DOR: 20.1001.1.2538161.1403.9.4.6.1

DOI: https://doi.org/10.22034/csj.2025.214938

زمستان ۱۴۰۳، دوره ۹، شماره ۴، پیاپی ۳۵

دریافت مقاله: ۱۴۰۳/۰۹/۰۹

پذیرش مقاله: ۱۴۰۳/۱۰/۰۸

نوع مقاله: پژوهشی

اندازه‌گیری تاثیر اصلاح نابسامانی‌های کد بر مصرف انرژی و پایداری سیستم

نسرین مقصودی شقاقی

دانش‌آموخته کارشناسی ارشد نرم‌افزار، گروه مهندسی کامپیوتر، دانشکده فنی و مهندسی، دانشگاه شهرکرد، شهرکرد، ایران

پست الکترونیکی: nasrin.msh.1389@gmail.com

لیلا صمیمی دهکردی*

استادیار، گروه مهندسی کامپیوتر، دانشکده فنی و مهندسی، دانشگاه شهرکرد، شهرکرد، ایران

پست الکترونیکی: samimi@sku.ac.ir

عباس حری

استادیار، گروه مهندسی کامپیوتر، دانشکده فنی و مهندسی، دانشگاه شهرکرد، شهرکرد، ایران

پست الکترونیکی: horri@sku.ac.ir

چکیده

در سال‌های اخیر، بهینه‌سازی مصرف انرژی و ارتقای پایداری نرم‌افزار به یکی از دغدغه‌های اصلی در حوزه مهندسی نرم‌افزار تبدیل شده است. نابسامانی‌های کد، به‌عنوان عواملی که منجر به افزایش پیچیدگی، کاهش کارایی و تأثیرات منفی بر مصرف منابع می‌شوند، توجه پژوهشگران را به خود جلب کرده‌اند. این پژوهش با هدف بررسی تأثیر رفع نابسامانی‌های کد بر مصرف انرژی و زمان اجرای نرم‌افزار انجام شده است. برای این منظور، سه سوال تحقیق اصلی تدوین شده است: ۱) اصلاح نابسامانی‌ها چگونه بر مصرف انرژی تأثیر می‌گذارد؟ ۲) کدام نابسامانی‌ها بیشترین تأثیر را بر زمان اجرا دارند؟ ۳) آیا رفع نابسامانی‌ها به بهبود پایداری نرم‌افزار منجر می‌شود؟ در این راستا، هشت مطالعه موردی بر روی انواع نابسامانی‌های کد، طراحی و پیاده‌سازی شده است. در هر مطالعه، نسخه اولیه و اصلاح شده کد از نظر زمان اجرا و مصرف انرژی با استفاده از ابزارهای دقیق اندازه‌گیری و تحلیل شدند. نتایج نشان داد که رفع نابسامانی‌ها در اغلب موارد منجر به بهبود خوانایی و کاهش پیچیدگی کد شده است. در چهار نمونه به ویژه در مورد مطالعه دستور سوییچ، زمان اجرای برنامه بهینه شده کمتر از برنامه نابسامان است. همچنین، در پنج نمونه میزان مصرف انرژی کد بهینه شده تقریبا کمتر از کد دارای نابسامانی است. برای مثال، در مطالعه استفاده افراطی از نوع داده اولیه ۱۵ درصد کاهش مصرف انرژی را شاهد بوده‌ایم. با این حال، در نابسامانی‌هایی نظیر شیء خداوند و متد طولانی، به‌دلیل افزایش فراخوانی‌ها و مدیریت اشیاء، سربار پردازشی و مصرف انرژی در تکرارهای بالا افزایش یافته است.

واژه‌های کلیدی: نابسامانی کد، پایداری نرم‌افزار،

* نویسندهٔ مسئول

مصرف انرژی، بازآرایی کد، مهندسی نرم‌افزار آزمایشی.

۱. مقدمه

با گسترش استفاده از تلفن‌های هوشمند، رایانه‌های لوحی و قابل حمل و افزایش دغدغه‌های محیط‌زیستی، بهینه‌سازی مصرف انرژی در توسعه برنامه‌های کاربردی به یک ضرورت تبدیل شده است [۱]. این موضوع از دو منظر قابل توجه است: نخست، به‌منظور کاهش مصرف باتری و کاهش هزینه‌های انرژی کاربران و دوم، با هدف حرکت به‌سوی واحدهای پردازشی دوستدار محیط‌زیست. در حال حاضر، روش‌های مختلفی برای بهینه‌سازی مصرف انرژی در سطح سخت‌افزار ارائه شده‌اند، از جمله بهبود معماری پردازنده‌ها و بهینه‌سازی مصرف انرژی در مراکز داده. با این‌حال، سهم نرم‌افزار در مصرف انرژی نیز بسیار حائز اهمیت است و این مسئله به یک چالش عمده در مهندسی نرم‌افزار تبدیل شده است؛ چرا که برنامه‌ها در زمان اجرا می‌توانند به میزان زیادی بر مصرف انرژی تأثیرگذار باشند [۷].

رویکردهای متعددی در جهت ارزیابی و بهینه‌سازی مصرف انرژی نرم‌افزارها در حال توسعه هستند و تحقیقات اخیر نشان داده‌اند که برخی نابسامانی‌های کد می‌توانند منشأ نشتی انرژی باشند [۱، ٦، ۷]. این نابسامانی‌ها که می‌توانند بر نگهداری، توسعه و پایداری سیستم نرم‌افزاری نیز تأثیر منفی بگذارند، معمولاً ناشی از مشکلات طراحی هستند که در زمان اجرا باعث افزایش مصرف انرژی می‌شوند.

نابسامانی کد به مواردی اطلاق می‌شود که ساختار و سازمان‌دهی کدهای نرم‌افزاری به گونه‌ای است که فهم، تغییر و نگهداری آن‌ها دشوار می‌شود. این مشکل‌ها می‌توانند به دلایلی مانند پیچیدگی زیاد، عدم رعایت اصول کدنویسی، استفاده از نام‌گذاری‌های غیراستاندارد یا افزایش وابستگی‌ها در بخش‌های مختلف کد رخ دهند [۲، ۳]. انواع نابسامانی‌های کد شامل مواردی مانند دوباره‌کاری‌ها (مناطقی از کد که چندین بار تکرار شده‌اند)، وابستگی‌های پنهان (وابستگی‌های بین بخش‌هایی از کد که به صورت غیرمستقیم با یکدیگر در ارتباطند) و افزایش پیچیدگی (بخش‌هایی از کد که به دلیل افزایش شرط‌ها و شاخه‌ها، پیچیده و دشوار شده‌اند) می‌شود [٤].

سوالات تحقیق عبارتند از:

سوال یک: اصلاح نابسامانی‌های کد چگونه بر مصرف انرژی برنامه‌های جاوا تأثیر می‌گذارد؟

سوال دو: کدام نابسامانی‌ها بیشترین تأثیر را بر زمان اجرای برنامه‌های جاوا دارند؟

سوال سه: آیا رفع نابسامانی‌های کد می‌تواند به افزایش پایداری نرم‌افزار کمک کند؟

در این پژوهش، با تمرکز اثر مصرف انرژی در برنامه‌های نرم‌افزاری، ابتدا کدهای جاوا دارای نابسامانی‌های طراحی شناسایی شدند. هشت نوع نابسامانی در نظر گرفته شد که به‌عنوان یکی از چالش‌های اصلی مهندسی نرم‌افزار در نظر گرفته شده‌اند. برای شناسایی نابسامانی در کد، از ابزارهای مهندسی معکوس استفاده شد تا مدل‌هایی از کدهای موجود استخراج و ساختار آن‌ها تحلیل شود. این تحلیل امکان بررسی دقیق نقاط ضعف طراحی را فراهم کرد.

در مرحله بعد، اصلاحات لازم برای بر طرف کردن نابسامانی‌های شناسایی‌شده اعمال شد و نسخه‌ای بازسازی شده از کد به‌دست آمد. نسخه بازسازی شده به‌گونه‌ای طراحی شد که ضمن حفظ عملکرد اصلی برنامه، مصرف انرژی آن بهینه شود. در ادامه، فرآیند مقایسه میان نسخه اولیه و نسخه اصلاح شده انجام شد که شامل اندازه‌گیری دقیق مصرف انرژی و زمان اجرا برای نسخه شامل نابسامانی و نسخه بازسازی شده بود و تأثیر تغییرات اعمال شده ارزیابی شد. نتایج نشان داد که در چهار نمونه نابسامانی زمان اجرای برنامه بهینه شده کمتر از برنامه نابسامان است. برای مثال، در دستور سوییچ تا ۳۰ درصد کاهش زمان اجرا برای صدهزار تکرار را

شاهد بوده‌ایم. همچنین، در پنج نمونه میزان مصرف انرژی کد بهینه شده تقریبا کمتر از کد دارای نابسامانی است. برای مثال، در مطالعه استفاده افراطی از نوع دادهٔ اولیه، ۱۵ درصد کاهش مصرف انرژی مشاهده شد. فقط در دو مورد مصرف انرژی کد بازسازی شده بیشتر از کد نابسامان است.

نوآوری این مقاله در ارائه یک ارزیابی کمّی است که نشان می‌دهد رفع کدام نابسامانی‌های کد به کاهش مصرف انرژی و زمان اجرای برنامه‌های جاوا منجر می‌شود و میزان این کاهش به چه اندازه است. این موضوع درک بهتری از ارتباط میان طراحی کد و بهره‌وری انرژی برای برنامه‌نویسان ایجاد می‌کند. چنین درکی می‌تواند به توسعه سیستم‌هایی منجر شود که علاوه بر عملکرد بهینه، تأثیرات منفی کمتری بر منابع سخت‌افزاری و محیط‌زیست داشته باشند.

در ادامه، سازمان‌دهی مقاله به شرح زیر است. در بخش ۲، کارهای مرتبط بررسی می‌شود. در بخش ۳ روش پژوهش بیان می‌شود. بخش ٤ به ارزیابی و محاسبه زمان اجرا و مصرف انرژی برای نابسامانی‌ها می‌پردازد. بخش ٥ به بحث روی نتایج ارزیابی می‌پردازد. در آخر، مقاله با بخش ٦ شامل نتیجه‌گیری خاتمه می‌یابد.

۲.کارهای مرتبط

برخی مطالعات بر تحلیل و بهینه‌سازی مصرف انرژی در سیستم‌ها و نرم‌افزارها متمرکز هستند. مطالعه [۱] چارچوبی را برای اندازه‌گیری بهره‌وری انرژی و منابع نرم‌افزار تحت عنوان «مدل اندازه‌گیری نرم‌افزار سبز» معرفی می‌کند. نویسندگان به نبود اجماع و استانداردسازی در پژوهش‌های جاری در زمینه نرم‌افزارهای سبز اشاره دارند، که این مسئله مقایسه و بازتولید نتایج را دشوار کرده است. هدف از این مدل پر کردن این خلأ با ارائه رویکردی استاندارد برای اندازه‌گیری و تحلیل تأثیرات محیط‌زیستی نرم‌افزار و در نهایت، کاهش مصرف و ترویج پایداری است. این چارچوب پنج جزء اصلی را شناسایی می‌کند: شیء و اهداف اندازه‌گیری، معیارها و اندازه‌گیری‌های مورد نیاز، مدل رویه اندازه‌گیری، تنظیمات خاص برای فرآیند اندازه‌گیری و مدل ارزیابی داده‌ها. این مدل می‌تواند به توسعه‌دهندگان و پژوهشگران در تولید نرم‌افزارهای پایدارتر و کم‌مصرف‌تر کمک کند.

مطالعه [٦] روش جدیدی برای ارزیابی بهره‌وری انرژی سیستم‌های یادگیری ماشین معرفی می‌کند. این روش که توسط متخصصان صنعتی توسعه یافته، از مجموعه بارهای کاری نمایشی استفاده می‌کند تا اندازه‌گیری‌های قابل بازتولید کارایی انرژی را جمع‌آوری کند. مقاله چالش‌های استانداردسازی اندازه‌گیری توان را در میان بن‌سازه‌های سخت‌افزاری متنوع و ویژگی‌های بار کاری بررسی کرده و کاربرد این روش را در مقیاس‌های مختلف سیستم‌های یادگیری ماشین (از مرکز داده گرفته تا لبه و دستگاه‌های کوچک) توصیف می‌کند. تحلیل‌ها نشان‌دهنده بهبودهای قابل توجه در کارایی انرژی در هوش مصنوعی تولیدی است. مقاله به مسیرهای آینده پایداری هوش مصنوعی و ضرورت همکاری‌های صنعتی برای طراحی و اجرای سیستم‌های یادگیری ماشین کم‌مصرف اشاره می‌کند.

مطالعه [۷] امکان استفاده از معیارهای وزنی به‌عنوان شاخص‌های عملکرد انرژی در نرم‌افزار را بررسی می‌کند. اگرچه معیارهای ایستا می‌توانند در زمان کامپایل محاسبه شوند، اما معمولاً پیش‌بینی‌کننده‌های قابل اعتمادی برای مصرف انرژی نیستند. در این پژوهش، نویسندگان پیشنهاد می‌دهند معیارهای کامپایل زمانی با استفاده از پروفایل‌های زمان اجرا وزن‌دهی شوند تا مصرف انرژی نرم‌افزار بهتر منعکس شود. همچنین، استدلال می‌کنند که اتکای صرف به معیارهای ایستا برای تخمین عملکرد انرژی در برنامه‌های پیچیده، که مسیرهای اجرایی آن‌ها می‌تواند به‌شدت متغیر باشد، واقع‌گرایانه نیست. برای اثبات این ایده، آن‌ها معیار جفت‌شدگی بین اشیاء را بررسی کردند

و دریافتند که افزایش وزن‌دهی معیار با مصرف انرژی کمتر مرتبط است. این یافته‌ها به‌دلیل استفاده از اصول برنامه‌نویسی خوب مانند اصل وارونگی وابستگی است که گرچه جفت‌شدگی را کاهش می‌دهد اما می‌تواند تعداد کلاس‌ها و فراخوانی متدها را افزایش دهد، که به مصرف بیشتر انرژی منجر می‌شود.

برخی مطالعات به تحلیل نابسامانی‌های کد و روش‌های شناسایی یا بازسازی آن‌ها می‌پردازند. مطالعه [۵] اثر بازسازی نابسامانی‌های کد را بر مصرف منابع روی ۳۱ برنامه واقعی جاوا و پایتون بررسی می‌کند. ۱۶ نوع مختلف نابسامانی‌های کد در این پژوهش مطالعه شده است. از سه ابزار بازسازی خودکار برای جاوا و پایتون برای شناسایی و اصلاح نابسامانی‌ها استفاده شده است. نتایج نشان می‌دهد که اصلاح نابسامانی‌هایی مانند «کلاس خدا» می‌تواند تأثیر منفی بر استفاده از پردازنده و حافظه داشته باشد، در حالی که اصلاح نابسامانی‌هایی مانند «کد تکراری» منجر به بهبود بهره‌وری منابع می‌شود. نویسندگان اصولی را برای توسعه‌دهندگان پیشنهاد می‌کنند تا با استفاده از آن‌ها نابسامانی‌هایی را انتخاب کنند که بازسازی‌شان بهینه‌ترین نتیجه را از نظر بهره‌وری منابع دارد. همچنین سازوکاری مبتنی بر تحلیل رگرسیون برای پیش‌بینی تأثیر بازسازی دسته‌ای قبل از تصمیم‌گیری ارائه شده است.

مطالعه [۸] بر اهمیت اولویت‌بندی نابسامانی‌های کد در سیستم‌های شیءگرا تأکید می‌کند، به‌ویژه زمانی که با افزایش اندازه سیستم، تعداد نابسامانی‌های کد نیز افزایش می‌یابد. نویسندگان اشاره می‌کنند که پرداختن به تمامی نابسامانی‌های کد می‌تواند زمان‌بر و پرهزینه باشد و بنابراین تمرکز بر موارد بحرانی ضروری است. مقاله تأثیرات منفی نابسامانی‌های کد مانند کاهش کیفیت کد، افزایش هزینه‌های نگهداری و احتمال بیشتر بروز اشکال‌ها را بررسی می‌کند. عوامل متعددی نظیر شدت نابسامانی، تأثیر آن بر کیفیت نرم‌افزار و اهداف تیم توسعه در فرآیند اولویت‌بندی بررسی شده است. این پژوهش ابزارها، معیارها و برنامه‌های موضوعی را برای ارزیابی تأثیر نابسامانی‌ها معرفی کرده و توصیه می‌کند توسعه‌دهندگان ابزارها و معیارهای مناسب برای نیازهای خاص خود انتخاب کنند. در پایان، مقاله بر اهمیت دانش و درک توسعه‌دهندگان از برنامه‌های موضوعی و ابزارهای انتخابی تأکید دارد تا بتوانند علل اصلی نابسامانی‌ها و تأثیرات آن‌ها بر کیفیت نرم‌افزار را شناسایی کنند.

مطالعه [۹] بر تشخیص نابسامانی‌های چندگانه کد در سطح متد متمرکز است، مشکلی که به‌عنوان «تشخیص چندبرچسبی نابسامانی‌های کد» شناخته می‌شود. نویسندگان به این نکته اشاره می‌کنند که کد واقعی اغلب چندین نابسامانی کد را به‌طور هم‌زمان داراست و بنابراین طبقه‌بندی چندبرچسبی ضروری است. در این پژوهش، عملکرد الگوریتم‌های مختلف یادگیری ماشین همراه با سه روش طبقه‌بندی چندبرچسبی شامل ارتباط باینری، توان برچسب و زنجیره طبقه‌بندی ارزیابی شده است. نتایج نشان می‌دهد که مدل درخت تصمیم‌گیری، به‌ویژه در ترکیب با روش زنجیره طبقه‌بندی، بالاترین دقت را در تشخیص نابسامانی‌های چندبرچسبی دارد.

مطالعه [۱۳] مطالعه‌ای تطبیقی درباره استفاده از تکنیک‌های یادگیری ماشین برای طبقه‌بندی شدت نابسامانی‌های کد در نرم‌افزار ارائه می‌دهد. نویسندگان محدودیت‌های روش‌های سنتی شناسایی نابسامانی کد را که بر اساس آستانه‌ها یا قوانین عمل می‌کنند و اغلب منجر به مثبت‌های کاذب می‌شوند، بررسی کرده‌اند. مدلی مبتنی بر یادگیری ماشین، شامل تکنیک‌هایی نظیر رگرسیون، طبقه‌بندی چندگانه و طبقه‌بندی رتبه‌ای، برای غلبه بر این محدودیت‌ها پیشنهاد شده است. داده‌های مورد استفاده شامل معیارهای نرم‌افزاری چهار نوع نابسامانی کد (کلاس خدا، کلاس داده، زیاده‌خواهی ویژگی و متد طولانی) بوده است. نتایج نشان می‌دهد که جنگل تصادفی بهترین عملکرد را در طبقه‌بندی شدت نابسامانی‌ها دارد. همچنین تأثیر

معیارهای نرم‌افزاری مانند پیچیدگی، اندازه و جفت‌شدگی بر شدت نابسامانی‌ها بررسی شده است.

مطالعه [۱۵] تأثیر سه نابسامانی کد خاص اندروید را بر مصرف باتری برنامه‌های موبایل بررسی می‌کند. نویسندگان ۱۶ برنامه متن‌باز اندروید را تحلیل و با استفاده از ابزار "aDoctor" این نابسامانی‌ها را شناسایی و به‌صورت دستی بازسازی کرده‌اند. نتایج نشان می‌دهد که بازسازی این نابسامانی‌ها مصرف باتری را کاهش می‌دهد. مقاله همچنین مدل‌های ریاضی مبتنی بر تحلیل رگرسیون چندگانه برای پیش‌بینی مصرف باتری بر اساس متریک‌های نرم‌افزاری توسعه داده است و نشان داده که این مدل‌ها دقت بالایی دارند.

دسته دیگری از مطالعات راهکارهای بهینه‌سازی انرژی در طراحی و پیاده‌سازی کد را بررسی می‌کنند. مطالعه [۱۰] به بررسی تأثیر تکنیک‌های برنامه‌نویسی بهینه از نظر انرژی بر نگهداشت و عملکرد یک برنامه دسکتاپ جاوا پرداخته است. نویسنده تکنیک‌های مختلفی از جمله بازسازی کد، الگوهای طراحی و انتخاب ساختارهای داده بهینه را بررسی کرده است. مصرف انرژی با استفاده از ابزار Intel Power Gadget و عملکرد از طریق JMeter و SonarQube اندازه‌گیری شده است.

نتایج نشان می‌دهد که اعمال این تکنیک‌ها مصرف انرژی را تا ۴۳ درصد کاهش داده و عملکرد را ۲۲ درصد بهبود می‌بخشد. تحلیل نگهداشت نیز پیشرفت‌هایی را نشان داده، اما این پیشرفت‌ها، جزئی تلقی شده است.

مطالعه [۱۱] کارایی انرژی کتابخانه‌های مختلف ورودی/ خروجی جاوا و متدهای مرتبط را بررسی می‌کند. مطالعه، ۲۷ متد ورودی/ خروجی مختلف را در سناریوهای متفاوت شامل خواندن کامل فایل‌ها، خواندن بخش‌های فایل، جستجوی داده در فایل‌ها و نوشتن داده‌ها تحلیل می‌کند. یافته‌ها نشان می‌دهد استفاده از کلاس FileChannel در عملیات خواندن، پایین‌ترین مصرف انرژی را دارد و ۱۰ تا ۲۰ درصد کاراتر از سایر روش‌هاست. برعکس، استفاده از RandomAccessFile برای عملیات خواندن و نوشتن به‌عنوان یکی از ناکارآمدترین روش‌ها شناسایی شده است. همچنین، مقاله تأثیر اندازه میانگیر بر مصرف انرژی در عملیات میانگیر شده را بررسی کرده و بازه‌ای بهینه برای اندازه میانگیر پیشنهاد می‌دهد. یافته‌ها از طریق جایگزینی روش‌های پیش‌فرض با کاراترین روش‌ها در پروژه‌های واقعی جاوا نیز تأیید شده و صرفه‌جویی قابل‌توجهی در مصرف انرژی نشان داده است.

مطالعه [۱۴] به بررسی کارایی انرژی الگوی طراحی Visitor، یک راه‌حل نرم‌افزاری رایج برای الگوریتم‌هایی که روی اشیای نوع‌های مختلف عمل می‌کنند، می‌پردازد. آزمایش‌ها بر روی سه پیاده‌سازی مختلف این الگو شامل نسخه سنتی، نسخه بدون الگو و نسخه‌ای مبتنی بر اعزام بازتابی (بررسی نوع و تبدیل) انجام شده است. نتایج نشان می‌دهد که حذف کامل این الگو به‌طور مداوم کارایی انرژی را در جاوا و سی++ بهبود می‌بخشد. همچنین، نسخه مبتنی بر اعزام بازتابی در جاوا عملکرد بهتری نسبت به الگوی سنتی داشته است، اما در سی++ مصرف انرژی را به‌طور قابل توجهی افزایش داده است. این یافته‌ها بر اهمیت ملاحظات خاص زبان و استفاده از پیاده‌سازی‌های جایگزین در بهبود کارایی انرژی تأکید می‌کنند.

دستهٔ دیگر مطالعاتی هستند که بر ارزیابی و بهبود نگهداشت و کیفیت کد تمرکز دارند. مطالعه [۱۲] نابسامانی‌های کد و بازسازی آن‌ها را در زمینه کدهای رمزنگاری، به‌ویژه در کد SEED.c بررسی می‌کند. نویسندگان استدلال می‌کنند که اگرچه امنیت در الگوریتم‌های رمزنگاری از اهمیت بالایی برخوردار است، اما کیفیت کد، به‌ویژه خوانایی و نگهداشت، اغلب نادیده گرفته می‌شود. مقاله تأکید دارد که حتی خطاهای کوچک در کد امنیتی می‌توانند پیامدهای قابل‌توجهی داشته باشند، بنابراین کیفیت کد ضروری است. تحلیل کد SEED شامل بررسی معیارهایی مانند تعداد خطوط کد، تعداد متدها، تعداد ویژگی‌ها، پیچیدگی دورانی و سطح حداکثر

تودرتویی است. مقاله راهکارهای بازآرایی، شامل تقسیم متدهای طولانی، پیمانه‌بندی کلاس‌های بزرگ، حذف کدهای تکراری و ساده‌سازی لیست پارامترها را ارائه می‌دهد که هدف آن‌ها افزایش خوانایی، نگهداشت و در نهایت امنیت کد است.

مطالعه [۱۷] مدل جدیدی برای ارزیابی نگهداشت محصولات نرم‌افزاری ارائه می‌دهد که محدودیت‌های مدل‌های ارزیابی کیفیت موجود را برطرف می‌کند. نویسندگان معتقدند مدل‌های فعلی اغلب اهمیت موجودیت‌های خاص در یک سیستم نرم‌افزاری را در محاسبه نمرات نگهداشت در نظر نمی‌گیرند. مدل پیشنهادی با الگوریتمی مشابه PageRank، بر تعامل موجودیت‌ها با محیط اطرافشان تمرکز می‌کند و وزن‌های متفاوتی به نقص‌ها براساس اهمیت موجودیت‌های آسیب دیده اختصاص می‌دهد. این روش اطمینان می‌دهد که اجزای حیاتی تأثیر بیشتری بر ارزیابی کلی نگهداشت دارند. مطالعه موردی و ارزیابی کارشناسان مؤثر بودن و قابلیت استفاده این مدل را تأیید می‌کند.

در آخر، برخی مطالعات به بررسی ویژگی‌های پروژه و تأثیر آن بر کیفیت نرم‌افزار می‌پردازد.

مطالعه [۱۶] در مروری نظام‌مند به بررسی اولویت‌بندی نابسامانی‌های کد در سیستم‌های شیءگرا پرداخته و تلاش دارد تا بینش‌هایی را برای پژوهشگران و توسعه‌دهندگان جهت بهبود کیفیت نرم‌افزار ارائه دهد. این مرور ۳۹ مطالعه اولیه منتشر شده بین سال‌های ۲۰۱۱ تا ۲۰۲۲ را تحلیل کرده و آن‌ها را بر اساس انواع نابسامانی‌های کد، عوامل اولویت‌بندی، تکنیک‌های ارزیابی، ابزارهای استفاده شده، فرمول‌های رتبه‌بندی، مجموعه داده‌ها و معیارهای عملکرد دسته‌بندی کرده است. نتایج نشان می‌دهد که سه رویکرد اصلی برای اولویت‌بندی نابسامانی‌ها وجود دارد: دستی، نیمه‌خودکار و کاملاً خودکار. رویکردهای دستی، علی‌رغم متکی بودن به قضاوت کارشناسان، ممکن است دچار سوگیری شوند، در حالی که رویکردهای نیمه‌خودکار و کاملاً خودکار ارزیابی‌های عینی‌تری ارائه می‌دهند. با این حال، بسیاری از این روش‌ها همچنان به عوامل تعیین‌شده توسط توسعه‌دهندگان وابسته هستند. این مطالعه نیاز به پژوهش بیشتر درباره تکنیک‌های کاملاً خودکار، به‌ویژه با استفاده از الگوریتم‌های پیشرفته مانند شبکه‌های عصبی و منطق فازی را برجسته می‌کند. همچنین کمبود فرمول‌های استاندارد برای رتبه‌بندی و معیارهای عملکرد را شناسایی کرده و بر توسعه چارچوب‌های جامع تأکید می‌کند.

مطالعه [۱۸] تأثیر اطلاعات غیرعددی پروژه، به‌ویژه وضعیت پروژه، بر دقت مدل‌های یادگیری ماشین در شناسایی نابسامانی «کلاس بزرگ» را بررسی می‌کند. نویسندگان مجموعه داده‌ای شامل ۲۲ پروژه جاوا با وضعیت‌های مختلف (تولیدی/ پایدار، بتا و بالغ) ایجاد کرده و از پنج ابزار شناسایی نابسامانی برای شناسایی کلاس‌های بزرگ استفاده کرده‌اند. نتایج نشان می‌دهد که ترکیب اطلاعات وضعیت پروژه به‌طور قابل‌توجهی عملکرد طبقه‌بندی‌ها را بهبود می‌بخشد، به‌طوری که جنگل تصادفی بهترین عملکرد را داشته است. نویسندگان نتیجه می‌گیرند که اطلاعات وضعیت پروژه برای ساخت مدل‌های دقیق‌تر شناسایی نابسامانی کد ارزشمند است و پیشنهاد می‌کنند تحقیقات بیشتری درباره استفاده از سایر اطلاعات غیرعددی نرم‌افزار انجام شود.

۳.روش پژوهش

برای تشخیص نابسامانی در کد از روش مهندسی معکوس مدل‌رانده بهره گرفته شده است. در این روش ابتدا از روی کد جاوا با استفاده از ابزاری به‌نام مودیسکو[1] مدل جاوا استخراج شد. سپس، این مدل به مدلی مبتنی بر فرامدل معیارها تبدیل می‌شود. فرامدل معیارها که در [۱۹] معرفی شده است شامل تعاریف معیارهایی است که در تشخیص نابسامانی‌های ساختاری اهمیت دارند، برای مثال تعداد خط کد، fanout یا fanin کلاس‌ها و پیچیدگی دورانی. پس از تشخیص نابسامانی برنامه‌نویس به‌صورت

1-https://eclipse.dev/MoDisco/

دستی آن را رفع می‌کند.

برای انجام این پژوهش ۸ مطالعه موردی برای ۸ نوع نابسامانی مختلف طراحی شده است. سپس، زمان اجرا و میزان مصرف انرژی قبل و بعد از رفع هر نابسامانی اندازه‌گیری شده و تاثیر هر نابسامانی بر مصرف سیستم بررسی شده است.

جهت اندازه‌گیری زمان اجرا از دستور System.nanotime () جاوا در کد برنامه استفاده شده است.

برای اندازه‌گیری میزان مصرف انرژی از ابزار JoularJX بهره گرفته شده است[۲]. ابزار JoularJX یک عامل مبتنی بر جاوا برای پایش مصرف توان در سطح کد منبع نرم‌افزار است. این ابزار اطلاعات مصرف توان آنی را برای هر متد در یک برنامه جاوا، همچنین مصرف کل انرژی در پایان برنامه را ارائه می‌دهد. JoularJX از سیستم‌عامل‌های مختلف پشتیبانی می‌کند. این عامل می‌تواند به یک ماشین مجازی جاوا پیوست شود تا مصرف توان را در سطح کد منبع پایش کند. چندین فایل CSV با داده‌های توان و انرژی تولید می‌کند که به کاربران امکان می‌دهد مصرف توان برنامه‌های جاوای خود را در سطح متد تحلیل کنند.

در ادامه، نابسامانی‌های بررسی شده در این پژوهش به همراه تعریف، ویژگی‌ها، معیارهای تشخیص و تأثیر آن بر کیفیت و پایداری نرم‌افزار شرح داده شده است.

۱-۳- نابسامانی زیاده‌خواهی ویژگی

نابسامانی زیاده‌خواهی[۳] (یا تمایل) به ویژگی‌های دیگر زمانی رخ می‌دهد که یک کلاس بیش از حد به داده‌ها و متدهای کلاس‌های دیگر دسترسی پیدا کند [۲]. در این وضعیت، کلاس به‌جای تمرکز بر داده‌ها و وظایف خود، دائماً به داده‌های کلاس‌های دیگر وابسته است و از متدها و ویژگی‌های آن‌ها استفاده می‌کند. این نابسامانی معمولاً نشان می‌دهد که مسئولیت‌های کلاس‌ها به‌درستی تقسیم نشده‌اند و وظایف باید به‌صورت موضعی در کلاس خودشان انجام شوند.

برای تشخیص این نابسامانی، می‌توان مشاهده کرد که آیا متدی در یک کلاس وجود دارد که به‌طور مستمر به داده‌های کلاس دیگر دسترسی پیدا کند و آن‌ها را تغییر دهد یا پردازش کند. به‌علاوه، اگر بخش عمده‌ای از کد یک متد به استفاده از متدهای کلاس دیگر اختصاص یافته باشد، می‌تواند نشانه‌ای از تمایل بیش از حد به ویژگی‌های دیگر باشد [۳].

این نابسامانی می‌تواند پیامدهای منفی متعددی بر پایداری نرم‌افزار داشته باشد. با افزایش وابستگی میان کلاس‌ها، تغییر در یکی از کلاس‌ها ممکن است تأثیر زیادی بر کلاس‌های دیگر داشته باشد که باعث کاهش استقلال و افزایش وابستگی می‌شود. همچنین، این مسئله نگه‌داری و درک کد را برای توسعه‌دهندگان پیچیده‌تر می‌کند و در نهایت ممکن است ساختار نرم‌افزار دچار آشفتگی شود و قابلیت انعطاف و توسعهٔ آن کاهش یابد.

برای رفع این نابسامانی، می‌توان مسئولیت‌ها را به کلاس مناسب منتقل کرد. برای مثال، اگر کلاسی متدی دارد که بیشتر از داده‌های یک کلاس دیگر استفاده می‌کند، بهتر است این متد به کلاسی منتقل شود که داده‌ها متعلق به آن است. استفاده از اصول طراحی شیءگرا مانند اصل «قانون دِمتر» یا «اصل حداقل دانش» نیز می‌تواند به کاهش تمایل به ویژگی‌های دیگر کمک کند. این اصول به برنامه‌نویسان توصیه می‌کنند که کلاس‌ها فقط با کلاس‌های مرتبط و نزدیک خود تعامل داشته باشند و از دسترسی غیرضروری به داده‌ها و متدهای کلاس‌های دیگر بپرهیزند. انجام این اقدامات باعث افزایش استقلال کلاس‌ها، بهبود پایداری و انعطاف‌پذیری نرم‌افزار و کاهش پیچیدگی در نگه‌داری آن می‌شود [٤].

۲-۳- نابسامانی مجموعه داده‌ها

نابسامانی مجموعه داده‌ها[٤] زمانی رخ می‌دهد که گروهی از متغیرهای مرتبط به‌طور هم‌زمان به بخش‌های مختلف برنامه ارسال شوند [۳]. این موضوع می‌تواند باعث تکرار

2-https://joular.github.io/joularjx/
3-Feature Envy
4-Data Clumps

وابســتگی‌ها و کاهش خوانایی کد شــود، زیرا با تغییر در این متغیرها، همه قسمت‌های وابسته نیاز به به‌روزرسانی دارند. مجموعه داده‌ها یکی از مشکلات متداول در کدنویسی است که به‌وجود آمدن گروه‌های داده مرتبط در کد اشاره دارد. وقتی داده‌هایی که با هم مرتبط هســتند در یک گروه داده قرار می‌گیرند، این موضوع به‌عنوان یک نابسامانی کد شناخته می‌شود.

برای تشخیص این نابسامانی از معیارهای زیر استفاده می‌شــود: اندازه مجموعه داده‌ها، پیچیــدگی الگوریتم‌های پردازش داده و تعداد دفعات دســترسی به داده‌ها. هر چه مقادیر این معیارها بیشــتر باشــند، احتمال وجود این نوع نابسامانی بیشــتر اســت. مدیریت ناکارآمد داده‌ها باعث کاهش خوانایی و افزایش خطا می‌شود و پایداری سیستم را کاهش می‌دهد [۳].

بــرای حل این مشــکل، می‌توان متغیرهــای مرتبط را در قالــب یک شیء تجمیع کرد و یا از اصل تک‌مســئولیتی بهره برد. ایجاد کلاس‌های جدید یا اســتفاده از متغیرهای محلی، آرگومان‌های تابع و ثابت‌هــا برای مدیریت داده‌ها نیز راهکارهای مناسبی است. این تغییرات منجر به کاهش تکــرار، بهبود خوانایی، افزایش قابلیــت تغییرپذیری و در نهایت ارتقای کیفیت نرم‌افزار خواهد شد.

۳-۳- نابسامانی روابط نامناسب

روابط نامناســب[5] زمانی رخ می‌دهد کــه یک کلاس به حوزه‌هــا و متدهای داخلی کلاس دیگری وابســته شــود. ایــن نابســامانی در مهنــدسی نرم‌افزار به‌معنــای وجود وابستگی‌های نامناسب و غیرضروری بین کلاس‌ها و اجزای مختلف سیستم است. در این وضعیت، اجزای سیستم به‌طور غیرمنطقی به یکدیگر وابســته می‌شوند، که منجر به کاهش انعطاف‌پذیری، افزایش پیچیدگی و کاهش قابلیت نگهداری کد می‌شود. به‌طوری‌که هر تغییر کوچک در یک بخش می‌تواند تغییرات غیرمنتظره‌ای در ســایر بخش‌هــا ایجاد کند و کار برنامه‌نویسان را پیچیده‌تر سازد [۲].

برای تشخیص این نابسامانی از معیارهای زیر استفاده می‌شود: تعداد وابستگی‌های بین کلاس‌ها و تعداد متدهایی که بــه داده‌های خصــوصی کلاس‌های دیگر دســترسی دارند. کاهــش قابلیت نگهداری و افزایــش احتمال خطا از تاثیرات اصلی این نابســامانی است. به دلیل وابستگی‌های زیــاد، تغییر در یک کلاس می‌تواند منجر به مشــکلات در کلاس‌های دیگر شود [٤].

بــرای جلوگیری از این نابســامانی، می‌توان از الگوها و اصول طراحی مانند اصل جداســازی دغدغه‌ها و الگوی معکوس وابستگی[6] استفاده کرد. با استفاده از این الگوها، می‌توان اجزا را به‌صورت مجزا و مستقل از یکدیگر طراحی کرده و وابســتگی‌های نامناسب را کاهش داد. هدف اصلی در اینجــا، ایجاد یک ســاختار کد منظــم، قابل‌نگهداری و توسعه‌پذیر با حداقل تأثیر را داشته باشد.

۴-۳- نابسامانی شیء خداوند

در نابسامانی شیء خداوند، یک شیء یا کلاس وظایف بسیار زیادی را به‌عهده می‌گیرد و در نتیجه وابستگی‌های بســیــاری با دیگــر اشیــاء و کلاس‌ها دارد. ایــن شیء یا کلاس به‌نوعی به‌عنوان «خداونــد» در نرم‌افزار موردنظر تلقی‌می‌شود، زیرا همه وظایف و عملیات در آن قرار دارند

این نابســامانی باعث می‌شود که کد نامنظم و ناخوانا با قابلیت نگهداری پایین ایجاد شــود. همچنین، تغییرات در یک شیء خداوند ممکن اســت تأثیرات وسیعی در ســایر بخش‌های نرم‌افزار داشته باشد و این امر می‌تواند منجر به افزایش پیچیدگی و آسیب‌پذیری‌های نرم‌افزار شود.

برای جلوگیری از وقوع این نابسامانی، طراحی نرم‌افزار باید بر اصولی مانند تفکیک مســئولیت‌ها، اصل جداکردن امور مربوط به دامنه، استفاده از الگوهای طراحی مناسب و تعامل موثر بین اشیاء تمرکز داشته باشد.

برای تشخیص این نابسامانی از معیارهای زیر استفاده می‌شود: تعداد متدها، تعداد ویژگی‌ها (Attributes) و تعداد وابستگی‌ها (Dependencies) به دیگر بخش‌ها. هر چه این

5-Inappropriate Intimacy

6-Dependency Inversion

مقادیر بالاتر باشند، احتمال وجود این نابسامانی بیشتر است. این نابسامانی باعث کاهش پایداری و افزایش پیچیدگی کلاس می‌شود و نگهداری، تغییر و آزمون را دشوار می‌کند.

۵-۳- نابسامانی استفاده افراطی از نوع دادهٔ اولیه

نابسامانی استفاده افراطی از نوع دادهٔ اولیه[7] زمانی رخ می‌دهد که از انواع داده ساده مانند عدد صحیح، رشته و بولین برای نمایش مفاهیم پیچیده استفاده شود [۲]. برای مثال، ممکن است برای ذخیره‌سازی کدملی از رشته استفاده شود، در حالی که تعریف یک نوع داده اختصاصی که شرایط و قواعد معتبر بودن کدملی را بررسی کند، بسیار مناسب‌تر است.

تشخیص نابسامانی استفاده افراطی از نوع دادهٔ اولیه می‌تواند با بررسی چند نشانه انجام شود. از جمله این نشانه‌ها، تکرار زیاد پارامترهایی از نوع داده‌های ساده در توابع و متدهاست. همچنین اگر نسبت استفاده از انواع داده اولیه به انواع اختصاصی بسیار بالا باشد، می‌تواند نشانه‌ای از این نابسامانی باشد. پیچیدگی زیاد در روش‌های پردازش و اعتبارسنجی این مقادیر ساده نیز حاکی از این است که به‌جای مدل‌سازی دقیق، از انواع اولیه استفاده شده است [٤].

این نابسامانی تأثیرات منفی متعددی بر پایداری نرم‌افزار دارد. اول این که باعث افزایش پیچیدگی کد و دشواری درک مفاهیم می‌شود، زیرا انواع اولیه قادر به بیان کامل یک مفهوم پیچیده نیستند. همچنین خوانایی کد را کاهش داده و فهم هدف و کاربرد متغیرها را برای توسعه‌دهندگان مشکل می‌سازد. یکی دیگر از مشکلاتی که این نابسامانی ایجاد می‌کند، دشواری در آزمایش کد است، زیرا نبود اعتبارسنجی متمرکز برای این مقادیر، روند آزمون را پیچیده‌تر می‌کند. در نهایت، استفاده از انواع اولیه، انعطاف‌پذیری نرم‌افزار را کاهش داده و تغییرات بعدی را دشوار می‌کند.

برای رفع این نابسامانی می‌توان از روش‌هایی مانند ایجاد کلاس‌های اختصاصی برای مفاهیم پیچیده بهره گرفت. برای مثال، به‌جای استفاده از رشته برای ذخیره‌سازی شماره تماس، می‌توان کلاس مخصوصی تعریف کرد که شامل قواعد و اعتبارسنجی شماره تماس باشد. همچنین به‌جای استفاده از انواع ساده و مقادیر ثابت، می‌توان از انواع داده‌های مناسب مانند شمارشی‌ها بهره برد که ضمن افزایش خوانایی، کنترل بر روی مقادیر را نیز افزایش می‌دهند. این راهکارها به بهبود پایداری و خوانایی نرم‌افزار کمک کرده و نگه‌داری و توسعه آن را ساده‌تر می‌کند [۳].

۶-۳- نابسامانی متد طولانی

متدی که وظایف بسیاری را برعهده دارد و از تعداد خطوط کد زیادی تشکیل شده است. معمولاً این متدها پیچیدگی بالایی دارند و انجام چندین کار مرتبط یا غیرمرتبط را در خود جای داده‌اند. از مشکلات آن کاهش خوانایی کد، پیچیدگی زیاد، دشواری در آزمون و نگهداری می‌باشد. این متدها معمولاً از وابستگی‌های زیادی به سایر بخش‌ها برخوردارند. در نتیجه، قابلیت اطمینان کد کاهش یافته و احتمال بروز خطاهای پنهان افزایش می‌یابد [۲].

برای تشخیص این نابسامانی از معیارهای زیر استفاده می‌شود: تعداد خطوط کد (LOC)، پیچیدگی دورانی (Cyclomatic Complexity) و تعداد پارامترهای ورودی. هر چه مقادیر این معیارها بالاتر باشند، احتمال وجود این نوع نابسامانی بیشتر است [۳].

برای رفع نابسامانی متد طولانی، می‌توان با شکستن متد به متدهای کوچک‌تر و با هدف مشخص، خوانایی و سادگی کد را افزایش داد. هر متد کوچک باید تنها یک وظیفه خاص را انجام دهد و نام‌گذاری دقیقی داشته باشد تا عملکرد آن به‌وضوح بیان شود. این روش به اصل «تک‌مسئولیتی» پایبند است و نگهداری و آزمون کد را آسان‌تر می‌کند. همچنین، استفاده از الگوهایی مانند Tem-

7-Primitive Obsession

plate Method یا Strategy می‌تواند به ساختاردهی بهتر کد کمک کند و از تکرار کد و پیچیدگی اضافی جلوگیری کند و امکان توسعه و تغییر کد با کمترین تأثیر در بخش‌های دیگر را فراهم کند [٤].

۳-۷- نابسامانی دستور سوئیچ

این نابسامانی به استفاده بیش از حد از دستور «switch» در برنامه‌نویسی اشاره دارد. در این حالت، دستور «switch» برای انتخاب و اجرای بخش‌های مختلف کد بر اساس یک مقدار خاص به کار می‌رود [۲]. اما با افزایش تعداد شرایط مختلف، کد پیچیده و ناخوانا می‌شود. این وضعیت می‌تواند باعث افزایش پیچیدگی، کاهش خوانایی و دشواری در نگهداری و کاهش انعطاف‌پذیری کد شود، زیرا هرگونه افزودن یا تغییر در بخش‌ها مستلزم اصلاح دستورات «switch» مرتبط است که احتمال بروز خطا را افزایش می‌دهد.

برای تشخیص این نابسامانی از معیارهای زیر استفاده می‌شود: تعداد شاخه‌های موجود در دستور switch، عمق تو در تو شدن دستورات شرطی. این نابسامانی باعث افزایش پیچیدگی و کاهش پایداری کد می‌شود. تغییرات در منطق تصمیم‌گیری ممکن است به اشتباهات جدیدی منجر شود.

برای پیشگیری از این مشکل، می‌توان از الگوهایی مانند Strategy یا Visitor بهره برد. این الگوها امکان جداسازی بخش‌های مختلف کد در کلاس‌های مجزا را فراهم می‌کنند و به اصل جداسازی دغدغه‌ها پایبند هستند. هدف اصلی، ایجاد ساختاری منظم و خوانا برای کد است که تغییرات به‌سادگی اعمال شوند و انعطاف‌پذیری بیشتری با کمترین تأثیر ممکن فراهم شود.

۳-۸- نابسامانی جراحی ساچمه‌ای

نابسامانی جراحی ساچمه‌ای[8] وضعیتی است که در آن، تغییر در یک بخش از کد، نیازمند اعمال تغییرات در بخش‌های مختلف کد است [۲]. این نابسامانی نشان‌دهنده وابستگی بالا بین بخش‌های مختلف کد و نقض اصل باز و بسته است. این مشکل معمولاً به دلیل وابستگی‌های زیاد و نامناسب بین ویژگی‌ها و کلاس‌ها در کد ایجاد می‌شود. این وابستگی‌ها باعث می‌شود که هر تغییر کوچک در یک قسمت، مانند شلیک گلوله‌ای، به نقاط مختلف کد آسیب برساند و تغییرات گسترده‌ای را به دنبال داشته باشد.

برای تشخیص این نابسامانی، می‌توان از معیارهایی مانند تعداد خطوط تغییر یافته در هر تغییر و تعداد کلاس‌ها و متدهای وابسته به یکدیگر استفاده کرد. این نوع نابسامانی منجر به وابستگی‌های زیاد و کاهش پایداری کد می‌شود [٤].

برای رفع این مشکل، می‌توان از روش‌های مختلفی استفاده کرد. اول، تحلیل مجدد ویژگی‌ها و اجزای نرم‌افزار و کاهش وابستگی‌ها، دوم، استفاده از اصل تک‌مسئولیتی برای تقسیم وظایف بین کلاس‌ها و ویژگی‌ها، سوم، استفاده از الگوهای طراحی مانند Observer و Mediator برای کاهش وابستگی‌ها و چهارم، استفاده از آزمون‌های واحد برای ارزیابی تغییرات و شناسایی مشکلات. با استفاده از این روش‌ها، می‌توان وابستگی‌ها را کاهش داد، انسجام کد را افزایش داد و تغییرات را بهتر مدیریت کرد.

۴. ارزیابی

در این بخش، تاثیر رفع نابسامانی‌های کد بر مصرف انرژی و بهبود کیفیت در نرم‌افزارهای جاوا مورد بررسی قرار گرفته است. به‌منظور ارزیابی، هشت نوع مختلف نابسامانی شناسایی شده و نمونه‌هایی از کدهای جاوا برای هر کدام مورد آزمایش قرار گرفته است۹. آزمایش‌ها بر روی پردازنده اینتل دوهسته‌ای با حافظهٔ ٤ گیگابایت انجام شده و مصرف انرژی هر کدام از کدها را با استفاده از ابزار JoularJX جمع‌آوری شده است. همچنین، زمان اجرا در نرم‌افزار اکلیپس و با دستور System.nanotime() قبل

8-Shotgun Surgery

۹-کدها از لینک زیر قابل دانلود می‌باشد:
https://github.com/leilasamimi/EnergyOfJavaCodeSmell

و بعد از دستورات اصلی اندازه‌گیری شده است. در ادامه، هر نابسامانی به‌طور مجزا توضیح داده شده و سپس نتایج حاصل از آزمایش‌ها، شامل نمودارهای زمان اجرا (شکل ۱) و مصرف انرژی (شکل ۲) ارائه می‌شود.

۱,۴- معرفی مطالعات موردی

۱-۱-۴ - نابسامانی زیاده‌خواهی ویژگی

در مطالعه موردی اول، کلاسی به‌نام ShoppingCart وظیفه مدیریت اقلام موجود در سبد خرید، محاسبه قیمت کل و چاپ رسید را بر عهده دارد. در این کد، متد calculateTotalPrice برای محاسبه مجموع قیمت اقلام و متد printReceipt برای نمایش جزئیات هر کالا و مجموع قیمت در رسید به‌کار گرفته می‌شود. این کد نمونه‌ای از نابسامانی زیاده‌خواهی ویژگی است، زیرا کلاس ShoppingCart به‌شدت به اطلاعات و ویژگی‌های کلاس Item، مانند قیمت و نام کالا وابسته است. به‌عبارت دیگر، کلاس ShoppingCart بیشتر به جزئیات داخلی Item علاقه‌مند است تا به انجام مسئولیت‌های خودش. این نوع وابستگی منجر به کد غیر بهینه و وابستگی نامناسب میان کلاس‌ها می‌شود و می‌تواند پیچیدگی کد را افزایش دهد.

برای بهبود این ساختار، بهتر است که عملیاتی مانند محاسبه قیمت و ارائه اطلاعات مربوط به کالا در خود کلاس Item پیاده‌سازی شوند. مثلاً می‌توان متدی مانند calculatePrice در کلاس Item تعریف کرد تا محاسبه قیمت کل به‌صورت مستقل و در سطح همان کلاس انجام گیرد. این رویکرد باعث کاهش وابستگی‌های اضافی بین کلاس‌ها و ارتقای خوانایی و انعطاف‌پذیری کد خواهد شد

۲-۱-۴- نابسامانی مجموعه داده‌ها

در مطالعه موردی دوم، اطلاعات مربوط به کاربر شامل نام، نام خانوادگی، نشانی و شماره تلفن به‌عنوان پارامترهای جداگانه به متد processUserData ارسال می‌شوند. این طراحی ظاهراً ساده و مؤثر به نظر می‌رسد، اما در عمل نابسامانی مجموعه داده را به همراه دارد که قابلیت نگهداری و خوانایی کد را تحت تأثیر قرار می‌دهد. پارامترهایی که به متد مذکور ارسال می‌شوند، همگی به کاربر مربوط هستند و به‌صورت گروهی در چندین بخش از کد با هم استفاده می‌شوند. این مسئله باعث تکرار داده‌ها و کاهش خوانایی می‌شود، چرا که به‌جای استفاده از یک ساختار واحد (مثلاً یک کلاس User)، اطلاعات به‌صورت پراکنده و غیر مجتمع در کد پخش شده‌اند.

این نوع نابسامانی نگهداری کد را دشوارتر می‌کند، زیرا در صورت تغییر در اطلاعات کاربر، باید تمامی نقاطی که از این پارامترها استفاده می‌کنند، به‌روزرسانی شوند. برای حل این مشکل، بهتر است به‌جای ارسال پارامترهای پراکنده، یک شیء مانند User تعریف و استفاده شود که تمام اطلاعات کاربر را در قالب یک واحد منسجم نگه دارد

۳-۱-۴- نابسامانی روابط نامناسب

در مطالعه موردی سوم، کلاس Customer مسئول مدیریت اطلاعات مشتری شامل نام و سن است و همچنین متدهایی مانند getDiscountMessage برای نمایش پیام تخفیف و getGreeting برای تولید پیام خوشامدگویی دارد. این نمونه کد به‌دلیل دسترسی بیش از حد به اطلاعات مشتری و انجام پردازش‌هایی که به سایر کلاس‌ها مربوط می‌شود، نشان دهندهٔ نابسامانی روابط نامناسب است، زیرا کلاس Customer در حال انجام عملیات اضافی است که بهتر است در سایر کلاس‌ها صورت گیرند. برای مثال، تعیین تخفیف به‌طور ایده‌آل باید به یک کلاس جداگانه مانند DiscountService واگذار شود که به‌طور خاص بر سیاست‌های تخفیف تمرکز داشته باشد. به همین ترتیب، پردازش خوشامدگویی می‌تواند به کلاسی مانند GreetingService منتقل شود.

این نوع روابط نامناسب بین کلاس‌ها، نه تنها باعث افزایش وابستگی‌ها می‌شود، بلکه به کاهش قابلیت انعطاف‌پذیری و نگهداری کد نیز منجر می‌شود. تفکیک مسئولیت‌ها و انتقال عملیات مربوط به تخفیف و

خوشامدگویی به کلاس‌های مجزا، باعث بهبود ساختار کد و کاهش وابستگی‌های نامناسب می‌شود، که این امر توسعه و تغییرات آینده را ساده‌تر می‌سازد.

۴-۱-۴- نابسامانی شیء خداوند

در مطالعه موردی چهارم، کلاس GodObject به‌عنوان یک شیء چندمنظوره طراحی شده که مجموعه‌ای از ویژگی‌ها و عملیات مختلف را در خود جای داده است. این کلاس اطلاعات متنوعی مانند نام، سن، نشانی و شماره تلفن را ذخیره می‌کند و در عین حال، شامل عملیات پیچیده‌تری نظیر محاسبه حقوق، اعتبارسنجی داده‌ها و ارسال اعلان نیز می‌باشد. این طراحی امکان دسترسی به ویژگی‌ها و اجرای عملیات مختلف را در یک شیء فراهم می‌کند، اما به همراه خود مشکلاتی را ایجاد کرده است که بر کیفیت و قابلیت نگهداری کد تأثیر منفی دارد.

کلاس GodObject نمونه‌ای از نابسامانی شیء خداوند است؛ چرا که تقریباً تمامی ویژگی‌ها و عملیات را در خود متمرکز کرده و مسئولیت‌های مختلفی را بر عهده دارد. این تمرکز وظایف باعث می‌شود که کلاس بزرگ و پیچیده شود و اصل تک‌مسئولیتی را نقض کند، زیرا هر نوع عملیات و داده‌ای در آن جای داده شده است. چنین ساختاری خوانایی و قابل‌فهم بودن کد را کاهش داده و تغییر و نگهداری را دشوار کرده است، چرا که هر تغییری می‌تواند بر بخش‌های متعددی از کلاس تأثیر بگذارد.

۵-۱-۴- نابسامانی استفاده افراطی از نوع دادهٔ اولیه

در مطالعه موردی پنجم، کلاسی به‌نام Order برای مدیریت اطلاعات سفارش طراحی شده و شامل جزئیات مشتری مانند نام و نشانی و همچنین اطلاعات مربوط به کالا نظیر نام کالا، قیمت و تعداد آن است. علاوه بر این، متدهایی مانند calculateTotalPrice برای محاسبه قیمت کل و generateOrderSummary برای تولید خلاصه سفارش در این کلاس پیاده‌سازی شده‌اند. این ساختار در ظاهر ساده به‌نظر می‌رسد، اما استفاده بیش از حد از انواع داده‌های اولیه، مشکلاتی را در نگهداری و توسعه کد به همراه دارد. در این نمونه، از انواع داده‌های اولیه برای نمایش اطلاعات پیچیده‌تری مانند اطلاعات مشتری و کالا استفاده شده است. استفاده از این داده‌های پراکنده و ابتدایی می‌تواند منجر به تکرار کد و کاهش انعطاف‌پذیری شود، چرا که هر تغییری در این اطلاعات نیازمند اصلاح چندین بخش از کد خواهد بود.

برای بهبود ساختار کد، بهتر است به‌جای استفاده از انواع داده‌های اولیه، کلاس‌های اختصاصی مانند Customer و Item ایجاد شوند که وظیفه مدیریت اطلاعات مشتری و کالا را بر عهده داشته باشند. این رویکرد نه تنها ساختار کد را منسجم‌تر می‌کند، بلکه خوانایی آن را افزایش داده و اعمال تغییرات را آسان‌تر می‌سازد.

۶-۱-۴- نابسامانی متد طولانی

در مطالعه موردی ششم، فرآیند پردازش سفارش در دو کلاس اصلی مدیریت می‌شود: کلاس Order که ویژگی‌های سفارش از جمله اعتبار سفارش، موجودی کالا و توانایی مالی مشتری را بررسی می‌کند و کلاس OrderProcessor که عملیات پردازش سفارش را انجام می‌دهد. در متد processOrder، وضعیت‌های مختلف سفارش به‌ترتیب بررسی می‌شوند و در صورتی که سفارش معتبر باشد، کالا موجود باشد و مشتری از لحاظ مالی شرایط پرداخت را داشته باشد، چندین عملیات شامل پردازش پرداخت، به‌روزرسانی موجودی انبار و ارسال ایمیل تاییدیه انجام می‌شود. این رویکرد به ظاهر کامل و کارا به‌نظر می‌رسد، اما در عمل مشکلاتی ایجاد می‌کند که می‌تواند بر کارایی و نگهداری کد تأثیر منفی بگذارد. متد processOrder نمونه‌ای از نابسامانی متد طولانی است؛ به‌دلیل این که وظایف متعددی از جمله بررسی شرایط سفارش، مدیریت عملیات‌های پرداخت، به‌روزرسانی انبار و ارسال ایمیل را به‌صورت یکجا مدیریت می‌کند. این ساختار نه تنها خوانایی و درک کد را پیچیده می‌کند، بلکه

اصل تک‌مسئولیتی را نیز نقض می‌کند. در نتیجه، این متد با چندین شرط تودرتو، خواندن و نگهداری کد را دشوار کرده و قابلیت باز استفاده از آن را کاهش داده است.

۴-۱-۷- نابسامانی دستور سوئیچ

در مطالعه موردی هفتم، برای بررسی نابسامانی دستور سوئیچ، قیمت هر نوع میوه با استفاده از دستور سوئیچ تعیین می‌شود. متغیری به‌نام fruit، بر اساس نوع میوه مقادیر مختلفی دریافت می‌کند و به‌ازای هر مقدار، قیمت متناظر آن تنظیم می‌شود. این ساختار به ظاهر ساده و کاربردی به‌نظر می‌رسد، اما استفاده از دستور سوئیچ برای تعیین رفتار می‌تواند مشکلاتی را در نگهداری و توسعه کد به همراه داشته باشد. چرا که با اضافه شدن انواع جدید میوه‌ها یا تغییر قیمت‌ها، باید ساختار دستور به‌روز شود و تغییرات در چندین قسمت از کد اعمال شود. این مسئله اصل باز- بسته بودن را نقض می‌کند، زیرا کد برای اضافه کردن انواع جدید نیاز به تغییر مکرر دارد. علاوه بر این، ساختار سوئیچ به‌صورت متمرکز کنترل رفتارها را در یک مکان خاص نگه می‌دارد، که این می‌تواند از اصول برنامه‌نویسی شیءگرا فاصله بگیرد.

یک راهکار مناسب برای حل این مشکل، استفاده از الگوی strategy است. می‌توان کلاس‌های مجزا برای هر نوع میوه تعریف کرد که هر کدام شامل متد مخصوص به‌خود برای محاسبه قیمت باشند. چنین ساختاری نه تنها کد را تمیزتر و خواناتر می‌کند، بلکه انعطاف‌پذیری بیشتری به برنامه می‌بخشد و نگهداری آن را آسان‌تر می‌سازد.

۴-۱-۸- نابسامانی جراحی ساچمه‌ای

در مطالعه موردی آخر، نمونه‌ای از نابسامانی جراحی ساچمه‌ای بررسی می‌شود که در آن کلاسی به‌نام Custom-er شامل اطلاعات مشتری نظیر نام، ایمیل و نشانی است و همچنین چندین متد مانند updateAddress، sendEmail و placeOrder را در خود جای داده است که وظایف مختلفی را انجام می‌دهند. در نگاه اول، این ساختار ساده و مناسب به‌نظر می‌رسد، اما در عمل مشکلاتی ایجاد می‌کند که به نگهداری و توسعه کد آسیب می‌رساند. هر بار که تغییری در یک ویژگی یا عملکرد مشتری ایجاد شود، این تغییرات در چندین قسمت از برنامه، از جمله متدهای مختلف و حتی کلاس‌های مرتبط دیگر، اعمال خواهد شد. برای مثال، اگر نیاز به تغییر ساختار ذخیره‌سازی نشانی باشد، تغییرات باید هم در متد updateAddress و هم در هر جای دیگری که از نشانی مشتری استفاده می‌شود، اعمال شود.

این پراکندگی تغییرات باعث افزایش ریسک خطا و کاهش قابلیت نگهداری کد می‌شود.

برای حل این مشکل، می‌توان از تفکیک مسئولیت‌ها و ایجاد کلاس‌های مجزا برای هر وظیفه، مانند کلاس‌های OrderService، EmailService و AddressService استفاده کرد. این رویکرد باعث می‌شود که هر کلاس تنها مسئولیت خاص خود را داشته باشد و تغییرات در یک بخش به سادگی بدون نیاز به اعمال تغییرات پراکنده صورت گیرد.

۴-۲- نتایج ارزیابی در محاسبه زمان اجرا

شکل ۱ شامل ۸ نمودار است که زمان اجرا را برای هر مطالعه موردی قبل و بعد از رفع نابسامانی نشان می‌دهند. نمودارها از راست به چپ و بالا به پایین به ترتیب توضیح داده می‌شوند.

نمودار اول[۱۰]: برای نابسامانی زیاده‌خواهی ویژگی مشاهده می‌شود که پس از بازآرایی، زمان اجرا در تکرارهای کمتر تقریباً مشابه نسخه اصلی است و در تکرارهای بالاتر هم تفاوت قابل توجهی میان نسخه بهینه و نسخه اصلی دیده نمی‌شود. این موضوع به‌دلیل تغییرات جزیی در ساختار و تقسیم وظایف است که تأثیرات زیادی در زمان اجرای برنامه ندارند. در نسخه اصلی، پراکندگی وظایف و وابستگی‌های میان اجزای مختلف باعث افزایش سربار پردازشی شده است. این سربار در تکرارهای بالا بیشتر خود را نشان می‌دهد، اما تغییرات ساختاری در نسخه بازآرایی شده هم تأثیر چشمگیری در کاهش

۱۰-با عنوان Feature Envy

زمان اجرا ندارنـد. بنابراین، در تکرارهـای بالاتر تفاوت زمان اجرا میان دو نسخه تقریباً یکسان است. بازآرایی با کاهش پیچیدگی‌ها و حذف عملیات اضافی، مسیر پردازش را ساده‌تر کرده و باعث کاهش بار روی حافظه و پردازنده می‌شـود. به‌نظر می‌رسد برای بهبود کیفیت نرم‌افزار بهتر است این نابسامانی رفع گردد.

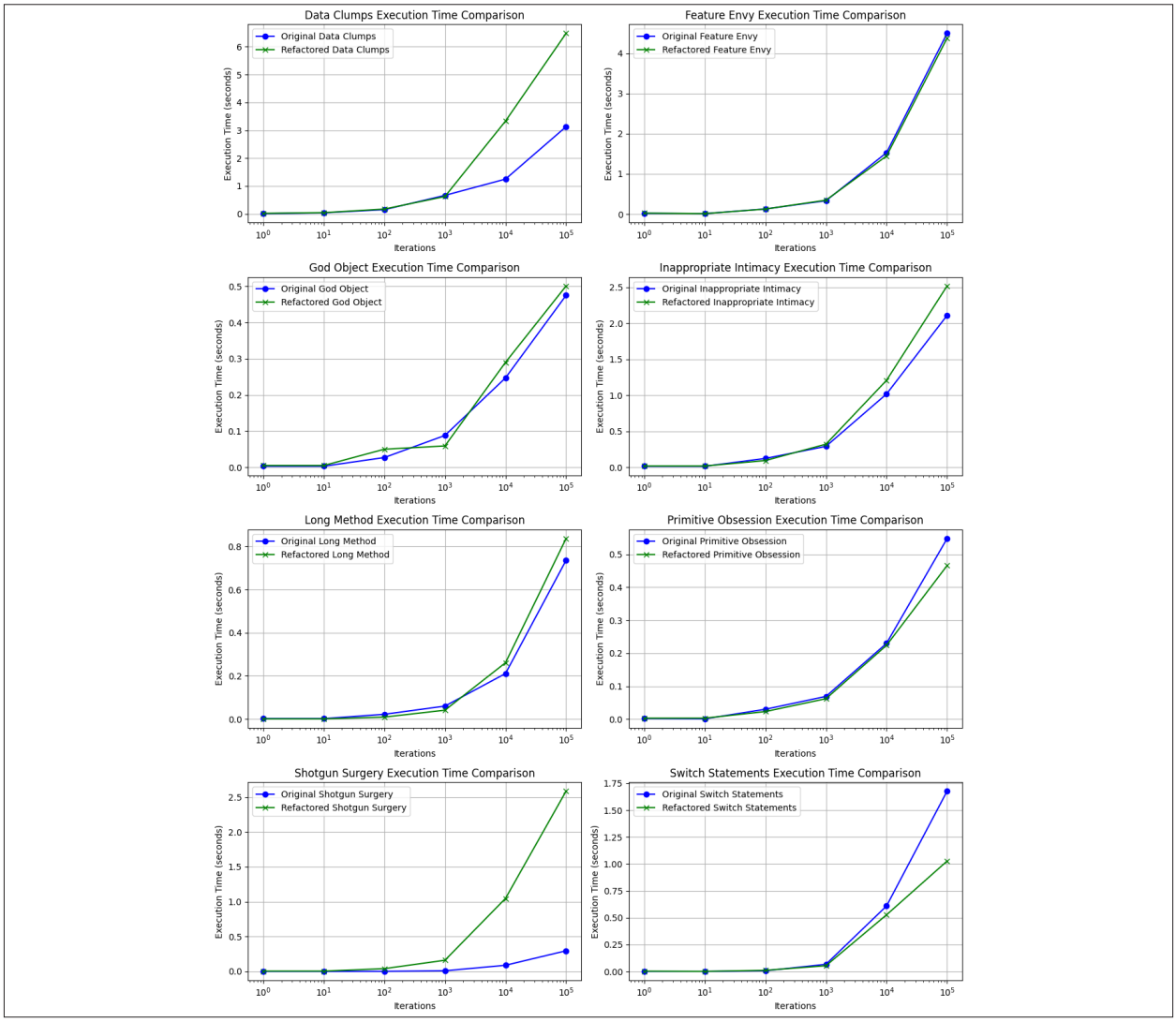
نمـودار دوم[۱۱]: در نابسـامانی مجموعـه داده، عبور پارامترهای مسـتقل به توابع و پـردازش آن‌ها به‌صورت جداگانـه باعث افزایش پیچیدگی محاسـباتی و زمان اجرا می‌شود. در نسخه بهینه‌شـده، با تعریف یک کلاس برای تجمیع داده‌ها، هدف بهبود روند پردازش بوده اسـت، چرا که انتقال و پـردازش یک شیء سـاده‌تر از انتقال چندین پارامتر مسـتقل است. با این حال، به‌طور جالبی در نمودار مشاهده می‌شود که علی‌رغم این تغییرات، نسخه بهینه در زمـان اجرا بسیار کندتر از نسـخه اصلی عمل می‌کند. در نسـخه بهینه‌شده، افزایش زمان اجرا به‌دلیل هزینه اضافی ایجاد شیء و فراخوانی عملیات‌ها به‌طور مکرر است. برای هر بـار اجرا، شیء جدیدی ایجاد می‌شـود که این فرآیند زمان‌بـر اسـت. علاوه بر این، برای دسـترسی به مقادیر مختلف شیء، عملیات‌های اضافی فراخوانی می‌شـوند که موجب افزایش زمان اجرا می‌شـود. ایجاد شیء برای هر بار اجرا باعث مصرف بیشتر حافظه و افزایش هزینه‌های جانبی از جمله مدیریت حافظه می‌شود.

نمودار سوم[۱۲]: با توجه به نمودار زمان اجرا، مشاهده می‌شود که زمان اجرا در نسخه بهینه شده نسبت به نسخه اصلی، به‌ویژه در تکرارهای بالا، بیشـتر شده است. علت اصلی این اسـت که در نسخه بهینه شـده، مسئولیت‌های محاسـبه تخفیف و تولید پیام خوشامدگویی به کلاس‌های جداگانه منتقل شده‌اند. هر فراخوانی به این کلاس‌ها نیاز به عبور از مرزهای مختلف و تخصیص منابع بیشتری دارد. این ساختار باعث افزایش هزینه‌های جانبی می‌شود، مانند فراخوانی‌هـا و مدیریت حافظه میان کلاس‌ها که در نهایت منجر به افزایش زمان اجرا می‌شود. نسخه بازآرایی شده در ابتدا سـریع‌تر عمل می‌کند، امـا در تکرارهای بالاتر با تفکیک بیشتر، هزینه‌های پردازشی و زمانی بالاتری را در مقیاس بزرگ‌تر می‌پردازد.

نمودار چهارم[۱۳]: همان‌طور کـه در نمودار زمان اجرا مشاهده می‌شود، نسخه بهینه شده در تعداد تکرارهای کم (۱۰ به توان ۳) عملکرد بهتری نسبت به نسخه اصلی دارد، اما با افزایش تکرارها، زمان اجرا در نسخه بهینه به‌تدریج افزایش یافته و برای مقیاس‌های بزرگ‌تر اندکی بیشـتر از نسخه اصلی می‌شـود. بهبود عملکرد در تعداد تکرارهای پایین به این دلیل است که انتقال مسئولیت‌ها به کلاس‌های جداگانه و کاهش تمرکز وظایـف در یک کلاس، پیچیدگی محاسـباتی را در این سـطح کاهش داده و اجرای عملیات را کارآمدتر کرده اسـت. با این حـال، در تکرارهای بالا، هزینه‌های مرتبط با ایجاد و مدیریت اشیاء متعدد و همچنین فراخوانی‌هـای مکـرر میان کلاس‌ها باعث ایجاد سـربار بیشتری می‌شـود. این سـربار ناشی از افزایش تعاملات میـان کلاس‌ها و عملیات‌های مرتبـط با تخصیص حافظه اسـت، که در مقیاس بزرگ‌تر تأثیـر قابل‌توجهی بر زمان اجرا دارد. در نهایت، در حالی که تفکیک وظایف در نسخه بازآرایی شـده اصول طراحی بهتری را نشان می‌دهد، این تغییر منجر به افزایش زمان اجرا شده است. برای مواردی کـه تعداد تکرارهـای زیادی دارنـد، بایـد از تکنیک‌هایی مانند کشـف عملکرد بهینه‌تر در طراحی کد یا بهینه‌سازی فرایندهای داخلی کلاس‌ها اسـتفاده کرد تا اثرات منفی بر مصرف منابع به حداقل برسد.

نمـودار پنجم[۱۴]: در نسـخه اصـلی، به‌دلیـل این‌که عملیات‌هـا در یک کلاس متمرکز هسـتند، هر عملیات باید به‌طور مسـتقیم از روی داده‌های مختلف در همان کلاس پردازش شـود. این باعث می‌شـود که زمان اجرا افزایش یابد، زیـرا هر بار که یک عملیات نیاز بـه انجام دارد، این عملیـات باید تمـامی داده‌ها را در خود پـردازش کرده و

۱۱-با عنوان Data Clumps
۱۲-با عنوان Inappropriate Intimacy
۱۳- با عنوان God Object
۱۴-با عنوان Primitive Obsession



شکل ۱. مقایسه زمان‌های اجرا در کدهای دارای نابسامانی و کدهای بهینه شده

تعاملات پیچیده‌ای بین بخش‌های مختلف انجام دهد. در نسخه بهینه، تفکیک وظایف باعث می‌شود که هر کدام از عملیات‌ها توسط یک واحد اختصاصی انجام شود. این باعث می‌شود که عملیات‌ها به صورت موازی یا مستقل از هم پردازش شوند که منجر به کاهش زمان اجرا می‌شود. در تکرارهای بالا، به‌دلیل تقسیم وظایف به واحدهای مختلف و انجام پردازش‌ها به‌طور تخصصی‌تر، زمان اجرا به‌طور چشمگیری کاهش می‌یابد.

نمودار ششم[۱۵]: در نسخه بهینه، زمان اجرای کد تا ۱۰۰۰ تکرار به‌طور مشابه نسخه اصلی باقی می‌ماند، اما از تکرارهای بعدی، زمان اجرا افزایش می‌یابد. این افزایش به‌دلیل تفکیک متدهای مختلف به‌صورت جداگانه است. در نسخه بهینه، هر عملیات مانند بررسی اعتبار سفارش، موجودی کالا و موجودی کافی به‌صورت جداگانه اجرا می‌شود که باعث می‌شود زمان اجرای هر کدام از این بررسی‌ها به‌طور جداگانه محاسبه شود. در تکرارهای بالاتر، این زمان‌های اضافی جمع می‌شوند و منجر به افزایش قابل توجه زمان اجرا می‌شوند. علاوه بر این، اجرای زنجیره‌ای متدها باعث می‌شود که پردازش‌ها به تدریج زمان بیشتری را در تکرارهای بالا به خود اختصاص دهند، که در نهایت به افزایش زمان اجرا منجر می‌شود.

نمودار هفتم[۱۶]: در نسخه بهینه، زمان اجرای برنامه

۱۵-با عنوان Long Method

۱۶-با عنوان Switch statement

کاهش یافته است چون تصمیم‌گیری‌ها با استفاده از دستور سوییچ به شکلی مؤثرتر و سریع‌تر انجام می‌شود. در نسخه اصلی، بررسی چندین شرط مختلف و اجرای دستورات پیچیده باعث افزایش زمان اجرا می‌شد، اما در نسخه بهینه، فرآیند تصمیم‌گیری به شکلی ساده‌تر و مستقیم‌تر صورت می‌گیرد که زمان کمتری برای پردازش نیاز دارد. این بهینه‌سازی‌های ساختاری باعث شده‌اند که زمان اجرا کاهش یابد و عملکرد کلی برنامه بهبود یابد.

نمودار هشتم[۱۷]: زمان اجرای نسخه بهینه به دلیل تقسیم وظایف به متدهای مختلف و جداگانه، نسبت به نسخه اصلی کمی بهتر است. این بهبود ناشی از انجام عملیات به‌صورت توزیع شده است که باعث می‌شود پردازش‌ها سریع‌تر انجام شوند. با این‌حال، چون جابه‌جایی‌های بیشتری بین متدها و اشیاء انجام می‌شود، به‌طورکلی زمان صرف شده برای اجرای هر عملیات کاهش می‌یابد، اما از آن‌جایی‌که تنها در برخی از سناریوها این بهینه‌سازی‌ها اثر مثبت دارند، تغییرات عمده‌ای در زمان کلی اجرا مشاهده نمی‌شود.

۳-۴- نتایج ارزیابی در محاسبه مصرف انرژی

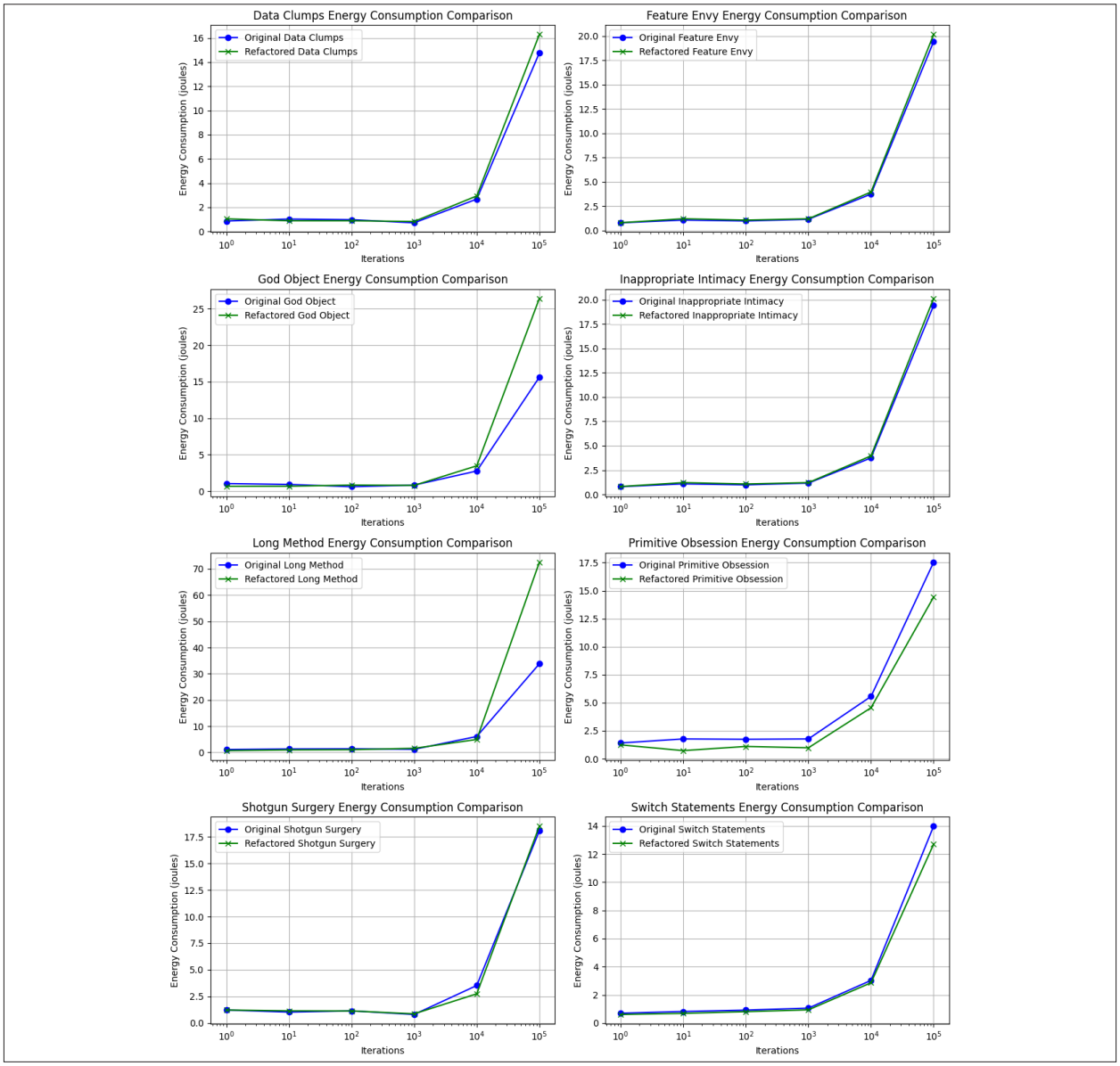
شکل ۲ شامل ۸ نمودار است که میزان مصرف انرژی را برای هر مطالعه موردی قبل و بعد از رفع نابسامانی نشان می‌دهند. نمودارها از راست به چپ و بالا به پایین به‌ترتیب توضیح داده می‌شوند.

نمودار اول[۱۸]: مصرف انرژی در دو نسخه تغییر محسوسی ندارد و تفاوتی در نمودار مشاهده نمی‌شود. این موضوع نشان می‌دهد که اصلاحات انجام شده در طراحی برنامه، تأثیری بر میزان انرژی مصرفی نداشته‌اند. هر چند نسخه بهینه شده از نظر ساختار کدنویسی و کاهش وابستگی‌های داخلی بهبود یافته است، اما این تغییرات نتوانسته‌اند بر مصرف انرژی تأثیرگذار باشند. دلیل این موضوع این است که عملیات پردازشی و دسترسی به داده‌ها در هر دو نسخه به شکلی مشابه اجرا می‌شوند و بهینه‌سازی طراحی تنها پیچیدگی منطقی را کاهش داده، نه مصرف انرژی را. با توجه به بهبود ساختار کد و کاهش وابستگی‌ها، رفع این نابسامانی همچنان توصیه می‌شود. این اصلاحات به‌ویژه در پروژه‌هایی که به تغییرات مکرر نیاز دارند، نگهداری کد را آسان‌تر می‌سازد.

نمودار دوم[۱۹]: در نسخه اصلی، اطلاعات کاربر به‌صورت پارامترهای مستقل پردازش می‌شود که می‌تواند باعث تکرار دسترسی به حافظه و پردازش جداگانه هر داده شود. این روش در تکرارهای کم تأثیری بر مصرف انرژی ندارد، اما در تکرارهای زیاد ممکن است به افزایش جزئی مصرف انرژی منجر شود. در نسخه بهینه، داده‌های کاربر در یک ساختار مجتمع ذخیره شده‌اند که مدیریت حافظه را ساده‌تر می‌کند و نیاز به دسترسی‌های مستقل را کاهش می‌دهد. این تجمیع در تکرارهای کم تفاوتی در مصرف انرژی ایجاد نمی‌کند، اما در تکرارهای بالا مصرف انرژی نسخه بهینه کمی بیشتر است، احتمالاً به دلیل سربار اضافی ناشی از ایجاد شیء باشد. با توجه به این‌که تغییر زیادی در مصرف انرژی مشاهده نمی‌شود، می‌توان نتیجه گرفت که این نوع نابسامانی بهتر است رفع گردد زیرا علاوه بر بهبود کیفیت کد، تاثیری منفی بر کارایی ندارد.

نمودار سوم[۲۰]: در نسخه اصلی، همه محاسبات مرتبط با تخفیف و پیام خوشامدگویی در یک کلاس انجام می‌شود، که باعث می‌شود بار پردازشی متراکم باشد. این ساختار ممکن است در مقیاس کم اثرات منفی بر مصرف انرژی نداشته باشد، اما با افزایش تعداد تکرارها، عملیات اضافی که به‌طور متمرکز انجام می‌شود، باعث افزایش مصرف انرژی می‌شود. در نسخه بهینه‌شده، اگرچه مسئولیت‌ها تفکیک شده‌اند، اما این تفکیک ممکن است باعث نیاز به فراخوانی‌های بیشتری میان کلاس‌ها شود که به معنای هزینه اضافی در پردازش و مصرف انرژی است. این افزایش فراخوانی‌ها می‌تواند منجر به مصرف بیشتر انرژی در نسخه بهینه شده نسبت به نسخه اصلی شود. با

۱۷- با عنوان Shotgun Surgery
۱۸-با عنوان Feature Envy
۱۹-با عنوان Data Clumps
۲۰-با عنوان Inappropriate Intimacy



شکل ۲. مقایسه مصرف انرژی در کدهای دارای نابسامانی و کدهای بهینه شده

توجه به این که این افزایش ناچیز است و با توجه به بهبود در ساختار کد و کاهش وابستگی‌ها، رفع این نابسامانی توصیه می‌شود. این اصلاحات به‌ویژه در پروژه‌هایی که نیاز به تغییرات مکرر دارند، نگهداری کد را ساده‌تر می‌کند

نمودار چهارم[۲۱]: در نسخه بهینه شده، تفکیک مسئولیت‌ها و سازمان‌دهی بهتر وظایف باعث شده است که در تکرارهای پایین، مصرف انرژی مشابه نسخه اصلی باقی بماند. اما در تکرار ۱۰ به توان ۵، مصرف انرژی نسخه بهینه به‌طور قابل توجهی افزایش یافته و بیشتر از نسخه اصلی شده است. این افزایش می‌تواند به‌دلیل سربار ناشی از فراخوانی‌های مکرر بین کلاس‌های مختلف و مدیریت اشیاء در نسخه اصلاح شده باشد که در بار پردازشی بالا منجر به مصرف انرژی بیشتر شده است.

نمودار پنجم[۲۲]: در نسخه اصلی، مصرف انرژی به دلیل عدم تفکیک مناسب مسئولیت‌ها و تعداد زیاد پردازش‌های غیرضروری افزایش می‌یابد. به ویژه در شرایط تکرار بالا (نظیر داده‌های بیشتر یا پردازش‌های سنگین‌تر)، این فرآیندها به‌طور مستقیم تأثیر منفی روی کارایی دارند. در

۲۱-با عنوان God Object

۲۲-با عنوان Primitive Obsession

نسخه بهینه، مسئولیت‌ها به‌طور صحیح بین چندین کلاس مختلف تقسیم شده و هر کلاس تنها یک کار خاص را انجام می‌دهد. این تفکیک مسئولیت‌ها موجب کاهش پیچیدگی و کاهش تعاملات غیرضروری می‌شود، که باعث کاهش مصرف انرژی در هر پردازش می‌شود. همچنین، در نسخه بهینه، داده‌ها به‌صورت مستقل و تخصصی‌تر مدیریت می‌شوند که می‌تواند منجر به پردازش سریع‌تر و بهینه‌تر شود. بدین‌ترتیب، در تکرارهای زیاد، این رویکرد باعث کاهش فشار روی پردازنده و صرفه‌جویی در مصرف انرژی می‌شود.

نمودار ششم[۲۳]: با افزایش تعداد تکرارها، مصرف انرژی در کد بهینه شده تا حدودی افزایش یافته که به‌دلیل اجرای هم‌زمان متدها و فراخوانی‌های مجزا بوده است. با این وجود، این بهینه‌سازی در صورت استفاده کم یا متوسط از متدها، کارایی قابل‌قبولی دارد و خوانایی کد را افزایش می‌دهد.

در نسخه بهینه، تا ۱۰۰۰۰ تکرار، مصرف انرژی مشابه نسخه اصلی است. اما از آن به بعد، مصرف انرژی به‌طور ناگهانی افزایش می‌یابد. این افزایش می‌تواند به‌دلیل جدا شدن فراخوانی‌های مختلف متدهای مجزا باشد. در نسخه اصلی، شرط‌ها به‌صورت تودرتو در یک متد قرار دارند که باعث می‌شود تعداد فراخوانی‌های متد کمتر و مصرف انرژی پایین‌تر باشد. اما در نسخه بهینه، تفکیک این شرط‌ها به متدهای جداگانه باعث افزایش تعداد فراخوانی‌ها و در نتیجه، مصرف بیشتر انرژی می‌شود. این اثر در تکرارهای بالاتر به‌وضوح مشهود است، زیرا با افزایش تعداد تکرارها، سیستم به تعداد بیشتری از متدهای مجزا نیاز دارد که باعث افزایش مصرف انرژی می‌شود.

نمودار هفتم[۲۴]: در نسخه بهینه شده، مصرف انرژی کاهش یافته است زیرا دستور سوییچ باعث ساده‌تر شدن جریان کنترل برنامه شده و نیاز به انجام محاسبات اضافی یا بررسی‌های پیچیده کاهش یافته است. این بهبود در ساختار کد باعث می‌شود که پردازش‌ها سریع‌تر و بهینه‌تر انجام شوند، بنابراین تعداد عملیات پردازشی کاهش می‌یابد و در نتیجه مصرف انرژی پایین‌تر می‌آید. کد بهینه شده انرژی کمتری را برای پردازش تصمیمات پیچیده مصرف می‌کند، چرا که تصمیم‌گیری‌ها به شکل مستقیم و مؤثرتری صورت می‌گیرد.

نمودار هشتم[۲۵]: در نسخه بهینه‌شده، مصرف انرژی کمی بهتر از نسخه اصلی است. این بهبود ناشی از سازمان‌دهی بهتر وظایف و کاهش وابستگی‌های غیرضروری میان اجزای مختلف برنامه است. هر چند تقسیم عملیات به متدهای جداگانه و ایجاد کلاس‌های جدید برای وظایف خاص ممکن است باعث افزایش تعداد فراخوانی‌ها و مدیریت اشیاء شود، اما بهینه‌سازی در طراحی کلی و کاهش پیچیدگی‌های مرتبط با وابستگی‌ها توانسته تأثیر این عوامل را جبران کرده و مصرف انرژی را کاهش دهد.

۵. بحث

با توجه به تحلیل‌های ارائه شده و بررسی تأثیر رفع نابسامانی‌ها بر زمان اجرا و مصرف انرژی، می‌توان به این نتیجه رسید که رفع برخی نابسامانی‌ها به بهبود پایداری سیستم کمک می‌کند، در حالی که برخی دیگر ممکن است تأثیرات منفی یا خنثی داشته باشند. در زیر، هر نابسامانی بررسی و توصیه کلی برای رفع یا عدم رفع آن ارائه شده است.

رفع نابسامانی زیاده‌خواهی ویژگی باعث کاهش پیچیدگی کد و بهبود نگهداری آن می‌شود. هر چند تأثیر قابل توجهی بر زمان اجرا یا مصرف انرژی در تکرارهای بالا مشاهده نمی‌شود، اما بهبود ساختار کد و کاهش وابستگی‌ها به‌وضوح مفید است. در نتیجه، رفع این نابسامانی توصیه می‌شود، زیرا به بهبود کیفیت کد، کاهش پیچیدگی و افزایش نگهداشت کمک می‌کند.

با تحلیل نتایج مجموعه داده می‌توان گفت تجمیع داده‌ها

۲۳-با عنوان Long Method
۲۴-با عنوان Switch Statement
۲۵-با عنوان Shotgun Surgery

در یک ساختار مجتمع، مدیریت حافظه را ساده‌تر می‌کند، اما در تکرارهای بالا ممکن است به‌دلیل سربار ایجاد اشیاء و عملیات اضافی باعث افزایش مصرف انرژی و زمان اجرا شود. رفع این نابسامانی به‌دلیل بهبود کیفیت کد و کاهش پیچیدگی توصیه می‌شود، اما باید در پروژه‌هایی با تکرارهای زیاد و سربار پردازشی بالا با دقت بیشتری انجام شود.

در رفع نابسامانی روابط نامناسب، تفکیک مسئولیت‌ها به بهبود ساختار کد و کاهش وابستگی‌ها کمک می‌کند، اما ممکن است در تکرارهای بالا به‌دلیل فراخوانی‌های بیشتر میان کلاس‌ها، باعث افزایش زمان اجرا و مصرف انرژی شود. رفع این نابسامانی توصیه می‌شود، به‌ویژه در پروژه‌هایی که تغییرات مکرر و نگهداری آسان‌تر کد اهمیت دارند.

رفع نابسامانی شیء خداوند تفکیک وظایف از یک شیء مرکزی به کلاس‌های جداگانه در تکرارهای پایین عملکرد بهتری دارد، اما در تکرارهای زیاد به دلیل سربار ناشی از مدیریت اشیاء و فراخوانی‌های مکرر، باعث افزایش قابل توجه زمان اجرا و مصرف انرژی می‌شود. رفع این نابسامانی نیازمند بررسی دقیق است. در صورت امکان، از راهبردهای بهینه‌سازی بیشتر برای کاهش سربار فراخوانی‌ها و مدیریت اشیاء استفاده شود.

با تحلیل نتایج نابسامانی استفاده افراطی از نوع دادهٔ اولیه می‌توان نتیجه گرفت، تفکیک مسئولیت‌ها و استفاده از اشیاء به‌جای داده‌های اولیه باعث کاهش پیچیدگی و بهبود زمان اجرا و مصرف انرژی به‌ویژه در تکرارهای بالا می‌شود. در نتیجه، رفع این نابسامانی توصیه می‌شود، چرا که به بهبود ساختار کد، کاهش پیچیدگی و افزایش کارایی کمک می‌کند.

تفکیک متدهای طولانی خوانایی کد را افزایش می‌دهد، اما در تکرارهای بالا ممکن است باعث افزایش زمان اجرا به دلیل تعداد فراخوانی‌های بیشتر شود. رفع این نابسامانی در پروژه‌هایی با پیچیدگی متوسط یا نیاز به خوانایی بیشتر توصیه می‌شود، اما در پروژه‌هایی با تکرارهای بالا باید با دقت بیشتری انجام شود.

بهینه‌سازی نابسامانی دستور سوییچ باعث کاهش زمان اجرا و مصرف انرژی می‌شود، زیرا جریان کنترل برنامه ساده‌تر و کارآمدتر شده است. رفع این نابسامانی توصیه می‌شود، زیرا به بهبود کارایی و کاهش مصرف منابع کمک می‌کند.

در رفع جراحی ساچمه‌ای، تقسیم وظایف باعث کاهش وابستگی‌های غیرضروری و بهبود سازمان‌دهی کد می‌شود. هر چند تغییرات عمده‌ای در زمان اجرا مشاهده نمی‌شود، اما مصرف انرژی کمی کاهش یافته است. رفع این نابسامانی در شرایطی که وابستگی‌های زیاد مانع نگهداری کد می‌شوند، توصیه می‌شود.

۱-۵- اولویت‌بندی نابسامانی‌ها

رفع نابسامانی‌های زیر به‌طورکلی توصیه می‌شود، زیرا علاوه بر بهبود کیفیت کد و خوانایی، در پایداری سیستم نیز تأثیر مثبت دارند:

- زیاده‌خواهی ویژگی: کاهش پیچیدگی و بهبود نگهداری.
- استفاده افراطی از نوع داده‌ی اولیه: کاهش زمان اجرا و مصرف انرژی.
- دستور سوییچ: بهبود کارایی و کاهش مصرف منابع.
- جراحی ساچمه‌ای: بهبود سازمان‌دهی و کاهش وابستگی‌ها.
- رفع نابسامانی‌های زیر نیز توصیه می‌شود، اما نیازمند بررسی دقیق‌تر است تا تأثیرات منفی احتمالی در پروژه‌های بزرگ و با تکرارهای زیاد کاهش یابد:
- مجموعه داده: بهبود کیفیت کد با مدیریت سربار.
- روابط نامناسب: کاهش وابستگی‌ها با بهینه‌سازی بیشتر.
- شیء خداوند: مدیریت سربار ناشی از تعاملات میان کلاس‌ها.
- متد طولانی: تعادل میان خوانایی و کارایی.

۲-۵- پاسخ به سوالات تحقیق

سوال یک: اصلاح نابسامانی‌های کد چگونه بر مصرف انرژی برنامه‌های جاوا تأثیر می‌گذارد؟

کاهش مصرف انرژی: نابسامانی‌هایی مانند «دستور سوئیچ» و « استفاده افراطی از نوع دادهٔ اولیه» پس از بازآرایی کد، کاهش قابل‌توجهی در مصرف انرژی نشان داده‌اند. برای مثال، در نابسامانی «دستور سوئیچ»، جایگزینی تصمیم‌گیری‌های پیچیده با الگوهای مناسب منجر به کاهش عملیات پردازشی غیرضروری و در نتیجه، مصرف کمتر انرژی شده است.

افزایش مصرف انرژی در تکرارهای بالا: در نابسامانی‌هایی مانند «شیء خداوند» و «روابط نامناسب»، به‌دلیل افزایش تعاملات بین کلاس‌ها و سربار ناشی از مدیریت اشیاء جدید، مصرف انرژی در تکرارهای بالا بیشتر شده است. برای مثال، در نابسامانی «شیء خداوند»، انتقال وظایف به کلاس‌های متعدد باعث افزایش هزینه‌های پردازشی در بارهای سنگین شده است.

تأثیر خنثی: برای برخی نابسامانی‌ها مانند «زیاده‌خواهی ویژگی»، اصلاحات انجام‌شده مصرف انرژی را در مقیاس‌های کوچک بهبود داده‌اند اما در تکرارهای زیاد تغییر محسوسی مشاهده نشده است.

سوال دو: کدام نابسامانی‌ها بیشترین تأثیر را بر زمان اجرای برنامه‌های جاوا دارند؟

افزایش زمان اجرا: اصلاح «مجموعه داده‌ها»، به‌دلیل سربار ناشی از ایجاد اشیاء جدید و فراخوانی‌های مکرر، منجر به افزایش زمان اجرا شده است. در اصلاح «شیء خداوند»، تفکیک وظایف این کلاس در تکرارهای زیاد باعث افزایش زمان اجرا به‌دلیل افزایش تعداد فراخوانی‌ها و مدیریت منابع شده است. با اصلاح «متد طولانی» در تکرارهای بالا، جدا شدن عملیات‌های مختلف به متدهای مجزا باعث افزایش قابل‌توجه زمان اجرا شده است.

کاهش زمان اجرا: اصلاح «دستور سوئیچ» با کاهش پیچیدگی تصمیم‌گیری، زمان اجرا را بهبود داده است. در اصلاح «جراحی ساچمه‌ای»، تقسیم وظایف به کلاس‌های تخصصی و کاهش وابستگی‌ها باعث کاهش زمان اجرا در نسخه بهینه شده است.

سوال سه: آیا رفع نابسامانی‌های کد می‌تواند به افزایش پایداری نرم‌افزار کمک کند؟

بهبود پایداری: نابسامانی‌هایی مانند «زیاده‌خواهی ویژگی»، «دستور سوئیچ» و «جراحی ساچمه‌ای» پس از اصلاح، ساختار کد را بهبود داده، وابستگی‌ها را کاهش داده و نگهداشت نرم‌افزار را ساده‌تر کرده‌اند. این اصلاحات تأثیر مثبتی بر پایداری نرم‌افزار در بلندمدت خواهند داشت

چالش‌های پایداری در مقیاس بزرگ: در نابسامانی «شیء خداوند» و «روابط نامناسب»، اگرچه اصلاحات باعث افزایش خوانایی و کاهش پیچیدگی شده است، اما سربارهای پردازشی در تکرارهای بالا می‌توانند نگهداشت نرم‌افزار را در مقیاس‌های بزرگ پیچیده‌تر کنند. بنابراین، استفاده از بهینه‌سازی‌های تکمیلی برای کاهش سربارها توصیه می‌شود.

۵. نتیجه‌گیری

در این پژوهش، تأثیر رفع نابسامانی‌های مختلف کد از جمله شیء خداوند، متد طولانی، استفاده افراطی از نوع دادهٔ اولیه و موارد دیگر بر مصرف انرژی و زمان اجرای برنامه‌های جاوا بررسی شد. نتایج نشان داد که اصلاح نابسامانی‌ها در بیشتر موارد باعث بهبود ساختار کد، افزایش خوانایی و کاهش پیچیدگی می‌شود. این تغییرات منجر به نگهداری آسان‌تر و افزایش پایداری سیستم می‌گردد. با این حال، در برخی موارد مانند نابسامانی شیء خداوند و متد طولانی، اصلاحات موجب افزایش سربار پردازشی و در نتیجه افزایش زمان اجرا و مصرف انرژی در شرایط خاص شدند، که این امر نیازمند استفاده از راهبردهای تکمیلی بهینه‌سازی است.

به‌طورکلی، یافته‌ها حاکی از آن است که رفع نابسامانی‌ها با وجود تأثیرات مختلف بر مصرف منابع، به‌ویژه در تکرارهای زیاد، برای پروژه‌هایی با نیاز به نگهداری آسان‌تر و تغییرات مکرر کد ضروری است.

بهبودهای ایجاد شـــده در کیفیت کد و کاهش وابستگی‌ها نشـــان‌دهنده اهمیت این روش‌ها در طراحی نرم‌افزارهای پایدار و کم‌مصرف است. با این حال، تحلیل نتایج نشان داد که برای برخی نابســـامانی‌ها، نیاز به بررسی‌های دقیق‌تر و به‌کارگیری تکنیک‌های تکمیلی بهینه‌ســـازی جهت کاهش سربار ضروری است.

در تحقیقـــات آتی می‌تـــوان رویکردهـــای خـــودکار و هوشـــمند برای شناسایی و رفع نابسامانی‌های کد توسعه داد. همچنیـــن، بـــررسی تأثیر ایـــن اصلاحات بر ســـایر شـــاخص‌های کیفیت نرم‌افزار نظیر امنیت و مقیاس‌پذیری، می‌تواند مسیر جدیدی برای بهبود نرم‌افزارهای مدرن باز کند. به‌کارگیری تکنیک‌های یادگیری ماشین برای پیش‌بینی تأثیرات بازســـازی و استفاده از روش‌های بهینه‌سازی در مقیاس بزرگ‌تر نیز از دیگر زمینه‌های پژوهش پیشنهادی است.

مراجع

1. A. Guldner, R. Bender, C. Calero, G. S. Fernando, M. Funke, J. Gröger, L. M. Hilty, J. Hörnschemeyer, G.-D. Hoffmann, D. Junger, T. Kennes, S. Kreten, P. Lago, F. Mai, I. Malavolta, J. Murach, K. Obergöker, B. Schmidt, A. Tarara, J. P. De Veaugh-Geiss, S. Weber, M. Westing, V. Wohlgemuth, and S. Naumann, "Development and evaluation of a reference measurement model for assessing the resource and energy efficiency of software products and components—Green Software Measurement Model (GSMM)," Future Generation Computer Systems, vol. 155, pp. 402–418, 2024.
2. A. Gupta, B. Suri, and S. Misra, "A systematic literature review: code bad smells in java source code," in Computational Science and Its Applications–ICCSA 2017: 17th International Conference, Trieste, Italy, July 3-6, 2017, Proceedings, Part V, vol. 17, pp. 665-682. Springer International Publishing, 2017.
3. M. Zhang, T. Hall, and N. Baddoo, "Code bad smells: a review of current knowledge," Journal of Software Maintenance and Evolution: Research and Practice, vol. 23, no. 3, pp. 179-202, 2011.
4. G. Lacerda, F. Petrillo, M. Pimenta, and Y. G. Guéhéneuc, "Code smells and refactoring: A tertiary systematic review of challenges and observations," Journal of Systems and Software, vol. 167, p. 110610, 2020.
5. A. Imran, T. Kosar, J. Zola, and F. Bulut, "Predicting the impact of batch refactoring code smells on application resource consumption," in Proc. ACM Conference, 2023.
6. T. Armand, R. Rajan, and A. T. Idgunji, "MLPerf Power: Benchmarking the energy efficiency of machine learning systems," arXiv preprint, arXiv: 2410. 12032, 2024.
7. C. Bree and D. Ó Cinnéide, "Weighted metrics for the development of energy efficient software," in Proc. ACM Designing, vol. 24, pp. 1–6, 2024.
8. R. Verma, K. Kumar, and H. K. Verma, "A study of relevant parameters influencing code smell prioritization in object-oriented software systems," in Proc. IEEE 6th International Conference on Signal Processing, Computing and Control (ISPCC), Solan, India, 2021, pp. 150–154, doi: 10.1109/ISPCC53510.2021.9609478.
9. P. S. Yadav, R. S. Rao, and A. Mishra, "An evaluation of multi-label classification approaches for method-level code smells detection," IEEE Access, vol. 9, pp. 3387856–3387870, 2024, doi: 10.1109/ACCESS.2024.3387856.
10. A. C. B. Ribeiro, "Effects of applying energy efficient patterns," M.S. thesis, Instituto Superior de Engenharia do Porto, Portugal, 2023.
11. Z. Ournani, R. Rouvoy, P. Rust, and J. Penhoat, "Comparing the energy consumption of Java I/O libraries and methods," in Proc. 37th International Conference on Software Maintenance and Evolution (ICSME), 2021, pp. 1–12.
12. J.-K. Hong, "Metrics for code quality check in SEED_mode.c," International Journal of Internet, Broadcasting and Communication, vol. 16, no. 3, pp. 184–191, 2024, doi: 10.7236/IJIBC.2024.16.3.184.
13. A. Abdou and N. Darwish, "Severity classification of software code smells using machine learning techniques: A comparative study," Journal of Software Evolution and Process, e2454, 2022, doi: 10.1002/smr.2454.
14. C. Bree and D. Ó Cinnéide, "Energy efficiency of the visitor pattern: Contrasting Java and C++ implementations," Empirical Software Engineering, vol. 28, no. 1, pp. 145, 2023, doi: 10.1007/s10664-023-10387-8.
15. A. Gupta, B. Suri, D. Sharma, S. Misra, and L. Fernandez-Sanz, "Code smells analysis for android applications and a solution for less battery consumption," Scientific Reports, vol. 14, no. 1, p. 18194, 2024.
16. R. Verma, K. Kumar, and H. K. Verma, "Code smell prioritization in object-oriented software systems: A systematic literature review," Journal of Software Evolution and Process, e2536, 2023, doi: 10.1002/smr.2536.
17. H. Oumarou and Kolyang, "A source-code maintainability evaluation model for software products," Science, Engineering and Technology, vol. 3, no. 2, pp. 97–105, 2023, doi: 10.54327/set2023/v3.i2.73.
18. K. Alkharabsheh, S. Alawadi, Y. Crespo, and J. Á. Taboada, "Exploring the role of project status information in effective code smell detection," Cluster Computing, vol. 28, no. 1, pp. 29, 2025, doi: 10.1007/s10586-024-04724-9.
19. نســـرین مقصودی شـــقاقی، لیلا صمیمی دهکردی، عبـــاس حری، ۱۴۰۱، شناســـایی نابســـامانی‌های کد به کمک مهندسی معکوس مدل‌رانده، ششـــمین کنفرانس ملی کاربرد فناوری‌های نوین در علوم مهندسی، تربت حیدریه.