ICWR2022
May 11-12, 2022; Tehran, Iran
8th International Conference on Web Research

Automatic Refactoring of Software Source Code to Eliminate Linguistic Antipatterns

Mohammad Amin Shahidi Nashroudkoli[1],Mehrdad Ashtiani*[2]

[1] Computer Engineering, Iran University of Science and Technology, Tehran, Iran
amin7799sh@gmail.com

[2] Assistant Professor, Iran University of Science and Technology, Tehran, Iran
m_ashtiani@iust.ac.ir

Abstract

Nowadays, due to the important role of software systems in our lives, without the use of software, we will not be able to get most of the services we are used to. The quality of these services depends on the quality of the software that implements them. There are many criteria for measuring the quality of a software. The presence or absence of antipatterns can be a measure of software quality. Some well-known Antipatterns are studied according to their effect on performance, reliability and other related criteria. One of these criteria is the readability of the software source code. Developers of a program are not necessarily the only ones developing it in the future, so it is important to follow tips that make it easier for potential developers to understand how the program works. This article examines the linguistic antipatterns associated with naming functions and provides solutions for their automatic resolution. Antipatterns that are related to the naming of members of the software source code are called linguistic antipatterns. This paper presents a method for automatic detection and elimination of these antipatterns using abstract syntax tree. The proposed method is then tested on the source code of several open source softwares.

Keywords: Antipattern, Linguistic Antipattern, Code Refactoring, Automated Code Refactoring, Clean Code.

ICWR2022
May 11-12, 2022; Tehran, Iran
8th International Conference on Web Research

بازسازی خودکار کد منبع نرم‌افزار برای رفع ضدالگوهای زبانشناختی

محمدامین شهیدی نشرودکلی[۱]، مهرداد آشتیانی*[۲]

[۱] مهندسی کامپیوتر، دانشکده‌ی مهندسی کامپیوتر، دانشگاه علم و صنعت ایران، تهران
amin7799sh@gmail.com

[۲] استادیار، دانشکده‌ی مهندسی کامپیوتر، دانشگاه علم و صنعت ایران، تهران
m_ashtiani@iust.ac.ir

چکیده

امروزه با توجه به نقش مهمی که نرم‌افزار در زندگی ما دارد، بدون استفاده از نرم‌افزار نمی‌توانیم بسیاری از خدماتی که به آن‌ها عادت کرده‌ایم را دریافت کنیم. کیفیت این خدمات به کیفیت نرم‌افزاری که آن‌ها را پیاده‌سازی می‌کند وابسته است. معیارهای فراوانی برای سنجش کیفیت یک نرم‌افزار وجود دارد که هرکدام از جهتی به سنجش کیفیت نرم‌افزار می‌پردازد. وجود یا عدم وجود ضدالگوها می‌تواند معیاری برای سنجش کیفیت نرم‌افزار باشد. برخی ضدالگوهای شناخته‌شده با توجه به تأثیرشان بر روی کارایی، اطمینان‌پذیری و سایر معیارهای مرتبط مطالعه می‌شوند. یکی از این معیارها خوانایی کد منبع نرم‌افزار است. سازندگان یک برنامه لزوماً تنها کسانی نیستند که به توسعه آن می‌پردازند، بدین جهت رعایت نکاتی که فهمیدن چگونگی کارکرد برنامه را برای توسعه‌دهندگان احتمالی ساده‌تر می‌کند ضروری است. این مقاله، به بررسی ضدالگوهای زبانشناختی مرتبط با نام‌گذاری توابع و ارائه‌ی راه‌حل برای رفع خودکار آن‌ها می‌پردازد. به ضدالگوهایی که مربوط به نام‌گذاری اعضای کد منبع نرم‌افزار هستند ضدالگوهای زبانشناختی گفته می‌شود. در این مقاله روشی برای تشخیص و رفع خودکار این ضدالگوها به کمک درخت نحو انتزاعی ارائه می‌شود. سپس روش ارائه‌شده بر روی کد منبع چند نرم‌افزار متن‌باز آزمایش می‌شود.

کلمات کلیدی

ضدالگو، ضدالگوی زبانشناختی، بازسازی کد، بازسازی خودکار کد، کد تمیز.

۱- مقدمه

تحقیقات اخیر نشان می‌دهد انتخاب واژگان نامناسب در کد منبع[۱] نرم‌افزار می‌تواند در فهم[۲]، قابلیت نگهداری[۳] و کیفیت کلی آن اثر منفی داشته باشد. کسی که یک نرم‌افزار را از ابتدا توسعه می‌دهد به احتمال قوی دچار سؤبرداشت نمی‌شود زیرا با تمام قسمت‌های برنامه آشنا است. اما برای فردی که وظیفه‌ی نگهداری، توسعه و یا تغییر یک نرم‌افزار موروثی[۴] را دارد این احتمال وجود دارد که در فهم کارکرد برنامه دچار اشتباه شود. بسیاری از نرم‌افزارها (یا قسمتی از آن‌ها) توسط افرادی نوشته می‌شود که تحصیلات رسمی در زمینه‌ی مهندسی نرم‌افزار ندارند [1]. حال اگر چنین فردی اقدام به توسعه‌ی کد منبع نرم‌افزاری کند که از قبل توسط فردی دیگر و با قواعد نام‌گذاری نامناسب توسعه یافته، به احتمال قوی دچار سؤبرداشت در درک نحوه‌ی کارکرد برنامه و یا حتی هدف آن می‌شود. یک کد منبع ناخوانا می‌تواند علاوه بر معیار خوانایی[۵]، بر روی سایر معیارها نیز اثر منفی داشته باشد. برای مثال نگهداری نرم‌افزار که یک فرایند هزینه‌بر است [2]، نیازمند فهمی صحیح از کارکرد قسمت‌های مختلف کد منبع است اما وجود واژگان نامناسب در نام توابع می‌تواند باعث گمراهی در این مورد شود و فرایند نگهداری[۶] را با مشکل روبرو، و پرهزینه‌تر کند. قابل‌درک بودن، خوانایی را نتیجه می‌دهد، سپس تغییرپذیری[۷]، گسترش‌پذیری[۸] و قابلیت نگهداری بدست می‌آید [3].

با رشد نیاز به توسعه‌ی زیرساخت‌های قدیمی، سهولت درک نرم‌افزارهای موروثی قدیمی یک پیشنیاز ضروری است [4]. این امر می‌تواند

با وجود مستندات مناسب برای توسعه‌دهندگان جدید نرم‌افزار قدیمی مهیا شود. همچنین نگهداری از سیستم نرم‌افزاری قدیمی به شدت به کیفیت کد سیستم وابسته است [5]. عدم همخوانی در مستندات و کد منبع نرم‌افزار نیز یکی دیگر از جنبه‌های ضدالگوهای زبانشناختی[9] است.

با توجه به این که ضدالگوها بخش بزرگی از مطالعات مربوط به مهندسی نرم‌افزار را معطوف به خود کرده‌اند، راه‌حل‌هایی نیز برای رفع آنها پیشنهاد شده است. رفع ضدالگوها در کد منبع نرم‌افزار که به آن بازسازی[10] کد گفته می‌شود برای یک کد منبع بزرگ، در صورتی که توسط انسان انجام شود کاری پیچیده و زمانبر است، بدین جهت این بازسازی را می‌توان به عهده‌ی رایانه گذاشت تا در زمان و هزینه صرفه‌جویی شود، در این حالت به آن بازسازی خودکار کد گفته می‌شود. بازسازی خودکار کد شامل دو مرحله‌ی اصلی تشخیص و اصلاح است. عمل پیدا کردن بوی بد[11] کد و رسیدن به ضدالگوها، در مرحله‌ی تشخیص است. پیاده‌سازی راه‌حل پیشنهادی مناسب در کد برای رفع ضدالگوها در مرحله‌ی اصلاح است. این مقاله به ارائه‌ی روشی برای تشخیص و اصلاح برخی از ضدالگوهای زبانشناختی در نام‌گذاری توابع در کد منبع جاوا به کمک زبان پایتون می‌پردازد. همچنین قابل ذکر است که مفاهیم مطرح‌شده در این مقاله از جهت اثرشان بر روی خوانایی کد منبع دارای شباهت‌های فراوان به مباحث کد تمیز[12] هستند. البته تعریف یگانه و مشخصی برای کد تمیز وجود ندارد [6].

۲- کارهای مرتبط

عمده‌ی تحقیقات صورت‌گرفته بر روی جنبه‌ی زبانشناختی کد منبع، به کیفیت، درک‌پذیری کد منبع نرم‌افزار و یا تشخیص ضدالگوهای زبانشناختی می‌پردازند. در این بخش خلاصه‌ای از تحقیقاتی که در این سه حوزه صورت گرفته، ارائه می‌شود.

۲-۱- تحقیقات مبتنی بر همزمانی اسامی

یکی از معیارهای مورد مطالعه برای سنجش درک‌پذیری نرم‌افزار، همزمانی اسامی[13] است. در پژوهش مربوط به لیون (۲۰۱۲) مطالعه بر روی نوشتار کد منبع ۳۱ سیستم متن‌باز نشان می‌دهد در ۸۳ درصد از توابع، همزمانی اسامی رخ داده است [۷]. یعنی در آن‌ها حداقل یک اسم هم در نام تابع و هم در شناسه‌های مختلف بدنه‌ی تابع موجود است. اگر همزمانی اسامی در تابعی رخ دهد، شانس این که آن تابع درک‌پذیر به حساب آید بیشتر می‌شود. در توابعی که طولانی‌اند یا فرایندهای پیچیده را پیاده‌سازی می‌کنند، استفاده از همزمانی اسامی توصیه می‌شود.

۲-۱-۱- اسامی توابع درک‌پذیر و همزمانی اسامی

از آنجایی که درک کردن یک تابع امری ذهنیست، افراد با ذهن‌های مختلف می‌توانند تحلیل متفاوتی از میزان درک‌پذیری تابع داشته باشند. وجود همزمانی اسامی برای اسامی‌ای 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 همزمانی اسامی را معیاری قابل‌اندازه‌گیری برای درک‌پذیری دانست. این امر با شمارش تعداد دفعاتی که یک اسم در پیاده‌سازی تابع آمده است انجام می‌گیرد. هرقدر این تعداد بیشتر باشد، نقش آن اسم در تابع پررنگ‌تر است.

با بررسی همزمانی 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 همزمانی اسامی‌اند اما در دیگری چنین شرطی وجود ندارد.

مزایای استفاده از همزمانی اسامی برای سنجش درک‌پذیری توابع:

۱. احتمال این که یک تابع دارای همزمانی اسامی، قابل‌درک نیز باشد ۱۹ درصد بیشتر از حالتی است که همزمانی اسامی نداشته باشد.
۲. اگر همزمانی اسامی مربوط به کلمات تخصصی حوزه‌ی کاری نرم‌افزار باشد، خواننده متن کد از موضوع حوزه‌ی نرم‌افزار مربوطه نیز مطلع می‌شود.
۳. با مقایسه تعداد تکرار کلمات مختلف می‌توان اهمیت آن‌ها را در کد منبع مقایسه و رده‌بندی کرد.

معایب استفاده از همزمانی اسامی برای سنجش درک‌پذیری توابع:

۱. این روش دقیق نیست. به عبارت دیگر، هیچ تضمینی وجود ندارد که رخدادن همزمانی اسامی باعث افزایش درک‌پذیری تابع شود.
۲. استفاده متعدد از یک کلمه در قسمت‌های مختلف کد، با وجود این که مصداق رخدادن همزمانی اسامی است، درصورت استفاده‌ی بی‌مورد می‌تواند درک‌پذیری کد منبع را کاهش دهد.
۳. رخدادن همزمانی اسامی در نام شناسه‌ها ممکن است باعث شباهت بیش از اندازه‌ی اسامی شناسه‌ها و در نتیجه، کاهش درک‌پذیری کد منبع شود.

۲-۱-۲- استفاده از افعال برای یافتن همزمانی اسامی

برنامه‌نویسان اغلب از یک فعل خاص در نام تابع استفاده می‌کنند اگر در پیاده‌سازی آن از اظهارات و اصطلاحات مشخصی استفاده شده باشد. اسامی‌ای مانند get، set، is از اینگونه افعال هستند. آزمایشات لیون (۲۰۱۲) نشان می‌دهد بیشترین افعال استفاده‌شده در نام توابع به ترتیب مانند جدول (۲) هستند [۷].

جدول (۱) : مقایسه‌ی درک پذیری توابع [8]

معیار	بدون همزمانی اسامی	دارای همزمانی اسامی
سخت بودن فهم پیاده‌سازی	۶	۱
تابع درک پذیر	۵۳	۸۲
تابع درک پذیر سخت	۳۱	۱۶
خطای Part of speech	۶	۱
ایجاد کننده‌ی Exeption	۴	۰
کل	۱۰۰	۱۰۰

همچنین آزمایشات لیون (۲۰۱۲) نشان می‌دهد همزمانی اسامی در توابعی که نامشان با افعال مختلفی آغاز می‌شود، در نقاط مختلفی از پیاده‌سازی آن‌ها رخ می‌دهد. تصاویر شکل (۱) و شکل (۲) نتایج تحقیقات لیون (۲۰۱۲) برای توابعیست که نامشان به ترتیب با get و set آغاز می‌شود.

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

۱. در صورت پیاده‌سازی روشی برای پیداکردن همزمانی اسامی در توابع، با توجه به نام توابع می‌توان قسمت‌های مختلف بدنه‌ی تابع را جهت جست‌وجو اولویت‌بندی کرد.
۲. می‌توان از این ابزار استفاده‌ی 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 همزمانی اسامی رخداده است، می‌توان با احتمال بالایی اظهار کرد که بهتر است نام تابع با کدام یک از کلمات جدول (۲) آغاز شود.
۳. با بررسی کلمه‌ی آغازین نام تابع و نقاط رخدادن همزمانی اسامی در بدنه‌ی آن، می‌توان مناسب بودن نام تابع را مورد ارزیابی قرار داد.

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

۱. بررسی نام تابع و کلمه‌ی آغازین آن، نیازمند شکستن نام تابع به عبارات تشکیل دهنده‌ی آن است. برای این کار نیز باید روشی مناسب انتخاب شود.
۲. اگر برنامه‌نویس از شیوه‌ی برنامه‌نویسی خاصی استفاده کرده باشد که در آن از کلمات جدول (۲) در نامگذاری توابع استفاده نمی‌شود، روش مطرح‌شده در این بخش کارآمد نخواهد بود.

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

قواعدی که در برنامه‌نویسی جهت افزایش کیفیت به کار می‌روند شامل اجزای مختلفی از کد منبع می‌شود. عدم رعایت این قوانین، با توجه به این که برای چه عضوی از کد منبع 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.

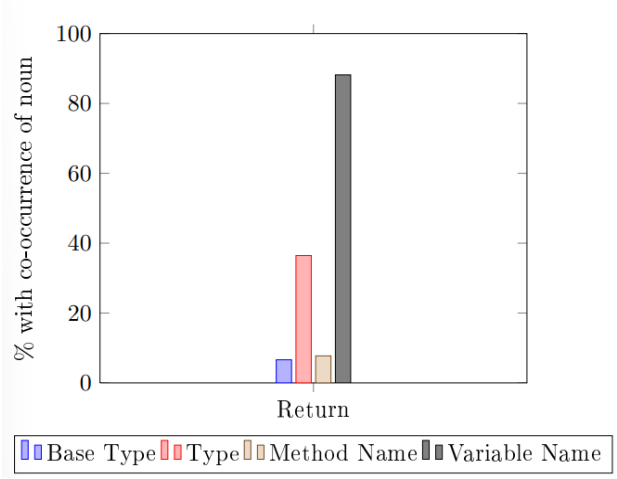
جدول (۲) : بیشترین افعال مورد استفاده در نام توابع [8]

رتبه	فعل	تعداد توابع	درصد توابع
۱	Get	34200	26.19
۲	Set	14408	11.03
۳	Is	5810	4.45
۴	Add	3858	2.95
۵	Create	3688	2.82
۶	Visit	1878	1.44
۷	Remove	1780	1.36
۸	Test	1462	1.12
۹	Has	1383	1.06
۱۰	Read	1040	0.80

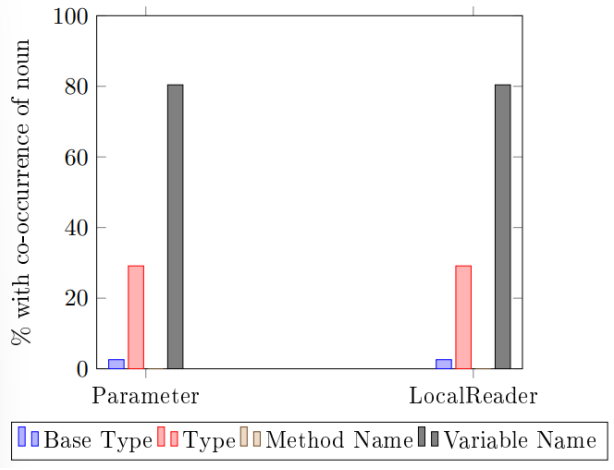
۲-۲-۱- تأثیر کامنت‌ها بر کیفیت کد منبع

بخش بزرگی از کد منبع در سیستم‌های نرم‌افزاری از کامنت‌ها تشکیل می‌شود که پیاده‌سازی‌ها را مستند می‌کنند و به توسعه‌دهندگان در درک کد کمک می‌کنند [8]. همچنین کامنت‌ها دومین نوع پراستفاده مستندات برای درک‌پذیری کد هستند [9]. توسعه‌دهندگان به طور گسترده‌ای معتقدند که مستندسازی ضعیف باعث سؤبرداشت می‌شود [10]. همچنین تحقیقات نشان می‌دهد مستندسازی ضعیف بر روی قابلیت نگهداری نرم‌افزار تاثیر منفی می‌گذارد. استفاده از کامنت یا هر روش مستندسازی دیگری ممکن است تحت تأثیر بازه‌های زمانی محدودی که برای تیم توسعه‌دهندگان تعریف می‌شود قرار گیرد. استیدل (۲۰۱۳) برای زبان‌های برنامه‌نویسی C/C++ و جاوا کامنت‌ها را دسته‌بندی می‌کند و بیان می‌کند تعداد کامنت‌ها در کد بیانگر استفاده‌ی مناسب از کامنت برای درک‌پذیری نیست زیرا برخی از این انواع کمکی به این هدف نمی‌کنند [۸]. دسته‌بندی کامنت‌ها به صورت زیر است:

۱. **کامنت‌های حق نسخه‌برداری** شامل اطلاعاتی درباره‌ی حق نسخه‌برداری فایل کد منبع هستند. معمولاً در ابتدای فایل مشاهده می‌شوند.



شکل (۱) : همزمانی اسامی در توابعی که نامشان با get آغاز می‌شود [۷]



شکل (۲) : همزمانی اسامی برای توابعی که نامشان با get آغاز می‌شود [۷]

۲. **کامنت‌های سرتیتر** دیدی کلی نسبت به عملکرد کلاس و همچنین اطلاعاتی مانند نویسنده، شماره‌ی تجدید نظر و غیره می‌دهند. در فایل جاوا اینگونه کامنت‌ها معمولاً بعد از گنجاندن کتابخانه‌ها و قبل از تعریف تابع دیده می‌شوند.

۳. **کامنت‌های عضو** عملکرد یک عضو (متود یا صفت) را شرح می‌دهند و در همان خط تعریف عضو و یا خط قبلی‌اش نوشته می‌شوند. این کامنت‌ها اطلاعاتی برای توسعه‌دهنده و یا رابط برنامه‌نویسی برنامه[۱۴] مهیا می‌کنند.

٤. **کامنت‌های خطی** تصمیمات مربوط به پیاده‌سازی را درون یک متود توضیح می‌دهد.

٥. **کامنت‌های بخش** به متودها یا صفت‌هایی اشاره می‌کنند که به جنبه‌ی عملکردی مشترکی تعلق دارند.

٦. **کامنت‌های کد** شامل قسمت‌هایی از کد می‌شوند که کامنت شده‌اند و کامپایلر از آن‌ها چشم‌پوشی می‌کند. معمولاً کد برای استفاده‌های آتی و یا اشکال‌زدایی کامنت می‌شود.

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

شکل (۳) رابطه‌ی میان درک‌پذیری کد و اثرات مثبت و منفی انواع کامنت‌ها بر روی آن را نشان می‌دهد.

مزایای ارزیابی نقش کامنت‌ها در کیفیت کد منبع به شرح زیر است:

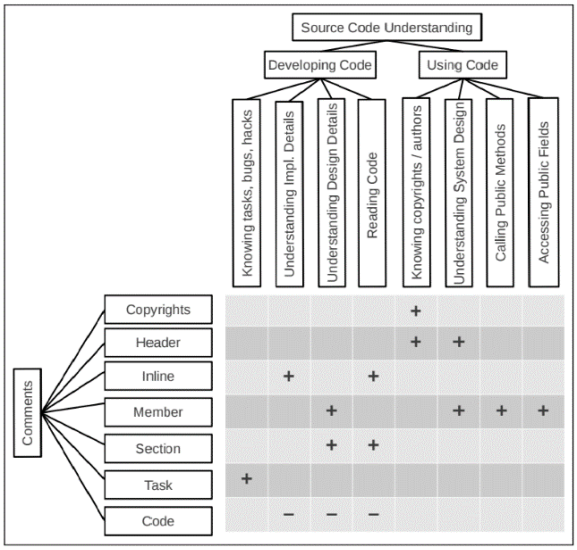
۱. انتخاب نوع کامنت متناسب با آثار مورد نظر بر روی درک‌پذیری کد.

۲. عدم استفاده از کامنت‌های کد به دلیل آثار منفی‌شان بر روی جنبه‌هایی از درک‌پذیری نرم‌افزار.

۳. طبقه‌بندی معیارهای درک‌پذیری کد منبع.

معایب ارزیابی نقش کامنت‌ها در کیفیت کد منبع به شرح زیر است:

۱. عدم ارائه‌ی چهارچوبی ساختارمند برای ایجاد کامنت‌های مناسب.



شکل (۳) : رابطه‌ی میان درک‌پذیری کد و اثرات انواع کامنت‌ها بر روی کد [8]

۲-۲-۲- رابطه نام‌گذاری شناسه‌ها و کیفیت کد منبع

در مقاله لوسیا (۲۰۱۱) به نقش مهم انتخاب شناسه‌های کد منبع در کیفیت نرم‌افزار اشاره شده است [۱۱]. اسنید (۱۹۹۶) 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 [۱۲]. تحقیقات مارکوس و هایدوک (۲۰۰۸) که چندین سیستم نرم‌افزاری متن‌باز را مورد مطالعه قرار می‌دهد، بیان می‌کند حدود ۴۰ درصد کلمات تخصصی حوزه مربوط به موضوع سیستم‌ها در کد منبع نیز تکرار شده‌اند [۱۳].

تحقیقات بسیاری درباره تحلیل کیفیت کد منبع صورت گرفته است. لاوری (۲۰۰۷) کیفیت شناسه‌ها را در بیش از پنجاه میلیون خط کد از ۴ زبان برنامه‌نویسی مختلف را که در یک بازه‌ی زمانی ۳۰ ساله نوشته شده‌اند بررسی کرده است [۱۴]. این خطوط شامل کد منبع‌های متن‌باز و اختصاصی می‌شود. در این تحقیق ارزیابی کیفیت شناسه‌ها با پیدا کردن آنها به کمک فرهنگ‌لغات و یا اختصارات شناخته‌شده انجام شده است. نتیجه تحقیقات لاوری (۲۰۰۷) نشان می‌دهد روش‌های جدید برنامه‌نویسی شناسه‌های باکیفیت‌تری تولید می‌کند. همچنین نشان داد عبارات اختصاری در کدهای اختصاصی، بیشتر از متن‌باز استفاده می‌شوند.

باتلر (۲۰۰۹) کیفیت شناسه‌های به‌کاررفته در ۸ نرم‌افزار متن‌باز نوشته‌شده به زبان جاوا را با استفاده از ۱۲ قاعده‌ی نام‌گذاری بررسی کرده است [۱۵]. در این تحقیقات رابطه‌ی بارزی 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 خطاتیره‌های متوالی نباید در کنار هم استفاده شود.

۳. اسامی شناسه‌ها باید از کلماتی تشکیل شوند که در فرهنگ‌لغات یافت می‌شود. همچنین باید از کلمات مخففی استفاده شود که شناخته‌شده هستند.

٤. اسامی شناسه‌ها باید حداکثر از ۴ کلمه یا مخفف تشکیل شده باشد.

٥. در صورت عدم وجود دلایل متقاعدکننده، ثابت‌های شمارش[۱۵] باید به ترتیب حروف اظهار شوند.

٦. شناسه‌ها نباید در ابتدا و انتهای نامشان دارای خطاتیره باشند.

۷. اطلاعات نوع داده‌ی شناسه نباید با استفاده از نماد مجارستانی[۱۶] 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 ۴ 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 منبع نرم‌افزارهای مربوطه را تحلیل کرد و تعداد اخطارهایی با اولویت یک و اولویت دو به دست آورد. شکل (۴) رابطه‌ی بین اشتباه در نام‌گذاری شناسه‌ها و اخطارهای FindBugs را در ۸ نرم‌افزار مربوطه نشان می‌دهد. شکل (۴) نشان می‌دهد در آن دسته از کلاس‌هایی که اخطارهای دارای اولویت دو به دست آمده است، احتمال وجود مصادیق نقض ۱۲ قاعده‌ی نام‌گذاری مطرح شده بیشتر است. همچنین برخی از این مصادیق رابطه مشهودتری با اخطارها دارند. برای مثال عدم رعایت استفاده‌ی 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.

مزایای بررسی رابطه‌ی نام‌گذاری شناسه‌ها و کیفیت کد منبع:

۱. طبقه‌بندی قواعد نامگذاری با توجه به تعدد و آثارشان بر روی کد منبع.
۲. پیروی از شیوه‌های برنامه‌نویسی‌ای که ۱۲ قاعده‌ی نامگذاری این تحقیقات را رعایت می‌کنند.
۳. استفاده از ابزار FindBugs برای پیدا کردن کد منبع‌هایی که احتمال نقض قوانین ۱۲ گانه‌ی نامگذاری در آن‌ها بالاست.

معایب بررسی رابطه نام‌گذاری شناسه‌ها و کیفیت کد منبع:

۱. روابطی که در این تحقیق بیان‌شده‌اند بر اساس احتمالات‌اند و نمی‌توان بصورت کامل آن‌ها را مبنای کارهای دیگر قرار داد.
۲. عدم وجود رابطه‌ی منطقی میان اخطارهای دارای اولویت یک و عدم ارائه‌ی توضیح منطقی درباره‌ی آن.

۲-۳- 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 منبع می‌توان از تکنیک‌های حوزه‌ی پردازش زبان‌های طبیعی بهره گرفت.

۲-۳-۱- آشکارساز ضدالگوی زبانشناختی (LAPD)

آرنودوا (۲۰۱۴) برای تشخیص ضدالگوهای زبانشناختی ابزاری به نام LAPD معرفی می‌کند [۱۶]. 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 کامنت‌ها و اجزای کد، ارتباط بین آن‌ها را کشف و بررسی می‌کند. همچنین برای بررسی معنی‌دار بودن کلمات و ارتباط بین آن‌ها از پایگاه‌داده‌ی واژگان وردنت[۱۷] استفاده می‌کند. آرنودوا کارایی این ابزار را با بررسی کد منبع چند نرم‌افزار ارزیابی کرده‌است. جدول (۳) برای هر یک از نرم‌افزارهای مورد بررسی تعداد ضدالگوهای تشخیص داده شده و آنچه در عمل وجود دارد را بیان می‌کند. همچنین دقت الگوریتم‌های ارائه‌شده نیز در جدول (۳) مشهود است.

مزایای روش پیشنهادی آرنودوا به شرح زیر است:

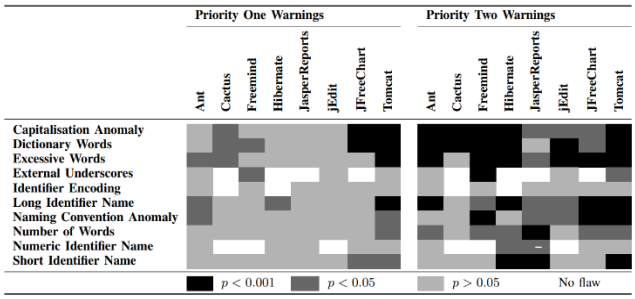
۱. در ابزار LAPD برای تمام ضدالگوهای زبانشناختی الگوریتمی برای آشکارسازی ارائه شده است.
۲. پایگاه داده‌ی وردنت ۲۰۰ زبان را پشتیبانی می‌کند و فقط به زبان انگلیسی متکی نیست.
۳. با توجه به جدول (۳) این روش در اکثر موارد از دقت بالایی برخوردار است.

معایب روش پیشنهادی آرنودوا به شرح زیر است:

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

۲-۳-۲- استفاده از یادگیری ماشین

لیو (۲۰۱۹) روشی مبتنی بر یادگیری ماشین و شبکه‌های عصبی پیچشی[۱۸] ارائه می‌دهد و در آن علاوه بر کشف ناسازگاری‌ها در نامگذاری توابع، پیشنهادهایی نیز برای رفع آن‌ها به توسعه‌دهنده می‌دهد [۱۷]. به طور خلاصه، در روش لیو توابع با توجه به بدنه‌ی آن‌ها طبقه‌بندی می‌شوند و در



شکل (٤) : رابطه‌ی اخطارهای FindBugs و قواعد نامگذاری [15]

جدول (۳) : ضدالگوهای تشخیص داده شده توسط LAPD [16]

		ArgoUML 0.10.1	*ArgoUML 0.34*	*Cocoon 2.2.0*	*Eclipse 1.0*	Validated	TP	Precision
A.1	*"Get" - more than an accessor*	0	2	1	15	15/18	9	60%
A.2	*"Is" returns more than a Boolean*	2	0	4	26	24/32	24	100%
A.3	*"Set" method returns*	4	30	6	53	47/93	46	98%
A.4	*Expecting but not getting a single instance*	7	3	8	33	34/51	26	77%
B.1	*Not implemented condition*	20	28	43	232	74/323	58	78 %
B.2	*Validation method does not confirm*	1	8	11	235	70/255	52	74%
B.3	*"Get" method does not return*	1	3	2	57	38/63	37	97%
B.4	*Not answered question*	0	2	0	34	26/36	26	100%
B.5	*Transform method does not return*	0	86	15	44	58/145	57	98%
B.6	*Expecting but not getting a collection*	8	39	12	135	64/194	47	73%
C.1	*Method name and return type are opposite*	0	0	0	6	6/6	3	50%
C.2	*Method signature and comment are opposite*	7	20	12	243	72/282	6	8%
D.1	*Says one but contains many*	15	92	42	103	70/252	40	57%
D.2	*Name suggests Boolean but type does not*	14	13	21	138	64/186	36	56%
E.1	*Says many but contains one*	45	117	24	116	73/302	55	75%
F.1	*Attribute name and type are opposite*	1	0	0	0	1/1	1	100%
F.2	*Attribute signature and comment are opposite*	1	0	3	19	19/23	1	5%

نتیجه متودهایی 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 (از نظر عملکر) برای آن تابع نامی پیشنهاد داد. لیو بیان می‌کند برخی از شبکه‌های عصبی می‌توانند معانی پیاده‌سازی توابع را درک و آن‌ها را دسته‌بندی کنند. این روش از سه گام اصلی پردازش داده، آموزش، تشخیص و ارائه‌ی پیشنهاد تشکیل شده است. در این روش منظور از ناسازگاری، عدم همانگی نام تابع با عملکرد آن است.

در گام **پردازش داده** نام توابع و بدنه‌ی آن‌ها نشانه‌گذاری می‌شود. در نشانه‌گذاری نام توابع، نام‌ها که خودشان نشانه محسوب می‌شوند به زیرنشانه‌های تشکیل‌دهنده‌ی خود تبدیل می‌شوند. در نشانه‌گذاری بدنه‌ی توابع از درخت نحو انتزاعی استفاده می‌شود.

در گام **آموزش** نشانه‌های به دست آمده از مرحله‌ی قبل ابتدا تبدیل به بردارهای عددی می‌شوند. زیرا برای خوراندن نشانه‌ها به شبکه‌های عصبی پیچشی باید آن‌ها را به صورت بردارهای عددی درآورده باشیم. نشانه‌های نام توابع با استفاده از روش بردار پاراگراف به بردارهای عددی تبدیل می‌شوند. زیرا هر نام را می‌توان مانند پاراگرافی دانست که تابع را توصیف می‌کند. نشانه‌های بدنه‌ی یک تابع ابتدا با استفاده از روش Word2Vec به بردار تبدیل می‌شوند. سپس تمام بردارهای بدنه‌ی تابع به شبکه‌ی عصبی پیچشی داده می‌شوند تا تبدیل به یک بردار عددی واحد برای بدنه‌ی تابع شوند. سپس بردارهای نام و بدنه‌ی توابع به صورت جداگانه طبقه‌بندی می‌شوند.

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

جدول (۴) نتایج ارزیابی تشخیص ناسازگاری‌ها را نشان می‌دهد. جدول تشخیص توابع دارا یا بدون ناسازگاری را برای مقادیر مختلف k 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 ۵ تابعی که بیشترین شباهت را دارند استفاده شده است.

مزایای روش ارائه شده توسط لیو:

۱. برقراری روشی برای بررسی رابطه میان نام و عملکرد تابع.
۲. روش پیشنهادی توابع را دسته‌بندی می‌کند و از این امر می‌توان برای بررسی دیگر معیارهای مؤثر بر روی کیفیت کد منبع استفاده کرد. به عبارت دیگر می‌توان موارد استفاده از این روش را گسترش داد.
۳. برخلاف بسیاری از تحقیقات صورت گرفته در زمینه‌ی کیفیت کد، لیو علاوه بر آشکارسازی، به چگونگی رفع ناسازگاری‌ها نیز می‌پردازد.

معایب روش ارائه شده توسط لیو:

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.

جدول (٤): نتیج ارزیابی تشخیص ناسازگاری‌ها [17]

EVALUATION RESULTS OF INCONSISTENCY IDENTIFICATION.

	Evaluation metrics	k = 1	k = 5	k = 10	k = 20	k = 30	k = 40
Inconsistent	Precision (%)	**56.8**	53.7	53.3	53.3	49.9	49.7
	Recall (%)	**84.5**	55.9	46.7	46.7	28.8	33.6
	F1-measure (%)	**67.9**	54.8	49.7	49.7	36.5	40.1
Consistent	Precision (%)	**72.0**	55.9	54.2	54.2	51.4	51.4
	Recall (%)	38.2	53.7	60.7	60.7	**72.2**	67.4
	F1-measure (%)	49.9	54.8	57.3	57.3	**60.0**	58.3
Accuracy (%)		**60.9**	54.8	53.8	50.8	50.9	51.1

۲. این روش ابتدا داده‌ها (کد منبع) 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 به عنوان پیشنهادی برای رفع ناسازگاری در یک تابع دیگر پیشنهاد شود.

۳- ضدالگوهای زبانشناختی مورد مطالعه

تمامی ضدالگوهای زبانشناختی می‌توانند بر جنبه‌هایی از کیفیت کد تأثیر بگذارند. مطمئناً مطالعه‌ی تمام آن‌ها و ارائه‌ی راه‌حل برای رفع آن‌ها از حوصله‌ی این مقاله خارج است. بدین جهت در این مقاله به مطالعه‌ی آن دسته از ضدالگوهایی می‌پردازیم که راه‌حل ارائه‌شده برای آن‌ها، می‌تواند برای رسیدن به راه‌حلی برای سایر ضدالگوهای زبانشناختی نیز تعمیم داده شود. تمامی ضدالگوهای مورد بحث مرتبط به توابع هستند. همچنین سعی شده نتایج بروز چنین ضدالگوهایی نیز ارائه شود.

۳-۱- بیشتر از آنچه انجام می‌دهد بیان می‌کند

گاهی نام تابع می‌تواند گمراه‌کننده باشد. این گمراهی ممکن است تا جایی پیش رود که خواننده‌ی کد منبع را قانع کند که تابع مورد بحث، عملکرد یا نتایجی دارد که در عمل چنین نیست. در واقع نام تابع عملکردی از تابع ارائه داده است که در واقعیت وجود ندارد یا کاستی دارد. بدین جهت در مقاله آرنودو (۲۰۱۳) که مبنای بسیاری مطالعات در این حوزه است، به چنین ضدالگوهایی «بیشتر از آنچه انجام می‌دهد بیان می‌کند» اطلاق شده است [۱۸].

۳-۱-۱- تابع پیاده‌سازی‌نشده

چنین ضدالگویی هنگامی رخ می‌دهد که تابع صرفاً تعریف شده است اما بدنه آن خالی مانده است. علت وقوع چنین ضدالگویی می‌تواند این باشد که توسعه‌دهندگان قبلی نرم‌افزار، طرحی برای توسعه‌ی برنامه داشته‌اند و با تعریف تابع (و عدم توسعه‌ی آن و در نتیجه خالی ماندن بدنه‌اش) قصد در بیان آنچه در آینده صورت می‌پذیرد داشته‌اند. اگر توسعه‌دهنده‌ی جدید با آگاهی از این که تابعی وجود دارد، آن را فراخوانی کند، در صورتی که تابع پیاده‌سازی نشده باشد، نتایج مورد نظر کاربر اعمال نمی‌شود و این امر احتمال بروز خطا در سیستم را بالا می‌برد. همچنین در صورت تست کردن، بخصوص تست کردن از نوع جعبه سیاه، تست‌کننده ممکن است با صرف وقت و تلاش اقدام

به طراحی تست برای چندین سناریو نماید، در صورتی که انجام تست برای تابعی با بدنه خالی نیاز نیست، و یا می‌توان آن را با یک سناریو پوشش داد. در برخی از زبان‌های برنامه‌نویسی وجود توابع گسترش می‌تواند باعث بروز توابعی با بدنه‌ی خالی شود [19]. این مبحث در این مقاله مطالعه نمی‌شود.

۳-۱-۲- تابع «Get» چیزی برنمی‌گرداند

توابع برگرداننده (یا توابع دهنده[۱۹]) که معمولاً نام آن‌ها با Get آغاز می‌شود این انتظار را ایجاد می‌کنند که در صورت فراخوانی، مقداری را برمی‌گردانند [18]. معمولاً از این توابع برای دسترسی (یا پایش دسترسی) به صفت‌های یک کلاس استفاده می‌شود. حال اگر چنین تابعی چیزی به عنوان خروجی نتیجه ندهد، بیانگر وجود یک ضدالگوی زبانشناختی است.

۳-۲- بیشتر از آنچه می‌گوید انجام می‌دهد

عدم تهیهی مستندات و یا استفاده از واژگان نامناسب می‌تواند خواننده کد منبع را از شناخت و یا درک صحیح برخی از ویژگی‌های کد منبع محروم کند. این ویژگی‌ها ممکن است برای استفاده بهینه از سیستم مربوطه حیاتی باشند و یا در صورت عدم استفاده، عملکرد سیستم مختل شود. بدین جهت در مقاله آرنودوا (۲۰۱۳) که مبنای بسیاری از مطالعات در این حوزه است، به چنین ضدالگوهایی «بیشتر از آنچه می‌گوید انجام می‌دهد» اطلاق شده است [۱۸].

۳-۲-۱- 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 [۲۰]. در صورت وقوع چنین شرایطی، ضدالگوی «تابع برگرداننده‌ای که بیشتر از یک راه دسترسی است» رخ می‌دهد.

۳-۲-۲- تابع «is» چیزی بیشتر از نوع داده بولی برمی‌گرداند

وقتی نام تابعی با عبارت is آغاز می‌شود، انتظار می‌رود نوع داده‌ی برگشتی آن از نوع بولی باشد. یعنی یکی از دو مقدار صحیح و غلط را برگرداند. در غیر این صورت تابع دارای ضدالگوی ذکر شده است.

۳-۲-۳- تابع «Set» مقدار بازگشتی دارد

توابع تغییردهنده (مانند توابع تخصیص‌دهنده[۲۰]) امکان اختصاص دادن مقداری را به صفات کلاس (که معمولا محافظت‌شده یا خصوصی هستند و در نتیجه نمی‌توان آنها را از خارج مورد دسترسی قرار داد) فراهم می‌کنند. طبق رسوم، توابع تخصیص‌دهنده مقداری را برنمی‌گردانند. بطور کلی، این توضیح برای توابعی که نامشان با set آغاز می‌شود، صحیح است. همچنین تابع تخصیص‌دهنده‌ای که چیزی برمی‌گرداند باید با داشتن یک نام مناسب (که دیگر با set آغاز نشود) این مسئله را شفاف کند [۱۸].

٤- طرح پیشنهادی

بازسازی کد منبع شامل دو مرحله‌ی اصلی تشخیص و رفع است. مطالعات صورت‌گرفته در این زمینه نیز گاهی یک، ویا هر دو مرحله را پوشش می‌دهند [۲۱]. در ادامه به ارائه‌ی طرح پیشنهادی جهت پیاده‌سازی این دو مفهوم در این مقاله می‌پردازیم.

۴-۱- تشکیل درخت نحو انتزاعی

بسیاری از مطالعات صورت‌گرفته بر روی ضدالگوها، از درخت نحو انتزاعی سود می‌برند. هر گره این درخت نشان‌دهنده‌ی یک ساختار در متن است [۲۲]. این درخت نه تنها کد منبع را به شکل مناسب و قابل‌درکی تجزیه می‌کند، بلکه با ارائه‌ی اطلاعاتی برای هر گره، کار تحلیل کد منبع را آسان می‌کند. به تهیه‌ی درخت نحو انتزاعی از کد منبع تجزیه گفته می‌شود [۲۳]. وظیفه‌ی تجزیه‌گر شامل استفاده از تحلیلگر واژه و قرار دادن خروجی‌های آن در کنار هم برای ساخت درخت می‌شود. ابزار استفاده‌شده برای تهیه درخت نحو انتزاعی در این مقاله، برای تمامی قسمت‌های ممکن در کد منبع جاوا کلاس‌هایی دارد. همچنین امکان از صافی ردکردن تمامی گره‌ها برای به دست آوردن گره‌هایی از یک کلاس خاص مهیا شده است.

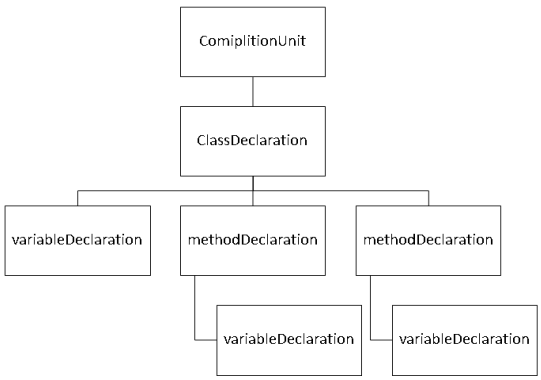
به هر یک از گره‌های یک درخت نحو انتزاعی یک نشانه[۲۱] نیز گفته می‌شود. یک نشانه در نرم‌افزار دارای نوع داده‌ی مخصوص به خود است. نشانه‌های قسمت‌های مختلف کد می‌توانند نوع داده‌ی متفاوتی داشته باشند (که در تمایز دادن آن‌ها استفاده خواهد شد).

۴-۲- دسته‌بندی نشانه‌ها

گرچه این مقاله در رابطه با ضدالگوهای توابع است، اما اطلاعاتی که در فرایند تشخیص و رفع آنها نیازداریم محدود به توابع نمی‌شود. رفع ضدالگوهای مربوط به توابع بدون درنظر گرفتن سایر جنبه‌های کد منبع می‌تواند باعث تشکیل ضدالگوهای دیگر شود. گره‌های درخت نحو انتزاعی دارای کلاس مربوط به خود هستند. ما می‌توانیم تمام گره‌ها را از صافی[۲۲] رد کنیم تا تنها گره‌های مربوط به یک کلاس خاص باقی بمانند.

اولین و پرکاربرد ترین صافی، جداکننده‌ی گره‌های مربوط به توابع است. شکل (۵) گره‌های اصلی درخت نحو انتزاعی یک کد منبع 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 (۵) سه



شکل (۵) : نمایش گره‌های اصلی درخت نحو انتزاعی یک کد منبع ساده

گره به دست می‌آید. در روش پیشنهادی توابعی برای از صافی گذراندن‌های ذکرشده، تعریف شده‌اند. دلیل اهمیت یافتن نشانه‌های مربوط به تعریف متغیرها این است که در مرحله‌ی 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 کد منبع توجه کنیم.

٤-٣- تشخیص و دسته‌بندی توابع نیازمند بازسازی

در بخش قبل نشانه‌ها را دسته‌بندی کردیم. حال نشانه‌های مربوط به توابع را برای یافتن ضدالگوهای مورد بحث مطالعه می‌کنیم. در صورت تشخیص وجود ضدالگو در یک تابع، یک شئ متناظر با آن تابع از کلاس مربوط به ضدالگوی تشخیص‌داده‌شده ایجاد می‌کنیم. به کمک این شئ، ضدالگو را ذخیره می‌کنیم و از صفات آن در فرایند اصلاح کد استفاده می‌کنیم. یکی از صفات تمام کلاس‌های ضدالگوها، مربوط به نشانه ایست که ضدالگوی مربوطه در آن رخ داده است. به تعداد ضدالگوهای مورد مطالعه، کلاس برای ضدالگوها داریم. در شکل (۶) فرایند تشخیص و ذخیره ضدالگوها با دیدی کلی نشان داده شده است. همچین در بررسی نام توابع از روش تقسیم‌بندی توابع برای تجزیه نام توابع به کلمات بامعنی تشکیل‌دهنده‌شان استفاده می‌کنیم.

٤-٤- اصلاح کد منبع

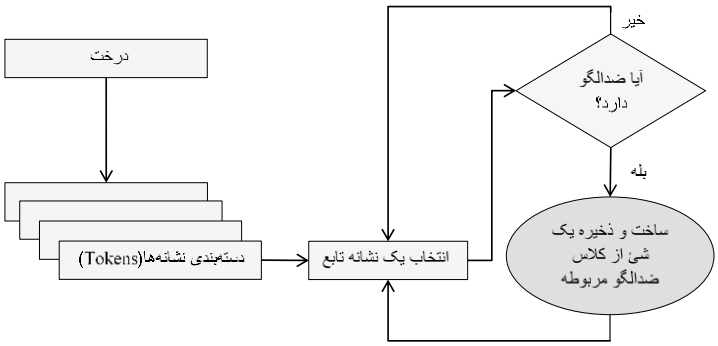
روش اصلاح هر یک از ضدالگوهای مطرح‌شده متفاوت است اما به طور کلی از دو روش تغییر نام و انتقال بخشی از بدنه‌ی تابع به تابعی دیگر (و یا ترکیب این دو روش) در این مقاله استفاده شده است. البته تغییر نام عضوی از کد منبع باید با شرایط خاصی انجام پذیرد تا شاهد نام‌های تکراری در کد منبع نباشیم. به عنوان مثال برای رفع ضدالگوی «تابع «Get» بیشتر از یک راه دسترسی است» تابع جدیدی می‌سازیم و قسمت‌های اضافی بدنه‌ی تابع دارای ضدالگو را به تابع جدید منتقل می‌کنیم. سپس تابع جدید را درون تابع اولیه فراخوانی می‌کنیم. در واقع یک تابع جدید با نام تابع اولیه می‌سازیم که در آن تابع اولیه فراخوانی می‌شود. اما نام تابع اولیه را تغییر می‌دهیم تا دیگر دارای این ضدالگو نباشد. شکل (۷) مراحل تشخیص و رفع این ضدالگو را به صورت شبه‌کد نشان می‌دهد.

همانطور که از خطوط ۱۵ تا ۱۸ شبه‌کد مشخص است، اگر نام جدیدی که برای تابع انتخاب می‌کنیم در کد تکراری باشد، به انتهای آن یک پسوند عددی اضافه می‌کنیم تا وقتی که که یک نام یکتا نیابیم آن پسوند را یک واحد افزایش می‌دهیم. همچنین پس از آن که در خط ۱ درخت نحو انتزاعی کد ساخته می‌شود، در خط ۳ با فیلتر کردن گره‌های آن، تمام گره‌های مربوط به تعریف توابع به دست می‌آیند.

٤-٥- ارزیابی طرح پیشنهادی

با استفاده از کد منبع چند نرم‌افزار منبع‌باز، به ارزیابی طرح پیشنهادی در مواجهه با سناریوهای واقعی پرداختیم. روش پیشنهادی توانست ۶۵ درصد از ضدالگوهای زبانشناختی موجود در کدهای منبع را بیابد و رفع کند. دلیل آن که نتوانست ۳۵ درصد ضدالگوهای زبانشناختی را تشخیص دهد، ناکارآمدی روش تقسیم‌بندی کلمات در تجزیه نام توابع بود. شکل (۸) و شکل (۹) تعداد ضدالگوهای تشخیص‌داده شده و ضدالگوهای موجود در برخی از کد منبع‌های دو نرم‌افزار آرگو یوام‌ال و بلینک را نشان می‌دهد.

همچنین تأثیر اصلاح هر نوع از ضدالگوهای بحث‌شده بر روی تعداد خطوط کد منبع مورد بررسی قرار گرفت. برای مثال به ازای رفع یک تابع دارای ضدالگوی «تابع «Is» چیزی بیشتر از نوع داده بولی برمی‌گرداند» طبق فرمول (۱) زیر هفت و نیم خط به کد منبع اضافه می‌شود:

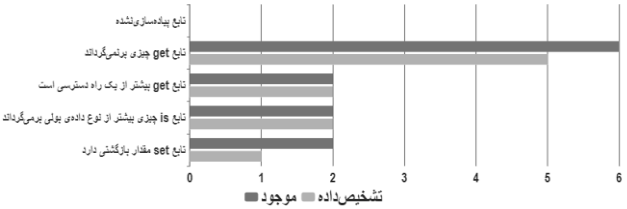


شکل (٦) : فرایند تشخیص و ذخیره ضدالگوها

```
1  AST = Java_to_AST(sourceCodeDirectory)
2  codeLines = java_to_lines(sourceCodeDirectoy)
3  methodNodes = AST.filter(type=methods)
4
5  for m in methodNodes:
6      if m.name.lower_case().starts_with("get"):
7          getMethods.append(m)
8
9  for g in getMethods:
10     if g.body.length > 1:
11         newMethodNameBase = "calculateFor_" + g.name
12         newMethodName = newMethodNameBase
13         index = 1
14
15         for i in methodNodes:
16             if i.name == newMethodName:
17                 newMethodName = newMethodNameBase + int_to_str(index)
18                 index++
19
20         newMethod = create_java_method(name = newMethodName,
21                                        modifiers = g.modifiers,
22                                        return_type = g.return_type,
23                                        body = g.body
24                                        inputs = g.inputs)
25         newMethod.insert_to_code(codeLines, position="end_of_class")
26
27         g.body = "return" + newMethod.name + "(" + str(newMethod.inputs.without_type) + ");"
28         g.save(codeLines)
29 lines_to_java(codeLines, sourceCodeDirectoy)
```

شکل (۷) : الگوریتم تشخیص و رفع ضدالگوی «تابع «Get» بیشتر از یک راه دسترسی است»

شکل (۸) : مقایسه تعداد تشخیصی و موجود ضدالگوها در نرم‌افزار آرگو یوام‌ال



شکل (۹) : مقایسه تعداد تشخیصی و موجود ضدالگوها در نرم‌افزار بلینک

$$\bar{L} = \left(\frac{1}{n}\right)\sum_{i=1}^{n} ۱ + B_i = ۷.۵ \quad (۱)$$

که در آن $\bar{L}$ میانگین خطوط اضافه شده به کد با رفع ضدالگو، n تعداد سناریوهای مختلف برای این ضدالگو و $۱ + B_i$ تعداد خطوط اضافه شده به کد به ازای هر سناریو است. وجود عدد ۱ در $۱ + B_i$ به دلیل نیاز به وجود حداقل یک خط برای تعریف یک تابع جدید در زبان جاوا است. زیرا در روش پیشنهادی این مقاله برای رفع این ضدالگو یک تابع جدید به کد منبع اضافه می‌شود. B_i نیز تعداد خطوط بدنه‌ی تابع جدید است. مزایا و معایب روش پیشنهادی به شرح زیر است.

مزایا:

۱. با تشکیل درخت نحو انتزاعی اطلاعاتی فراوانی بدست می‌آید. این اطلاعات می‌تواند در توسعه این پایان‌نامه در آینده مفید باشد.
۲. نیاز به بررسی و اعتبار سنجی بازسازی 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 ضدالگو داشته باشد، برای هر ضدالگو مستقلاً عمل شده و رفع می‌گردد.
٥. با از میان بردن ضدالگوهای زبان شناختی، کیفت کد منبع از جهت تمیز بودن نیز بالا می‌رود.
٦. به دلیل وجود کلاس‌های مختلف برای هر ضدالگو، می‌توان اشیأ ساخته‌شده به جهت تشخیص ضدالگوها را ذخیره و از عملیات صورت‌گرفته مستنداتی تهیه کرد.

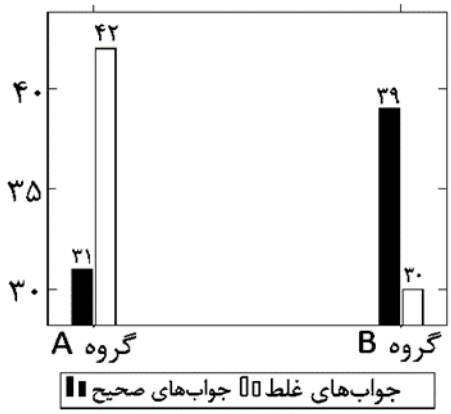
معایب:

۱. با رفع ضدالگوها در نتیجه‌ی تغییر کد منبع، باید پس از رفع هر ضدالگو درخت نحو انتزاعی جدیدی ساخت.
۲. به دلیل پرداختن به هر ضدالگو به صورت مستقل، بار اضافه پردازشی به وجود می‌آید و بسیاری از فرایندها چندین بار اجرا می‌شوند.
۳. از آنجایی که کد منبع دارای پسوند java. است، باید پس از هر تغییر در رشته‌ی متنی کد منبع، آن را ذخیره نمود و برای تغییرات بعدی دوباره این اتفاق خواهد افتاد. یعنی بار پردازشی حلقه‌ی تبدیل فایل جاوا به رشته‌ی متنی و بالعکس بالاست.
۴. به دلیل اضافه کردن پسوند یا پیشوند به نام توابع در راه‌حل رفع برخی ضدالگوها، نام آنها طولانی‌تر می‌شود که از جهت کیفیت کد منبع، یک تأثیر منفیست.
۵. درباره ضدالگو «تابع «Get» بیشتر از یک راه دسترسی است»، اگر در بدترین حالت، تمام توابع دارای این ضدالگو باشند، پس رفع ضدالگوها تعداد توابع کد منبع دو برابر می‌شود (زیرا به ازای هر تابع یکی دیگر نیز ساخته می‌شود).
۶. درخت نحو انتزاعی استفاده‌شده در این پایان‌نامه قابلیت تشخیص کامنت‌ها را ندارد. به همین دلیل زیرساخت مناسب برای مطالعه‌ی ضدالگوهای زبانشناختی مرتبط با کامنت‌گذاری را فراهم نمی‌کند.
۷. از آنجایی که برای رفع اکثر ضدالگوهای مطرح‌شده، به کد منبع خطوطی اضافه و کم می‌شود، خطوط اضافه‌شده داره کامنت‌گذاری نیستند و ممکن است برای توسعه‌دهنده، درک تغییراتی که پس از بازسازی کد اعمال شده‌اند سخت باشد.

۵- نتایج به کارگیری روش پیشنهادی

با ارزیابی روش پیشنهادی نواقصی در فرایند تشخیص آن آشکار شد. همچنین مهم‌ترین اثر اصلاح ضدالگوهای کد منبع، بهبود درک‌پذیری آن است. بدین جهت در این بخش به این دو مسئله پرداخته می‌شود. تحقیقات کرمانسراوی (۲۰۱۹) در پاسخ به این پرسش که «آیا ضدالگوهای زبانشناختی می‌توانند بر درک توسعه‌دهندگان از کد تاثیر گذارند؟» آزمایشی ترتیب داده است. در این آزمایش تابعی با ضدالگو «تابع «Is» چیزی بیشتر از نوع داده بولی بازمی‌گرداند» و تابعی که بازسازی‌شده‌ی آن است به ترتیب به گروه A و B داده شد [۲۴]. سپس از هرکدام از اعضای هر دو گروه سوالاتی درباره چگونگی کارکرد تابع پرسیده شد. نتایج شکل (۱۰) نشان می‌دهد درک گروهی که تابع بدون ضدالگو (بازسازی شده) را مطالعه می‌کردند، از تابع مورد نظر بالاتر از گروه دیگر است، زیرا در این گروه ۳۹ درصد از شرکت کنندگان 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.

مشکلات مشاهده‌شده در تشخیص ضدالگوهای کد منبع نرم‌افزارهای ذکرشده قالباً مربوط به ناکارآمدی روش تقسیم‌بندی کلمات است. مشکلات این روش که با استفاده از کتابخانه wordsegment در پایتون انجام گرفته، ممکن است مربوط به روش پیاده‌سازی‌اش باشد و لزوماً ماهیت استفاده از این روش نادرست نباشد. همچنین با وجود این که روش جایگزین تقسیم‌بندی



شکل (۱۰): تأثیر اصلاح ضدالگوی «تابع «Is» چیزی بیشتر از نوع داده بولی بازمی‌گرداند» [۲۴]

کلمات نتایج صحیحی برای آزمایشات 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 و این مشکل برطرف شد. پس از مشاهده عدم توانایی روش تقسیم‌بندی کلمات بر روی نام‌هایی مانند setLenght، عملکرد برنامه برای نام‌هایی مانند set_lenght بررسی و صحیح ارزیابی شد. یعنی روش تقسیم‌بندی کلمات در صورتی که کلمات یک عبارت با خط تیره از یکدیگر جدا شده باشند، عملکرد بهتری دارد. آرنودوا (۲۰۱۴) در روش پیشنهادی‌اش [16]، برای تحلیل اسامی توابع از تجزیه‌گر زبان طبیعی استنفورد و پایگاه‌داده‌ی واژگان وُردنت استفاده می‌کند.

٦- نتیجه

در این مقاله روشی برای تشخیص و رفع ضدالگوهای زباشناختی موجود در کد منبع نرم‌افزار جهت افزایش کیفیت نرم‌افزار تشریح شد. اهمیت این روش در پیوند دادن مباحث نظری موجود در این حوزه به یک پیاده‌سازی عملی است. ابتدا به ایجاد درخت نحو انتزاعی توسط تحلیلگر واژه، جهت تسریع فرایند تشخیص ضدالگوهای زبانشناختی موجود در متن پرداخته شد. سپس برای انواع ضدالگوهای مورد بحث روشی جهت رفع آن‌ها و اعمال تغییرات مربوطه بر روی کد منبع ارائه شد. سپس با بررسی تأثیر روش پیشنهادی بر روی کد منبع چندین نرم‌افزار متن‌باز جهت ارزیابی اثربخشی آن، عملکرد روش پیشنهادی قابل قبول دانسته شد. از عوامل شناخته شده‌ی کاستی اثربخشی روش پیشنهادی می‌توان به کارکرد نادرست روش تقسیم‌بندی کلمات در برخی موارد اشاره کرد. از آنجایی که پژوهش‌های مرتبط اکثراً به طور جداگانه به تأثیرات ضدالگوهای زبانشناختی، عوامل ایجاد کننده، معرفی و یا رفع آن‌ها پرداخته‌اند، در این مقاله سعی شده شکاف بین این موارد پر شود. با توجه به این که روش ارائه شده برای ضدالگوهای زبانشناختی مختص به توابع است، تعمیم این روش جهت پوشش ضدالگوهای زبانشناختی مربوط به شناسه‌ها نیز امکان‌پذیر است.

مراجع

[1] 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.

[2] 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.

[3] R. C. Martin. "Clean Code: A Handbook of Agile Software Craftsmanship," *Prentice Hall*, Dec,2009.

[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] B. Latte, "Clean Code: On the Use of Practices and Tools to Produce Maintainable Code for Long-Living," in *Proceedings of the EMLS 2019: 6th Collaborative Workshop on Evolution and Maintenance of Long-Living Systems*, Stuttgart, Germany, January 18, 2019. pp. 96-99.

[6] M. Anaya. "Clean Code in Python," *Packt Publishing Ltd*, Aug, 2018.

[7] D. v. Leewen, "Comprehensible Method Names: Focusing on the Nouns," *University of Amsterdam*, 2012.

[8] 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.

[9] D. Souza, "A Study of the Documentation Essentials to Software Maintenance," in *Proceedings of the 23rd annual international confrence on Design of communication: documenting & designing for pervasive information*, New York, NY, USA, September 21-23, 2005, pp. 68-75.

[10] 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.

[11] 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.

[12] H. Snead, "Object-oriented COBOL recycling," in *Proceedings of the WCRE '96: 4rd Working Conference on Reverse Engineering*, DC, United States, November 8-10, 1996, pp. 169-178.

[13] 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.

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

[15] 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.

[16] V. Arnaoudova, "Towards Improving the Code Lexicon and its Consistency (PhD thesis)," *École Polytechnique de Montréal* ,2014.

[17] 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.

[18] 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.

[19] "Extension Methodes - C# Programming Guide | Microsoft Docs," Microsoft, URL: https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/classes-and-structs/extension-methods/, Access Date: 4 Apr 2022.

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

[21] A. Baqais and M. Alshayeb, "Automatic software refactoring: a systematic literature review," *Kluwer Academic Publishers*, vol. 28, no. 2, pp. 459-502, 2020

[22] "abstract syntax tree," wikipedia, [Online]. Available: https://wioikjod.com/rt.

[23] "Parsing - Wikipedia," Wikipedia, URL: https://en.wikipedia.org/wiki/Parsing/, Access Date: 5 Feb 2022.

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

زیرنویس‌ها

[1] Source Code
[2] Comprehensibility
[3] Maintainability
[4] Legacy Software
[5] Readability
[6] Maintenance
[7] Changeability
[8] Scalability
[9] Linguistic Anti-Patterns
[10] Refactoring
[11] Bad Smell
[12] Clean Code
[13] Co-Occurrence of Nouns
[14] Application Programming Interface (API)
[15] Enum Constants
[16] Hungarian Notation
[17] Wordnet
[18] Convolutional Neural Networks
[19] Getter Methods
[20] Setter Methods
[21] Token
[22] Filter