضوع می‌تواند تا زمانی که کلاس شروع به استفاده مجدد توسط سایر بخش‌های محصول کند، منطقی باشد. بسته موردنظر نیز ممکن است خیلی بزرگ شده باشد. اغلب بهتر است به این کلاس به بسته‌ای که بیشتر از نظر شکل یا عملکرد به آن مرتبط است منتقل شود. این کار می‌تواند به حذف وابستگی‌های پیچیده سطح بسته کمک کند و پیدا کردن و استفاده مجدد از کلاس‌ها را برای توسعه دهندگان آسان‌تر کند. اگر وابستگی‌های زیادی برای کلاس در بسته خودش وجود داشته باشد، می‌توان ابتدا از بازسازی استخراج کلاس برای جداکردن بخش‌های مربوطه استفاده کرد. به‌عنوان مثال، شکل الف-۱ نمونه‌ای از بازآرایی انتقال کلاس را نشان می‌دهد.



شکل الف-۱ نمونه‌ای از بازآرایی انتقال کلاس

جدول الف-۱ جزئیات و مراحل انجام بازآرایی انتقال کلاس را نشان می‌دهد. همچنین شکل الف-۲ شبه کد اعمال خودکار این بازآرایی را بیان می‌کند. این الگوریتم بسته مبدأ، بسته مقصد و نام کلاس را به‌عنوان ورودی دریافت می‌کند و کلاس موردنظر را به بسته مقصد انتقال می‌دهد و تغییرات حاصل از انتقال را در سطح برنامه منتشر می‌کند.

در ابتدا قبل از اینکه تغییری در برنامه ایجاد شود، لیستی از فایل‌هایی که از کلاس موردنظر استفاده کرده‌اند را پیدا می‌کند. به دلیل انتقال کلاس تمامی خطوط مربوط به وارد کردن این کلاس باید تغییر کنند. سپس کلاس از بسته مبدأ حذف می‌شود و به بسته مبدأ انتقال می‌یابد.

جدول الف-۱ بازسازی انتقال کلاس

بازآرایی	استخراج کلاس
بو (ها) ی کد مرتبط	کلاس بی‌جا[۱]
تشخیص موقعیت بازآرایی[۲]	به‌صورت تصادفی
ورودی‌ها	نام کلاس، بسته مبدأ، بسته مقصد
خروجی(ها)	حذف کلاس مبدأ و ایجاد آن در کلاس مقصد.
پیش‌شرط‌ها	۱. یکسان نبودن بسته مبدأ و مقصد. ۲. بسته مبدأ یا مقصد نباید بسته ریشه[۳]باشد. ۳. معتبر بودن داده‌های ورودی (در متن کد وجود داشته باشند.). ۴. کلاسی با نام کلاس موردنظر نباید در بسته مقصد وجود داشته باشد.
پس شرط‌ها	۱. تغییر خطوط مربوط به وارد کردن کلاس
مراحل	۱. پیدا کردن فایل‌هایی که از کلاس موردنظر استفاده می‌کنند. ۲. حذف کلاس از بسته مبدأ. ۳. ایجاد همان کلاس در بسته مقصد. ۴. اعمال تغییرات در کد جهت رفع خطاهای کامپایل.
تعداد گذرهای لازم	۲
ملاحظات عملی	فرض شده است که داده‌های ورودی معتبر هستند.

[۱] Misplaced class

[۲] refactoring opportunity

[۳] Root (or default) package

[۴] pass

Algorithm 2. Move Class Refactoring Pseudocode

```
Input: className
Input: sourcePackage
Input: targetPackage
Output: refactored project
1.  // Check pre-conditions
2.  if sourcePackage = targetPackage then
3.      return false; // cannot move to the same package
4.  end if
5.  if sourcePackage = ROOT_PACKAGE or targetPackage = ROOT_PACKAGE then
6.      return false; // cannot move to the root(default) package in java.
7.  end if
8.  if not checkPackageExist(sourcePackage, targetPackage) then
9.      return false; // check if given packages really exists.
10. end if
11. if not checkClassExist(sourcePackage, className) then
12.     return false; // check if given class really exists.
13. end if
14. if checkClassExist(targetPackage, className) then
15.     return false; // cannot move if there is an exact class in the target package
16. end if
17. classReferences ← findReferences(sourcePackage, className)
18. currentClassPath ← generateClassPath(sourcePackage, className)
19. newClassPath ← generateClassPath(targetPackage, className)
20. classContent ← getCassContent(sourcePackage, className)
21. removePath(currentClassPath) // delete class from source package
22. writeFile(newClassPath, classContent) // move class to target package
23. for reference in classReferences do
24.     UpdateImportListener(reference, sourcePackage, targetPackage, className) // ANTLR Listener
25. endfor
```

شکل الف-۲ شبه کد بازسازی انتقال کلاس

به‌عنوان مثال شکل الف-۱را در نظر بگیرید. کلاس Test که در بسته target_package تعریف شده است از کلاس Sample که دربسته source_package استفاده شده است اما در خود بسته source_package هیچ استفاده‌ای از کلاس Sample نشده است؛ بنابراین منطقی است که کلاس Sample را به target_package انتقال دهیم.

```java
package target_package;

import source_package.Sample;

public class Test {
    public Test(){
        Sample sample = new Sample();
        System.out.println("" + sample.toString());
    }
}
```

پس از اعمال بازسازی کلاس علاوه بر عملیات انتقال کلاس، کلاس Test به‌صورت زیر تغییر خواهد

کرد:

```
package target_package;

public class Test {
    public Test(){
        Sample sample = new Sample();
        System.out.println("" + sample.toString());
    }
}
```

مشاهده می‌شود که خط مربوط به وارد کردن کلاس Sample از source_package باید حذف شود، زیرا این کلاس دیگر در source_package وجود ندارد.

برای خودکار انجام‌دادن این بازسازی از ۱ بار پیمایش DFS درخت تجزیه استفاده شده است. همان‌طور که پیش‌تر توضیح داده شده برای به دست آوردن درخت تجزیه و پیمایش آن از ابزار ANTLR استفاده شده است.

کلاس MoveClassAPI

در این کلاس عملیات‌های اصلی مربوط به بازسازی مانند بررسی پیش‌شرط‌ها، انجام بازسازی و سایر عملیات لازم پیاده‌سازی شده است. ابتدا در تابع check_preconditions پیش‌شرط‌های زیر به ترتیب بررسی می‌شوند:

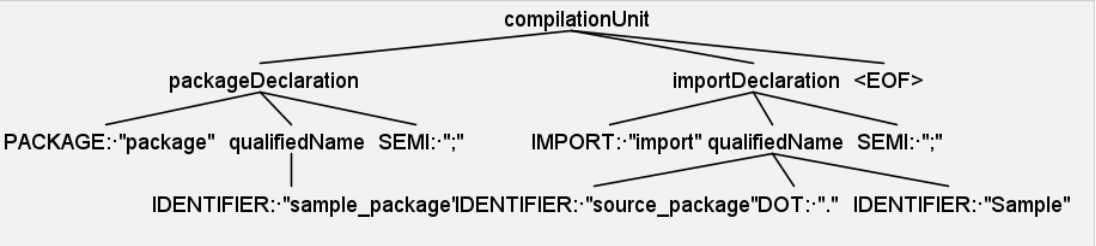
- یکسان نبودن پکیج‌های مبدأ و مقصد.
- پکیج مبدأ یا مقصد نباید پکیج ریشه یا پیش‌فرض جاوا باشد.
- داده‌ها ورودی باید معتبر باشند.

کد این تابع به‌صورت زیر است:

```
if self.source_package == self.target_package:
            logger.error("Source and target packages are same.")
            return False
        if self.source_package == ROOT_PACKAGE or self.target_package ==
ROOT_PACKAGE:
            logger.error("Can not move package to/from root package.")
            return False
        # Get package directories
        source_package_dir, target_package_dir =
self.get_package_directories()
        if source_package_dir is None or target_package_dir is None:
            logger.error("Package entity does not exists.")
            return False
        if not os.path.exists(os.path.join(source_package_dir,
f"{self.class_name}.java")):
            logger.error("Class does not exists in source package.")
            return False
        # Get class directory
        class_dir, class_content, usages = self.get_class_info()
        if class_dir is None or class_content is None:
            logger.error("Class entity does not exists.")
            return False
        new_class_path = os.path.join(target_package_dir,
f"{self.class_name}.java")
        if os.path.exists(new_class_path):
            logger.error("Class already exists in target package.")
            return False
```

کلاس UpdateImportsListener

پس از چک کردن پیش‌شرط‌ها، توسط دستورات کتابخانه سیستم‌عامل زبان پایتون عملیات انتقال کلاس (فایل) انجام می‌شود. پس از انتقال کلاس نیاز هست تا در import های سایر فایل‌های پروژه تغییرات ایجاد شده انجام شود. بدین منظور تمام فایل‌هایی که از کلاس مبدأ استفاده می‌کنند توسط کلاس UpdateImportsListener پیمایش و ویرایش می‌شوند. برای درک بهتر درخت شکل الف-۳ را در نظر بگیرید. کلاس Sample به پکیج target_package منتقل شده است و نیاز است import های مربوط به آن اصلاح شوند.



شکل الف-۳ درخت تجزیه برای اصلاح importها

برای اصلاح import دو حالت می‌تواند رخ بدهد. حالت اول این است که فایل مورد پیمایش داخل همان پکیج مقصد باشد. در این حالت باتوجه‌به قواعد جاوا نیازی به import نیست و کافی است import موردنظر حذف شود. در حالت دوم نیاز داریم تا import اصلاح شود. این دو حالت در enterImportDeclaration بررسی می‌شوند و در exitCompilationUnit بر روی فایل نوشته می‌شوند. کد این بخش را می‌توانید در زیر مشاهده کنید:

```
class UpdateImportsListener(JavaParserLabeledListener):
    def __init__(self, rewriter: TokenStreamRewriter, source_package:
str, target_package: str, class_name: str):
        self.rewriter = rewriter
        self.source_package = source_package
        self.target_package = target_package
        self.class_name = class_name
        self.current_package = None
        self.imported = False
        self.import_loc = None

    def enterPackageDeclaration(self, ctx:
JavaParserLabeled.PackageDeclarationContext):
        self.current_package = ctx.qualifiedName().getText()

    def exitPackageDeclaration(self,
ctx:JavaParserLabeled.PackageDeclarationContext):
        self.import_loc = ctx.stop

    def enterImportDeclaration(self, ctx:
JavaParserLabeled.ImportDeclarationContext):
        # import source_package.Sample;
        if self.target_package in ctx.getText():
            self.imported = True
            if self.class_name in ctx.getText():
                if self.target_package == self.current_package:
                    replace_text = ""
                else:
                    replace_text = f"import
{self.target_package}.{self.class_name};\n"
                self.rewriter.replaceRangeTokens(
                    from_token=ctx.start,
                    to_token=ctx.stop,
                    text=replace_text,
                    program_name=self.rewriter.DEFAULT_PROGRAM_NAME
                )
        elif f"{self.source_package}.{self.class_name}" in ctx.getText():
            self.rewriter.delete(
                program_name=self.rewriter.DEFAULT_PROGRAM_NAME,
                from_idx=ctx.start.tokenIndex,
                to_idx=ctx.stop.tokenIndex
            )
    def exitCompilationUnit(self,
ctx:JavaParserLabeled.CompilationUnitContext):
        if not self.imported and self.current_package !=
self.target_package:
            self.rewriter.insertAfterToken(
                token=self.import_loc,
                text=f"\nimport
{self.target_package}.{self.class_name};\n",
                program_name=self.rewriter.DEFAULT_PROGRAM_NAME
            )
```

پیوست ب: بازسازی استخراج کلاس

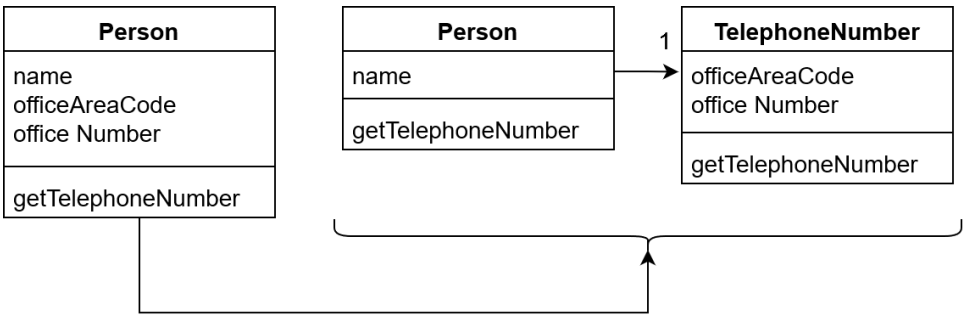
تعریف: بازآرایی استخراج کلاس به انتقال بعضی از توابع و فیلدهای یک کلاس به یک کلاس جدید گفته می‌شود.

مشکل: ممکن است به دلیل گسترده شدن کدهای برنامه و عدم رعایت اصول کدنویسی تمیز، بعضی از کلاس‌ها بیش از حد بزرگ شده و در اصطلاح تشکیل یک کلاس بزرگ می‌دهند. معمولاً در این نوع کلاس‌ها اصل تک مسئولیتی (SRP) رعایت نمی‌شود.

راه‌حل: ایجاد یک کلاس جدید و شکستن کلاس بزرگ به دو کلاس کوچک‌تر.

دلیل: کلاس‌ها همیشه در ابتدا واضح و قابل‌فهم نوشته می‌شوند و سعی بر آن است تا هر کلاس کار مربوط به خود را انجام دهد. اما با گسترش برنامه، به هر کلاس توابع و فیلدهایی به مرور زمان اضافه می‌شوند که در صورت عدم نظارت دقیق، به کلاس‌ها چندین مسئولیت مختلف واگذار می‌شود. طبق اصول کدنویسی تمیز بهتر است از این واقعه دوری کرد.

به‌عنوان مثال. شکل ب-۱ نمونه‌ای از بازآرایی استخراج کلاس را نشان می‌دهد.



شکل ب-۱ نمودار کلاس برای بازآرایی استخراج کلاس

جدول ب-۱ جزئیات و مراحل انجام بازآرایی استخراج کلاس را نشان می‌دهد. همچنین شکل ب-۲ شبه کد اعمال خودکار این بازآرایی را بیان می‌کند. این الگوریتم لیست حاوی متدها و لیست حاوی فیلدهایی که بایستی از کلاس خارج شده و به کلاس جدید انتقال یابند را به‌عنوان ورودی دریافت کرده و کلاس جدید با متشکل از متدها و فیلدهای دریافتی ایجاد می‌کند. همچنین تغییرات موردنیاز در کل برنامه منتشر می‌شوند.

ایده اصلی برای خودکارسازی بازآرایی استخراج کلاس استفاده از فن delegation است. ممکن است

فیلدها و توابع منتقل شده در مکان‌های مختلفی از کلاس اصلی و خارج از آن مورداستفاده قرار گرفته باشد که در صورت انتقال فیلد بدون اعمال هیچ تغییراتی با خطای کامپایل مواجه خواهیم شد. برای رفع این چالش از روش نمایندگی یا delegation استفاده شده است. بدین صورت که یک نمونه از کلاس جدید در کلاس اصلی ساخته می‌شود و هرجایی که از فیلد کلاس اصلی استفاده شده است را به نماینده ایجاد شده تغییر می‌دهیم. بدین ترتیب از وقوع هرگونه خطا در قسمت‌های استفاده‌کننده از کلاس مبدأ اجتناب می‌شود.

جدول ب-۱ بازسازی استخراج کلاس

بازآرایی	استخراج کلاس
بو(ها)ی کد مرتبط	کلاس بزرگ
تشخیص موقعیت بازآرایی[۱]	به‌صورت تصادفی
ورودی‌ها	نام کلاس، آرایه‌ای از نام فیلدها، آرایه‌ای از نام توابع
خروجی(ها)	کلاسی جدید در فایلی جدید شامل فیلدها و توابع داده شده در ورودی
پیش‌شرط‌ها	۵. معتبر بودن ورودی‌های داده شده (در کد پروژه وجود داشته باشند)
پس شرط‌ها	۲. پیدا کردن مکان تمام استفاده‌هایی که از فیلدهای انتقال‌یافته شده و تغییر آنها (استفاده از نمونه کلاس جدید)
مراحل	۵. ایجاد کلاس جدید. ۶. عمومی کردن فیلدها و توابع داده شده. ۷. انتقال فیلدها و توابع مذکور به کلاس جدید. ۸. ساخت یک نمونه عمومی از کلاس جدید در کلاس اصلی. [۲] ۹. اعمال تغییرات در کد جهت رفع خطاهای کامپایل.
تعداد گذرهای لازم	3
ملاحظات عملی	فرض شده است که داده‌های ورودی معتبر هستند.

[۱] refactoring opportunity

[۲] Delegation

[۳] pass

Algorithm 3. Extract Class Refactoring Pseudocode

```
Input: className
Input: moveFields
Input: moveMethods
Output: refactored project
1. methodFieldMap ← getClassMap(sourceClass) // list of used fields per each method in the source class
2. classPath ← generateClassPath(sourceClass)
3. newClassName ← generateNewName(sourceClass)
4. delegatedObjectName ← generateObjectName(newClassName)
5. newClassPath← generateClassPath(newClassName)
6. // Extract class ANTLR listerner
7. listener ← ExtractClassRefactoringListener(sourceClass, moveFields,
                                               moveMethods, newClassName,
                                               methodFieldMap)
8. writeFile(classPath, listener.getText())
9. // Update internal references with ANTLR listener
10. listener ← NewClassPropagation(sourceClass, moveFields, methodFieldMap, delegatedObjectName)
11. writeFile(newClassPath, listener.getText())
12. fieldReferences ← getFieldReferences(sourceClass, moveFields)
13. for field in fieldReferences do
14.    // Updates field references in project-level
15.    listener← PropagateFieldUsageListener(usage, delegatedObjectName)
16.    writeFile(usage.getFilePath(), listener.getText())
17. end for
```

شکل ب-۲ شبه کد بازسازی استخراج کلاس

به‌عنوان مثال تصور کنید می‌خواهیم در کلاس Person فیلدهای officeAreaCode و officeNumber و تابع getTelephonenNumber را به کلاس جدیدی انتقال دهیم:

```
public class Person {
    private String name;
    private int officeAreaCode;
    private int officeNumber;

    public String getTelephoneNumber(){
        return String.valueOf(this.officeAreaCode) + "-" +
String.valueOf(this.officeNumber);
    }

    public void testMethod(){
        String testVar = this.getTelephoneNumber() +  " " + this.name;
        System.out.println(testVar);
    }
}
```

همان‌طور که مشاهده می‌شود در هر دو تابع کلاس فوق از فیلدهای موردنظر استفاده شده است. بعد از اعمال بازسازی داریم:

```
public class Person {
  public PersonExtracted personExtracted = new PersonExtracted();
  private String name;

  public String getTelephoneNumber() {
    return this.personExtracted.getTelephoneNumber();
  }

  public void testMethod() {

    String testVar = this.getTelephoneNumber() + " " + this.name;

    System.out.println(testVar);
  }
}
```

مشاهده می‌شود که فیلد حذف شده اما از کلاس جدید یک نمونه ایجاد شده است و در تابع getTelephoneNumber به این نمونه اشاره شده است. زیرا کنون فیلد مذکور در کلاس جدید است. برای سایر استفاده‌هایی که در خارج از این کلاس رخداده است نیز همین فرایند تکرار می‌شود.

برای خودکار انجام‌دادن این بازسازی از ۴ بار پیمایش DFS درخت تجزیه استفاده شده است. برای به دست آوردن درخت تجزیه و پیمایش آن به‌صورت DFS از کتابخانه ANTLR استفاده شده است.

کلاس MethodMapListener

قبل از انجام بازسازی ضروری است بدانیم در تابع‌ها از چه فیلدهای یا تابع‌های دیگری استفاده شده است که قرار نیست انتقال داده شوند. بدین‌جهت وظیفه این کلاس به‌دست‌آوردن نگاشت بین تابع‌های انتقالی و فیلدها یا تابع‌های ثابت است. به‌عنوان مثال در نمونه کد فوق این نگاشت به این صورت است:

```
{
        'getTelephoneNumber': set(), 'testMethod':
        {'name'}
}
```

این نگاشت به این معنی است که در تابع انتقالی getTelephoneNumber از هیچ تابع ثابت یا فیلد ثابت دیگری استفاده نشده است، اما در تابع testMethod از فیلد ثابت name استفاده شده است.

برای به دست آوردن این نگاشت از ۳ قانون گرامر جاوا استفاده شده است:

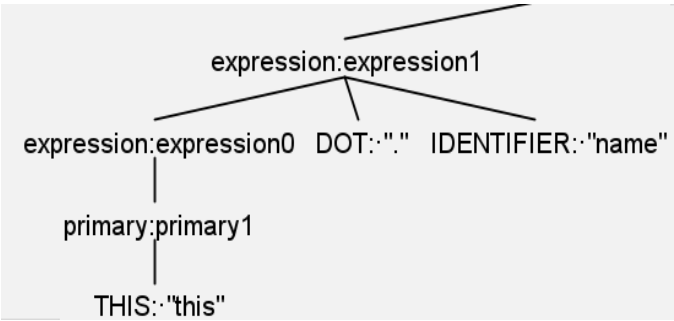
۱. هنگام تعریف تابع، نام تابع به‌عنوان کلید و یک مجموعه خالی به‌عنوان مقدار آن در نظر گرفته می‌شود.

```
def enterMethodDeclaration(self, ctx:
JavaParserLabeled.MethodDeclarationContext):
        self.detected_method = ctx.IDENTIFIER().getText()
        self.map[self.detected_method] = set()

def exitMethodDeclaration(self, ctx:
JavaParserLabeled.MethodDeclarationContext):
        self.detected_method = None
```

۲. اگر در عبارتی از this استفاده شده باشد ینی یک ارجاع به فیلد یا تابعی از کلاس صورت گرفته است.

```
def enterPrimary1(self, ctx: JavaParserLabeled.Primary1Context):
        if ctx.THIS():
            self.detected_usage = True
```

برای درک بهتر این موضوع به درخت موجود در شکل ب-۳ ترسیم شده است.



شکل ب-۳ بخشی از درخت تجزیه برای پیدا کردن فیلدها و توابع مورداستفاده

۳. در هنگام خروج از قاعده expression1 بررسی می‌شود که آیا فیلد تشخص داده شد ینی name جز فیلدهای انتقالی است یا خیر. اگر انتقالی نبود به نگاشت تابع اضافه می‌شود. این فرایند برای هر فیلد و تابع تکرار می‌شود.

```
def exitExpression1(self, ctx: JavaParserLabeled.Expression1Context):
        if self.detected_method and self.detected_usage:
            if ctx.IDENTIFIER():
                if ctx.IDENTIFIER().getText() not in self.moved_fields:
                    self.map[self.detected_method].add(ctx.IDENTIFIER().g
etText())
                self.detected_usage = False
            elif ctx.methodCall():
                if ctx.methodCall().IDENTIFIER().getText() not in
self.moved_methods:
                    self.map[self.detected_method].add(ctx.methodCall().I
DENTIFIER().getText())
                self.detected_usage = False
```

کلاس ExtractClassRefactoringListener

وظیفه اصلی این کلاس قرار دادن توابع و فیلدهای داده شده در یک کلاس جدید است. هم‌زمان با این کار عمل delegation را نیز انجام می‌دهد.

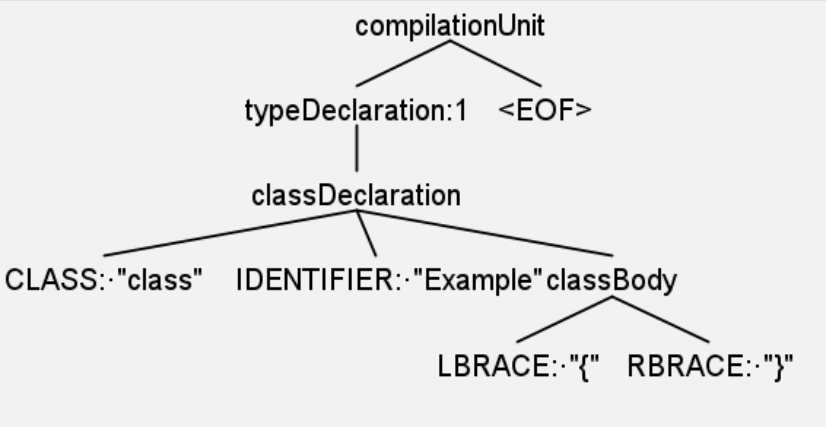
برای ساخت کلاس جدید متغیری با نام code تعریف شده است که محتویات کلاس جدید به‌صورت یک رشته در این متغیر ذخیره می‌شود. در ابتدا هنگامی که وارد کلاس مبدأ می‌شویم نام پکیج و اسم کلاس را تعریف می‌کنیم:

```
def enterClassDeclaration(self, ctx:
JavaParserLabeled.ClassDeclarationContext):
        class_identifier = str(ctx.children[1])
        if class_identifier == self.source_class:
            self.is_source_class = True
            self.code += self.NEW_LINE * 2
            if self.package_name:
                self.code += f"package
{self.package_name};{self.NEW_LINE}"
            self.code += f"// New class({self.new_class}) generated by
CodART" + self.NEW_LINE
            self.code += f"class {self.new_class}{self.NEW_LINE}" + "{" +
self.NEW_LINE
        else:
            self.is_source_class = False
```

همان‌طور که ملاحظه می‌شود، بعد از نوشتن نام کلاس، بلاک کلاس باز می‌شود. لازم است هنگام خروج از کلاس مبدأ آن را ببندیم:

```
def exitClassDeclaration(self, ctx:
JavaParserLabeled.ClassDeclarationContext):
        if self.is_source_class:
            self.code += "}"
            self.is_source_class = False
```

برای راحتی درک این موضوع به شکل ب-۴ توجه کنید:



شکل ب-۴ تعریف کلاس جدید با پیمایش از روی درخت

برای قرار دادن فیلدها و تابع‌ها به طریق مشابه از قواعد FieldDeclaration و MethodDeclaration استفاده می‌شود.

برای انجام delegation کافی است در ابتدای کلاس مبدأ یک نمونه از کلاس جدید بسازیم. هنگامی که وارد بدنه کلاس شدیم می‌توانید در ابتدای بدنه کلاس فیلد موردنظر را قرار دهیم:

```
def enterClassBody(self, ctx: JavaParserLabeled.ClassBodyContext):
        if self.is_source_class:
            self.token_stream_rewriter.insertAfterToken(
                token=ctx.start,
                text="\n\t" + f"public {self.new_class}
{self.object_name} = new {self.new_class}();",
                program_name=self.token_stream_rewriter.DEFAULT_PROGRAM_N
AME
```

کلاس PropagateFieldUsageListener

هم در کلاس مبدأ و هم در سایر کلاس، هرجایی که از فیلدها انتقال‌یافته استفاده شده است باید تغییر کند. در واقع چون از delegation استفاده شده است، این تغییر بسیار جزئی است و کافی است this

را به this.object_name تغییر دهیم که object_name نام نمونه جدید از کلاس استخراج شده است. این کار نیز با استفاده از قاعده expression1 به‌صورت زیر انجام می‌شود:

```
def enterExpression1(self, ctx: JavaParserLabeled.Expression1Context):
        identifier = ctx.IDENTIFIER()
        if identifier is not None:
            if identifier.getText() == self.field_name:
                self.token_stream_rewriter.replaceSingleToken(
                    token=ctx.expression().primary().start,
                    text=f"this.{self.object_name}"
                )
```

کلاس NewClassPropagation

در توابعی که از فیلدهای ثابت در آن استفاده شده است کافی است نمونه کلاس مبدأ یعنی this را با نام ref که مخفف reference است را در توابع کلاس مقصد پاس دهیم، سپس در کلاس مقصد تمامی استفاده‌ها از this به ref تغییر پیدا می‌کند. به‌عنوان مثال اگر در تابع getTelephoneNumber از تابع testMethod استفاده شده باشد، آنگاه بعد از بازسازی داریم:

```
public String getTelephoneNumber() {
        return this.personExtracted.getTelephoneNumber(this);
    }
```

```
public String getTelephoneNumber(Person ref){
        ref.testMethod();
        return String.valueOf(this.officeAreaCode) + "-" +
String.valueOf(this.officeNumber);
    }
}
```

همان‌طور که مشاهده می‌شود، در کلاس مبدأ نمونه کلاس تحت عنوان this پاس داده شده است و در کلاس مقصد تحت عنوان ref استفاده شده است و this.testMethod() به ref.testMethod() تغییر پیدا کرده است.

این کار نیز با استفاده از دو قاعده گرامر جاوا قابل انجام است. در قاعده تعریف تابع می‌توان به پارامترها ref را که نمونه‌ای از کلاس مبدأ است اضافه کرد و همچنین با استفاده از قاعده expression1 می‌توان this را با ref عوض کرد:

```
def enterMethodDeclaration(self, ctx:
JavaParserLabeled.MethodDeclarationContext):
        self.fields = self.method_map.get(ctx.IDENTIFIER().getText())
        if self.fields:
            if ctx.formalParameters().getText() == "()":
                text = f"{self.source_class} ref"
            else:
                text = f", {self.source_class} ref"

            self.token_stream_rewriter.insertBeforeToken(
                token=ctx.formalParameters().stop,
                text=text,
                program_name=self.token_stream_rewriter.DEFAULT_PROGRAM_N
AME
            )

    def exitMethodDeclaration(self, ctx:
JavaParserLabeled.MethodDeclarationContext):
        self.fields = None
```

```
def enterExpression1(self, ctx: JavaParserLabeled.Expression1Context):
        if self.fields and ctx.expression().getText() == "this":
            for field in self.fields:
                if field in ctx.getText():
                    self.token_stream_rewriter.replaceSingleToken(
                        token=ctx.expression().primary().start,
                        text="ref"
```

پیوست پ: بازسازی انتقال تابع

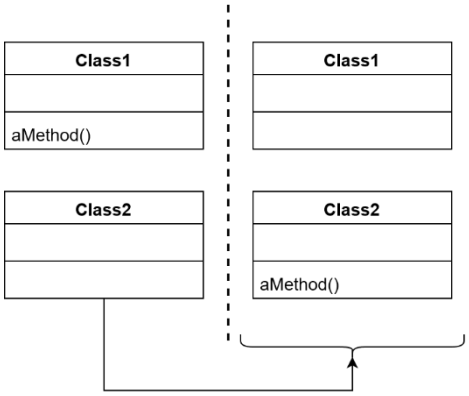
تعریف: بازآرایی انتقال تابع به انتقال یک تابع از کلاسی به کلاس دیگر و تغییرات مربوطه به این انتقال گفته می‌شود.

مشکل: به دلیل پیچیدگی برنامه، ممکن است یک تابع در کلاس دیگری بیشتر از کلاس خودش استفاده شده باشد.

راه‌حل: انتقال تابع به کلاسی مناسب.

دلیل: انتقال تابع به کلاسی که حاوی بیشتر داده‌های استفاده شده توسط آن تابع باشد، باعث می‌شود که کلاس‌ها از نظر داخلی منسجم‌تر شوند. همچنین با جابه‌جا کردن یک تابع ممکن است فراخوانی آن تابع را به کلاسی که در آن قرار دارد کاهش دهد. اگر کلاس هدف به این کلاس وابسته باشد، این امر مفید است و وابستگی بین کلاس‌ها را کاهش می‌دهد.

به‌عنوان مثال شکل پ-۱ نمونه‌ای از بازآرایی انتقال تابع را نشان می‌دهد.



شکل پ-۱ نمودار کلاس برای بازآرایی انتقال تابع

جدول پ-۱ جزئیات و مراحل انجام بازآرایی انتقال تابع را نشان می‌دهد. همچنین شکل پ-۲ شبه کد اعمال خودکار این بازآرایی را بیان می‌کند. این الگوریتم اطلاعات کلاس مبدأ، تابع مدنظر جهت انتقال و کلاس مقصد را دریافت می‌کند و عمل بازآرایی را انجام می‌دهد.

ایده اصلی برای خودکارسازی بازآرایی انتقال تابع استفاده از فن delegation است. ممکن است تابع منتقل شده در مکان‌های مختلفی از کلاس اصلی و خارج از آن مورداستفاده قرار گرفته باشد. برای رفع این چالش از روش نمایندگی یا delegation استفاده شده است. بدین صورت که یک نمونه از کلاس مقصد

در کلاس مبدأ ساخته می‌شود و هرجایی که از تابع کلاس اصلی استفاده شده است را به نماینده ایجاد شده تغییر می‌دهیم. بدین ترتیب از وقوع هرگونه خطا در قسمت‌های استفاده‌کننده از کلاس مبدأ اجتناب می‌شود.

جدول پ-۱ بازسازی انتقال تابع

بازآرایی	انتقال تابع
بو(ها)ی کد مرتبط	دسترسی بیشتر به داده های بیرونی[۱]
تشخیص موقعیت بازآرایی[۲]	به‌صورت تصادفی
ورودی‌ها	پکیج مبدأ، کلاس مبدأ، نام تابع، پکیج مقصد، کلاس مقصد.
خروجی(ها)	کلاس مقصد شامل تابع انتقال‌یافته
پیش‌شرط‌ها	۶. معتبر بودن ورودی‌های داده شده (در کد پروژه وجود داشته باشند) ۷. یکسان نبودن کلاس مبدأ و کلاس مقصد ۸. وجود نداشتن تابعی با نام مشابه در کلاس مقصد ۹. عدم وجود وابستگی حلقوی یا Cyclic Dependency بین دو کلاس مبدأ و مقصد ۱۰. نبودن کلاس در سلسله مراتب ارث‌بری و پیاده‌سازی (Extend, Implement)
پس شرط‌ها	۳. پیدا کردن مکان تمام استفاده‌های تابع انتقال‌یافته شده و تغییر آنها (استفاده از کلاس مقصد)
مراحل	۱۰. حذف تابع از کلاس مبدأ ۱۱. قرار دادن تابع در کلاس مقصد ۱۲. به‌روزرسانی محل‌های استفاده از تابع
تعداد گذرهای لازم	۵
ملاحظات عملی	فرض شده است که داده‌های ورودی معتبر هستند.

[۱] Feature Envy

[۲] refactoring opportunity

[۳] pass

Algorithm 4. Move Method Refactoring Pseudocode

```
Input: sourceClass
Input: sourcePackage
Input: targetClass
Input: targetPackage
Input: methodName
Output: refactored project
// check pre-conditions
if not checkMethodExists(sourcePackage, sourceClass, methodName) then
   return false // invalid input values
end if
if checkMethodExists(targetPackage, targetClass, methodName) then
   return // a similar method already exists
end if
if sourcePackage = targetPackage and sourceClass = targetClass then
   return // invalid input values
end if
if checkCyclicDependeny(sourceClass, targetClass) then
   return // cannot move method if there is a cyclic dependency between classes
end if
// Cut method content with ANTLR listener
listener ← CutMethodListener(sourcePackage, sourceClass, methodName)
// Paste method content with ANTLR listener
PasteMethodListener(targetPackage, targetClass, methodName, listener.getText())
methodReferences ← getReferences(sourcePackage, sourceClass, methodName)
for reference in methodReferences do
   PropagateListener(reference) // propagate changes in project with ANTLR listener
end for
// Reference injection with ANTLR listener
ReferenceInjectorAndConstructorListener(targetPackage, targetClass)
```

شکل پ-۲ شبه کد بازآرایی انتقال تابع

به‌عنوان مثال تصور کنید می‌خواهیم در کلاس Class1 تابع aMethod را انتقال دهیم. زیرا در خود این کلاس هیچ فراخوانی به آن صورت نگرفته است اما در کلاس Class2 به این تابع فراخوانی داریم. پس کلاس مقصد را کلاس Class2 در نظر می‌گیریم.

```
package source_package;

public class Class1 {
    public void aMethod(){
        System.out.println("Running aMethod");
    }
}
```

و در کلاس Class2 داریم:

```
package target_package;

import source_package.Class1;

public class Class2 {
    public void testMethod(){
        Class1 class1 = new Class1();
        class1.aMethod();
    }
}
```

پس از اجرا بازسازی دو کلاس فوق به‌صورت زیر تغییر می‌کنند. مشاهده می‌شود که تابع از Class1 حذف و به Class2 انتقال‌یافته است.

```
package source_package;

import target_package.Class2;

public class Class1 {
    public Class2 class2ByCodArt = new Class2();
}
```

و در کلاس Class2 داریم:

```
package target_package;

import source_package.Class1;

public class Class2 {
    public Class2() {}

    public void aMethod() {
        System.out.println("Running aMethod");
    }

    public void testMethod() {
        Class1 class1 = new Class1();
        this.aMethod();
    }
}
```

تغییرات ایجاد شده به‌صورت زیر است:

۱. از کلاس مقصد، یک نمونه در کلاس مبدأ ایجاد می‌شود.

۲. به دلیل اینکه در کلاس مقصد سازنده بدون ورودی وجود ندارد، یک سازنده بدون ورودی ایجاد

می‌شود.

۳. تابع موردنظر انتقال‌یافته است.

۴. برای مراجع تابع مورد انتقال در کلاس مقصد از this استفاده شده است.

برای سایر مراجع، اعم از مراجع درون کلاس مبدأ و سایر کلاس‌ها، از نمونه ایجاد شده استفاده می‌شود.

```
this.aMethod(); → this.class2ByCodArt.aMethod();

Class1 class1 = new Class1();
class1.aMethod(); → class1.class2ByCodArt.aMethod();
```

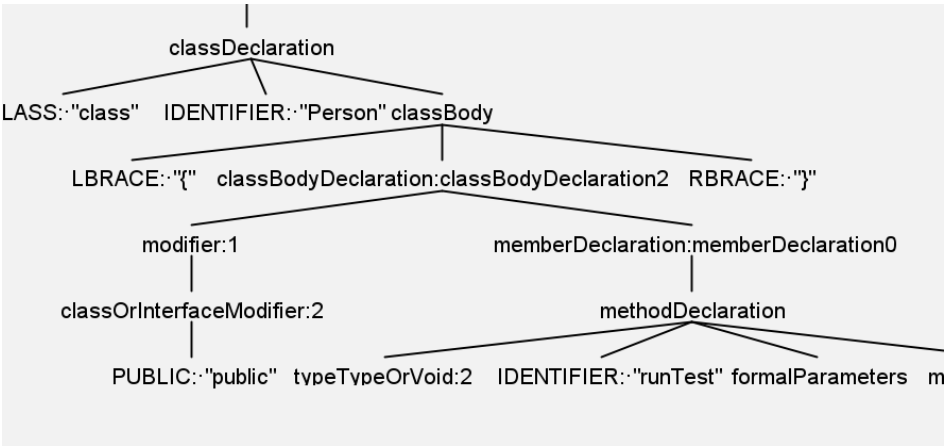
برای انجام این بازسازی به‌صورت خودکار از ۴ بار پیمایش DFS درخت تجزیه استفاده شده است که جزئیات هرکدام به شرح زیر است.

کلاس CutMethodListener

وظیفه این کلاس حذف تابع از کلاس مبدأ و پیاده‌سازی روش نمایندگی یا delegation است. چون داریم از کلاس مقصد یک نماینده ایجاد می‌کنیم پس لازم است جهت جلوگیری از خطای کامپایل، آن کلاس import شود. این کار در exitPackageDeclaration پیاده‌سازی شده است. زیرا همان‌طور که قبلاً بیان شده، در زبان جاوا import ها پس از package مشخص می‌شوند.

```
def exitPackageDeclaration(self, ctx:
JavaParserLabeled.PackageDeclarationContext):
        if self.import_statement:
            self.rewriter.insertAfterToken(
                token=ctx.stop,
                text=self.import_statement,
                program_name=self.rewriter.DEFAULT_PROGRAM_NAME
            )
            self.import_statement = None
```

برای حذف تابع موردنظر و قرار دادن نماینده کلاس مقصد ابتدا نیاز است تا محل تابع را در درخت تجزیه مشخص کنیم. شکل پ-۳ را در نظر بگیرید.



شکل پ-۳ بخشی از درخت تجزیه شامل تعریف تابع در یک کلاس

همان‌طور که مشاهده می‌شود، methodDeclaration زیرمجموعه قانون memberDeclaration است. پس یک متغیر باینری به نام is_member تعریف شده است تا هنگام پیمایش مشخص کند که درون memberDeclaration هستیم. سپس در classBodyDeclaration کل متن تابع ابتدا استخراج و ذخیره می‌شود و سپس با نماینده کلاس مقصد جایگزین می‌شود. در کد زیر تمامی این مراحل به طور کاملاً واضح نوشته شده است.

```
class CutMethodListener(JavaParserLabeledListener):
    #.... __init___

    def exitPackageDeclaration(self, ctx:
JavaParserLabeled.PackageDeclarationContext):
        if self.import_statement:
            self.rewriter.insertAfterToken(
                token=ctx.stop,
                text=self.import_statement,
                program_name=self.rewriter.DEFAULT_PROGRAM_NAME
            )
            self.import_statement = None

    def enterMemberDeclaration0(self, ctx:
JavaParserLabeled.MemberDeclaration0Context):
        self.is_member = True

    def exitMemberDeclaration0(self, ctx:
JavaParserLabeled.MemberDeclaration0Context):
        self.is_member = False

    def enterMethodDeclaration(self, ctx:
JavaParserLabeled.MethodDeclarationContext):
        if self.is_member and ctx.IDENTIFIER().getText() ==
self.method_name:
            self.do_delete = True

    def exitClassBodyDeclaration2(self, ctx:
JavaParserLabeled.ClassBodyDeclaration2Context):
        if self.do_delete:
            self.method_text = self.rewriter.getText(
                program_name=self.rewriter.DEFAULT_PROGRAM_NAME,
                start=ctx.start.tokenIndex,
                stop=ctx.stop.tokenIndex
            )
            if self.is_static:
                replace_text = f"public static {self.class_name}
{self.instance_name} = new {self.class_name}();"
            else:
                replace_text = f"public {self.class_name}
{self.instance_name} = new {self.class_name}();"
            self.rewriter.replace(
                program_name=self.rewriter.DEFAULT_PROGRAM_NAME,
                from_idx=ctx.start.tokenIndex,
                to_idx=ctx.stop.tokenIndex,
                text=replace_text
            )

            self.do_delete = False
```

یکی از وظایف دیگر این پیمایشگر، ذخیره کردن import ها است. از آنجا که داریم که تابع به‌صورت کامل انتقال می‌دهیم، ممکن از است از کلاس‌هایی استفاده کرده باشد که در کلاس مقصد استفاده نشده باشد؛ بنابراین نیاز است تا import ها انجام شده در کلاس مبدأ ضبط و ذخیره شود. این کار در تابع enterImportDeclaration پیاده‌سازی می‌شود.

```
def enterImportDeclaration(self, ctx:
JavaParserLabeled.ImportDeclarationContext):
    self.imports += self.rewriter.getText(
        program_name=self.rewriter.DEFAULT_PROGRAM_NAME,
        start=ctx.start.tokenIndex,
        stop=ctx.stop.tokenIndex
    ) + "\n"
```

کلاس PasteMethodListener

وظیفه این کلاس قرار دادن تابع موردنظر و import های ذخیره شده، جهت انتقال داخل کلاس مقصد است. کد کامل این تابع و import ها به‌وسیله کلاس قبلی استخراج شده است. برای قرار دادن import ها در جای مناسب، دو حالت ممکن است رخ دهد.

حالت اول این است که کلاس در یک بسته مشخص تعریف شده است. در این حالت می‌توان import ها را هنگام خروج از قانون packageDeclaration در متن کد قرار داد.

```
def exitPackageDeclaration(self, ctx:
JavaParserLabeled.PackageDeclarationContext):
    if self.has_package and self.imports:
        self.rewriter.insertAfter(
            index=ctx.stop.tokenIndex,
            text="\n" + self.imports,
            program_name=self.rewriter.DEFAULT_PROGRAM_NAME
        )
```

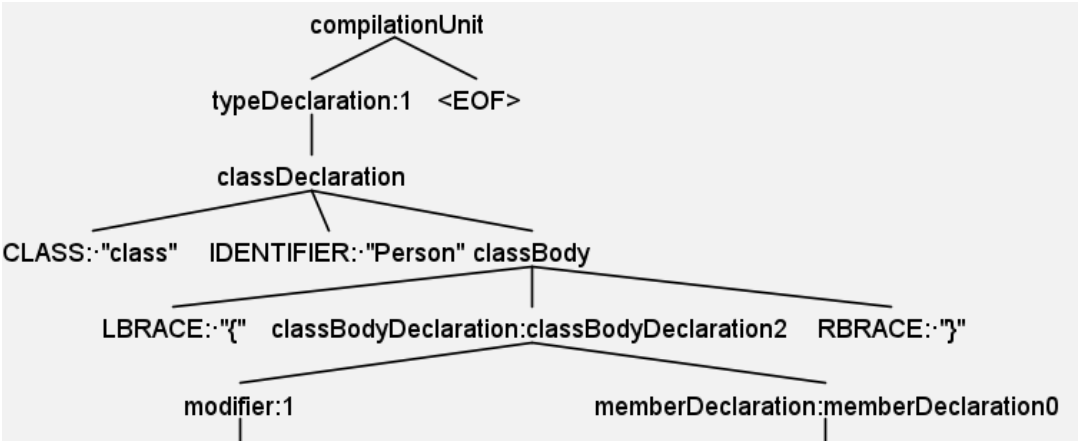
در حالت دوم، برای کلاس، بسته‌ای تعریف نشده است. (کلاس در بسته پیش‌فرض قرار دارد.) در این حالت می‌توان import ها را در هنگام خروج از قانون compilationUnit در متن کد جاگذاری کرد.

```
def exitCompilationUnit(self, ctx:
JavaParserLabeled.CompilationUnitContext):
    if not self.has_package and self.imports:
        self.rewriter.insertBefore(
            index=ctx.start.tokenIndex,
            text="\n" + self.imports,
            program_name=self.rewriter.DEFAULT_PROGRAM_NAME
        )
```

از متغیر دوحالته has_package برای متمایز ساختن این دو حالت استفاده شده است.

```
def enterPackageDeclaration(self, ctx:
JavaParserLabeled.PackageDeclarationContext):
    self.has_package = True
```

همانطور که در شکل پ-۴ مشخص است، برای جاگذاری متن تابع، کافی است در enterClassBody کد تابع را بعد از توکن شروع قرار دهیم.



شکل پ-۴ بخشی از درخت تجزیه جهت فهمیدن محل توابع

طبق کد زیر، کد تابع در بدنه کلاس قرار داده می‌شود:

```
class PasteMethodListener(JavaParserLabeledListener):
    def __init__(self, method_text: str, rewriter: TokenStreamRewriter):
        self.method_text = method_text
        self.rewriter = rewriter

    def enterClassBody(self, ctx: JavaParserLabeled.ClassBodyContext):
        self.rewriter.insertAfterToken(
            token=ctx.start,
            text="\n" + self.method_text + "\n",
            program_name=self.rewriter.DEFAULT_PROGRAM_NAME
        )
```

همچنین در این پیمایش مشخص می‌شود که آیا سازنده بدون ورودی، تعریف شده است یا کلاس بدون سازنده بدون ورودی است. از این متغیر دوحالته has_empty_cons در پیمایش بعدی برای تعریف یا عدم تعریف سازنده استفاده می‌شود.

کلاس ReferenceInjectorAndConstructorListener

این کلاس دو وظیفه مهم را انجام می‌دهد.

۱. تعریف کلاس سازنده بدون ورودی در صورت نیاز

۲. کنترل کردن مراجعی که در متن تابع به کلاس مبدأ صورت گرفته است.

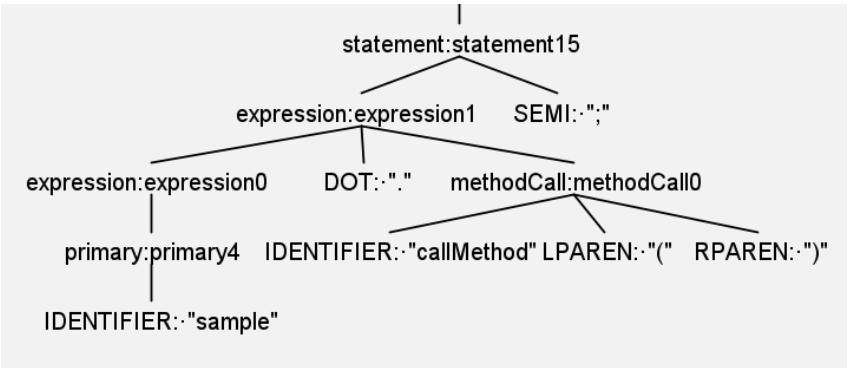
باتوجه‌به متغیر دوحالته has_empty_cons از کلاس قبل، کلاس سازنده در بدنه کلاس تعریف می‌شود.

```
def enterClassDeclaration(self, ctx:
JavaParserLabeled.ClassDeclarationContext):
    self.class_name = ctx.IDENTIFIER().getText()

def enterClassBody(self, ctx: JavaParserLabeled.ClassBodyContext):
    if not self.has_empty_cons:
        self.rewriter.insertAfterToken(
            token=ctx.start,
            text="\n" + f"public {self.class_name}()" + "{}\n",
            program_name=self.rewriter.DEFAULT_PROGRAM_NAME
        )
```

برای کنترل کردن مراجعی که در متن تابع به کلاس مبدأ صورت گرفته است، می‌بایست یک نمونه از کلاس مبدأ به تابع انتقال‌یافته تحت نام ref پاس داده شود و در متن تابع به‌جای this از ref استفاده شود. جزئیات پیاده‌سازی این قسمت پیش‌تر در بازسازی استخراج کلاس بیان شده است.

کلاس PropagateListener

چون از روش نمایندگی یا delegation استفاده شده است بنابراین در تمام بخش‌هایی که تابع انتقال‌یافته صدا زده شده است کافی است به‌جای اینکه مستقیم به کلاس مبدأ ارجاع شود، به نماینده کلاس مقصد ارجاع شود. برای درک بهتر کافی است درخت تجزیه را هنگام صدا زدن تابع در شکل پ-۵ مشاهده کنیم.



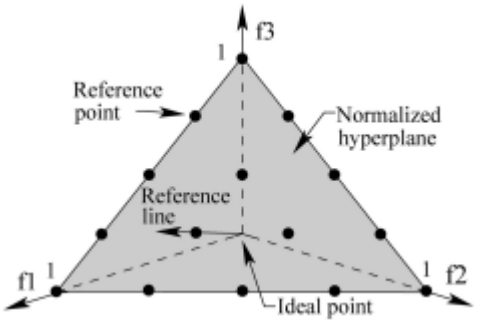
شکل پ-۵ بخشی از درخت تجزیه مربوط به صدا زدن یک تابع

همان‌طور که مشاهده می‌شود کافی است در قانون methodCall به identifier آن، نام نماینده اضافه شود تا تابع از طریق نماینده کلاس مقصد صدا زده شود. همچنین، در صورت ارجاع به کلاس مبدأ، باید هنگام صدا زدن تابع یک نمونه از کلاس مبدأ نیز پاس داده شود. در کد زیر این عملیات پیاده‌سازی شده است.

```
class PropagateListener(JavaParserLabeledListener):
def __init__(self, method_name: str, new_name: str, lines: list,
is_in_target_class: bool, method_map: dict,
                 rewriter: TokenStreamRewriter):
    self.method_name = method_name
    self.new_name = new_name
    self.lines = lines
    self.method_map = method_map
    self.fields = None
    self.rewriter = rewriter
    self.is_in_target_class = is_in_target_class

def enterMethodCall0(self, ctx: JavaParserLabeled.MethodCall0Context):
    identifier = ctx.IDENTIFIER()
    self.fields = self.method_map.get(identifier)
    if identifier and ctx.start.line in self.lines and
identifier.getText() == self.method_name:
        if self.fields:
            parent = ctx.parentCtx
            caller = parent.children[0]
            caller = self.rewriter.getText(
                program_name=self.rewriter.DEFAULT_PROGRAM_NAME,
                start=caller.start.tokenIndex,
                stop=caller.stop.tokenIndex
            )
            if ctx.expressionList():
                self.rewriter.insertAfterToken(
                    token=ctx.expressionList().stop,
                    text=", " + caller,
                    program_name=self.rewriter.DEFAULT_PROGRAM_NAME
                )
            else:
                self.rewriter.insertAfter(
                    index=ctx.stop.tokenIndex - 1,
                    text=caller,
                    program_name=self.rewriter.DEFAULT_PROGRAM_NAME
                )
        if self.is_in_target_class:
            self.rewriter.replaceSingleToken(
                token=ctx.parentCtx.start,
                text="this"
            )
        else:
            self.rewriter.replaceSingleToken(
                token=ctx.start,
                text=self.new_name
            )

def exitMethodCall0(self, ctx: JavaParserLabeled.MethodCall0Context):
    self.fields = None
```

پیوست ت: مقدمه‌ای بر الگوریتم NSGA-III

الگوریتم NSGA-III یک الگوریتم جدید مبتنی بر الگوریتم ژنتیک مرتب‌سازی نا مغلوب است که در سال ۲۰۱۴ توسط پروفسور دب و یکی از دانشجویانش به نام جِین ارائه شده است. اصول پایه این الگوریتم مانند الگوریتم NSGA-II است. هر دو الگوریتم از عملگرهای تقاطع و جهش برای تولید فرزندان استفاده می‌کنند و از رویکرد مرتب‌سازی نا مغلوب برای تعیین رتبه نامغلوبی اعضای جمعیت استفاده می‌کنند. با این حال، بر خلاف الگوریتم NSGA-II که از مفهوم فاصله ازدحام برای ایجاد تمایز بین اعضای مربوط به یک جبهه پرتو استفاده می‌کند، الگوریتم NSGA-III از یک عملگر انتخاب بر مبنای تعدادی نقطه مرجع برای این منظور استفاده می‌کند که می‌تواند منجر به پراکندگی بیشتر جواب‌های نا مغلوب حاصل شود.



شکل ت-۱ صفحه موسوم به صفحه بالایی

مراحل مختلف اجرای این الگوریتم به‌صورت زیر هستند:

۱. تولید یک جمعیت اولیه
۲. انجام عملگرهای جهش و تقاطع
۳. ایجاد جمعیت ترکیبی از والدین و فرزندان
۴. مرتب‌سازی نا مغلوب
۵. تعیین نقاط مرجع بر روی یک صفحه موسوم به صفحه بالایی
۶. نرمال‌سازی وفقی اعضای جمعیت
۷. عملگر ارتباط
۸. حفظ تورفتگی

مراحل ۱ تا ۴ شبیه به نسل قبلی یعنی NSGA-II و مشابه سایر الگوریتم‌های ژنتیک است. مراحل

۵ تا ۸ به ترتیب به شرح زیر هستند.

تعیین نقاط مرجع بر روی یک صفحه موسوم به صفحه بالایی

الگوریتم NSGA-III از یک مجموعه از پیش تعیین شده نقاط مرجع استفاده می‌نماید تا از تنوع پاسخ‌های به دست آمده نهایی اطمینان حاصل شود. (شکل ت-۱) این نقاط مرجع انتخاب شده می‌توانند در یک روش ساختاری از پیش تعیین شده یا به طور ترجیحی توسط کاربر مشخص شوند. در حالتی که هیچ‌گونه اطلاعات ترجیحی وجود نداشته باشد، هیچ روش ساختاری جایابی نقاط مرجع نمی‌تواند تطبیق داده شود. با این حال می‌توان از روش سیستماتیک که نقاط را روی یک صفحه جای می‌دهد، استفاده نمود. اگر P بخش در امتداد هر تابع هدف در نظر گرفته شود، تعداد کل نقاط مرجع در حالت مسئله M هدفه به‌صورت $C(M+p-1.m)$ محاسبه می‌شود.

برای مثال، در یک مسئله سه هدفه نقاط مرجع روی یک فضای مثلثی شکل که رئوس آن روی نقاط (۱،۰،۰) و (۰،۱،۰) و (۰،۰،۱) قرار دارند ایجاد می‌شوند. اگر چهار بخش در امتداد هر تابع هدف در نظر گرفته شوند، $P=4$، آنگاه ۱۵ نقطه مرجع تولید خواهد شد. برای درک بهتر این نقاط در ۰ نشان داده شده‌اند.

در الگوریتم NSAG-III علاوه بر تاکید بر پاسخ‌های نا چیره، پاسخ‌های در جهت وابسته به این نقاط مرجع نیز مطرح هستند. از آنجا که نقاط مرجع ایجاد شده فوق عمدتاً روی کل ناحیه صفحه توزیع شده‌اند، پاسخ‌های به دست آمده نیز احتمالاً روی جبهه بهینه پارتو یا نزدیک آن توزیع می‌شوند.

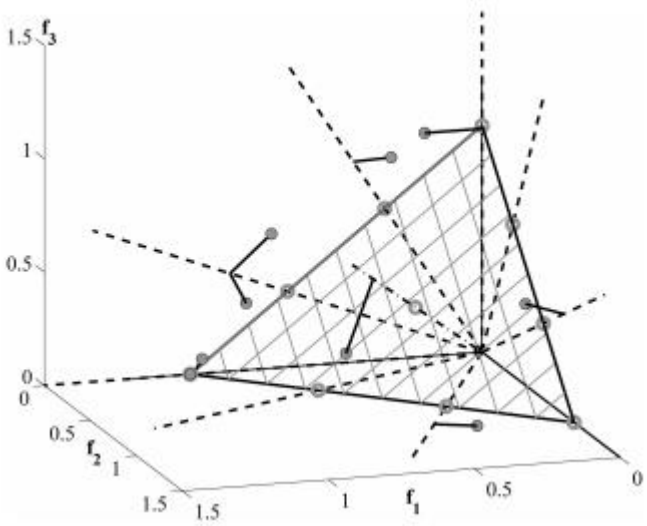
نرمال‌سازی وفقی اعضای جمعیت

در این مرحله یک فرایند نرمالیزاسیون انجام می‌شود. نخست نقاط ایده‌آل از جمعیت St با شناسایی مقدار مینیمم Z_i^{min} برای هر تابع هدف ساخته می‌شود. هر مقدار هدف St با تفریق هدف f_i از Z_i^{min} به دست می آید چنان که نقطه‌ایده‌آل به دست آمده St یک بردار صفر می‌شود. ما این مقدار را به‌صورت $f_i'(x)$ نشان می‌دهیم. سپس نقاط افراطی یا extreme (Z_i^{max}) در هر محور هدف با پیدا کردن جواب‌هایی $(x \in St)$ که تابع مقیاس زایی حاصله متناظر را مینیمم می نماید، شناسایی می‌شوند. این M بردار افراطی سپس برای ساختن یک صفحه M بعدی استفاده می‌شوند.

عملگر ارتباط

بعد از نرمالیزه کردن هر هدف به طور انطباقی بر اساس توسعه اعضای S_t در فضای تابع هدف، ما باید هر عضو از جمعیت را با نقاط مرجع مرتبط نماییم. بدین منظور، ما یک خط مرجع متناظر با هر نقطه

مرجع روی صفحه با اتصال نقطه مرجع به مبدأ تعریف می‌کنیم. سپس، فاصله عمودی هر عضو از جمعیت S_t را از هر خط مرجع محاسبه می‌کنیم. نقطه مرجع که خط مرجعش نزدیک به عضو جمعیت در فضای هدف نرمالیزه شده است به‌عنوان عضو جمعیت مرتبط با آن تلقی می‌شود. این موضوع در شکل ت-۲ نشان داده شده است.



شکل ت-۲ مثالی از نقاط مرجع

حفظ تورفتگی

این نکته هم ارزشمند خواهد بود که نقطه مرجع ممکن است یک یا چند عضو جمعیت را وابسته به خود داشته باشد یا نیازی نیست که هیچ عضوی از جمعیت را داشته باشد. ما در اینجا تعداد عضوهای جمعیت از $P_{t+1} = S_t/Fl$ که مرتبط با نقاط مرجع هستند را به شمار می‌آوریم. اجازه دهید این تورفتگی را به‌صورت ρj برای h امین نقطه مرجع به‌حساب آوریم. ما حالا یک عملگر حفظ تورفتگی به فرم زیر ابداع می‌کنیم: نخست، ما مجموعه نقطه مرجع $J_{min} = \{j : argmin_j^{\rho j}\}$ با داشتن مینیمم ρj را شناسایی می‌کنیم. در حالت نقاط مرجع چندگانه، یک ($\bar{j} \in Jmin$) به طور تصادفی انتخاب می‌شود.

اگر $\rho\bar{j} = 0$ (بدین معنی که هیچ عضو P_{t+1} به نقطه مرجع $\bar{j}$ وابسته نیست) می‌تواند دو سناریو با $\bar{j}$در مجموعه Fl وجود دارد. نخست یک یا چند عضو در جبهه Fl وجود دارد که با نقطه مرجع $\bar{j}$ وابسته می‌شوند. در این حالت، یک عضو با کوتاه‌ترین فاصله عمودی از خط مرجع به P_{t+1} اضافه می‌شود. تعداد $\rho\bar{j}$ برای نقطه مرجع $\bar{j}$ سپس به‌اندازه یک افزایش می‌یابد. دوم، جبهه Fl هیچ عضو وابسته به نقطه مرجع $\bar{j}$ نداشته باشد. در این حالت، نقطه مرجع از بررسی بیشتر برای نسل فعلی حذف می‌شود.

در حالت $\rho_{\bar{j}} \geq 1$ (بدین معنی که یک عضو وابسته به نقطه مرجع در S_t/Fl وجود دارد)، یک عضو انتخاب شده تصادفی اگر وجود داشته باشد، از جبهه Fl که به نقطه مرجع $\bar{j}$ مرتبط است به P_{t+1} اضافه می‌شود. تعداد $\rho_{\bar{j}}$ سپس به‌اندازه یک افزایش می‌یابد. بعد از اینکه تعداد فرورفتگی‌ها آپدیت شد، فرایند برای k بار دیگر تکرار می‌شود تا همه شیارهای جمعیت خالی P_{t+1} تکرار می‌شود.

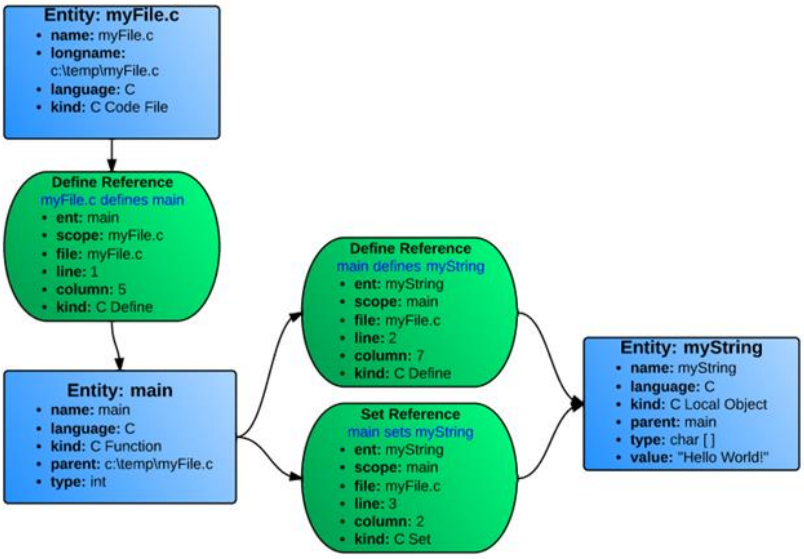
پیوست ث: آشنایی بیشتر با پروژه OpenUnderstand

به‌عنوان مثال قطعه کد زیر را که به زبان سی نوشته شده است را در نظر بگیرید:

myFile.c

```
    void main(){
       char myString[];
       myString = "Hello World!";
}
```

در شکل ث-۱، موجودیت‌ها و روابط بین آنها (مراجع) ترسیم شده است.

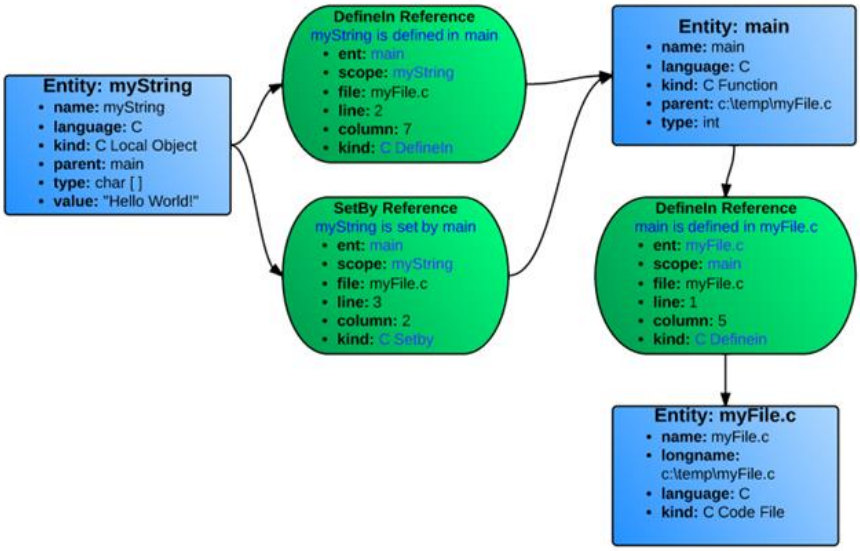


شکل ث-۱ ساختار داده Understand برای کد سی مذکور

هر رابطه دوطرفه است. به‌عنوان مثال اگر تابع main متغیر myString را تعریف می‌کند.[۱]آنگاه رابطه برعکس آن نیز موجود است، یعنی متغیر myString در تابع main تعریف شده است.[۲]در شکل زیر، روابط معکوس شکل ث-۱ ترسیم شده است.

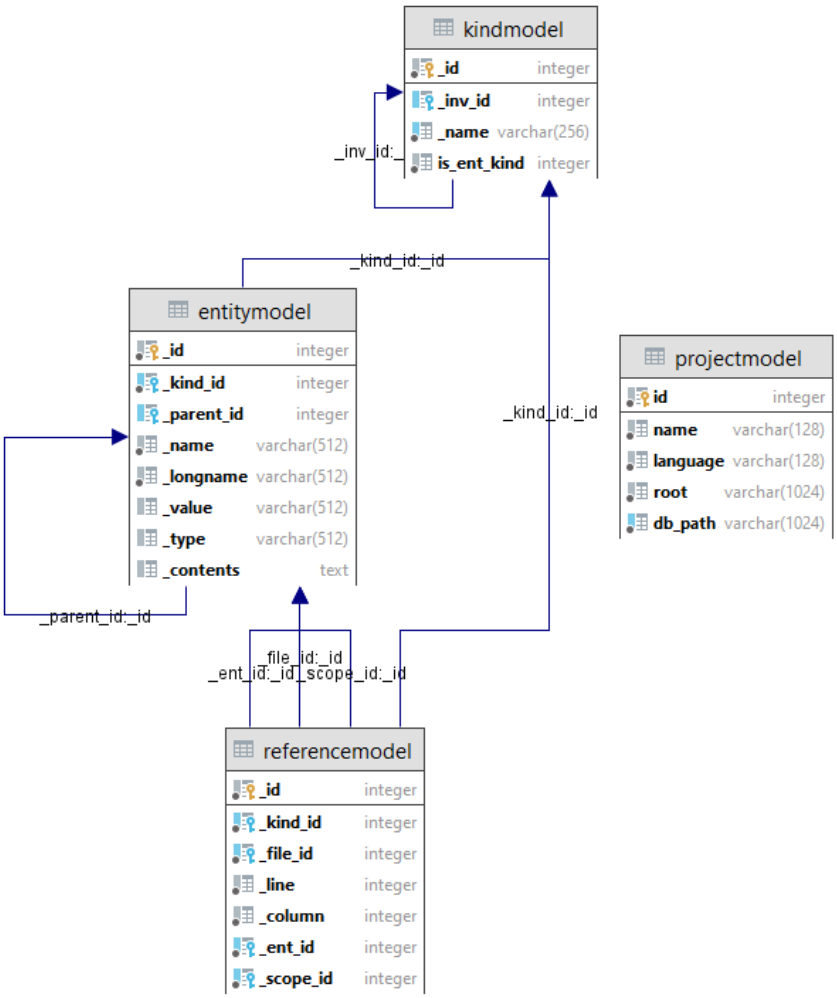
[۱] Define

[۲] Define In



شکل ث-۲ معکوس شده روابط موجود در شکل ث-۱

با بررسی روابط مختلف و تحلیل داده‌های Understand بر روی پروژه‌های واقعی و بزرگ مانند Weka توانستیم ساختار تقریبی پایگاه‌داده این ابزار را مهندسی معکوس کنیم. در شکل زیر نمودار موجودیت - رابطه پایگاه‌داده Open Understand ترسیم شده است.



شکل ث-۳ نمودار موجودیت - رابطه Open Understand

جهت پیاده‌سازی این ساختار و ارتباط مستقیم با پایگاه‌داده از نگاشت شی-رابطه‌ای کتابخانه peewee و پایگاه‌داده SQLite3 استفاده شده است (فایل models.py در لایه db). این نگاشت در شکل ث-۳ ترسیم شده است. برای سازگاری کامل با Understand سعی کردیم با استفاده از مستندات ارائه شده، در فایل api. y توابع و کلاس‌ها را با همان نام و عملکرد ابزار شبیه‌سازی کنیم. برای توضیح کامل نحوه عملکرد این سامانه، مثالی از مرجع ساختن Create/CreateBy توضیح داده شده است.

این مرجع تنها هنگامی رخ می‌دهد که یک شی جدید از یک کلاس در یک تابع ایجاد شود. به‌عنوان مثال قطعه کدی که در ادامه آمده است را در نظر بگیرید.

```
class c1 {
 ...
}

class c2 {
 c1 a = new c1();
}
```

در مثال فوق دو مرجع داریم که در جدول ث-۱ لیست شده است.

جدول ث-۱ مراجع create/createby در کد مذکور

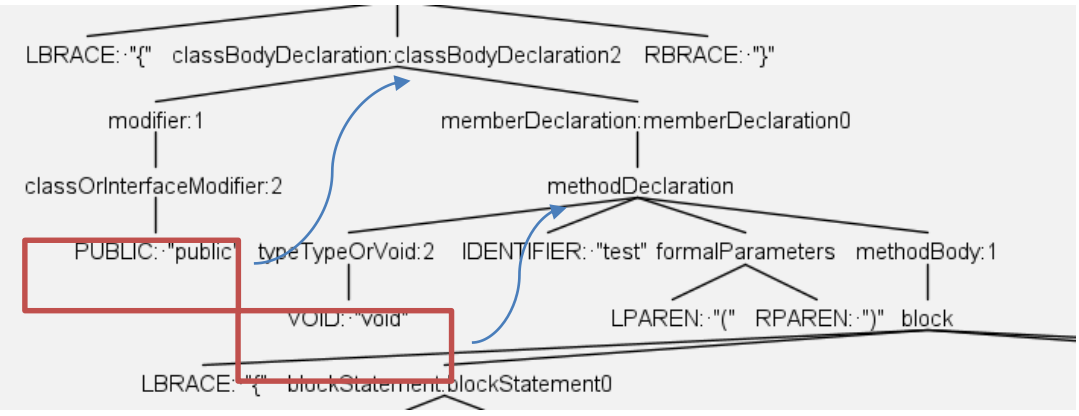
نام مرجع	موجودیتی که مرجع را در برمی‌گیرید.	موجودیتی که مورد ارجاع است.
Java Create	c2	c1
Java Createby	c1	c2

هر مرجع در فایل مخصوص به خود با استفاده از ابزار ANTLR که پیش‌تر مفصل توضیح داده شده است، پیاده‌سازی می‌شود. به‌عنوان مثال این مرجع در فایل create_create.py در لایه analysis_passes پیاده‌سازی شده است. در ادامه به جزئیات پیاده‌سازی این مرجع می‌پردازیم.

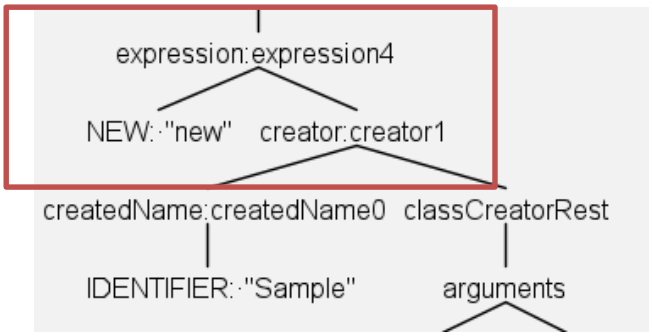
ابزار Understand برای مرجع create به سطح توابع ینی ,public, private و غیره و همچنین به نوع داده بازگشتی تابع مانند void, int و غیره نیاز دارد که برای به‌دست‌آوردن آن از دو تابع find method access و find method return type استفاده شده است. درواقع درخت را روبه‌بالا پیمایش کرد تا به قانون‌های method declaration و class body Declaration رسید.

```
def findmethodreturntype(self, c):
    parents = ""
    context = ""
    current = c
    while current is not None:
        if type(current.parentCtx).__name__ ==
"MethodDeclarationContext":
            parents = (current.parentCtx.typeTypeOrVoid().getText())
            context = current.parentCtx.getText()
            break
        current = current.parentCtx

    return parents, context

def findmethodacess(self, c):
    parents = ""
    modifiers = []
    current = c
    while current is not None:
        if "ClassBodyDeclaration" in type(current.parentCtx).__name__:
            parents = (current.parentCtx.modifier())
            break
        current = current.parentCtx
    for x in parents:
        if x.classOrInterfaceModifier():
            modifiers.append(x.classOrInterfaceModifier().getText())
    return modifiers
```

عملیات فوق در شکل ث-۴، بر روی درخت تجزیه نشان داده شده است.



شکل ث-۴ درخت دسترسی و خروجی تابع



شکل ث-۵ درخت ساخته‌شدن شی جدید

در قانون Expression4 ممکن است ساختن یک شی اتفاق افتد (شکل ث-۵)؛ بنابراین در این قانون وجود ساختن شی جدید بررسی می‌شود و در صورت وجود، اطلاعات آن به یک لیست اضافه می‌شود تا در پایگاه‌داده ذخیره شود.

```
def enterExpression4(self, ctx: JavaParserLabeled.Expression4Context):
    modifiers = self.findmethodacess(ctx)
    mothodedreturn, methodcontext = self.findmethodreturntype(ctx)

    if ctx.creator().classCreatorRest():
        allrefs = class_properties.ClassPropertiesListener.findParents(
            ctx)  # self.findParents(ctx)
        refent = allrefs[-1]
        entlongname = ".".join(allrefs)
        [line, col] = str(ctx.start).split(",")[3].split(":")

        self.create.append({"scopename": refent, "scopelongname":
entlongname, "scopemodifiers": modifiers,
                            "scopereturntype": mothodedreturn,
"scopecontent": methodcontext,
                            "line": line, "col": col[:-1], "refent":
ctx.creator().createdName().getText(),
                            "scope_parent": allrefs[-2] if len(allrefs) >
2 else None,
                            "potential_refent": ".".join(
                                allrefs[:-1]) + "." +
ctx.creator().createdName().getText()})
```

به این ترتیب، اطلاعات مربوط به create استخراج می‌شود. حال کافی است آن را در پایگاه‌داده ذخیره کنیم. این کار در تابع addCreateRefs صورت می‌گیرد.

```
listener = CreateAndCreateBy()
listener.create = []
p.Walk(listener, tree)
p.addCreateRefs(listener.create, file_ent, file_address)
```

کارکرد این تابع به این صورت است که در ابتدا موجودیت‌ها به پایگاه‌داده اضافه می‌شوند و سپس مرجع‌های پیدا شده توسط ANTLR اضافه می‌شوند.

```
def addCreateRefs(self, ref_dicts, file_ent, file_address):
    for ref_dict in ref_dicts:
        scope =
EntityModel.get_or_create(_kind=self.findKindWithKeywords("Method",
ref_dict["scopemodifiers"]),
                                            _name=ref_dict["scopename"],
                                            _type=ref_dict["scopereturnty
pe"]
                                            ,
_parent=ref_dict["scope_parent"] if ref_dict[

          "scope_parent"] is not None else file_ent
                                            ,
_longname=ref_dict["scopelongname"]
                                            ,
_contents=["scopecontent"])[0]
        ent = self.getCreatedClassEntity(ref_dict["refent"],
ref_dict["potential_refent"], file_address)
        Create = ReferenceModel.get_or_create(_kind=190, _file=file_ent,
_line=ref_dict["line"],
                                                _column=ref_dict["col"],
_scope=scope, _ent=ent)
        Createby = ReferenceModel.get_or_create(_kind=191,
_file=file_ent, _line=ref_dict["line"],
                                                _column=ref_dict["col"],
_scope=ent, _ent=scope)
```

فرایند مطرح شده برای مرجع create برای سایر مراجع مشابه است و تنها کافی است پیمایشگرهای درخت متناسب با هر مرجع نوشته شود. باتوجه‌به مستندات Understand حدود ۲۴ نوع مرجع برای زبان جاوا در نظر گرفته شده است. اگر این پروژه را برای هر ۲۴ نوع مرجع تکمیل کنیم آنگاه می‌توان Open Understand را جایگزین Understand کنیم.

مراجع

[1] W. F. Opdyke, "Refactoring object-oriented frameworks," 1992.

[2] M. Fowler, *Refactoring: improving the design of existing code*. Addison-Wesley Professional, 2018.

[3] T. Mariani and S. R. Vergilio, "A systematic review on search-based refactoring," *Information and Software Technology,* vol. 83, pp. 14-34, 2017.

[4] S. A. A. Morteza Zakeri. "Source Code Automated Refactoring Toolkit (CodART)." https://github.com/m-zakeri/CodART (accessed 2022).

[5] T. Parr. "ANother Tool for Language Recognition (ANTLR)." https://www.antlr.org/ (accessed 17 February, 2022).

[6] J. Bansiya and C. G. Davis, "A hierarchical model for object-oriented design quality assessment," *IEEE Transactions on software engineering,* vol. 28, no. 1, pp. 4-17, 2002.

[7] SciTools. "Understand by SciTools." https://www.scitools.com/ (accessed 17 Feburary, 2022).

[8] S. Leary. "JSON in Java." https://github.com/stleary/JSON-java (accessed.

[9] S. Müller. "JOpenChart Library and Toolkit." http://jopenchart.sourceforge.net/ (accessed 17 February, 2022).

[10] H. Burchardt. "jVLT - a vocabulary learning tool." http://jvlt.sourceforge.net/ (accessed 17 February, 2022).

[11] S. A. A. Morteza Zakeri. "OpenUnderstand." https://github.com/m-zakeri/OpenUnderstand (accessed 17 February, 2022).

[12] K. Deb and H. Jain, "An evolutionary many-objective optimization algorithm using reference-point-based nondominated sorting approach, part I: solving problems with box constraints," *IEEE transactions on evolutionary computation,* vol. 18, no. 4, pp. 577-601, 2013.

[13] N. Tsantalis, T. Chaikalis, and A. Chatzigeorgiou, "Ten years of JDeodorant: Lessons learned from the hunt for smells," in *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, 2018: IEEE, pp. 4-14.

[14] S. Kaur, L. K. Awasthi, and A. Sangal, "A brief review on multi-objective software refactoring and a new method for its recommendation," *Archives of Computational Methods in Engineering,* vol. 28, no. 4, pp. 3087-3111, 2021.

[15] A. Ouni, M. Kessentini, H. Sahraoui, K. Inoue, and K. Deb, "Multi-criteria code refactoring using search-based software engineering: An industrial case study," *ACM Transactions on Software Engineering and Methodology (TOSEM),* vol. 25, no. 3, pp. 1-53, 2016.

[16] A. Ouni, M. Kessentini, H. Sahraoui, and M. S. Hamdi, "Search-based refactoring: Towards semantics preservation," in *2012 28th IEEE International Conference on Software Maintenance (ICSM)*, 2012: IEEE, pp. 347-356.

[17] M. W. Mkaouer, M. Kessentini, S. Bechikh, and M. Ó Cinnéide, "A robust multi-objective approach for software refactoring under uncertainty," in *International Symposium on Search Based Software Engineering*, 2014: Springer, pp. 168-183.

[18] A. Ouni, M. Kessentini, H. Sahraoui, and M. Boukadoum, "Maintainability defects detection and correction: a multi-objective approach," *Automated Software Engineering*, vol. 20, no. 1, pp. 47-79, 2013.

[19] M. W. Mkaouer, M. Kessentini, S. Bechikh, K. Deb, and M. Ó Cinnéide, "Recommendation system for software refactoring using innovization and interactive dynamic optimization," in *Proceedings of the 29th ACM/IEEE international conference on Automated software engineering*, 2014, pp. 331-336.

[20] A. Ouni, M. Kessentini, and H. Sahraoui, "Search-based refactoring using recorded code changes," in *2013 17th European Conference on Software Maintenance and Reengineering*, 2013: IEEE, pp. 221-230.

[21] M. W. Mkaouer, M. Kessentini, M. Ó. Cinnéide, S. Hayashi, and K. Deb, "A robust multi-objective approach to balance severity and importance of refactoring opportunities," *Empirical Software Engineering*, vol. 22, no. 2, pp. 894-927, 2017.

[22] A. Ouni, M. Kessentini, H. Sahraoui, and M. S. Hamdi, "The use of development history in software refactoring using a multi-objective evolutionary algorithm," in *Proceedings of the 15th annual conference on Genetic and evolutionary computation*, 2013, pp. 1461-1468.

[23] V. Alizadeh, M. Kessentini, M. W. Mkaouer, M. Ocinneide, A. Ouni, and Y. Cai, "An interactive and dynamic search-based approach to software refactoring recommendations," *IEEE Transactions on Software Engineering*, vol. 46, no. 9, pp. 932-961, 2018.

[24] A. Ouni, M. Kessentini, and H. Sahraoui, "Multiobjective optimization for software refactoring and evolution," in *Advances in computers*, vol. 94: Elsevier, 2014, pp. 103-167.

[25] H. Wang, M. Kessentini, W. Grosky, and H. Meddeb, "On the use of time series and search based software engineering for refactoring recommendation," in *Proceedings of the 7th International Conference on Management of computational and collective intElligence in Digital EcoSystems*, 2015, pp. 35-42.

[26] A. Ouni, M. Kessentini, H. Sahraoui, K. Inoue, and M. S. Hamdi, "Improving multi-objective code-smells correction using development history," *Journal of Systems and Software*, vol. 105, pp. 18-39, 2015.

[27] K. Deb, A. Pratap, S. Agarwal, and T. Meyarivan, "A fast and elitist multiobjective genetic algorithm: NSGA-II," *IEEE transactions on evolutionary computation*, vol. 6, no. 2, pp. 182-197, 2002.

[28] M. Mohan and D. Greer, "MultiRefactor: automated refactoring to improve software quality," in *International Conference on Product-Focused Software Process Improvement*, 2017: Springer, pp. 556-572.

[29] D. Heuzeroth. "RECODER." https://sourceforge.net/projects/recoder/ (accessed 2022.

[30] M. W. Mkaouer, M. Kessentini, S. Bechikh, M. Ó Cinnéide, and K. Deb, "On the use of many quality attributes for software refactoring: a many-objective search-based software engineering approach," *Empirical Software Engineering*, vol. 21, no. 6, pp. 2503-2545, 2016.

[31] J. Blank and K. Deb, "Pymoo: Multi-objective optimization in python," *IEEE Access*, vol. 8, pp. 89497-89509, 2020.

[32] J. Hamano. "Git." https://git-scm.com/ (accessed 17 February, 2022).

[33] C. Li, X. Chu, Y. Chen, and L. Xing, "A knowledge-based technique for initializing a genetic algorithm," Journal of Intelligent & Fuzzy Systems, vol. 31, no. 2, pp. 1145-1152, 2016.

Iran University of Science and Technology
School of Computer Engineering

Design and Implementation of a Tool for Automatic Source Code Refactoring to Improve Software Quality

A Thesis Submitted in Partial Fulfillment
of the Requirements for the Degree of
Bachelor of Science
in
Computer Engineering

Student:
Seyyed Ali Ayati

Supervisor:
Dr. Saeed Parsa

Feburary 2022