DOR: 20.1001.1.2538161.1403.9.4.3.8

DOI: https://doi.org/10.22034/csj.2025.214934

زمستان ۱۴۰۳، دوره ۹، شماره ۴، پیاپی ۳۵

دریافت مقاله: ۱۴۰۳/۰۸/۰۳

پذیرش مقاله: ۱۴۰۳/۰۹/۳۰

نوع مقاله: پژوهشی

بازسازی خودکار کد زبان اصلی نرم‌افزار برای رفع پادالگوهای زبان‌شناختی مربوط به توابع

مهرداد آشتیانی*

استادیار، دانشکده مهندسی کامپیوتر، دانشگاه علم و صنعت، تهران، ایران

پست الکترونیکی: m_ashtiani@iust.ac.ir

محمدامین شهیدی نشرودکلی

دانشجوی کارشناسی ارشد، مهندسی کامپیوتر، دانشگاه علم و صنعت، تهران، ایران

پست الکترونیکی: amin7799sh@gmail.com

فاطمه بخشی

کارشناسی ارشد مهندسی کامپیوتر، دانشگاه علم و صنعت، تهران، ایران

پست الکترونیکی: fatemeh_bakhshi@comp.iust.ac.ir

چکیده

امروزه معیارهای فراوانی برای سنجش کیفیت نرم‌افزار وجود دارد که هر کدام از جهتی به سنجش آن می‌پردازند. وجود یا عدم وجود پادالگوها می‌تواند معیاری برای سنجش کیفیت کدِ زبان اصلی نرم‌افزار باشد. برخی پادالگوهای شناخته شده با توجه به تاثیرشان بر روی کارایی، اطمینان‌پذیری و سایر معیارهای مرتبط مطالعه می‌شوند. یکی از این معیارها، خوانایی کدِ زبان اصلی نرم‌افزار است. سازنده یا سازندگان یک برنامه تنها کسانی نیستند که به توسعه آن می‌پردازند. به همین دلیل رعایت نکاتی که فهمیدن چگونگی کارکرد برنامه را برای توسعه‌دهندگان احتمالی ساده‌تر می‌کند ضروری است. این مقاله به بررسی پادالگوهای مرتبط با نام‌گذاری توابع و ارائه راه‌حل برای رفع خودکار آن‌ها می‌پردازد. به پادالگوهایی که مربوط به نام‌گذاری توابع، متغیرها و سایر اعضا در کدِ زبان اصلی نرم‌افزار و همچنین مستندات آن هستند پادالگوهای زبان‌شناختی گفته می‌شود. در این مقاله روشی برای تشخیص و رفع خودکار پادالگوهای زبان‌شناختی به کمک درخت نحو انتزاعی ارائه می‌شود. سپس روش ارائه شده بر روی کدِ زبان اصلی چهار نرم‌افزار متن‌باز که در مجموع بیش از ۱۵۰۰۰۰ خط کد دارند، آزمایش می‌شود. در نتیجه این آزمایش‌ها ۵۱ پادالگو به درستی شناسایی و اصلاح می‌شوند. اصلاح شدن به گونه‌ای است که حدود ۵۰ خط کد به مجموعه کدها اضافه می‌کند. مزیت این پژوهش نسبت به پژوهش‌های مشابه، ارائه روشی برای رفع پادالگوها و اعمال تغییرات در کدِ زبان اصلی است. زیرا بسیاری از پژوهش‌ها تنها به تشخیص پادالگوهای زبان‌شناختی اکتفا کرده‌اند. همچنین، تاثیری که این بازسازی بر روی تعداد خطوط کدِ زبان اصلی می‌گذارد نیز بررسی شده است.

* نویسندهٔ مسئول

واژه‌های کلیدی: پادالگو، پادالگوی زبان‌شناختی، بازساخت خودکار کد، کد تمیز، درک‌پذیری.

۱. مقدمه

تحقیقات اخیر نشان می‌دهد انتخاب واژگان نامناسب در کدِ زبان اصلی نرم‌افزار می‌تواند در فهم، قابلیت نگهداری و کیفیت کلی نرم‌افزار تاثیر منفی بگذارد [۱]. کسی که یک برنامه را از ابتدا توسعه می‌دهد به احتمال قوی دچار سوء برداشت نمی‌شود زیرا تسلط کاملی روی کد برنامه دارد. اما برای فردی که وظیفه نگهداری، توسعه و یا تغییر یک برنامه را بر عهده دارد این احتمال وجود دارد که در فهم کارکرد برنامه دچار اشتباه شود. بسیاری از نرم‌افزارها (یا قسمتی از آن‌ها) توسط افرادی نوشته می‌شوند که تحصیلات رسمی در زمینه مهندسی نرم‌افزار ندارند [۲]. حال اگر چنین افرادی اقدام به توسعه کدِ زبان اصلی نرم‌افزاری کنند که از قبل توسط فردی دیگر و با قواعد نام‌گذاری نامناسب توسعه یافته، به احتمال قوی دچار برداشت ناصحیح در درک نحوه کارکرد برنامه و یا حتی هدف آن می‌شود.

یک کدِ زبان اصلی ناخوانا می‌تواند علاوه بر معیار خوانایی، بر روی سایر معیارها نیز اثر منفی داشته باشد. برای مثال نگهداری نرم‌افزار که یک فرایند هزینه‌بر است نیازمند فهمی صحیح از کارکرد قسمت‌های مختلف کدِ زبان اصلی برنامه است [۳]. اما، وجود واژگان نامناسب در نام توابع می‌تواند باعث گمراهی در این مورد شود و فرایند نگهداری را با مشکل روبه‌رو، و پرهزینه‌تر کند. یک مسئله مهم در هنگام مواجهه با نرم‌افزارهای قدیمی، سهولت درک آن‌ها بوده که یک پیش‌نیاز ضروری است [٤]. این امر می‌تواند با وجود مستندات مناسب برای توسعه‌دهندگان جدید نرم‌افزار قدیمی فراهم شود. عدم هم‌خوانی در مستندات و کدِ زبان اصلی نرم‌افزار نیز یکی دیگر از جنبه‌های پادالگوهای زبان‌شناختی است.

با توجه به این که پادالگوها بخش بزرگی از مطالعات مربوط به مهندسی نرم‌افزار را معطوف به خود کرده‌اند، راه‌حل‌هایی نیز برای رفع آنها پیشنهاد شده است. رفع پادالگوها در کدِ زبان اصلی نرم‌افزار که به آن بازساخت کد[1] گفته می‌شود برای یک کدِ زبان اصلی بزرگ، در صورتی که توسط انسان انجام شود کاری پیچیده و زمان‌بر است، بدین‌جهت این بازسازی را می‌توان به عهده رایانه گذاشت تا در زمان و هزینه صرفه‌جویی شود، در این حالت به آن بازساخت خودکار کد[2] گفته می‌شود. بازساخت خودکار کد شامل دو گام اصلی تشخیص و اصلاح است. عمل پیدا کردن نابسامانی کد[3] و رسیدن به پادالگوها، در گام تشخیص است. پیاده‌سازی راه‌حل پیشنهادی مناسب در کد برای رفع پادالگوها در گام اصلاح است. مطالعات قبلی اهمیت پادالگوها را برای توسعه‌دهندگان نرم‌افزار بررسی کرده است. با این حال، پژوهش‌ها در جهت اصلاح کد و رفع پادالگوها کمتر تمرکز داشته‌اند و این تحقیق به این شکاف می‌پردازد. این مقاله به ارائه روشی برای تشخیص و اصلاح برخی از پادالگوهای زبان‌شناختی در نام‌گذاری توابع در کدِ زبان اصلی جاوا به کمک زبان پایتون می‌پردازد. این کار با کمک درخت نحو انتزاعی، ایجاد تمامی نشانه‌های مربوط به توابع و در نتیجه مشخصات توابع و بررسی آن با پادالگوهای شناخته شده انجام می‌شود. سپس روش ارائه شده بر روی کد زبان اصلی چهار نرم‌افزار متن‌باز که در مجموع بیش از ۱۵۰۰۰۰ خط کد دارند، مورد آزمایش قرار می‌گیرد. در نتیجه این آزمایش‌ها، ۵۱ پادالگو به‌طور صحیح شناسایی و اصلاح می‌شوند.

در ادامه، در بخش ۲ معانی و مفاهیم مورد نیاز برای درک مقاله ارائه می‌شود. در بخش ۳ به بررسی کارهای مرتبط و مزایای و معایب هر کدام پرداخته می‌شود. در بخش ٤ پادالگوهای زبان‌شناختی مورد مطالعه معرفی می‌شوند. روش پیشنهادی در بخش ۵ ارائه شده و در بخش ٦ مورد ارزیابی قرار می‌گیرد. بخش ۷ به نتیجه‌گیری اختصاص دارد.

1-Code refactoring
2-Automatic code refactoring
3-Code smell

۲. مفاهیم و تعاریف مبنایی

آنچه در این بخش به نوشتار در آمده است شرحی است بر معانی و مفاهیمی که برای درک آنچه این مقاله بر روی آن مطالعه داشته ضروری است. در تشخیص و رفع پادالگوها از برخی از این مفاهیم استفاده شده و یا مستقیماً مورد بحث قرار نگرفته‌اند.

۲-۱- پادالگو

ابتدا به مفهوم الگو بپردازیم. یک الگو، راه‌حلی متداول است که در پاسخ به یک مسئله ارائه می‌شود [۵]. الگو یک روش ایده‌آل است که با استفاده از دانش تخصصی در حوزه نرم‌افزار به گونه‌ای به دست می‌آید که بتوان از آن دُر طراحی انواع مختلف نرم‌افزار استفاده مجدد کرد [۵]. پادالگو تعریفی مشابه به الگو دارد، زیرا یک پادالگو نیز برای پاسخ به یک مسئله مکررا استفاده می‌شود، اما دلیلی که به آن پادالگو گفته می‌شود این است که استفاده از آن آثار منفی برجای می‌گذارد.

پادالگوها یادآور مسئله اختراع دوباره چرخ هستند. مهندسان نرم‌افزار در طول تاریخ راه‌حل‌های مختلفی را برای مشکلات پرتکرارشان اختراع کرده‌اند. برخی از این راه‌حل‌ها در طول زمان درستی خود را ثابت کرده‌اند. الگوها مجموعه‌ای از این راه‌حل‌ها هستند، و پادالگوها آن راه‌حل‌هایی هستند که با گذشت زمان اثرات منفی‌شان آشکار شد.

۲-۲- پادالگوی زبان‌شناختی

پادالگوهای زبان‌شناختی به راه‌حل‌های بد در نام‌گذاری، مستندسازی و پیاده‌سازی موجودیت‌های کد اطلاق می‌شود که می‌تواند کیفیت سامانه‌های رایانه‌ای را کاهش دهد و تاثیر منفی بر روی درک‌پذیری برنامه داشته باشد [۶]. یک پادالگو زبان‌شناختی به طور مستقیم بر روی عملکرد برنامه تاثیر نمی‌گذارد اما با پیچیده کردن مواردی همچون نگهداری و توسعهٔ نرم‌افزار، می‌تواند در کاهش عملکرد نیز مؤثر باشد. برای مثال تابعی را در نظر بگیرید که از نامش به نظر می‌رسد متغیری را مقداردهی می‌کند. چنین توابعی که توابع مقداردهنده[4] نام دارند معمولاً نامشان با set شروع می‌شود. توسعه‌دهنده‌ای که شناختی از جزئیات کارکرد این تابع ندارد فرض می‌کند این تابع متغیر یا متغیرهایی را مقداردهی می‌کند ولی مقداری را برنمی‌گرداند (خروجی ندارد). در نتیجه توسعه‌دهنده مقدار برگردانده شده را حتی ذخیره نیز نمی‌کند. پس منابع سخت‌افزاری و نرم‌افزاری صرف شده برای برگشت دادن یک مقدار در تابع، به هدر رفته است. در شکل ۱ مثالی از یکی از توابع نرم‌افزار آرگو یوامال آورده شده است. تابع که از نامش مشخص است یک تابع مقداردهنده است، مقداری را نیز برگردانده است. توسعه‌دهنده جدید ممکن است اطلاعاتی مفید یا ضروری را به دلیل عدم اطلاع از وجود یک مقدار بازگردانده شده از دست بدهد.

```
public Dimension setBreadth
        (Dimension target, int source) {
  if (orientation == VERTICAL)
    return new Dimension(source,
        (int)target.getHeight());
  else
    return new Dimension(
                (int)target.getWidth(), source);
}
```

شکل ۱. تابع مقداردهنده‌ای که مقدار برمی‌گرداند [۱].

وجود پادالگوها در پروژه‌های نرم‌افزاری می‌تواند مشکل‌ساز باشد. برای مثال نتایج مقاله [۲۱] نشان داد که احتمال بروز اشکال برای توسعه‌دهندگان، هنگامی که از واسطهای کاربری که در آن‌ها از پادالگوها استفاده شده است در مقایسه با واسطهای کاربری بدون پادالگو، ۲۹ درصد بیشتر افزایش می‌یابد.

۲-۳- درخت نحو انتزاعی

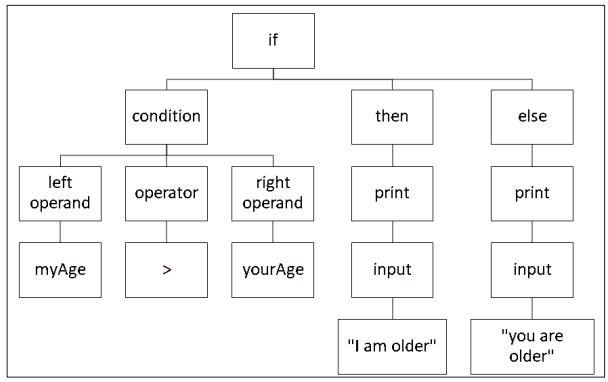
در علوم رایانه، درخت نحو انتزاعی، نمایشی درختی از ساختار نحوی انتزاعی متن (اغلب کدِ زبان اصلی) نوشته‌شده به زبان رسمی است. هر گره درخت نشان‌دهنده یک ساختار در متن است [۷]. علت انتزاعی نامیدن این درخت این است که تمامی جزئیات همان‌گونه که در متن

4-Set methods

کدِ زبان اصلی مشاهده می‌شوند در درخت نیستند. برای مثال در زبان‌های برنامه‌نویسی که در آنها از پرانتز برای نوشتن شرط‌ها استفاده می‌شود، هنگام نمایش درخت نحو متن کدِ زبان اصلی، پرانتزها به صورت مستقیم نمایش داده نمی‌شوند. برای مثال می‌توان با قرار دادن تمامی اجزایی که درون یک پرانتز قرار دارند، در زیر یک گره والد (گره پدر)، به صورت ضمنی نشان داد که تمام آن‌ها درون یک پرانتز هستند. در شکل ۲ چنین شرطی در یک قطعه کد نمایش داده شده است. نمایش این قطعه کد به صورت درخت نحو در شکل ۳ آمده است.

```
if ( myAge > yourAge) {
   print("I am older");
   }
else {
   print("You are older");
   }
```

شکل ۲. مثال یک شرط



شکل ۳. درخت نحو انتزاعی برای قطعه کد شکل ۲.

۳.کارهای مرتبط

عمده تحقیقات صورت‌گرفته بر روی جنبه زبان‌شناختی کدِ زبان اصلی، به کیفیت، درک‌پذیری کد و یا تشخیص پادالگوهای زبان‌شناختی می‌پردازند. در این بخش خلاصه‌ای از تحقیقاتی که در این سه حوزه صورت گرفته، ارائه می‌شود.

۳-۱- تحقیقات مبتنی بر همایندی اسامی

یکی از معیارهای مورد مطالعه قرارگرفته برای سنجش درک‌پذیری نرم‌افزار، همایندی اسامی[5] است. در پژوهش مربوط به لیون (۲۰۱۲) [۸] مطالعه بر روی نوشتار کدِ زبان اصلی ۳۱ سیستم متن‌باز[6] نشان می‌دهد در ۸۳ درصد از توابع، همایندی اسامی رخ داده است. یعنی در آن‌ها حداقل یک اسم هم در نام تابع و هم در شناسه‌های مختلف بدنه تابع موجود است. اگر همایندی اسامی در تابعی رخ دهد، شانس این که تابع درک‌پذیر به حساب آید بیشتر می‌شود. در توابعی که طولانی‌اند یا فرایندهای پیچیده‌ای را پیاده‌سازی می‌کنند، استفاده از همایندی اسامی توصیه می‌شود.

۳-۱-۱- اسامی توابع درک‌پذیر و همایندی اسامی

از آنجایی که درک یک تابع امری ذهنی است، افراد مختلف می‌توانند تحلیل متفاوتی از میزان درک‌پذیری تابع داشته باشند. وجود همایندی برای اسامی که می‌توانند بیانگر خوبی از عملکرد تابع باشند مفید است. همچنین، استفاده از کلمات کوتاه در به خاطر ماندن تابع در ذهن افراد کمک می‌کند. می‌توان همایندی اسامی را معیاری قابل‌اندازه‌گیری برای درک‌پذیری دانست. این امر با شمارش تعداد دفعاتی که یک اسم در پیاده‌سازی تابع آمده است انجام می‌گیرد. هر چقدر این تعداد بیشتر باشد، نقش آن اسم در تابع پررنگ‌تر است.

با بررسی همایندی اسامی بین نام تابع و پیاده‌سازی آن می‌توان تا حدی دریافت که نام تابع چقدر مناسب عملکرد آن است. این کار برای تمامی کلمات موجود در نام تابع تکرار می‌شود تا دریابیم کدام یک ارتباط محکم‌تری با عملکرد تابع دارد. جدول ۱ نتایج بررسی درک‌پذیری دو گروه از توابع است. در یک گروه، توابع دارای همایندی اسامی هستند اما در دیگری چنین شرطی وجود ندارد.

مزایای استفاده از همایندی اسامی به عنوان معیار سنجش درک‌پذیری توابع:

۱. احتمال این که یک تابع دارای همایندی اسامی قابل‌درک باشد ۱۹ درصد بیشتر از حالتی است که همایندی اسامی نداشته باشد.

5-Co-occurrence of nouns
6-Open source

جدول ۱. مقایسه درک پذیری توابع [۸].

دارای همایندی اسامی	بدون همایندی اسامی	معیار
۱	۶	سخت بودن فهم پیاده‌سازی
۸۲	۵۳	تابع درک‌پذیر
۱۶	۳۱	تابع درک‌پذیر سخت
۱	۶	خطای Part of speech
۰	۴	ایجاد کنندهٔ Exeption
۱۰۰	۱۰۰	کل

۲. اگر همایندی اسامی مربوط به کلمات تخصصی حوزهٔ کاری نرم‌افزار باشد، خواننده متن کد از موضوع حوزهٔ نرم‌افزار مربوطه نیز مطلع می‌شود.

۳. با مقایسه تعداد تکرار کلمات مختلف می‌توان اهمیت آن‌ها را در کدِ زبان اصلی مقایسه و رده‌بندی کرد.

معایب استفاده از همایندی اسامی به عنوان معیار سنجش درک‌پذیری توابع:

۱. این روش دقیق نیست. به عبارت دیگر، هیچ تضمینی وجود ندارد که رخدادن همایندی اسامی باعث افزایش درک‌پذیری تابع شود.

۲. استفاده متعدد از یک کلمه در قسمت‌های مختلف کد، با وجود این که مصداق رخدادن همایندی اسامی است، در صورت استفادهٔ بی‌مورد می‌تواند درک‌پذیری کدِ زبان اصلی را کاهش دهد.

۳. رخدادن همایندی اسامی در نام شناسه‌ها ممکن است باعث شباهت بیش از اندازهٔ اسامی شناسه‌ها و در نتیجه، کاهش درک‌پذیری کد شود.

٤. اگر کلمه‌ای که مکررا تکرار شده به عملکرد کدِ زبان اصلی نامربوط باشد، نتیجه معکوس خواهد داشت و درک‌پذیری کد به شدت کاهش می‌یابد.

۳-۱-۲- استفاده از افعال برای یافتن همایندی اسامی

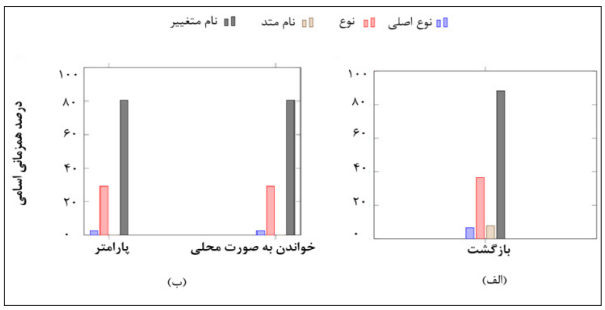
برنامه‌نویسان اغلب از یک فعل خاص در نام تابع استفاده می‌کنند. اسامی مانند is، set، get این‌گونه هستند.

آزمایش‌های لیون (۲۰۱۲) [۸] نشان می‌دهد بیشترین افعال استفاده شده در نام توابع به ترتیب موارد نشان داده شده در جدول ۲ هستند.

جدول ۲. بیشترین افعال مورد استفاده در نام توابع [۸].

درصد توابع	تعداد توابع	فعل	رتبه
۲۶/۱۹	۳۴۲۰۰	Get	۱
۱۱/۰۳	۱۴۴۰۸	Set	۲
۴/۴۵	۵۸۱۰	Is	۳
۲/۹۵	۳۸۵۸	Add	۴
۲/۸۲	۳۶۸۸	Create	۵
۱/۴۴	۱۸۷۸	Visit	۶
۱/۳۶	۱۷۸۰	Remove	۷
۱/۱۲	۱۴۶۲	Test	۸
۱/۰۶	۱۳۸۳	Has	۹
۰/۸۰	۱۰۴۰	Read	۱۰

همچنین آزمایش‌های لیون(۲۰۱۲) نشان می‌دهد همایندی اسامی در توابعی که نامشان با افعال مختلفی آغاز می‌شود، در نقاط مختلفی از پیاده‌سازی تابع رخ می‌دهد. تصویر شکل ٤ نتایج تحقیقات لیون (۲۰۱۲) برای توابعی است که نامشان با get و set آغاز می‌شود.



شکل ۴. همایندی اسامی در توابعی که نامشان با get (الف) و set (ب) آغاز می‌شود [۸].

مزایای استفاده از افعال برای یافتن همایندی اسامی به شرح زیر است:

۱. در صورت پیاده‌سازی روشی برای پیداکردن همایندی اسامی در توابع، با توجه به نام توابع می‌توان

قسمت‌های مختلف بدنه تابع را جهت جست‌وجو اولویت‌بندی کرد.

۲. می‌توان از این ابزار استفاده معکوس کرد. در شرایطی که توسعه‌دهنده از کلمات جدول ۲ در ابتدای نام تابعی استفاده نکرده باشد، با توجه به آن قسمت‌های تابع که در آن‌ها همایندی اسامی رخ داده است، می‌توان با احتمال بالایی اظهار کرد که بهتر است نام تابع با کدام یک از کلمات جدول ۲ آغاز شود.

۳. با بررسی کلمهٔ آغازین نام تابع و نقاط رخ دادن همایندی اسامی در بدنه آن، می‌توان مناسب بودن نام تابع را مورد ارزیابی قرار داد.

معایب استفاده از افعال برای یافتن همایندی اسامی به شرح زیر است:

۱. بررسی نام تابع و کلمه آغازین آن، نیازمند شکستن نام تابع به عبارات تشکیل‌دهنده آن است. برای این کار نیز باید روشی مناسب انتخاب شود.

۲. اگر برنامه‌نویس از شیوه برنامه‌نویسی خاصی استفاده کرده باشد که در آن از کلمات جدول ۲ در نامگذاری توابع استفاده نمی‌شود، روش مطرح شده در این بخش کارآمد نخواهد بود.

۳-۲- تحقیقات مبتنی بر اثر اجزای کد بر کیفیت کد

قواعدی که در برنامه‌نویسی جهت افزایش کیفت به کار می‌روند شامل اجزای مختلفی از کدِ زبان اصلی می‌شود. عدم رعایت این قوانین، با توجه به این که برای چه عضوی از کدِ زبان اصلی است می‌تواند آثار مختلفی داشته باشد. برای مثال انتخاب یک نام نامرتبط برای تابع می‌تواند بر روی درک‌پذیری کد تاثیر منفی بگذارد اما لزوماً بر روی خوانایی تاثیری نداشته باشد.

۳-۲-۱- تاثیر توضیح‌ها[7] بر کیفیت کدِ زبان اصلی

بخش بزرگی از کدِ زبان اصلی در سیستم‌های نرم‌افزاری از توضیح‌ها تشکیل می‌شود که پیاده‌سازی‌ها را مستند می‌کنند و به توسعه‌دهندگان در درک کد کمک می‌کنند [۹]. همچنین توضیح‌ها دومین نوع پراستفاده مستندات برای درک‌پذیری کد هستند [۱۰]. توسعه‌دهندگان به طور گسترده‌ای معتقدند که مستندسازی ضعیف باعث سوء برداشت می‌شود [۱۱]. همچنین تحقیقات نشان می‌دهد مستندسازی ضعیف بر روی قابلیت نگهداری نرم‌افزار تاثیر منفی می‌گذارد. استفاده از توضیح یا هر روش مستندسازی دیگری ممکن است تحت تاثیر بازه‌های زمانی محدودی که برای تیم توسعه‌دهندگان تعریف می‌شد قرار گیرد. استیدل(۲۰۱۳) [۹] برای زبان‌های برنامه‌نویسی C/C ++ و جاوا توضیح‌ها را دسته‌بندی می‌کند و بیان می‌کند تعداد توضیح‌ها در کد بیانگر استفاده مناسب از توضیح برای درک‌پذیری نیست زیرا برخی از این انواع کمکی به این هدف نمی‌کنند. دسته‌بندی توضیح‌ها به صورت زیر است:

۱. توضیح‌های حق نسخه‌برداری شامل اطلاعاتی درباره حق نسخه‌برداری فایل کدِ زبان اصلی هستند. این توضیحات معمولاً در ابتدای فایل مشاهده می‌شوند.

۲. توضیح‌های سرتیتر دیدی کلی نسبت به عملکرد کلاس و همچنین اطلاعاتی مانند نویسنده، شمارهٔ تجدید نظر و غیره می‌دهد. در فایل جاوا این‌گونه توضیح‌ها معمولاً بعد از گنجاندن کتابخانه‌ها و قبل از تعریف تابع دیده می‌شوند.

۳. توضیح‌های عضو عملکرد یک عضو (تابع یا صفت) را شرح می‌دهند و در همان خط تعریف عضو و یا خط قبلی‌اش نوشته می‌شوند. این توضیح‌ها اطلاعاتی برای توسعه‌دهنده و یا رابط برنامه‌نویسی کاربردی[8] مهیا می‌کنند.

٤. توضیح‌های خطی تصمیماست مربوط به پیاده‌سازی را درون یک تابع توضیح می‌دهد.

٥. توضیح‌های بخش به توابع یا صفت‌هایی اشاره می‌کنند که به جنبهٔ عملکردی مشترکی تعلق دارند.

٦. توضیح‌های کد شامل قسمت‌هایی از کد می‌شوند که

7-Comments

8-Apllication programming interface(API)

جدول ۳. رابطه‌ی میان درک‌پذیری کد و اثرات انواع توضیح‌ها بر روی کد [۹].

		درک‌پذیری کد زبان اصلی							
		استفاده از کد				توسعه کد			
		شناخت وظیفه، خطا، تکنیک	درک جزئیات پیاده‌سازی	درک جزئیات طراحی	خوانایی کد	شناخت کپی‌رایت / نویسنده	درک طراحی سیستم	صدا زدن متدهای عمومی	دسترسی به فیلدهای عمومی
توضیح‌ها	حق نسخه‌برداری					+			
	سرتیتر					+	+		
	خطی		+		+				
	عضویت			+			+	+	+
	بخش		+	+					
	وظیفه	+							
	کد		−	−	−				

توضیح شده و کامپایلر از آن چشم‌پوشی می‌کند. معمولاً کد برای استفاده‌های آتی و یا اشکال‌زدایی توضیح گذاری می‌شود.

۷. توضیح‌های وظیفه یادداشت‌های توسعه‌دهنده هستند که کارهای باقی‌مانده، اشکال‌های رفع‌نشده و میانبرهای استفاده‌شده در پیاده‌سازی را یادآوری می‌کنند.

در جدول ۳ رابطه میان درک‌پذیری کد و اثرات مثبت و منفی انواع توضیح‌ها گزارش داده شده است.

مزایای ارزیابی نقش توضیح‌ها در کیفیت کدِ زبان اصلی به شرح زیر است:

۱. انتخاب نوع توضیح متناسب با آثار موردنظر بر روی درک‌پذیری کد.
۲. عدم استفاده از توضیح‌های کد به دلیل آثار منفی بر روی جنبه‌هایی از درک‌پذیری آن.
۳. طبقه‌بندی معیارهای درک‌پذیری کدِ زبان اصلی.

معایب ارزیابی نقش توضیح‌ها در کیفیت کدِ زبان اصلی به شرح زیر است:

۱- عدم ارائه چارچوبی ساختارمند برای ایجاد توضیح‌های مناسب.

۳-۲-۲- رابطه نام‌گذاری شناسه‌ها و کیفیت کدِ زبان اصلی

در مقاله لوسیا (۲۰۱۱) [۱۲] به نقش مهم انتخاب شناسه‌های کد در کیفیت نرم‌افزار اشاره شده است. اسنید (۱۹۹۶) [۱۳] در مشاهداتش بیان می‌کند در بسیاری از سیستم‌های قدیمی، توسعه‌دهندگان نام‌گذاری توابع، شناسه‌ها و ساختارهای داده را به صورت دلخواه انجام داده‌اند، بدون این که نام آن‌ها را به عملکردشان ربط داده باشند. تحقیقات مارکوس و هایدوک (۲۰۰۸) [۱۴] که چندین سیستم نرم‌افزاری متن‌باز را مورد مطالعه قرار می‌دهد، بیان می‌کند حدود ۴۰ درصد کلمات تخصصی حوزه مربوط به موضوع سیستم‌ها در کدِ زبان اصلی نیز تکرار شده‌اند.

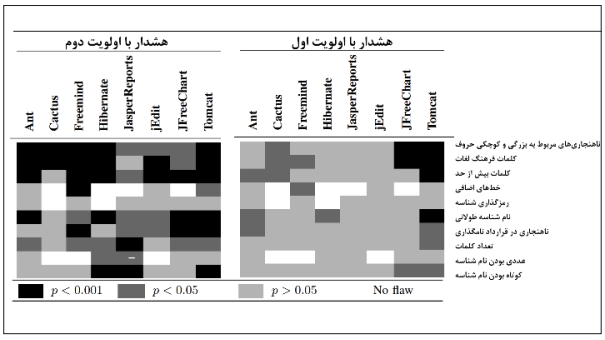
تحقیقات بسیاری درباره تحلیل کیفیت کد صورت گرفته است. لاوری (۲۰۰۷) [۱۵] کیفیت شناسه‌ها را در بیش از پنجاه میلیون خط کد از ۴ زبان برنامه‌نویسی مختلف را که

در یک بازهٔ زمانی ۳۰ ساله نوشته شده‌اند بررسی کرده است. این خطوط شامل کدِ متن‌باز و اختصاصی می‌شود. در این تحقیق ارزیابی کیفیت شناسه‌ها با پیدا کردن آنها به کمک فرهنگ لغات و یا اختصارات[9] شناخته شده انجام شده است. نتیجه تحقیقات لاوری (۲۰۰۷) نشان می‌دهد روش‌های جدید برنامه‌نویسی شناسه‌های با کیفیت‌تری تولید می‌کند. همچنین نشان داد عبارات اختصاری در کدهای اختصاصی، بیشتر از متن‌باز استفاده می‌شوند.

باتلر (۲۰۰۹) [۱۶] کیفیت شناسه‌های به‌کار رفته در ۸ نرم‌افزار متن‌باز نوشته شده به زبان جاوا را با استفاده از ۱۲ قاعده نام‌گذاری بررسی کرده است. در این تحقیقات رابطه مشخصی میان شناسه‌هایی که در آن‌ها حداقل یکی از قواعد نام‌گذاری رعایت نشده بود و مشکلات مربوط به کیفیت کدی که توسط ابزار FindBugs گزارش شده بود وجود داشت. ۱۲ قاعده‌ای که باتلر در این تحقیقات در نظر گرفت به شرح زیر است:

۱. در شناسه‌ها باید از حروف بزرگ به درستی استفاده شود.
۲. از خط‌تیره‌های متوالی نباید در کنار هم استفاده شود.
۳. اسامی شناسه‌ها باید از کلماتی تشکیل شوند که در فرهنگ لغات یافت می‌شود. همچنین باید از کلمات مخففی استفاده شود که شناخته شده هستند.
٤. اسامی شناسه‌ها باید حداکثر از ٤ کلمه یا مخفف تشکیل شده باشد.
۵. در صورت عدم وجود دلایل متقاعدکننده، ثابت‌های شمارش[10] باید به ترتیب حروف اظهار شوند.
٦. شناسه‌ها نباید در ابتدا و انتهای نامشان دارای خط‌تیره باشند.
۷. اطلاعات نوع دادهٔ شناسه نباید با استفاده از نماد مجارستانی[11] و یا روش‌های مشابه در نام شناسه گنجانده شود.
۸. تا جای ممکن باید از انتخاب اسامی طولانی برای شناسه‌ها خودداری کرد.
۹. اسامی شناسه‌ها نباید دارای ترکیبی نامنظم از حروف بزرگ و کوچک باشد.
۱۰. نام شناسه‌ها باید متشکل از ۲ تا ٤ کلمه باشد.
۱۱. نام شناسه‌ها نباید تماماً از کلمات عددی و یا اعداد تشکیل شده باشد.
۱۲. نام شناسه‌ها باید حداقل از ۸ حرف به غیر از c، d، e، g، i، in، inOut، j، k، m، n، o، out، t، x، y و z تشکیل شده باشد.

باتلر به کمک ابزار FindBugs کد نرم‌افزارهای مربوطه را تحلیل کرد. این ابزار برای تجزیه و تحلیل کد ایستای برنامه‌های جاوا به کار می‌رود و اولویت داده شده توسط ابزار نشان‌دهنده اهمیت هشدار است. باتلر تعداد اخطارهایی با اولویت یک و اولویت دو به دست آورد. شکل ۵ رابطهٔ بین اشتباه در نام‌گذاری شناسه‌ها و اخطارهای FindBugs را در ۸ نرم‌افزار مربوطه نشان می‌دهد. شکل ۵ نشان می‌دهد در آن دسته از کلاس‌هایی که اخطارهای دارای اولویت دو به دست آمده است، احتمال وجود مصادیق نقض ۱۲ قاعدهٔ نام‌گذاری مطرح شده بیشتر است. همچنین برخی از این مصادیق رابطه مشهودتری با اخطارها دارند. برای مثال عدم رعایت استفادهٔ مناسب از حروف بزرگ در کلاس یا کلاس‌های ۵ مورد از نرم‌افزارهایی که در آن‌ها اخطار با اولویت دو رخ داده است وجود داشته است.



شکل ۵. رابطه اخطارهای FindBugs و قواعد نامگذاری [۱۶].

مزایای بررسی رابطه نام‌گذاری شناسه‌ها و کیفیت کدِ زبان اصلی:

9-Abbreviations
10-Enum constants
11- Hungarian notation

۱. طبقه‌بندی قواعد نامگذاری با توجه به تعدد و آثارشان بر روی کدِ زبان اصلی.
۲. پیروی از شیوه‌های برنامه‌نویسی‌ای که ۱۲ قاعدهٔ نامگذاری این تحقیقات را رعایت می‌کنند.
۳. استفاده از ابزار FindBugs برای پیدا کردن کدهایی که احتمال نقض قوانین ۱۲گانه نامگذاری در آن‌ها بالا است.

معایب بررسی رابطه نام‌گذاری شناسه‌ها و کیفیت کد:

۱. روابطی که در این تحقیق بیان‌شده‌اند احتمالی بوده و نمی‌توان به صورت کامل آن‌ها را مبنای کارهای دیگر قرار داد.
۲. عدم وجود رابطه منطقی میان اخطارهای دارای اولویت یک و عدم ارائه توضیح منطقی دربارهٔ آن.

۳-۳- تحقیقات مبتنی بر آشکارسازی

با دانستن انواع مختلفی از ایرادات که می‌توانند در مورد نامگذاری‌ها رخ داده باشند، می‌توان چگونگی تشخیص آن‌ها را تحت روش‌هایی نظام‌مند ارائه داد. استخراج ویژگی‌های کد یکی از ویژگی‌های مهم این روش‌ها است. همچنین در بررسی نام‌ها و عبارات به کار رفته در کد می‌توان از تکنیک‌های حوزهٔ پردازش زبان‌های طبیعی بهره گرفت.

۳-۳-۱- آشکارساز پادالگوی زبان‌شناختی (LAPD [۱۲])

آرنودوا (۲۰۱٤) [۱۷] برای تشخیص پادالگوهای زبان‌شناختی ابزاری به نام LAPD معرفی می‌کند. وی در این ابزار الگوریتم‌های تشخیص را برای زبان‌های برنامه‌نویسی جاوا و C++ پیاده‌سازی کرده است. LAPD از تجزیه‌گر زبان طبیعی استنفورد استفاده می‌کند. سپس با تحلیل توضیح‌ها و اجزای کد، ارتباط بین آن‌ها را کشف و بررسی می‌کند. همچنین برای بررسی معنی‌دار بودن کلمات و ارتباط بین آن‌ها از پایگاه دادهٔ واژگان وردنت[۱۳] استفاده می‌کند. آرنودوا کارایی این ابزار را با بررسی کدِ زبان اصلی چند نرم‌افزار ارزیابی کرده است. جدول ۳ برای هر یک از نرم‌افزارهای مورد بررسی تعداد پادالگوهای تشخیص داده شده و آنچه در عمل وجود دارد را بیان می‌کند. همچنین دقت الگوریتم‌های ارائه شده نیز مشهود است.

مزایای روش پیشنهادی آرنودوا به شرح زیر است:

۱. در ابزار LAPD برای تمام پادالگوهای زبان‌شناختی الگوریتمی برای آشکارسازی ارائه شده است.
۲. پایگاه دادهٔ وردنت ۲۰۰ زبان را پشتیبانی می‌کند و فقط به زبان انگلیسی متکی نیست.
۳. با توجه به جدول ٤ این روش در اکثر موارد از دقت بالایی برخوردار است.

معایب روش پیشنهادی آرنودوا به شرح زیر است:

۱. برخی از الگوریتم‌های معرفی‌شده نیازمند بررسی کلمهٔ آغازین نام توابع و شناسه‌ها هستند اما این روش چگونگی انجام این کار را بیان نمی‌کند.
۲. دقت این روش با توجه به جدول ۳ برای تشخیص پادالگوهای مرتبط با توضیح‌ها پایین است.
۳. این روش راهکاری برای اصلاح پادالگوهای آشکارشده ارائه نمی‌دهد

۳-۳-۲- استفاده از یادگیری ماشین

لیو (۲۰۱۹) [۱۸] روشی مبتنی بر یادگیری ماشین و شبکه‌های عصبی هم‌آمیختی[۱٤] ارائه می‌دهد و در آن علاوه بر کشف ناسازگاری‌ها در نامگذاری توابع، پیشنهادهایی نیز برای رفع آن‌ها به توسعه‌دهنده می‌دهد. به طور خلاصه، در روش لیو توابع با توجه به بدنهٔ آن‌ها طبقه‌بندی می‌شوند و در نتیجه توابعی که عملکرد مشابهی دارند در یک گروه قرار می‌گیرند. حال اگر تابعی دارای نامی ناسازگار با عملکردش باشد، می‌توان با توجه به نام توابع مشابه برای آن تابع، نامی پیشنهاد داد. لیو بیان می‌کند برخی از شبکه‌های عصبی می‌توانند ماهیت عملکرد توابع را درک و آن‌ها را دسته‌بندی کنند. این روش از سه گام

12-Linguistic AntiPattern Detector
13-Wordnet
14-Convolutional neural networks

جدول ۴. پادالگوهای تشخیص داده شده توسط LAPD برای چند نرم‌افزار [۱۷].

پادالگوها	Precision	TP	Validated	Eclipse	Coco	Argo UML	Argo ...
تابع «Get» بیشتر از یک راه دسترسی است.	60%	9	15.18	15	1	2	0
تابع «Is» چیزی بیشتر از نوع داده بولی	100%	24	24.32	26	4	0	2
تابع «Set» مقدار بازگشتی دارد.	98%	46	47.93	53	6	30	4
وجود انتظار بازگشت و عدم برگشت نمونه	77%	26	34.51	33	8	3	7
تابع	78%	58	74.32	232	43	28	20
متد اعتبارسنجی تایید	74%	52	70.25	235	11	8	1
تابع «Get» مقدار بازگشتی ندارد.	97%	37	38.63	57	2	3	1
سوال پاسخ داده نشد	100%	26	26.36	34	0	2	0
متد تغییرشک مقدار بازگشتی ندارد.	98%	57	58.14	44	15	86	0
انتظار مجموعه و عدم دریافت آن	73%	47	64.19	135	12	39	8
نام متد و نوع مقداری که برمیگردد تناقض دارند.	50%	3	6.6	6	0	0	0
امضای متد و توضیح با هم تناقض دارد.	8%	6	72.28	243	12	20	7
بیشتر از آنچه هد.	57%	40	70.25	103	42	92	15
نام نشاندهنده نوع داده بولی اس اما نوع نه.	56%	36	64.19	138	21	13	14
بیشتر از آنچه انجام میدهد بی	75%	55	73.30	116	24	117	45
نام و نوع صفت متضاد هستند.	100%	1	1.1	0	0	0	1
امضای صفت و توضیح با هم تناقض دارد.	5%	1	19.23	19	3	0	1

اصلی تشکیل شده است: پردازش داده، آموزش، تشخیص و ارائه پیشنهاد. در این روش منظور از ناسازگاری، عدم هماهنگی نام تابع با عملکرد آن است.

۱. در گام پردازش داده نام توابع و بدنهٔ آن‌ها نشانه‌گذاری[۱۵] می‌شوند. در نشانه‌گذاری نام توابع، نام‌ها که خودشان نشانه محسوب می‌شوند به زیرنشانه‌های تشکیل‌دهنده خود تبدیل می‌شوند. زیرنشانه‌های تشکیل شده در نشانه‌گذاری بدنه توابع و در درخت نحو انتزاعی استفاده می‌شود.

۲. در گام آموزش نشانه‌های به‌دست آمده از گام قبل ابتدا تبدیل به بردارهای عددی می‌شوند. زیرا برای خوراندن نشانه‌ها به شبکه‌های عصبی هم‌آمیختی باید آن‌ها را به صورت بردارهای عددی درآورده باشیم. نشانه‌های نام توابع با استفاده از روش بردار پاراگراف[۱۶] به بردارهای عددی تبدیل می‌شوند. زیرا هر نام را می‌توان مانند پاراگرافی دانست که تابع را توصیف می‌کند. نشانه‌های بدنه یک تابع ابتدا با استفاده از روش Word2Vec به بردار تبدیل می‌شوند. سپس تمام بردارهای بدنه تابع به شبکهٔ عصبی هم‌آمیختی داده می‌شوند تا تبدیل به یک بردار عددی واحد برای بدنهٔ تابع شوند. سپس بردارهای نام و بدنه توابع به صورت جداگانه طبقه‌بندی می‌شوند.

۳. در گام تشخیص و ارایهٔ پیشنهاد با داشتن بردار نام و بدنه یک تابع می‌توان وجود ناسازگاری بین نام و بدنه‌اش را تشخیص داد. در صورت وجود ناسازگاری، نام توابعی که بدنه مشابهی دارند پیشنهاد داده می‌شود.

شکل ٦ سه گام مشروح را نشان می‌دهد. جدول ٤ نیز تشخیص توابع دارا یا بدون ناسازگاری برای مقادیر مختلف k را نشان می‌دهد. مقدار k نشان می‌دهد برای سنجش ناسازگاری، الگوریتم پیشنهادی تا چند تابع مشابه را در نظر می‌گیرد. برای مثال k=5 یعنی از ٥ تابعی که بیشترین شباهت را دارند استفاده شده است. با توجه به جدول ٥ میتوان نتیجه گرفت عمده روش‌های به کار برده شده در تشخیص سازگاری و ناسازگاری در صورت افزایش تعداد توابعی که برای مقایسه استفاده می‌شوند، با کاهش دقت مواجه خواهند شد.

زیرا هر قدر تعداد توابع مذکور افزایش داده شود، ناچار به برگزیدن توابعی خواهیم بود که شباهت کمتری با تابع مورد بررسی دارند و استفاده از آن‌ها در بررسی تابع مورد بررسی کاهش دقت را به همراه خواهد داشت.

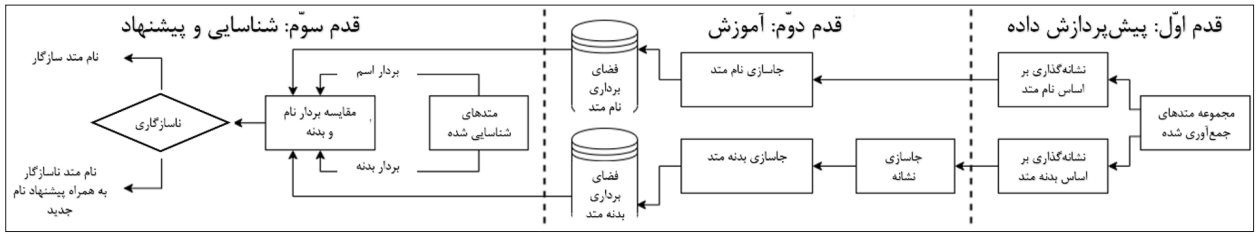
مزایای روش ارائه شده توسط لیو به شرح زیر است:

۱. معرفی روشی برای بررسی رابطه میان نام و عملکرد تابع.

۲. روش پیشنهادی توابع را دسته‌بندی می‌کند و از این امر می‌توان برای برای بررسی دیگر معیارهای مؤثر بر روی کیفیت کدِ زبان اصلی استفاده کرد. به عبارتی دیگر می‌توان موارد استفاده از این روش را گسترش داد.

15-Tokenize

16-Paragraph vector



شکل ۶. مراحل روش استفاده از یادگیری ماشین [۱۸].

جدول ۵. نتیجه ارزیابی تشخیص ناسازگاری‌ها به ازای درنظر گرفتن تعداد مختلفی از توابع مشابه [۱۸].

	معیار ارزیابی (٪)	K=1	K=5	K=10	K=20	K=30	K=40
سازگار	دقت	۵۶.۸	۵۳.۷	۵۳.۳	۵۳.۳	۴۹.۹	۴۹.۷
	پوشش	۸۴.۵	۵۵.۹	۴۶.۷	۴۶.۷	۲۸.۸	۳۳.۰۶
	معیار F	۶۷.۹	۵۴.۸	۴۹.۷	۴۹.۷	۳۶.۵	۴۰.۱
ناسازگار	دقت	۷۲.۰	۵۵.۹	۵۴.۲	۵۴.۲	۵۱.۴	۵۱.۴
	پوشش	۳۸.۲	۵۳.۷	۶۰.۷	۶۰.۷	۷۲.۲	۶۷.۴
	معیار F	۴۹.۹	۵۴.۸	۵۷.۳	۵۷.۳	۶۰.۰	۵۸.۳
صحت		۶۰.۹	۵۴.۸	۵۳.۸	۵۰.۸	۵۰.۹	۵۱.۱

جدول ۶. مزایا و معایب روش‌های مختلف.

	مزایا	معایب
همایندی	●امکان قرار دادن زمینه کد در همایندی اسامی تخصصی. ●سادگی وجود رابطه بین تکرار واهمیت کلمات. ●امکان پیشنهاد نام تابع بر اساس همایندی.	●نادقیق. ●امکان استفاده‌ی بی‌مورد. ●کاهش خوانایی در نتیجه شباهت زیاد شناسه‌ها. ● نیازمند شکستن نام تابع به عبارات تشکیل دهنده آن ●با فرض استفاده از کلمات استاندارد کارایی دارد.
تاثیر اجزای کد	●انتخاب نوع توضیح متناسب با آثار مورد نظر بر روی درک‌پذیری کد. ●پیروی از قاعده‌ی برنامه‌نویسی و عدم استفاده از توضیح‌هایی با آثار منفی بر روی درک‌پذیری.	●عدم ارائه چارچوبی ساختارمند برای ایجاد توضیح‌های مناسب.
آشکارسازی	●ارائه الگوریتم برای تمام پادالگوهای زبان‌شناختی . ●دقت بالاتر.	●معمولاً راهکاری برای اصلاح پادالگوها ارائه نمیدهد. ●در روش‌های مبتنی بر یادگیری ماشین نیاز به طبقه‌بندی تعداد زیادی تابع از جهت نام و بدنه وجود دارد ●سربار پردازش بالا .

۳. برخلاف بسیاری از تحقیقات صورت‌گرفته در زمینهٔ کیفیت کد، لیو علاوه بر آشکارسازی، به چگونگی رفع ناسازگاری‌ها نیز می‌پردازد.

معایب روش ارائه شده توسط لیو به شرح زیر است:

۱. برای امکان‌پذیر بودن این روش باید تعداد زیادی تابع را از جهت نام و بدنه طبقه‌بندی و ذخیره کنیم تا بتوانیم نام صحیحی پیشنهاد بدهیم.

۲. این روش ابتدا داده‌ها (کدِ زبان اصلی) را پردازش می‌کند و تمام توابع را طبقه‌بندی می‌کند و سپس به تشخیص ناسازگاری‌ها می‌پردازد. حتی اگر هیچ ناسازگاری‌ای یافت نشود، باز هم بخش بزرگی از بار پردازشی این روش باید انجام شود.

۳. اگر توابع طبقه‌بندی شده خودشان دارای ناسازگاری باشند، ممکن است نام یک تابع دارای ناسازگاری پیشنهاد شود.

در جدول ٦ به منظور جمع‌بندی، معایب و مزایای تحقیقات مبتنی بر همایندی، تاثیر اجزای کد و آشکارسازی به اختصار آورده شده است.

۴. پادالگوهای زبان‌شناختی مورد مطالعه

تمامی پادالگوهای زبان‌شناختی می‌توانند بر جنبه‌هایی از کیفیت کد تاثیر بگذارند. مطمئنا مطالعه تمام آن‌ها و ارائه

راه‌حل بـــرای رفع آن‌ها از حوصله این مقاله خارج اســـت. بدین‌جهت در این مقاله به مطالعه آن دســـته از پادالگوهایی می‌پردازیم که راه‌حل ارائه شـــده برای آن‌ها، می‌تواند برای رسیدن به راه‌حلی برای سایر پادالگوهای زبان‌شناختی نیز تعمیم داده شود. تمامی پادالگوها مورد بحث مرتبط به توابع هستند. روش‌های یافتن و اصلاح پادالگوهای زبان‌شناختی مربوط به اســـامی متغیرها به مراتب ساده‌تر هستند اما از همان منطق روش پیشنهادی پیروی می‌کنند. در واقع به جای مقایسه نام تابع با بدنه آن، نام متغیر با داده‌ای که به آن داده می‌شود مقایسه می‌گردد. بدین جهت در این تحقیق به مطالعه پادالگوهای زبان‌شـــناختی توابع بسنده شده است. همچنین سعی شده است برخی از نتایج وجود چنین پادالگوهایی در کدِ زبان اصلی نرم‌افزار بیان شود.

۴-۱- بیشتر از آنچه انجام می‌دهد بیان می‌کند

گاهی نام تابع می‌تواند گمراه‌کننده باشـــد. این گمراهی ممکن اســـت تا جایی پیش رود که خوانندهٔ کدِ زبان اصلی را قانـــع کند که تابع مورد بحث، عملکرد یا نتایجی دارد که در عمل چنین نیســـت. در واقع نـــام تابع عملکردی از تابع ارائه داده است که در واقعیت وجود ندارد یا کاستی دارد. بدین‌جهت در مقاله آرنودوا (۲۰۱۳) [۱] که مبنای بسیاری مطالعات در این حوزه است، به چنین پادالگوهایی «بیشتر از آنچه انجام می‌دهد بیان می‌کند» اطلاق شده است.

۴-۱-۱- تابع پیاده‌سازی‌نشده

چنیـــن پادالگویی هنـــگامی رخ می‌دهد کـــه تابع صرفا تعریف شده است اما بدنه آن خالی مانده است. علت وقوع چنین پادالگویی می‌تواند این باشد که توسعه‌دهندگان قبلی نرم‌افزار طرحی برای توسعه برنامه داشته‌اند و با تعریف تابع (و عدم توســـعهٔ آن و در نتیجه خالی ماندن بدنه‌اش) قصد در بیان آنچه در آینده صورت می‌پذیرد داشته‌اند.

اگر توسعه‌دهندهٔ جدید با آگاهی به این که تابعی وجود دارد، آن را فراخوانی کند، در صورتی که تابع پیاده‌سازی نشده باشد، نتایج موردنظر کاربر اعمال نمی‌شود و این امر احتمال بروز خطا در سیســـتم را بـــالا می‌برد. همچنین در صورت آزمایش کردن، به خصوص آزمایش کردن از نوع جعبه سیاه[17]، آزمایش‌کننده ممکن اســـت با صرف وقت و تلاش اقدام به طراحی آزمایش برای چندین سناریو نماید، در صورتی که انجام آزمایش برای تابعی با بدنه خالی نیاز نیست، و یا می‌توان آن را با یک سناریو پوشش داد.

۴-۱-۲- تابع «Get» چیزی برنمی‌گرداند

توابـــع برگرداننده (یـــا دهنده)[18] که معمـــولا نام آن‌ها با Get آغاز می‌شـــود این انتظار را ایجـــاد می‌کنند که در صـــورت فراخوانی، مقداری را برمی‌گردانند [۱]. معمولاً از این توابع برای دسترسی (یا پایش دسترسی) به صفت‌های یک کلاس اســـتفاده می‌شـــود. حال اگر چنین تابعی چیزی بـــه عنوان خروجی نتیجه ندهد، بیانگـــر وجود یک پادالگو زبان‌شـــناختی است. شـــکل ۷ قســـمتی از کدِ زبان اصلی نرم‌افـــزار اکلیپس آی‌دی‌ای نســـخه ۱.۰ را نشـــان می‌دهد که در آن تابـــع getMethodeBodies بـــا توجه به نامش بایـــد مقداری را خروجی بدهد اما این اتفاق نیفتاده اســـت. کلیدواژه void که در هنگام تعریف تابع آمده است نشان از خروجی نداشتن این تابع است.

توسعه‌دهنده‌ای که از وجود چنین مسئله‌ای بی‌خبر است ممکن است اقدام به ذخیره خروجی چنین تابعی در قسمتی از برنامـــه کند، اما از آنجایی که چنیـــن خروجی‌ای هرگز تولید نمی‌شود، توسعه‌دهنده باید به بدنه تابع مراجعه کرده و راهی برای به دست آوردن خروجی مطلوب بیابد و یا از ذخیره خروجی‌ای که وجود ندارد منصرف شود.

۴-۲- بیشتر از آنچه می‌گوید انجام می‌دهد

عدم تهیه مســـتندات و یا اســـتفاده از واژگان نامناسب می‌توانـــد خوانندهٔ کدِ زبان اصلی را از شـــناخت و یا درک صحیح برخی از ویژگی‌هـــای کدِ زبان اصلی محروم کند. این ویژگی‌ها ممکن اســـت برای اســـتفاده بهینه از سیستم مربوطه حیاتی باشد و یا در صورت عدم استفاده، عملکرد

17-Black box testing
18-Get methods or getters or accessor methods

```
protected void getMethodBodies
  (CompilationUnitDeclaration unit,int place){
  //fill the methods bodies in order
  //for the code to be generated
  if (unit.ignoreMethodBodies) {
    unit.ignoreFurtherInvestigation = true;
    return; // if initial diet parse did not
    //work, no need to dig into method bodies.
  }
  if (place < parseThreshold)
    return; //work already done ...
  //real parse of the method....
  parser.scanner.setSourceBuffer(
     unit.compilationResult.
     compilationUnit.getContents());
  if (unit.types != null) {
    for (int i = unit.types.length; --i >= 0;)
      unit.types[i].parseMethod(parser, unit);
        }
}
```

شکل ۷. قسمتی از کد اکلیپس آی‌دی‌ای دارای پادالگوی «تابع برگرداننده چیزی برنمی‌گرداند» [۱].

سیستم مختل شود. بدین جهت در مقاله آرنودوا (۲۰۱۳) [۱] که مبنای بسیاری مطالعات در این حوزه است، به چنین پادالگوهایی «بیشتر از آنچه می‌گوید انجام می‌دهد» اطلاق شده است.

۴-۲-۱- تابع «Get» بیشتر از یک راه دسترسی است

در جاوا، توابع دسترسی‌دهنده راهی برای دسترسی به صفات یک کلاس فراهم می‌کنند. همچنین متداول نیست که این توابع عملی غیر از برگرداندن صفت مربوطه انجام دهند [۱۹]. در صورت وقوع چنین شرایطی، پادالگو «تابع برگرداننده‌ای که بیشتر از یک راه دسترسی است» رخ می‌دهد. وجود هرگونه عملی به غیر از دادن دسترسی به صفات کلاس باید مستند شود و احتمالاً نام تابع دیگر شبیه چیزی مثل getSomething نباشد. شکل ۱۰ قطعه کدی از نرم‌افزار اکلیپس آی‌دی‌ای نسخه ۱.۰ را نشان می‌دهد که در آن تابع getImageData به غیر از عمل برگرداندن یک مقدار، اعمال دیگری نیز داشته است. برای مثال مقداری که برمی‌گرداند را در بدنه‌اش ایجاد کرده و صرفاً یک دسترسی‌دهنده نبوده است. در واقع مقدار خروجی در بدنه خود تابع ایجاد شده است.

از عواقب چنین رخدادی می‌توان به رفتار غیرمنتظره برنامه با فراخوانی چنین تابعی اشاره کرد. زیرا کاربر انتظار کاری به غیر از دسترسی دادن به صفات را از تابع

```
public ImageData getImageData() {
  Point size = getSize();
  RGB black = new RGB(0, 0, 0);
  RGB[] rgbs = new RGB[256];
  rgbs[0] = black; // transparency
  rgbs[1] = black; // black
  PaletteData dataPalette=new PaletteData(rgbs);
  imageData = new
    ImageData(size.x,size.y,8,dataPalette);
  imageData.transparentPixel = 0;
  drawCompositeImage(size.x, size.y);
  for (int i = 0; i < rgbs.length; i++)
    if (rgbs[i] == null) rgbs[i] = black;
  return imageData;
}
```

شکل ۸. قسمتی از کد اکلیپس آی‌دی‌ای دارای پادالگوی « تابع برگرداننده‌ای که بیشتر از یک راه دسترسی است» [۱].

ندارد، و با فراخوانی آن نتایجی در برنامه می‌بیند که انتظار ندارد منشأشان تابع دسترسی‌دهنده باشد. همچنین، اگر تابع در بدنه‌اش با ایجاد تغییراتی بر روی مقادیر، قصد آماده کردن و مناسب‌سازی آن‌ها برای خروجی دادن دارد، در صورتی که در صفات کلاس تغییراتی ایجاد شود، ممکن است این تغییرات (اعمالی که باعث می‌شوند تابع دیگر فقط یک دسترسی‌دهنده ساده نباشد) دیگر توسط کاربر قابل ردیابی نباشد. این مسئله وقت و تلاش مضاعفی از توسعه‌دهندگان برای رفع عیب، خطایابی و سرویس و نگهداری از نرم‌افزار طلب می‌کند.

۴-۲-۲- تابع «Is» چیزی بیشتر از نوع داده بولی برمی‌گرداند

وقتی نام تابعی با عبارت is آغاز می‌شود، انتظار می‌رود نوع داده برگشتی آن از نوع بولی باشد. یعنی یک از دو مقدار صحیح و غلط را برگرداند. در زبانی مانند پایتون می‌تواند به جای استفاده صریح از عبارت True یا False از موارد دیگری مانند عدد صحیح، رشته (شکل ۹) و غیره استفاده کرد. برای مثال برای نشان دادن مقدار غلط از عدد صفر و برای مقدار صحیح از اعداد غیر صفر استفاده می‌شود.

همچنین در پایتون می‌توان از یک لیست خالی به عنوان مقدار غلط، و از یک لیست غیر خالی به عنوان مقدار صحیح استفاده کرد (شکل ۱۰).

توسعه‌دهنده ممکن است برای مقدار برگردانده شده یک تابع که نامش با is شروع می‌شود، مقداری به غیر از

```
newValue = “a  new string”
if newValue:
  print(“True”)
else:
  print(“False”)
output>> True
```

شکل ۹. مثال استفاده از سایر انواع داده به جای بولی.

```
newValue = list() #empty list
if newValue:
  print(“True”)
else:
  print(“False”)
output>> False
```

شکل ۱۰. مثال استفاده از سایر انواع داده به جای بولی.

نوع داده بولی در نظر بگیرد که برایش تداعی‌کننده صحیح یا غلط اســت (و لزوما برای فرد دیگری این چنین نیست). برای مثال توســعه‌دهنده جاوا ممکن اســت بــرای تابعی خروجی صفــر و یک در نظر بگیرد و صفــر را به عنوان غلط و یک را به عنوان صحیح در نظر بگیرد. در شــکل ۱۱ تابع isExpired به جای برگرداندن نوع داده بولی، رشته‌ای را برمی‌گرداند که تداعی‌کننده یکی از این دو مفهوم است.

در صورت اقدام به ذخیره مقدار بازگشتی چنین تابعی در یــک متغیر از نوع داده بــولی، خطا در زمان کامپایل و یا حتی توســط محیط توســعه یکپارچه[19] تشخیص داده می‌شود، با این حال باعث بروز برداشت ناصحیح از طرف مسئولان نگهداری نرم‌افزار می‌شود.

۴-۲-۳- تابع «Set» مقدار بازگشتی دارد

توابع تغییردهنده (مانند توابع تخصیص‌دهنده[20]) امکان اختصــاص دادن مقداری را به صفــات کلاس(که معمولا محافظت‌شده[21] یا خصوصی[22] هستند و در نتیجه نمی‌توان آنها را از خارج مورد دســترسی قرار داد) فراهم می‌کنند. طبق رسوم، توابع تخصیص‌دهنده مقداری را برنمی‌گردانند. به‌طورکلی، این توضیح برای توابعی که نامشان با set آغاز می‌شود، صحیح است. همچنین تابع تخصیص‌دهنده‌ای که

19-Integrated development environment(IDE)
20-Setters
21-Protected
22-Private

```
public String isExpired {
  long currentTime = System.currentTimeMillis();
  if (currentTime <= this.expires) {
    return “F”;
    }
  else {
    return “T”;
    }
```

شکل ۱۱. مثال استفاده از نوع داده‌ای غیر از بولی برای تابعی که نامش با is شروع می‌شود.

```
public Dimension setBreadth
        (Dimension target, int source) {
  if (orientation == VERTICAL)
    return new Dimension(source,
        (int)target.getHeight());
  else
    return new Dimension(
                (int)target.getWidth(), source);
}
```

شکل ۱۲. وجود مقدار بازگشتی در تابعی که نامش با set آغاز می‌شود [۱].

چیزی برمی‌گرداند باید با داشتن یک نام مناسب (که دیگر با set آغاز نشــود) این مسئله را شفاف کند [۱]. شکل ۱۲ قطعه‌ای از کــدِ زبان اصلی نرم‌افزار آرگو یوامال نســخه ۰.۱۰.۰ اســت که در آن تابع تخصیص‌دهندهٔ setBreadth مقداری را نیز برمی‌گرداند. در چنین شرایطی، استفاده‌کننده از این تابع ممکن اســت مقدار بازگشــتی را ذخیره نکند و اطلاعات مفیدی را مانند خطاها از دست بدهد [۱].

۵. طرح پیشنهادی

روش‌های بازساخت کد عموماً شامل دو مرحله اصلی تشخیص و اصلاح می‌شوند. روش پیشنهادی در این گزارش نیز از این امر پیروی می‌کند. مطالعات صورت گرفته در این زمینه نیز گاهی یک، و یا هر دو گام را پوشش می‌دهند.

۵-۱- تشکیل درخت نحو انتزاعی

بسیاری از مطالعات صورت گرفته بر روی پادالگوها، از درخت نحو انتزاعی سود می‌برند. این درخت نه تنها کدِ زبان اصلی را به شکل مناسب و قابل درکی تجزیه می‌کند، بلکــه با ارائه اطلاعاتی برای هر گــره، کار تحلیل کدِ زبان اصلی را آســان می‌کند. به تهیهٔ درخت نحو انتزاعی از کدِ زبان اصلی تجزیه[23] گفته می‌شود. وظیفهٔ تجزیه‌گر[24] شامل استفاده از تحلیلگر واژه[25] و قرار دادن خروجی‌های آن در

23-Parse
24-Parser
25-Lexer or Lexical analyzer

کنار هم برای ساخت درخت می‌شود.

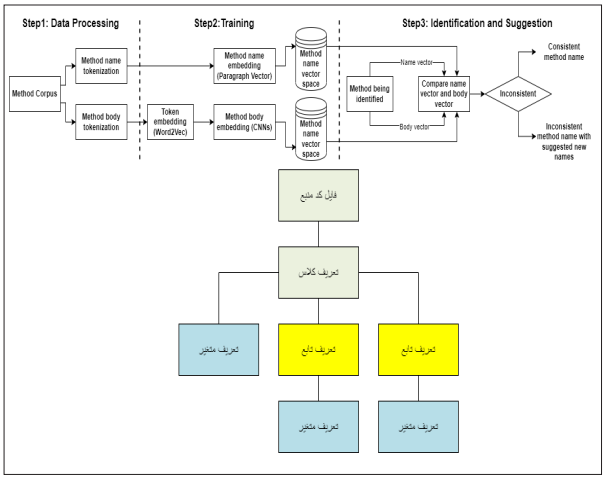
درخت نحو انتزاعی مورد استفاده در این مقاله کدِ زبان اصلی جاوا را تجزیه می‌کند. به کمک درخت حاصل، می‌توان متغیرها، صفات، کتابخانه‌های گنجانده شده، توابع و سایر گره‌ها را از یکدیگر تمایز داد. زیرا هر گره در این درخت می‌تواند یک شیء از کلاس‌هایی مشخص باشد. ابزار استفاده شده برای تهیه درخت نحو انتزاعی در این مقاله، برای تمامی قسمت‌های ممکن در کدِ زبان اصلی جاوا کلاس‌هایی دارد. همچنین امکان از صافی ردکردن[۲۶] تمامی گره‌ها برای به دست آوردن گره‌هایی از یک کلاس خاص مهیا شده است.

به هر یک از گره‌های یک درخت نحو انتزاعی یک نشانه[۲۷] نیز گفته می‌شود. یک نشانه در نرم‌افزار دارای نوع داده مخصوص به خود است. نشانه‌های قسمت‌های مختلف کد می‌توانند نوع داده متفاوتی داشته باشند(که در تمایز دادن آن‌ها استفاده خواهد شد).

۵-۲- دسته‌بندی نشانه‌ها

گرچه این مقاله در رابطه با پادالگوهای توابع است، اما اطلاعاتی که در فرایند تشخیص و رفع آنها نیاز داریم محدود به توابع نمی‌شود. رفع پادالگوهای مربوط به توابع بدون در نظر گرفتن سایر جنبه‌های کدِ زبان اصلی می‌تواند باعث تشکیل پادالگوهای دیگر شود. همان‌طور که گفته شد، گره‌های درخت نحو انتزاعی دارای کلاس مربوط به خود هستند. ما می‌توانیم تمام گره‌ها را از صافی رد کنیم تا تنها گره‌های مربوط به یک کلاس خاص باقی بمانند. اولین و پرکاربردترین صافی، جداکننده گره‌های مربوط به توابع است. شکل ۱۳ گره‌های اصلی درخت نحو انتزاعی یک کدِ زبان اصلی ساده را نمایش می‌دهد.

در چنین شرایطی با گذراندن گره‌ها از صافی توابع، دو گره (نشانه) باقی می‌ماند. زیرا در درخت از کلاس methodDeclaration دو گره وجود دارد. اگر این فرایند برای صافی متغیرها نیز انجام شود، با توجه به شکل ۱۵



شکل ۱۳. نمایش گره‌های اصلی درخت نحو انتزاعی یک کدِ زبان اصلی ساده.

سه گره به دست می‌آید. در روش پیشنهادی توابعی برای از صافی گذراندن‌های ذکر شده، تعریف شده‌اند. دلیل اهمیت یافتن نشانه‌های مربوط به تعریف متغیرها این است که در گام اصلاح ممکن است بخواهیم مقدار خروجی تابعی را در متغیر جدیدی ذخیره کنیم و برای انتخاب نام متغیر جدید باید به تکراری نبودن آن نام در کدِ زبان اصلی توجه کنیم.

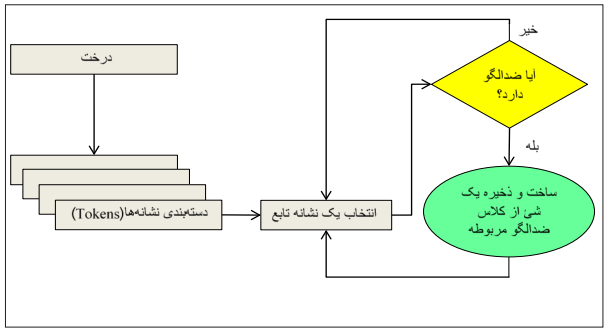
۵-۳- تشخیص و دسته‌بندی توابع نیازمند بازسازی

در بخش ٤-۲ نشانه‌ها را دسته‌بندی کردیم. حال نشانه‌های مربوط به توابع را برای یافتن پادالگوهای مورد بحث مطالعه می‌کنیم. در صورت تشخیص وجود پادالگو در یک تابع، یک شیء متناظر با آن تابع از کلاس مربوط به پادالگوی تشخیص داده شده ایجاد می‌کنیم. به کمک این شیء، پادالگو را ذخیره می‌کنیم و از صفات آن در فرایند اصلاح کد استفاده می‌کنیم. یکی از صفات تمام کلاس‌های پادالگوها، مربوط به نشانه‌ای است که پادالگوی مربوطه در آن رخ داده است. به تعداد پادالگوهای مورد مطالعه، کلاس برای پادالگوها داریم. در شکل ١٤ فرایند تشخیص و ذخیره پادالگوها با دیدی کلی نشان داده شده است.

۵-۳-۱- تابع پیاده‌سازی‌نشده

با داشتن تمامی نشانه‌های مربوط به توابع و در

26-Filter
27-Token



شکل ۱۴. نمایش فرایند تشخیص و ذخیره پادالگوهای توابع.

نتیجه مشخصات توابع، تشخیص یک تابع با بدنه خالی کار راحتی است. گره (نشانه) مربوط به یک تابع می‌تواند چندین گره فرزند داشته باشد. اگر گره مربوط به بدنهٔ تابع هیچ فرزندی نداشته باشد، بدین معنا است که بدنهٔ تابع خالی است.

۵-۳-۲- تابع «Get» چیزی برنمی‌گرداند

در تشخیص این پادالگو باید دو گام طی شود. گام اول بررسی وجود یا عدم وجود مقدار خروجی در تابع موردنظر است و گام دوم بررسی نام تابع از جهت آغاز شدن با عبارت get است. بین فرزندان گره مربوط به تعریف توابع در درخت، یکی از آن‌ها نوع خروجی[۲۸] است. برای توابعی که مقداری را خروجی نمی‌دهند، نوع خروجی (یا به بیان بهتر نوع دادهٔ خروجی) void است. پس تشخیص این که تابعی خروجی دارد و یا ندارد با بررسی گره فرزند مربوط به نوع خروجی امکان‌پذیر است. برای تشخیص وجود عبارت get در ابتدای نام تابع، ساده‌ترین راه این است که سه حرف اول را بررسی کنیم تا به ترتیب g، e و t باشند. لازم به ذکر است باید به کوچک و یا بزرگ بودن حروف نیز توجه کرد. برای مثال نام تابع می‌تواند با GET، Get، get و یا سایر نوشتارهای مشابه عبارت get آغاز شود. این روش دقت پایینی دارد. زیرا آغاز نام تابع با سه حرف نام برده لزوما به معنای آغاز نام تابع با کلمه get نیست. برای مثال اگر نام تابع getableObjectsList باشد، آنگاه دیگر نام تابع با کلمه get شروع نشده است، بلکه با

28-Return type

جدول ۷. نتیجه اعمال تقسیم‌بندی کلمات بر روی چند نام تابع.

نام تابع	کلمات استخراج شده (به ترتیب)
getMyAgeValue	get my age value
Get_allObjects_list	get all objects list
getableAttributes_tuple	getable attributes tuple

کلمه getable آغاز شده است و دیگر از نام آن اینچنین برداشت نمی‌شود که یک تابع دسترسی‌دهنده است.

راه‌حل دیگر استخراج کلمات با معنی از نام تابع است. این روش به وسیله تقسیم‌بندی کلمات[۲۹] میسر می‌شود. تقسیم‌بندی کلمات مسئله تقسیم یک رشته از متن نوشته شده به یک زبان، به کلمات تشکیل‌دهنده‌اش است [۲۰]. جدول ۷ چند مثال از نتایج مربوط به اعمال تقسیم‌بندی کلمات بر روی چند نام تابع را نشان می‌دهد. به کمک تقسیم‌بندی کلمات تشخیص پادالگو «تابع «Get» چیزی برنمی‌گرداند» میسر می‌شود.

۵-۳-۳- تابع «Get» بیشتر از یک راه دسترسی است

برای تشخیص آغاز شدن نام تابع با کلمه get مانند روش گفته شده در بخش قبل عمل می‌شود. برای بررسی «بیشتر از راه دسترسی» بودن تابع باید به بدنهٔ آن (با استفاده فرزندان گره مربوط به تابع در درخت نحو انتزاعی) مراجعه کرد. قسمتی از بدنه تابع که وظیفه برگرداندن مقدار خروجی را دارد، خود نشانه است و در درخت نحو انتزاعی فرزند گره بدنهٔ تابع است. اگر به غیر از این گره، گره‌های دیگری نیز فرزند گره بدنهٔ تابع باشند می‌توان دریافت که تابع مربوطه، به غیر از برقراری روشی برای دسترسی به یک صفت، اعمال دیگری نیز انجام می‌دهد. ممکن است مقدار خروجی نیز لزوماً یک صفت کلاس نباشد و بررسی این موضوع در حوزه این گزارش نیست. به طور خلاصه، در صورت وجود گرهی به غیر از گره مربوط به عبارت بازگشت در بدنهٔ تابعی که نامش با کلمه get آغاز می‌شود، پادالگو «تابع (Get) بیشتر از یک راه دسترسی است» در تابع رخ داده است.

29-Word segmentation

۵-۳-۴- تابع «Is» چیزی بیشتر از نوع داده بولی برمی‌گرداند

برای بررسی آغاز شـــدن نام تابـــع با کلمه is از روش معرفی‌شـــده در بخش ۵-۳-۲، یعـــنی تقسیم‌بندی کلمات استفاده می‌کنیم. سپس توابع به‌دست آمده را از جهت نوع داده خروجی‌شـــان بررسی می‌کنیـــم. در صورتی که نوع داده خروجی تابعی که نامش با کلمه is شـــروع می‌شود از نوع داده بولی نبود، وجود پادالگوی «تابع (Is) چیزی بیشتر از نوع داده بولی برمی‌گرداند» در آن تابع آشکار می‌شود

۵-۳-۵- تابع «Set» مقدار بازگشتی دارد

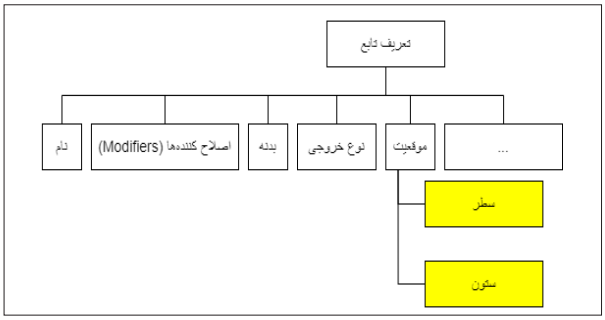
با اســـتفاده از روش مشـــروح در بخش ۵-۳-۲، یعنی تقسیم‌بندی کلمات، تمامی توابعی که نامشـــان با کلمه set آغاز می‌شـــود را به‌دست می‌آوریم. سپس بدنه تمام آن‌ها را از جهت وجود عبارت بازگشـــت بـــررسی می‌کنیم. این بـــررسی با توجه به فرزندان گره بدنه تابع در درخت نحو انتزاعی انجام می‌شود. در صورت وجود عبارت بازگشت، پادالگوی مورد بحث وجود داشـــته و آن را با اســـتفاده از کلاس مربوطه ذخیره می‌کنیم.

۵-۴- اصلاح کدِ زبان اصلی

با داشتن اطلاعات دقیق درباره پادالگوهای زبان‌شناختی مورد مطالعه موجود در کدِ زبان اصلی، زیرساخت مورد نیاز برای رفع آن‌ها فراهم می‌شود. فرایند بازساخت کد در این روش شـــامل اعمال تغییراتی بـــر روی کدِ زبان اصلی نرم‌افزار اســـت. این بدان معنی است که پس از این فرایند، کدِ زبان اصـــلی تغییر می‌کند. پس برای امکان بازگشـــت به نســـخه قبل از بازســـاخت، باید از قبل نسخه پشتیبان تهیه شود. کوشش شـــده که راه‌حل‌های ارائه شده در این گزارش برای رفع پادالگوها تا جای ممکن ساده و اثربخش باشند و نیاز به بررسی دستی توسط توسعه‌دهنده، بعد از بازساخت خودکار به حداقل برسد.

۵-۴-۱- تابع پیاده‌سازی‌نشده

روش ارائـــه شـــده برای رفـــع این پادالگـــو حذف آن



شکل ۱۵. قسمتی از یک درخت نحو انتزاعی شامل موقعیت قرارگیری تابع.

اســـت. یکی از فرزندان گره مربـــوط به یک تابع در درخت نحو انتزاعی، موقعیت آن اســـت. گره موقعیت خودش یک نوع داده اســـت که دو صفـــت دارد: (۱) خط(ردیف) و (۲) ستون(شکل ۱۵).

با دانســـتن خط و ســـتونی که تابع در آن شروع شده است، کافی است خط و ستونی که در آن بدنه تابع به پایان رسیده اســـت را پیدا کنیم تا بدانیم بـــرای حذف تابع، چه خطوط و چه قســـمت‌هایی از خطوط باید پاک شوند. برای ایـــن کار از آکولادها در زبان جاوا کمک گرفته می‌شـــود. اگر کدِ زبان اصلی نرم‌افزار در زمان اجرا، خطای نحوی[۳۰] نداشته باشـــد (چنین خطایی در زمان کامپایل و یا توسط محیط توســـعه یکپارچه تشـــخیص داده می‌شـــود)، تعداد آکولادهای باز({) با تعداد آکولادهای بسته(}) برابر خواهد بود. اگر از خط اول تابع شروع به شمارش آکولادهای باز و بسته در تمام خطوط (به ترتیب) تابع کنیم، خط و ستونی که در آن با شـــمارش یک آکولاد بسته، تعداد آکولادهای باز و بســـته برابر شـــود، نقطه پایانی آن تابع است. شکل ۱۶ شبه‌کد الگوریتم تشـــخیص و اصلاح این پادالگو را به صورت الگوریتم بیان می‌کند.

۵-۴-۲- تابع «Get» چیزی برنمی‌گرداند

روش ارائه شده برای رفع این پادالگو، تغییر نام تابع به گونه‌ای اســـت که دیگر یک تابع دسترسی‌دهنده به حساب نیاید. البته روش دیگر آن اســـت کـــه از روی نام تابع نوع خروجی را به دســـت آوریم. اما از آنجـــایی که نام توابع گاهی خلاصه شـــده اســـت، چنین راه‌کاری لزوماً کارآمد

30-Syntax error

```
AST = Java_to_AST(sourceCodeDirectory)

codeLines = java_to_lines(sourceCodeDirectoy)

methodNodes = AST.filter(type=methods)



for m in methodNodes:

    if m.body.is_empty():

        notImplementedMethods.append(m)

        startLine = m.position.startLine

        startColumn = m.position.startLine

        endLine = m.position.endLine

        endColumn = m.position.endColumn

        for i in range(startLine + 1 , endLine + 1):

            codeLines.delete[i]

        codeLines[startLine] = codeLines[startLine][startColumn : endOfLine]

        codeLines[endLine] = codeLines[endLine][0 : endColumn]



lines_to_java(codeLines, sourceCodeDirectoy)
```

شکل ۱۶. الگوریتم تشخیص و اصلاح پادالگوی «تابع پیاده‌سازی‌نشده»

نیســت. یکی از فرزندان گره‌های مربوط به توابع، گره نام تابع اســت. این گره نــوع داده مخصوص به خود را ندارد و صرفاً از نوع داده رشــته در پایتون استفاده می‌کند. در بخش ۵-۲ به دسته‌بندی گره‌ها با استفاده از صافی اشاره شد و لیستی از گره‌های مربوط به توابع به دست آمد.

این بدان معنا اســت که می‌توانیم از تکراری نبودن نام جدیدی که برای تابع دارای پادالگو انتخاب می‌کنیم اطمینان یابیم. در خطی از کدِ زبان اصلی که نام تابع تعریف شــده اســت، نام قبلی را با نام جدید جایگزین می‌کنیم. نام جدید همان نام قبلی به همراه یک پیشــوند است تا دیگر نام تابع با get آغاز نشــود. اگر نام جدید در کدِ زبان اصلی از قبل وجود داشت، یک پسوند عدد صحیح به آن اضافه می‌کنیم و ایــن فرایند را تا هنگامی ادامه می‌دهیم که دیگر نام جدید در کدِ زبان اصلی تکراری نباشد. شکل ۱۷ شبه‌کد الگوریتم تشخیص و اصلاح این پادالگو را نمایش می‌دهد.

۵-۴-۳- تابع «Get» بیشتر از یک راه دسترسی است

در این روش تمام قســمت‌های بدنه تابع را درون یک تابــع جدید قرار می‌دهیــم و برای آن تابــع جدید یک نام مناســب انتخاب می‌کنیم (برای تکراری نبــودن نام، مانند روش مشــروح در قســمت قبل عمل می‌کنیم.). سپس تابع

```
AST = Java_to_AST(sourceCodeDirectory)
codeLines = java_to_lines(sourceCodeDirectoy)
methodNodes = AST.filter(type=methods)

for m in methodNodes:
    if m.name.lower_case().starts_with("get"):
        getMethods.append(m)

for g in getMethods:
    if g.return_type.name == "void":
        newNameBase = "calculateFor_" + g.name
        newName = newNameBase
        index = 1

        for i in methodNodes:
            if i.name == newName:
                newName = newNameBase + int_to_str(index)
                index++

        codeLines[g.position.startLine].replace_substring(g.name, newName)
lines_to_java(codeLines, sourceCodeDirectoy)
```

شکل ۱۷. الگوریتم تشخیص و اصلاح پادالگوی «تابع «Get» چیزی برنمی‌گرداند»

جدید را به کدِ زبان اصلی اضافه می‌کنیم. سپس بدنه تابع اول را تغییــر می‌دهیم، به گونه‌ای که در آن، خروجی تابع جدید، خروجی داده شود.

تابع جدید باید تمام اصلاح‌کننده‌های تابع اول را داشته باشد. در واقع این تابع همان تابع اول است که نامش عوض شده تا دیگر پادالگو نداشته باشد. با این روش، قسمت‌های اضافی را از تابع اول حذف کردیم تا فقط قسمت مربوط به عبارت بازگشت در آن وجود داشته باشد. شکل ۱۸ شبه‌کد الگوریتم تشخیص و اصلاح این پادالگو را بیان می‌کند.

۵-۴-۴- تابع «Is» چیزی بیشتر از نوع داده بولی برمی‌گرداند

این که تابع دقیقاً چه چیزی را به جای نوع داده‌ی بولی برمی‌گردانــد در چگونگی رفع این پادالگو مؤثر اســت. در این روش، با بررسی نوع دادهٔ خروجی و عبارات بازگشت مربوط به تابع، تصمیم‌گیری می‌شود که هر کدام از عبارات بازگشت موجود در تابع، مقدار صحیح یا غلط را خروجی دهند. ابتدا تابعی مانند تابع موردنظر اما با نام مناسب (به طوری که دیگر پادالگو تشــخیص داده نشود) می‌سازیم. ســپس در تابع فعلی، بدنه قبلی را حذف می‌کنیم. نوع داده بازگشتی را به بولی تغییر می‌دهیم. سپس تابع جدیدی که ساخته‌ایم را در داخل تابع فعلی فراخوانی می‌کنیم. خروجی آن را در متغیری ذخیره می‌کنیم. سپس با بررسی مقدار و نوع داده متغیر، ســعی می‌کنیم درک کنیم که توسعه‌دهنده قبــلی از خروجی دادن این مقدار، ســعی در تداعی مفهوم

```
AST = Java_to_AST(sourceCodeDirectory)
codeLines = java_to_lines(sourceCodeDirectoy)
methodNodes = AST.filter(type=methods)

for m in methodNodes:
    if m.name.lower_case().starts_with("get"):
        getMethods.append(m)

for g in getMethods:
    if g.body.length > 1:
        newMethodNameBase = "calculateFor_" + g.name
        newMethodName = newMethodNameBase
        index = 1

        for i in methodNodes:
            if i.name == newMethodName:
                newMethodName = newMethodNameBase + int_to_str(index)
                index++

        newMethod = create_java_method(name = newMethodName,
                                       modifiers = g.modifiers,
                                       return_type = g.return_type,
                                       body = g.body
                                       inputs = g.inputs)
        newMethod.insert_to_code(codeLines, position="end_of_class")

        g.body = "return" + newMethod.name + "(" + str(newMethod.inputs.without_type) + ");"
        g.save(codeLines)
lines_to_java(codeLines, sourceCodeDirectoy)
```

شکل ۱۸. الگوریتم تشخیص و اصلاح پادالگوی «تابع «Get» بیشتر از یک راه دسترسی است»

«صحیح» داشته است یا «غلط».

جدول ۸ نشان‌دهنده نتیجه تصمیم‌گیری تابع تصمیم‌گیرنده بر اساس خروجی تابع جدید است (که خروجی‌اش بولی نیست). شکل ۱۹ شبه‌کد الگوریتم تشخیص و اصلاح این پادالگو را بیان می‌کند.

۵-۴-۵- تابع «Set» مقدار بازگشتی دارد

از آنجایی که ممکن است مقدار خروجی شامل اطلاعات مفیدی (برای مثال خطایابی) باشد، ذخیره آن توصیه می‌شود، بدین جهت درون برنامه آن را ذخیره می‌کنیم. به نام تابع پسوندی را اضافه می‌کنیم تا ذخیره کردن مقداری در متغیر از اسم آن قابل فهم شود. همچنین متغیر را در جایی خارج از تابع (یک خط بالاتر از آن) می‌سازیم و نام آن را به گونه‌ای تعیین می‌کنیم که در کدِ زبان اصلی تکراری نباشد. همچنین در هر قسمتی از بدنه که تابع مقداری را برمی‌گرداند، عبارت بازگشت را حذف و به جای آن، عبارت اختصاص[۳۱] را جایگزین می‌کنیم. شکل ۲۰ شبه‌کد الگوریتم تشخیص و اصلاح این پادالگو را بیان می‌کند.

۶. ارزیابی

برای ارزیابی، ابتدا کدِ زبان اصلی چهار سیستم نرم‌افزاری در آن‌ها کدهایی به زبان جاوا وجود دارد و

جدول ۸. تصمیم‌گیری برای اصلاح پادالگوی «تابع «Is» چیزی بیشتر از نوع داده بولی برمی‌گرداند».

ردیف	نوع داده	تصمیم
۱	int	اگر مقدار برابر با ۰ بود ← غلط در غیر این صورت ← صحیح
۲	byte	اگر مقدار برابر با ۰ بود ← غلط در غیر این صورت ← صحیح
۳	short	اگر مقدار برابر با ۰ بود ← غلط در غیر این صورت ← صحیح
۴	long	اگر مقدار برابر با ۰ بود ← غلط در غیر این صورت ← صحیح
۵	float	اگر مقدار برابر با ۰ بود ← غلط در غیر این صورت ← صحیح
۶	double	اگر مقدار برابر با ۰ بود ← غلط در غیر این صورت ← صحیح
۷	char	اگر برابر با "f"یا "۰" مشابه بود ← غلط در غیر این صورت ← صحیح
۸	String	اگر برابر با "f" یا "false" یا "۰" مشابه بود ← غلط در غیر این صورت ← صحیح
۹	int[]	اگر بیش از ۱ عضو داشت ← صحیح اگر عضوی نداشت ← غلط اگر یک عضو داشت مانند ردیف ۱ با آن برخورد شود
۱۰	byte[]	اگر بیش از ۱ عضو داشت ← صحیح اگر عضوی نداشت ← غلط اگر یک عضو داشت مانند ردیف ۲ با آن برخورد شود
۱۱	short[]	اگر بیش از ۱ عضو داشت ← صحیح اگر عضوی نداشت ← غلط اگر یک عضو داشت مانند ردیف ۳ با آن برخورد شود
۱۲	long[]	اگر بیش از ۱ عضو داشت ← صحیح اگر عضوی نداشت ← غلط اگر یک عضو داشت مانند ردیف ۴ با آن برخورد شود
۱۳	float[]	اگر بیش از ۱ عضو داشت ← صحیح اگر عضوی نداشت ← غلط اگر یک عضو داشت مانند ردیف ۵ با آن برخورد شود
۱۴	double[]	اگر بیش از ۱ عضو داشت ← صحیح اگر عضوی نداشت ← غلط اگر یک عضو داشت مانند ردیف ۶ با آن برخورد شود
۱۵	char[]	اگر بیش از ۱ عضو داشت ← صحیح اگر عضوی نداشت ← غلط اگر یک عضو داشت مانند ردیف ۷ با آن برخورد شود
۱۶	String[]	اگر بیش از ۱ عضو داشت ← صحیح اگر عضوی نداشت ← غلط اگر یک عضو داشت مانند ردیف ۸ با آن برخورد شود

31-Assign

```
AST = Java_to_AST(sourceCodeDirectory)
codeLines = java_to_lines(sourceCodeDirectoy)
methodNodes = AST.filter(type=methods)

for m in methodNodes:
    if m.name.lower_case().starts_with("is"):
        isMethods.append(m)

for m in isMethods:
    if m.return_type.name != "boolean":
        newMethodNameBase = "complement_" + m.name
        newMethodName = newMethodNameBase
        index = 1

        for i in methodNodes:
            if i.name == newMethodName:
                newMethodName = newMethodNameBase + int_to_str(index)
                index++

        newMethod = create_java_method(name = newMethodName,
                                       modifiers = m.modifiers,
                                       return_type = m.return_type,
                                       body = m.body
                                       inputs = m.inputs)
        newMethod.insert_to_code(codeLines, position="end_of_class")

        newBody = table5_2_decision(return_type = m.return_type.name,
                                    methodToCall = newMethodName)
        m.body = newBody
        m.save(codeLines)
lines_to_java(codeLines, sourceCodeDirectoy)
```

شکل ۱۹. الگوریتم تشخیص و اصلاح پادالگوی «تابع «is» چیزی بیشتر از نوع داده بولی برمی‌گرداند»

شامل پادالگوهای بحث شده هستند، گردآوری شده است. سپس تمام پادالگوها را با روش‌های ارائه شده تشخیص می‌دهیم و سپس رفع می‌کنیم. کدِ زبان اصلی قبل و بعد از بازسازی را با مقایسه می‌کنیم و به ارزیابی عملکرد برنامه می‌پردازیم. در مورد سیستم‌های نرم‌افزاری انتخابی در ادامه توضیح داده می‌شوند.

برای انتخاب این سیستم‌های نرم‌افزاری تلاش شده تا استخراج شده از سیستم‌های واقعی باشند تا از تطبیق بالای یافته‌های این پژوهش با شرایط دنیای واقعی اطمینان حاصل شود. همچنین سیستم‌های انتخاب شده از نظر پیچیدگی و مقیاس متفاوت هستند تا اطمینان حاصل شود که روش پیشنهادی می‌تواند در مقیاس‌های متفاوت کاربردی باشد. از طرفی سعی شده تا تنوع کاربرد در انتخاب سیستم‌های نرم‌افزاری لحاظ گردد. با در نظر گرفتن این تنوع می‌توان در مورد قابلیت تعمیم نتایج بهتر قضاوت کرد. از طرفی هر چهار سیستم انتخابی متن‌باز بوده و در گیت‌هاب در دسترس هستند. قابلیت دسترسی، تکرارپذیری آزمایش را تسهیل می‌کند و می‌تواند توسط سایر محققان مورد استفاده قرار گیرد. از طرفی، فرض بر این است که سیستم‌های چون توسط جامعه فعال پشتیبانی می شود، از استانداردهای صنعت پیروی می‌کنند و می‌توان اطمینان حاصل کرد که روش ما بر روی سیستم‌هایی که بر اساس استاندارد صنعت توسعه یافته‌اند، ارزیابی شده است.

در این ارزیابی نقاط قوت و ضعف روش ارائه شده در مواجه با کدِ زبان اصلی استفاده شده در سناریوهای کاربردی آشکار می‌شود. گفتنی است اصلاح تمام پادالگوها در بخش‌های آتی در تمام کدِ زبان اصلی‌ها به شکل صحیح نتیجه داده است. اما در تشخیص آن‌ها کاستی‌هایی وجود داشته که در ادامه به آن‌ها پرداخته شده است. تمامی توابعی که در کدِ زبان اصلی نرم‌افزارهای مورد بحث وجود دارند بررسی شده‌اند. لازم به ذکر است نتیجه عمل بازساخت کد در هر فایل کدِ زبان اصلی، بر روی همان فایل منعکس می‌شود. در واقع اگر تابعی دارای پادالگو باشد، تغییرات لازم بر روی همان تابع یا تابع جایگزین و در همان فایل منعکس می‌گردد. همچنین تمامی خطوط نرم‌افزارهای این بخش بررسی شده‌اند و این بررسی مختص به بخشی از آنها نیست. شکل‌های ۲۳ تا ۲۶ برای هر کدام از پادالگوهای مورد مطالعه، تعداد موجود و تشخیص داده شده در هر نرم‌افزار را نشان می‌دهد.

۶-۱- نرم‌افزار آرگو یوام‌ال

کد سیستم نرم‌افزاری آرگو یوام‌ال که در گیت‌هاب در دسترس است[32]، دارای ۳۶۵۷۸۷ خط کد و ۲۶۳۳ متد است. این پروژه دارای چندین فایل جاوا است اما در این ارزیابی برخی از آن‌ها استفاده شده‌اند زیرا تعدادشان زیاد است (حدود ۲۰۰۰۰ خط کد). نمودار شکل ۲۱ تعداد پادالگوهای تشخیص داده شده در این فایل‌ها را با تعداد موجود مقایسه می‌کند. پس از بررسی و مقایسه نتایج جدول فوق با نتایج دستی، مشاهده شد که تعداد پادالگوهای تشخیص‌داده شده در فایل‌ها بیشتر از مقدار به دست آمده با روش پیشنهادی است. پس از بررسی

32-https://github.com/argouml-tigris-org

```
AST = Java_to_AST(sourceCodeDirectory)
codeLines = java_to_lines(sourceCodeDirectoy)
methodNodes = AST.filter(type=methods)
variableNodes = AST.filter(type=variables)

for m in methodNodes:
    if m.name.lower_case().starts_with("set"):
        setMethods.append(m)

for m in setMethods:
    if m.return_type.name != "void":
        newMethodNameBase = m.name + "WithLog"
        newMethodName = newMethodNameBase
        index = 1

        for i in methodNodes:
            if i.name == newMethodName:
                newMethodName = newMethodNameBase + int_to_str(index)
                index++

        newVariableNameBase = "logFor" + newMethodName
        newVariableName = newVariableNameBase
        index = 1

        for v in variableNodes:
            if v.name == newVariableName:
                newVariableName = newVariableNameBase + int_to_str(index)
                index++


        newVariable = create_java_variable(name = newVariableName,
                                           modifiers = m.modifiers,
                                           data_type = m.return_type)

        newVariable.insert_to_code(codeLines, position = m.position)

        newMethod = create_java_method(name = newMethodName,
                                       modifiers = m.modifiers,
                                       return_type = void,
                                       body = m.body
                                       inputs = m.inputs)
        for j in newMethod.body:
            if type(j) == java.returnStatement:
                j = j.change_return_to_assign_statement(target_variable = newVariable)
        m.delete(codeLines)
        newMethod.insert_to_code(codeLines, position=m.position)

lines_to_java(codeLines, sourceCodeDirectoy)
```

مشخص شد روش تقسیم‌بندی کلمات نمی‌تواند در برخی موارد کلمات را به طور صحیح از هم جدا کند. در صورت وجود کلمات غیر انگلیسی نیز کارایی فرایند تقسیم‌بندی کلمات به شدت کاهش می‌یابد.

۶-۲- نرم‌افزار بلینک

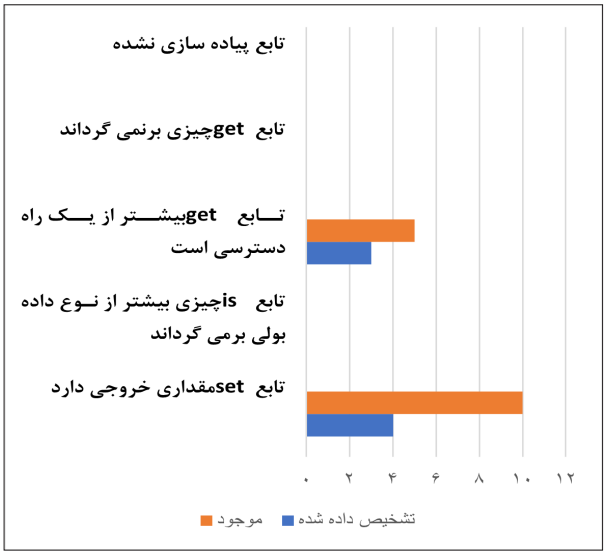
بلینک یک سرویس رایگان برای شبیه‌سازی سناریوهای اینترنت اشیا[۳۳] است. کدِ زبان اصلی این نرم‌افزار به صورت منبع‌باز در گیت‌هاب در دسترس قرار داده شده است[۳۴] و دارای ٤٨٦ خط کد جاوا و ۱۱ متد است.

نمودار شکل ۲۲ برای این نرم‌افزار تعداد پادالگوهای تشخیص داده شده را با تعداد موجود مقایسه می‌کند.

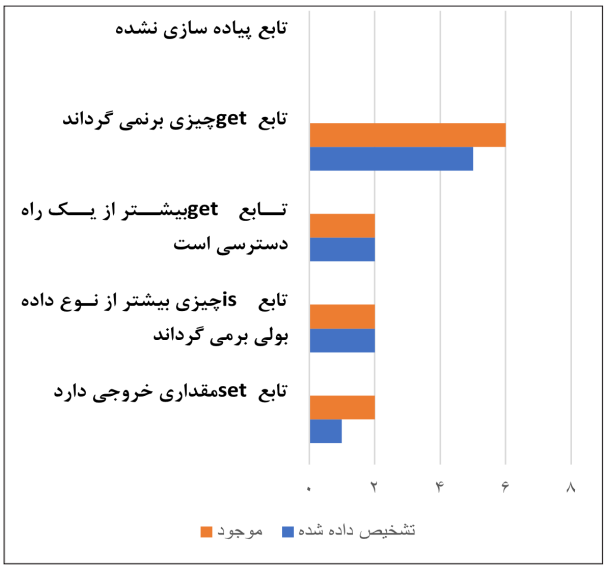
۶-۳- نرم‌افزار Tsunami Security Scanner

این نرم‌افزار جهت بررسی وضعیت امنیتی شبکه استفاده می‌شود و کد آن در گیت‌هاب در دسترس قرار دارد[۳۵]. تعداد خط‌های کد جاوا موجود در این سیستم ۲۱۷۷۵ خط و تعداد متدها ۸۰۷ است. نمودار شکل ۲۳

33-Internet of things(IoT)
34-https://github.com/blynkkk
35-https://github.com/google/tsunami-security-scanner



شکل ۲۱. مقایسه تعداد تشخیص داده شده و موجود پادالگوها.



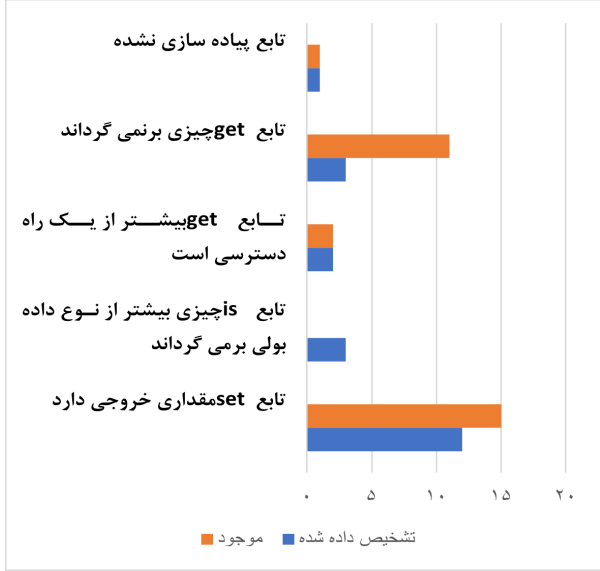
شکل ۲۲. مقایسه تعداد تشخیص داده شده و موجود پادالگوها.

تعداد پادالگوهای تشخیص داده شده را با تعداد موجود مقایسه می‌کند.

۶-۴- نرم‌افزار فرسکو

نرم‌افزار فرسکو برای نمایش تصاویر در سیستم عامل اندروید و توسط فیسبوک ساخته شده و کد آن در گیت‌هاب در دسترس است[۳۶]. این پروژه دارای ۱۰۸۲۳۵ خط کد جاوا و ۵۰۹۱ متد است. نمودار شکل ٢٤ تعداد پادالگوهای تشخیص داده شده را با تعداد موجود مقایسه می‌کند.

36-https://github.com/facebook/fresco

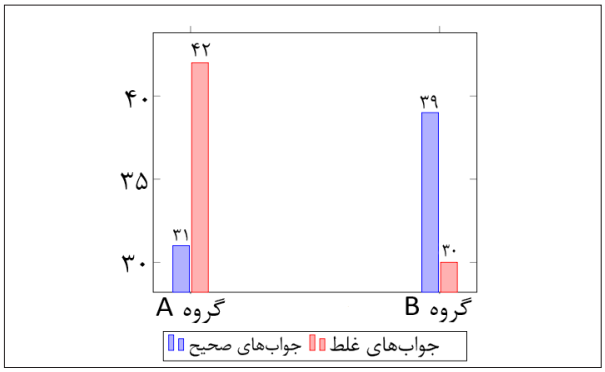


شکل ۲۳. مقایسه تعداد تشخیص داده شده و موجود پادالگوها.

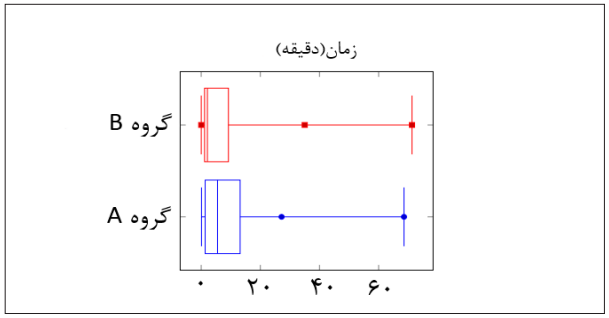
شکل ۲۴. مقایسه تعداد تشخیصی و موجود پادالگوها

۶-۵- بهبود درک‌پذیری کدِ زبان اصلی

تحقیقات کرمانسراوی (۲۰۱۹) [٦] در پاسخ به این پرسش که «آیا پادالگوهای زبان‌شناختی می‌توانند بر روی درک توسعه‌دهندگان از کد تاثیر گذارند؟» آزمایشی ترتیب داده است. در این آزمایش، تابعی با پادالگوی «تابع «is» چیزی بیشتر از نوع داده بولی باز می‌گرداند» و تابعی که بازسازی شده آن است به ترتیب به گروه A و B داده شد. سپس، از هر کدام از اعضای هر دو گروه سوالاتی درباره چگونگی کارکرد تابع پرسیده شد. نتایج شکل ۲۵ نشان می‌دهد درک گروهی که تابع بدون پادالگو (بازسازی) شده



شکل ۲۵. تاثیر اصلاح پادالگوی «تابع «is» چیزی بیشتر از نوع داده بولی بازمی‌گرداند» [۶].



شکل ۲۶. زمان مورد نیاز برای پاسخگویی به سوالات [۶].

را مطالعه می‌کردند، از تابع موردنظر بالاتر از گروه دیگر است، زیرا در این گروه ۳۹ درصد از شرکت‌کنندگان به سوال جواب صحیح دادند اما در گروه دیگر این مقدار ۳۱ درصد بوده است. همچنین کرمانسراوی بیان کرد که زمان مورد نیاز گروه B برای پاسخ دادن به سوالات در مجموع به صورت میانگین کمتر بوده است (شکل ۲٦).

۶-۶- بهبود فرایند تشخیص

مشکلات مشاهده شده در تشخیص پادالگوهای کدِ زبان اصلی نرم‌افزارهای ذکر شده در بخش قبل، غالباً مربوط به ناکارآمدی روش تقسیم‌بندی کلمات است. مشکلات این روش که با استفاده از کتابخانه wordsegment در پایتون انجام گرفته، ممکن است به روش پیاده‌سازی آن در این کتابخانه مربوط باشد و لزوماً ماهیت استفاده از این روش نادرست نباشد. همچنین، با وجود این که روش جایگزین تقسیم‌بندی کلمات نتایج صحیحی برای آزمایش‌ها داشت، اما بدیهی است چنین روشی نمی‌تواند در مواجهه با تمام

سناریوها عملکرد صحیحی داشته باشد. همچنین، برنامه در مواجهه با توابع انتزاعی عملکرد صحیحی نداشت که این مشکل برطرف شد. پس از مشاهده عدم توانایی روش تقسیم‌بندی کلمات بر روی نام‌هایی مانند setLenght، عملکرد برنامه برای نام‌هایی مانند set_breadth بررسی و صحیح ارزیابی شد. یعنی روش تقسیم‌بندی کلمات در صورتی که کلمات با خط تیره[37] از یکدیگر جدا شده باشند، عملکرد بهتری دارد.

۶-۷- تغییر تعداد خطوط کد

کم و یا زیاد شدن خطوط کد به نوع و تعداد پادالگوهای موجود در آن وابسته است. در ادامه تاثیری که هر کدام از پادالگوها پس از بازسازی بر روی تعداد خطوط می‌گذارند آورده شده است.

۶-۷-۱- تابع پیاده‌سازی‌نشده

این پادالگو نمی‌تواند باعث افزایش خطوط شود زیرا روش ارائه شده برای آن، حذف کردن آن است. تعداد خطوط حذف شده به واسطه رفع این پادالگو به صورت زیر است.

$$D = ۱ + B \ >= ۱ \quad (۱)$$

که در آن معنی متغیرها به شرح زیر است:

D تعداد خطوط پاک شده.

N تعداد خطوط بدنه تابع.

مقدار ۱ ثابتی که در سمت راست قرار دارد، به دلیل وجود خطی است که تعریف تابع از آن آغاز می‌شود.

۶-۷-۲- تابع «Get» چیزی برنمی‌گرداند

از آنجایی که روش ارائه شده برای رفع این پادالگو تغییر نام آن است، تغییری در تعداد خطوط کد به وجود نمی‌آید.

$$D = ۰ \quad (۲)$$

۶-۷-۳- تابع «Get» بیشتر از یک راه دسترسی است

در رفع این پادالگو خطوط حذف نمی‌شوند و بالعکس خطوطی اضافه می‌شوند. زیرا یک تابع به کدِ زبان اصلی اضافه می‌شود. تغییر نام توابع تغییری در تعداد خطوط ایجاد نمی‌کند. تعداد خطوط اضافه شده در کد با رابطه زیر مشخص می‌شود.

$$\boldsymbol{I = 1 + B = 3} \quad (۳)$$

علت اضافه شدن ۳ خط آن است که تابع اضافه شده تنها شامل ۳ خط است. ۱ خط برای خط آغازین که شامل نام تابع است و ۲ خط دیگری که آکولادها پوشش می‌دهند

۶-۷-۴- تابع «Is» چیزی بیشتر از نوع داده بولی برمی‌گرداند

با توجه به این که تابع به جای نوع داده بولی چه نوع داده‌ای برمی‌گرداند، تعداد خطوط اضافه شده می‌تواند متفاوت باشد. در رفع این پادالگو هیچ خطی حذف نمی‌شود زیرا در آن یک تابع به کدِ زبان اصلی اضافه می‌کنیم و تابع قبلی و نوع داده خروجی‌اش تغییر نام می‌یابد. تعداد خطوط تابع اضافه شده ثابت نیست و به نوع داده خروجی تابع اولیه وابسته است. بدین جهت تعداد خطوط تابع جدید را به صورت میانگین حالت‌های مختلف در نظر می‌گیریم و با $\bar{L}$ نشان می‌دهیم. این میانگین شامل خط آغازین تابع نیز می‌شود.

$$\bar{L} = \left(\frac{۱}{n}\right)\sum_{i=۱}^{n} ۱ + B_i = ۷.۵ \quad (٤)$$

در رابطه بالا n تعداد سناریوهایی است که در آن‌ها تابع Is نوع داده خروجی‌ای به غیر از بولی دارد. در این مقاله ۱۸ سناریو مختلف در نظر گرفته شده است و به ازای آن‌ها، مقدار $\bar{L}$ برابر با ۷.۵ است.

پس از اجرای کد بازسازی روی هر ٤ پروژه در مجموع حدود ۵۰ خط کد به پروژه اضافه شده است.

۶-۷-۵- تابع «Set» مقدار بازگشتی دارد

در رفع این پادالگو نام تابع تغییر می‌کند و یک صفت به کلاس اضافه می‌شود. اضافه شدن صفت به معنای افزایش تعداد خطوط کد به اندازه یک واحد است.

$$I = ۱ \quad (۵)$$

37-Underline

۶-۸- مقایسه با روش لیو

در روش لیو (۲۰۱۹) [۱۸] به دلیل استفاده از یادگیری ماشین نیاز به داشتن داده‌هایی وجود دارد که قبلاً دسته‌بندی[38] شده باشند و به کمک آن‌ها ماشین آموزش داده شود تا بتواند روش لیو را اجرا کند. روش پیشنهادی از این جهت ساده‌تر است اما می‌توان با تکنیک‌های یادگیری ماشین آن را تقویت نمود. همچنین در روش لیو، با توجه به این که برای شناسایی عدم تطابق نام و بدنه تابع از توابعی که عملکرد یکسانی دارند استفاده می‌شود، در صورتی که توابع مشابه تعدادشان به اندازه کافی بزرگ نباشد، احتمال این که برای یک تابع بشود تابعی مشابه پیدا کرد پایین می‌آید.

لیو در گام بازسازی (پس از تشخیص) تنها نام تابع را تغییر می‌دهد. از آنجا که این نام با توجه به توابع با عملکرد مشابه پیشنهاد می‌شود، اگر تابعی که نامش پیشنهاد شده درون همان کدِ زبان اصلی تابع اول باشد، مشکل وجود دو تابع با نام یکسان پیش می‌آید. این در حالی است که روش پیشنهادی بر روی اصلاح بدنه توابع تمرکز دارد و در موارد محدودی نام توابع را تغییر می‌دهد و این تغییر با توجه به نام دیگر توابع نیست.

با توجه به جدول ٤ در روش لیو اگر بتوان یک تابع بسیار مشابه را پیدا کرد (k=1) آنگاه دقت تشخیص ناسازگاری میان نام و بدنه تابع حداکثر ۸٤/٥ درصد خواهد بود. روش پیشنهادی با توجه به بررسی نتایج اعمال آن بر روی کدِ زبان اصلی نرم‌افزارهای مذکور در بخش ٦-۱ تا ٦-٤ با دقت ٦۹/٤ پادالگوهای موجود را تشخیص داد. پیش‌بینی می‌شود در صورت استفاده از روش‌های کارآمدتر در تقسیم‌بندی کلمات بتوان این دقت را به شکل قابل ملاحظه‌ای افزایش داد.

۷.نتیجه‌گیری

با ارزیابی روش پیشنهادی نواقصی در فرایند تشخیص آن آشکار شد. همچنین مهم‌ترین اثر اصلاح پادالگوهای کدِ زبان اصلی، بهبود درک‌پذیری آن است. بدین جهت در بخش نتیجه‌گیری به این دو مسئله پرداخته می‌شود. مزایا و معایب روش پیشنهادی در زیر مورد بحث قرار گرفته است. مزایای کار پیشنهادی عبارتند از:

۱. با تشکیل درخت نحو انتزاعی اطلاعاتی فراوانی به دست می‌آید. این اطلاعات می‌تواند در توسعه این مقاله در آینده مفید باشد.
۲. نیاز به بررسی و اعتبارسنجی بازساخت خودکار کد توسط نیروی انسانی به حداقل می‌رسد.
۳. اگر نام‌گذاری تابعی تغییر کند، فراخوانی‌های تابع در جای جای کد نیز با اسم جدید بروز می‌شوند.
٤. اگر یک تابع بیش از یک پادالگو داشته باشد، برای هر پادالگو مستقلاً عمل شده و رفع می‌گردد.
٥. با از میان بردن پادالگوهای زبان شناختی، کیفت کدِ زبان اصلی از جهت تمیز بودن نیز بالا می‌رود.
٦. به دلیل وجود کلاس‌های مختلف برای هر پادالگو، می‌توان اشیاء ساخته شده به جهت تشخیص پادالگوها را ذخیره و از عملیات صورت‌گرفته مستنداتی تهیه کرد.

اما معایب کار پیشنهادی عبارتند از:

۱. با رفع پادالگوها در نتیجه تغییر کدِ زبان اصلی، باید پس از رفع هر پادالگو درخت نحو انتزاعی جدیدی ساخت.
۲. به دلیل پرداختن به هر پادالگو به صورت مستقل، بار اضافه پردازشی به وجود می‌آید و بسیاری از فرایندها چندین بار اجرا می‌شوند.
۳. از آنجایی که کدِ زبان اصلی دارای پسوند .java است، باید پس از هر تغییر در رشته متنی کدِ زبان اصلی، آن را ذخیره نمود و برای تغییرات بعدی دوباره این اتفاق خواهد افتاد. یعنی بار پردازشی حلقه تبدیل فایل جاوا به رشته متنی و بالعکس بالا است.
٤. به دلیل اضافه کردن پسوند یا پیشوند به نام توابع در راه‌حل رفع برخی پادالگوها، نام آنها طولانی‌تر می‌شود که از جهت کیفیت کدِ زبان اصلی، یک تاثیر

38-Classification

منفی است.

۵. درباره پادالگو «تابع «Get» بیشتر از یک راه دسترسی است»، اگر در بدترین حالت، تمام توابع دارای این پادالگو باشند، با رفع پادالگوها تعداد توابع کدِ زبان اصلی دو برابر می‌شود (زیرا به ازای هر تابع یکی دیگر نیز ساخته می‌شود).

۶. درخت نحو انتزاعی استفاده شده در این مقاله قابلیت تشخیص توضیح را ندارد. به همین دلیل زیرساخت مناسب برای مطالعه پادالگوهای زبان‌شناختی مرتبط با توضیح‌گذاری را فراهم نمی‌کند.

۷. از آنجایی که برای رفع اکثر پادالگوهای مطرح شده، به کدِ زبان اصلی خطوطی اضافه و کم می‌شود، خطوط اضافه شده داره توضیح نیستند و ممکن است برای توسعه‌دهنده، درک تغییراتی که پس از بازساخت کد اعمال شده‌اند سخت باشد.

۷-۱- کارهای آینده

موضوعات مرتبطی که به عنوان توسعه‌های آتی احتمالی این کار پژوهشی می‌توانند موردنظر قرار گیرند عبارتند از:

۱. ارائه روشی برای به دست آوردن اطلاعات مربوط به توضیح‌گذاری‌ها در کدِ زبان اصلی. همان‌طور که در این گزارش ذکر شد، درخت نحو انتزاعی توضیح‌های کدِ زبان اصلی را در نظر نمی‌گیرد. بدین جهت ارائه یک روش جایگزین مفید است.

۲. ارائه روشی برای بررسی میزان صحت اطلاعات درون مستندات به خصوص توضیح‌ها. از آنجایی که مستندات صحیح می‌توانند درک‌پذیری نرم‌افزار را بالا ببرند، ارائه روشی برای صحت‌سنجی مستندات امری مفید خواهد بود.

۳. از آنجایی که اطلاعات شناسه‌ها و نامشان از درخت نحو انتزاعی قابل برداشت است، مطالعه پادالگوهای زبان‌شناختی مربوط به شناسه‌ها نیز می‌تواند مورد مطالعه قرار گیرد.

۴. بررسی تاثیر اصلاح پادالگوهای زبان‌شناختی کدِ زبان اصلی بر روی عملکرد و همچنین مقایسه تلاش لازم برای نگهداری و گسترش سیستم‌های دارا و فاقد پادالگوهای زبان‌شناختی می‌تواند دروازه‌ای برای دعوت برنامه‌نویسان به رعایت هر چه بیشتر اصول و قواعد برنامه‌نویسی در زمینه واژگان باشد.

۵. ارائه یک روش دقیق و صحیح برای پیاده‌سازی تقسیم‌بندی کلمات. در فصل ارزیابی مشاهده شد که عدم کارایی این روش چگونه می‌تواند بر تشخیص پادالگوهای موجود در کدِ زبان اصلی اثر منفی بگذارد.

۶. برقراری روشی جهت بررسی کدِ زبان اصلی برای تشخیص وجود پادالگوهای زبان‌شناختی به صورت بی‌درنگ. این روش می‌تواند علاوه بر از بین بردن نیاز به زمان اضافی برای بررسی کد پس از توسعه، مشکلات ناشی از نام‌گذاری‌های نامناسب را در لحظه به برنامه‌نویس اطلاع دهد و باعث یک خوداصلاحی در روش برنامه‌نویسی توسعه‌دهنده شود. پیاده‌سازی چنین روشی می‌تواند در قالب یک افزونه برای محیط‌های توسعه یکپارچه باشد.

۷. از آنجایی که روش پیشنهادی بهینه نیست و در آن برخی از روندها چندین بار تکرار می‌شوند، اصلاح آن به گونه‌ای که تمام قسمت‌های کدِ زبان اصلی را به صورت یکپارچه و هم‌زمان بازسازی کند مفید است.

۸. معرفی روشی بی‌درنگ و هوشمند جهت ارائه پیشنهاد، هنگام نام‌گذاری توابع و یا شناسه‌ها. در بسیاری از توابع، شناسه‌ها دارای نام‌هایی مرتبط با حوزه مربوط به نرم‌افزار هستند. برای مثال اگر نرم‌افزاری در حوزه ارزهای دیجیتال توسعه یافته باشد، احتمال این که در توابع آن از کلمه Crypto و یا Trade استفاده شده باشد بسیار بالا است. بنابراین، با داشتن حوزه کاری نرم‌افزار و تحلیل کارکرد هر یک از توابع آن، می‌توان نام‌هایی مناسب برای توابع

پیشنهاد داد.

۹. بررسی رابطه میان نوع و تعداد پادالگوهای زبان‌شناختی موجود در نرم‌افزار و مشخصه‌های محیطی تیم توسعه‌دهنده. دانستن این که شرایط محیطی چگونه می‌تواند بر روی سبک برنامه‌نویسی افراد تاثیر بگذارد می‌تواند باعث ایجاد یک چارچوب استاندارد جهت برقراری محیطی شود که در آن برنامه‌نویسان از جهت ذهنی توانایی بالایی در رعایت اصول و قواعد نام‌گذاری داشته باشند.

۱۰. از آنجایی که تحقیقات آرنودوا (۲۰۱٤) [۱۷] ابزاری به نام LAPD را برای آشکارسازی پادالگوها به صورت جامع معرفی می‌کند، گسترش این روش در جهت رفع خودکار پادالگوهای آشکار شده می‌تواند قابل توجه باشد.

۸. مراجع

[1] V. Arnaoudova, “A New Family of Software Anti-Patterns: Linguistic Anti-Patterns,” in Proceedings of the 2013 17th European Conference on Software Maintenance and Reengineering, Genova, Italy, March 5-8, 2013, pp. 187-196.

[2] M. Wahler and U. Drofenik, “Improving Code Maintainability: A Case Study on,” in Proceedings of the 2016 IEEE International Conference on Software Maintenance and Evolution (ICSME), Raleigh, NC, USA, October 2-7, 2016, pp. 493-501.

[3] F. Arcelli Fontana, “Metrics and Antipatterns for Software Quality Evaluation,” in Proceedings of the 2011 IEEE 34th Software Engineering Workshop, Milano, Italy, June 20-21, 2011, pp. 48-56.

[4] A. Stoianov, “Detecting Patterns and Antipatterns in Software using Prolog Rules,” in Proceedings of the IEEE 2010 International Joint Conference on Computational Cybernetics and Technical Informatics, Timisoara, Romania, May 27-29, 2010, pp. 253-258.

[5] C. U. Smith, “Software Performance AntiPatterns,” in Proceedings of the 27th International Computer Measurement Group Conference, Boulder, Colorado, December 2-7, 2001, pp. 797-806.

[6] Z. Kermansaravi, “Linguistic Anti-Patterns: Impact Analysis on Code Quality,” arXiv preprint arXiv:2104.00058v1. March 31, 2019.

[7] “Abstract syntax tree,” Wikipedia, URL: https://en.wikipedia.org/wiki/Abstract_syntax_tree/, Access Date: 4 Feb 2022.

[8] D. v. Leewen, “Comprehensible Method Names: Focusing on the Nouns,” University of Amsterdam, 2012.

[9] D. Steidl and B. Hummel, “Quality analysis of source code comments,” in Proceedings of the 2013 21st International Conference on Program Comprehension (ICPC), San Fransisco, CA, USA, May 20-21, 2013. pp. 83-92.

[10] D. Souza, “A Study of the Documentation Essentials to Software Maintenance,” in Proceedings of the 23rd annual International Conference on Design of Communication: Documenting & Designing for Pervasive Information, New York, NY, USA, September 21-23, 2005, pp. 68-75.

[11] C. S. Hartzman and C. F. Austin, “Maintenance Productivity: Observations Based on an Experience in a Large System Environment vol. 1,” IBM Press, Oct, 1993.

[12] A. D. Lucia and M. D. Penta, “Improving Source Code Lexicon via Traceability and Information Retrieval,” IEEE Transactions on Software Engineering, vol. 37, no. 2, pp. 205-227, 2011.

[13] S. Harry, “Object-oriented COBOL recycling,” in Proceedings of the WCRE ‘96: 3rd Working Conference on Reverse Engineering, DC, United States, November 8-10, 1996, pp. 169-178.

[14] S. Haiduc and A. Marcus, “On the Use of Domain Terms in Source Code,” in Proceedings of the 2008 16th IEEE International Conference on Program Comprehension, Amsterdam, Netherlands, July 10-13, 2008, pp. 113-122.

[15] D. Lawrie, “Quantifying identifier quality: an analysis of trends,” Kluwer Academic Publishers, vol. 12, no. 4, pp. 359-388, 2007.

[16] S. Butler, M. Wermelinger, Y. Yu and H. Sharp, “Relating Identifier Naming Flaws and Code Quality: An Empirical Study,” in Proceedings of the 2009 16th Working Conference on Reverse Engineering, Lille, France, October 13-16, 2009, pp. 31-35.

[17] V. Arnaoudova, “Towards Improving the Code Lexicon and its Consistency (PhD thesis),” École Polytechnique de Montréal ,2014.

[18] L. Kui, “Learning to Spot and Refactor Inconsistent Method Names,” in Proceedings of 2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE), Montreal, Canada, May 25-31, 2019, pp. 1-12.

[19] V. Arnaoudova, “Linguistic Antipatterns: What They Are and How Developers Perceive Them,” Empirical Software Engineering, vol. 21, no. 1, pp. 104-158, 2016.

[20] “Text segmentation,” Wikipedia, URL: https://en.wikipedia.org/wiki/Text_segmentation/, Access Date: 5 Feb 2022.

[21] E. Aghajani, C. Nagy, G. Bavota, M. Lanza, “A large-scale empirical study on linguistic antipatterns affecting APIs,” In 2018 IEEE International conference on software maintenance and evolution (ICSME) IEEE, Madrid, Spain, 2018, pp. 25-35.