Passive Defence
Vol. 16, No. 1, Spring 2025, Serial No. 61
ISSN: 2980-8030 , E-ISSN: 2008-6849

A Survey and Evaluating Types of Code Amells in Software Pprogram Code Rrefactoring Process

Ali Karimi* , Ali Tolui far , Farhad Karimi
*Assistant Professor, Department of Computer Engineering, Imam Hossein (AS) University, Tehran, Iran
(Received: 01/06/2024, Revised: 24/08/2024, Accepted: 23/02/2025, Published: 19/04/2025)
DOR: 20.1001.1.20086849.1404.16.1.2.6

ABSTRACT

Code smells are common characteristics in programming that may indicate the presence of an issue within the program. Detecting and identifying code smells is one of the key factors in software development, as it can improve quality of the program and make it easier to maintain and extend the code over time. Passive defense in software refers to a set of measures taken to increase the security and reduce the vulnerability of software against threats. These measures include secure design, the use of appropriate architectural patterns, and avoiding unnecessary complexity in the software code. It is obvious that one of the most effective ways to increase the quality of the software is to reconstructing or refactoring the code, which has a direct relationship with identification and repair of code smells. So far, a lot of research has been done in the field of identifying and removing code smells of software systems. However, among them, four types of code smell include; Long method, feature envy, large class and data class have attracted the most attention of researchers. In this article, by reviewing 58 code smells found in the software source code, they are classified into eight categories: Bloaters, Abusers, Change preventers, Dispensables, Couplers, Obfuscators, Data dealers and other code smells that are not placed in the mentioned categories, and while examining each one, they are presented as a new taxonomy. This article helps developers and software development teams to produce high quality and more efficient softwares by identifying and resolving code smells.

Keywords: Code Smell, Improving Software Quality, Software Engineering, Software Development, Software Code

This article is an open-access article distributed under the terms and conditions of the Creative Commons Attribution (CC BY) license.

Publisher: Imam Hussein University Authors

* Corresponding Author Email: a.karimi@ihu.ac.ir

پدافند غیرعامل

سال پانزدهم، شماره ۱، بهار ۱۴۰۴، (پیاپی ۶۱): صص ۳۲-۱۱

شاپای چاپی: ۶۹۴۹-۲۰۰۸ | شاپای الکترونیکی: ۸۰۳۰-۲۹۸۰

علمی – پژوهشی

بررسی و ارزیابی انواع کدهای مشکوک در فرآیند بازآرایی کد برنامه نرم‌افزاری

علی کریمی[۱*]، علی طلوعی‌فر[۲]، فرهاد کریمی[۳]

DOR: 20.1001.1.20086849.1404.16.1.2.6

تاریخ دریافت: ۱۴۰۳/۰۳/۱۲ | تاریخ پذیرش: ۱۴۰۳/۱۲/۰۵

تاریخ بازنگری: ۱۴۰۳/۰۶/۰۳ | تاریخ انتشار: ۱۴۰۴/۰۱/۳۰

چکیده

کدهای مشکوک مشخصه‌های رایج برنامه‌نویسی هستند که ممکن است نشان‌دهنده وجود مشکل در برنامه باشند. شناسایی و حذف کدهای مشکوک یکی از عوامل کلیدی در توسعه نرم‌افزار است؛ زیرا می‌تواند کیفیت کد برنامه را بهبود بخشد و نگهداری و گسترش آن را در طول زمان تسهیل نماید. پدافند غیرعامل در نرم‌افزار به مجموعه اقداماتی اشاره دارد که برای افزایش امنیت و کاهش آسیب‌پذیری نرم‌افزار در مقابل تهدیدات، انجام می‌شود. این اقدامات شامل طراحی امن، استفاده از الگوهای معماری مناسب و پرهیز از پیچیدگی‌های غیرضروری در کد نرم‌افزار است. بدیهی است که یکی از روش‌های مؤثر در افزایش کیفیت نرم‌افزار، بازآرایی کد است که رابطه مستقیمی با شناسایی و ترمیم کدهای مشکوک دارد. تاکنون تحقیقات زیادی در حوزه شناسایی و برطرف کردن کدهای مشکوک سامانه‌های نرم‌افزاری انجام گرفته است. لیکن، از میان آنها چهار نوع کد مشکوک شامل؛ متد طولانی، ویژگی حسادت، کلاس بزرگ و کلاس داده بیشترین توجه محققین را به خود جلب کرده است. در این مقاله، ۵۸ کد مشکوک که در کد منبع نرم‌افزار یافت می‌شوند، بررسی شده و در هشت دسته‌بندی در قالب یک آرایه‌شناسی جدید ارائه می‌شوند. این مقاله به توسعه‌دهندگان و تیم‌های توسعه نرم‌افزار کمک می‌کند تا با شناسایی و رفع کدهای مشکوک، نرم‌افزارهای باکیفیت بالاتر و بهینه‌تر را ارایه نمایند.

کلیدواژه‌ها: کد مشکوک، بهبود کیفیت نرم‌افزار، مهندسی نرم‌افزار، توسعه نرم‌افزار، کد نرم‌افزار

---

[۱] استادیار، گروه مهندسی کامپیوتر، دانشگاه جامع امام حسین (ع)، تهران، ایران (a.karimi@ihu.ac.ir)- نویسنده مسئول

[۲] دانشجوی کارشناسی ارشد، مهندسی کامپیوتر گرایش نرم‌افزار، دانشگاه جامع امام حسین (ع)، تهران، ایران

[۳] پژوهشگر، گروه مهندسی کامپیوتر، دانشگاه جامع امام حسین (ع)، تهران، ایران

* این مقاله یک مقاله با دسترسی آزاد است که تحت شرایط و ضوابط مجوز Creative Commons Attribution (CC BY) توزیع شده است.

ناشر: دانشگاه جامع امام حسین (ع) © نویسندگان

۱- مقدمه

کد مشکوک (بوی کد)[1] نوعی ساختار کد[2] است که اصول طراحی را نقض می‌کند و بر کیفیت کد تأثیر منفی می‌گذارد و نشانه‌ای از طراحی ضعیف نرم‌افزار و عادت‌های بد کدنویسی محسوب می‌شود [۱]. به عبارتی، کد مشکوک یک ویژگی ساختاری نرم‌افزار است که نشان‌دهنده جنبه‌ای در کد یا طراحی است که می‌تواند در توسعه و نگهداری نرم‌افزار مشکل ایجاد کند [۲].

بر خلاف اشکالات در کد منبع[3]، یک برنامه با کد مشکوک می‌تواند به‌راحتی اجرا شود. این کدها، اگرچه تأثیر کوتاه‌مدتی بر اجرای برنامه ندارند، اما تأثیر طولانی‌مدت کد مشکوک باعث کاهش کیفیت نرم‌افزار شده و درک و نگهداری آن را دشوار می‌سازد [۱].

از آنجایی‌که مفهوم کد مشکوک در اواخر دهه ۱۹۹۰ مطرح شد، تحقیقات زیادی به‌طور مداوم در این حوزه توسط بخش‌های دانشگاهی و صنعتی انجام شده است [۳]. بیشتر کارها بر شدت [۴, ۵]، همبستگی [۶, ۷]، تشخیص [۸-۱۰] و اصلاح [۱۱, ۱۲] کدهای مشکوک متمرکز هستند. در میان این کارها، بیش از ۹۰٪ آنها، به تشخیص کدهای مشکوک وابسته هستند که نشان می‌دهد شناسایی و برطرف کردن آنها همیشه یک موضوع مهم در زمینه تکامل نرم‌افزار بوده است [۱۳].

برای شناسایی کدهای مشکوک، محققان از هر دو روش تشخیص دستی و خودکار استفاده می‌کنند. تشخیص زودهنگام کدهای مشکوک به‌صورت دستی انجام می‌شود. بااین‌حال، این رویکرد نه‌تنها به تخصص کافی نیاز دارد، بلکه نیازمند زمان و تلاش زیادی است. همچنین این رویکرد، از دقت شناسایی قابل‌قبولی برخوردار نیست [۱].

بنابراین، اکثر محققان رویکردهای خودکار را برای شناسایی سریع و دقیق کدهای مشکوک ترجیح می‌دهند. از زمانی که اولین ابزار تشخیص خودکار کدهای مشکوک معرفی شد، بسیاری از ابزارهای معروف (مانند DECOR، JDedorant، iPlasma و غیره) برای بهبود دقت تشخیص کدهای مشکوک پیشنهاد شدند [۱].

همان‌طور که تجربه نشان می‌دهد، پروژه‌های طولانی و نیمه‌طولانی گاهی ممکن است به مدت چند سال یا بیشتر نیز ادامه یابند؛ بنابراین، هنگامی که در مرحله راه‌اندازی یا توسعه، متوجه می‌شویم که چیزی ممکن است اشتباه باشد، بسیار مهم است که از وجود کدهای مشکوک آگاه باشیم و آنها را هرچه سریع‌تر برطرف نماییم.

چالش‌های زیادی در تشخیص کدهای مشکوک برای توسعه‌دهندگان نرم‌افزار وجود دارد. انواع مختلف کدهای مشکوک این فرآیند را دشوار می‌کند. عدم تعریف رسمی از مفهوم کد مشکوک چالش دیگری است که توسعه‌دهندگان نرم‌افزار با آن مواجه هستند [۱۵].

از زمانی که مفهوم کد مشکوک توسط فاولر[4] تعریف شد، فعالیت‌های زیادی در زمینه‌های علمی، عملی و آموزشی برای شناسایی و رفع آنها انجام گرفت. به‌عنوان‌مثال، برخی کتاب‌ها با پیشنهادهای جدید درباره کدهای مشکوک و روش‌های مناسب بازآرایی آنها معرفی شدند، مقالات علمی جدیدی تلاش کردند تا یک چارچوب مشخص‌تر (تعاریف، پیش‌بینی‌ها، تأثیر) را ارائه دهند و از فنون یادگیری ماشین برای شناسایی آنها استفاده کنند [۱۴].

با توجه به مطالعات زیاد انجام‌شده در حوزه کدهای مشکوک و با بررسی مقالات پیشین، آرایه‌شناسی[5] انواع کدهای مشکوک در هشت دسته‌بندی در شکل (۱) و (۲) ارائه شده است. در این مقاله، هشت دسته‌بندی مذکور به همراه زیربخش‌های هرکدام، جداگانه بررسی شده‌اند.

بخش باقی‌مانده مقاله به شرح زیر تنظیم شده است. بخش دوم شامل مروری بر کارهای تحقیقاتی انجام‌شده توسط دیگر پژوهشگران می‌باشد. بخش سوم، مفهوم کد مشکوک را مورد بحث قرار می‌دهد. بخش چهارم شامل بررسی سلسله‌مراتب و تعاریف مشکوک بودن[6] است. در بخش پنجم، هشت دسته‌بندی انواع کدهای مشکوک به همراه بررسی زیربخش‌های هر کدام، ارائه شده است. در بخش ششم، مفهوم بازآرایی شرح داده شده است و در نهایت، بخش هفتم شامل نتیجه‌گیری حاصل از مفاهیم بیان‌شده در این مقاله است.

---

[1] Code smell
[2] Code structure
[3] Source code
[4] Fowler
[5] Taxonomy
[6] Smell Hierarchies and Definitions

۲- پیشینه تحقیق

ژانگ و همکاران [۱] در سال ۲۰۲۴، ۸۶ مقاله تشخیص کدهای مشکوک براساس یادگیری ماشین نظارت‌شده از ژانویه ۲۰۱۰ تا آوریل ۲۰۲۳ را جمع‌آوری کردند. در این پژوهش، آنها در مجموع ۷ سوال تحقیقاتی را از جنبه‌های مختلف مانند ساخت مجموعه داده‌ها، پیش‌پردازش داده‌ها، انتخاب ویژگی، آموزش مدل و غیره به‌صورت تجربی ارزیابی کردند. آنها نتیجه گرفتند که کارهای موجود با مشکلاتی مانند عدم‌تعادل نمونه‌ها، تمرکز متفاوت بر روی انواع کدهای مشکوک و انتخاب ویژگی محدود روبرو هستند.

عبدو و درویش[1][۲] در سال ۲۰۲۴، در پژوهش خود بر اندازه‌گیری طبقه‌بندی شدت کدهای مشکوک بسته به چندین مدل یادگیری ماشین مانند مدل‌های رگرسیون[2]، مدل‌های چندجمله‌ای[3] و مدل‌های طبقه‌بندی ترتیبی[4] تمرکز کردند. آنها از الگوریتم توضیح‌های آگنوستیک مدل قابل تفسیر محلی[5] (LIME) بیشتر برای توضیح پیش‌بینی‌ها و قابلیت تفسیر مدل یادگیری ماشین استفاده کردند. همچنین، آنها قوانین پیش‌بینی تولیدشده توسط الگوریتم نظریه تشدید انطباقی تصویری[6] (PART) را استخراج کردند تا اثربخشی استفاده از معیارهای نرم‌افزاری برای پیش‌بینی کدهای مشکوک را بررسی کنند. نتایج آزمایش‌ها نشان داد که دقت مدل طبقه‌بندی شدت نسبت به خط‌پایه افزایش یافته و همبستگی رتبه‌بندی بین مدل پیش‌بینی‌شده و واقعی با استفاده از معیار همبستگی اسپیرمن[7] به ۰/۹۷ - ۰/۹۲ می‌رسد.

سندوکا و الجمان[8] [۱۷] در سال ۲۰۲۳، مجموعه داده کدهای مشکوک زبان پایتون را برای کدهای مشکوک کلاس بزرگ[9] و متد طولانی[10] تولید کردند. مجموعه داده ساخته‌شده شامل ۱۰۰۰ نمونه برای هر کد مشکوک، با ۱۸ ویژگی استخراج‌شده از کد منبع بود. همچنین، عملکرد تشخیص شش مدل یادگیری ماشین را به عنوان خطوط‌پایه در تشخیص کد مشکوک زبان پایتون بررسی کردند. خطوط‌پایه براساس معیارهای دقت[11] و ضریب همبستگی متیوز[12] (MCC) ارزیابی شدند. نتایج نشان‌دهنده برتری گروه جنگل تصادفی[13] در شناسایی کد مشکوک کلاس بزرگ در پایتون با دستیابی به بالاترین عملکرد تشخیص نرخ MCC، ۰/۷۷ بود، در حالی که درخت تصمیم[14] با دستیابی به بالاترین نرخ MCC برابر با ۰/۸۹ بهترین مدل در تشخیص کد مشکوک متد طولانی پایتون بود.

واتاناپاکورن[15] و همکاران [۱۸] در سال ۲۰۲۲، پژوهش خود را در رابطه با تشخیص کدهای مشکوک متد طولانی، تعداد پارامترهای طولانی[16]، کلاس بزرگ، لیست طولانی از کلاس‌های پایه[17] و زنجیره دامنه طولانی[18] در هر دو سطح کلاس و متد، با استفاده از انتخاب ویژگی مبتنی بر همبستگی[19] و انتخاب گام‌به‌گام رگرسیون لجستیک به جلو[20] (شرطی) برای بهبود عملکرد مدل مبتنی بر یادگیری ماشین برای برنامه‌های زبان پایتون انجام دادند. آنها هشت مدل یادگیری ماشین را با یک مجموعه داده مبتنی بر ۱۱۵ پروژه منبع‌باز زبان پایتون، ۳۹ معیار نرم‌افزار در سطح کلاس و ۲۲ معیار نرم‌افزار در سطح متد، آموزش دادند. نتایج پژوهش آنها نشان داد که روش یادگیری ماشین هنگام شناسایی متد طولانی و لیست طولانی از کلاس‌های پایه، دقت ۹۹/۷۲٪ را به دست آورد.

عبدو و درویش [۱۹] در سال ۲۰۱۸، یک مطالعه تطبیقی بین یادگیرنده‌های تکی و گروهی را برای پیش‌بینی نقص نرم‌افزار پیشنهاد کردند. روش نمونه‌برداری مجدد SMOTE به‌عنوان یک مرحله پیش‌پردازش برای حل مشکل عدم‌تعادل داده‌ها استفاده شده است. فنون مختلف یادگیری مجموعه‌ای برای پیش‌بینی نقص‌های نرم‌افزار استفاده شده است. نتایج نشان داد که یادگیری مجموعه‌ای با استفاده از جنگل چرخشی[21] بهتر از سایر فنون مجموعه‌ای است.

---
1 Abdou and Darwish
2 Regression models
3 Multinominal models
4 Ordinal classification models
5 Local Interpretable Model Agnostic Explanations (LIME) algorithm
6 Projective Adaptive Resonance Theory (PART) algorithm
7 Spearman's correlation measure
8 Sandouka and Aljamaan
9 Large Class
10 Long Method
11 Accuracy
12 Matthews correlation coefficient (MCC)
13 Random Forest
14 Decision Tree
15 Vatanapakorn
16 Long parameter list
17 Long base class list
18 Long scope chaining
19 Correlation-based feature selection (CFS)
20 logistic regression-forward stepwise
21 Rotation forest

شکل (۱): آرایه‌شناسی انواع کدهای مشکوک - بخش اول

شکل (۲): آرایه‌شناسی انواع کدهای مشکوک - بخش دوم

فونتانا[1] و همکاران [۴] در سال ۲۰۱۶، مدل پیش‌بینی را با استفاده از ۳۲ رویکرد یادگیری ماشینی نظارت‌شده (۱۶ الگوریتم اصلی و فنون تقویت آنها) برای تشخیص کد مشکوک ساختند. آرسلی فونتانا بر روی چهار نوع کد مشکوک (کلاس بزرگ، کلاس داده، کلاس طولانی و ویژگی حسادت) تمرکز کرد و آزمایش‌های گسترده‌ای روی ۷۴ سیستم برای انتخاب بهترین الگوریتم‌ها با بهترین پارامترها برای تشخیص کدهای مشکوک انجام داد. نتایج نشان داد که الگوریتم‌های آزمایش‌شده مانند J48، بیز ساده[2] و جنگل تصادفی عملکرد بالایی را به دست آوردند که به محدوده ۹۶/۶۴٪ تا ۹۹/۰۲٪ دقت بدون توجه به

[1] Fontana

[2] Naive Bayes

انواع کد مشکوک رسید.

بسیاری از محققان مرورهایی در مورد کد مشکوک انجام داده‌اند. سوبرینیو[1] و همکاران [۲۰] در سال ۲۰۱۸، ۳۵۱ مقاله تحقیقاتی مرتبط با کدهای مشکوک منتشرشده بین سال‌های ۱۹۹۰ تا ۲۰۱۷ را از پنج منظر، شامل انواع کد مشکوک، تکامل توجه پژوهشگران، تنظیمات آزمایشی، پژوهشگران/تیم‌هایی که در حوزه کد مشکوک فعالیت می‌کنند و توزیع نشریه مقاله، تجزیه‌وتحلیل کردند.

برخی از محققان مشکلات مربوط به کدهای مشکوک را از جنبه‌های مختلف بررسی و پیشرفت تحقیقات مربوطه را در سال‌های اخیر مرتب کرده‌اند. تیان و همکاران[2] [۲۱] در سال ۲۰۲۳، به‌طور سیستماتیک بیش از ۳۰۰ مقاله مرتبط با کدهای مشکوک منتشرشده از سال ۱۹۹۰ تا ژوئن ۲۰۲۰ را تجزیه‌وتحلیل و طبقه‌بندی کردند. آنها روند توسعه کد مشکوک را تجزیه‌وتحلیل کردند، به‌طور کمّی جریان اصلی و نقاط مهم تحقیقات مرتبط را نشان دادند، عوامل کلیدی کدهای مشکوک مورد توجه دانشگاه را شناسایی و همچنین تفاوت‌های نگرانی‌ها بین صنعت و دانشگاه را بررسی کردند.

برخی از پژوهشگران بررسی‌های سیستماتیکی درباره تشخیص کد مشکوک مبتنی بر یادگیری ماشین انجام داده‌اند. کارام و همکاران[3] [۲۲] در سال ۲۰۱۹، ۲۶ مقاله را از سال ۱۹۹۰ تا دسامبر ۲۰۱۶ برای تجزیه‌وتحلیل آماری استفاده از تشخیص کدهای مشکوک مبتنی بر یادگیری ماشین و مقایسه عملکرد فنون مختلف یادگیری ماشین طبقه‌بندی کردند. عظیم و همکاران[4] [۲۳] در سال ۲۰۱۹، در مجموع ۱۵ مقاله را در مورد تشخیص کدهای مشکوک مبتنی بر یادگیری ماشین بین سال‌های ۲۰۰۰ تا ۲۰۱۷ مورد بررسی قرار دادند و در مورد کدهای مشکوک مورد بررسی، پیکربندی مدل یادگیری ماشین، طراحی استراتژی ارزیابی و عملکرد به‌دست‌آمده توسط مدل، تحلیلی سیستماتیک انجام دادند.

۳-کد مشکوک

توسعه نرم‌افزارهای با کیفیت بالا در حوزه مهندسی نرم‌افزار از اهمیت زیادی برخوردار است. امروزه سامانه‌های نرم‌افزاری از نظر اندازه و پیچیدگی در حال رشد هستند، در نتیجه حفظ کیفیت مطلوب آنها یکی از مهمترین مشکلات پیش‌روی صنعت نرم‌افزار است [۲۴]. وقتی صحبت از نگهداری نرم‌افزارهای بزرگ و پیچیده می‌شود، کدهای مشکوک یک چالش واقعی است [۲۵]. آنها ناهنجاری‌های طراحی هستند که نرم‌افزار را پیچیده کرده و مدیریت و درک آن را دشوار می‌کنند و منجر به هزینه‌های مالی برای توسعه و نگهداشت نرم‌افزار می‌شوند [۲۶].

به‌طور کلی و بنا به گفته فاولر [۲۷]، «کد مشکوک یک نشانه سطحی است که معمولاً مربوط به یک مشکل عمیق‌تر در سیستم است» و آنها مشخصه‌هایی در کد منبع نرم‌افزار هستند. در حالی که کد مشکوک ممکن است همیشه نشان‌دهنده یک مشکل جدی خاص نباشد، اغلب منجر به کشف این مسائل می‌شود [۲۸]. ابزارهای مختلف شناسایی کدهای مشکوک براساس بصری‌سازی، رویکرد مبتنی بر ماشین و ارزیابی براساس سنجه‌های مختلف وجود دارند و این روش‌ها به‌صورت دستی، خودکار و نیمه‌خودکار هستند [۲۵].

در توسعه و برنامه‌نویسی نرم‌افزار، مشخصه‌ای که می‌توان در کد منبع قرار داد و به‌طور بالقوه می‌تواند باعث ایجاد خطا، آسیب‌پذیری یا هر چیز ناخواسته‌ای شود را می‌توان کد مشکوک نامید. این یک خطای مستقیم نیست، اما به‌طور بالقوه می‌تواند باعث بروز آن شود. الگوهای کد مشکوک براساس زبان برنامه‌نویسی، استاندارد کدنویسی توسعه‌دهنده، شیوه‌های کدنویسی استاندارد و روش‌های توسعه، متفاوت هستند [۲۹]. برای اولین‌بار این اصطلاح توسط فاولر در کتاب «بازآرایی: بهبود طراحی کدهای موجود» استفاده شد و بعدها توسط کنت بک[5] رایج شد [۳۰].

به‌طورکلی، کد مشکوک به عنوان علائمی از مشکلات احتمالی کد یا طراحی شناخته می‌شود که به‌طور بالقوه ممکن است تأثیر منفی بر کیفیت نرم‌افزار مانند قابلیت نگهداری، قابلیت خوانایی کد و قابلیت آزمون آن داشته باشد [۳, ۳۱, ۳۲]. تعداد زیادی از مطالعات بر فنون تشخیص کدهای مشکوک و حذف آنها با بسیاری از ابزارهای تجزیه‌وتحلیل استاتیک برای تشخیص کدهای مشکوک متمرکز شده‌اند. این موارد شامل ابزارهایی مانند Pmd1، Sonarqube2 و Designite3 هستند.

در طول نگهداری و تکامل نرم‌افزار، سیستم‌های نرم‌افزاری

[1] Sobrinho
[2] Tian
[3] Caram
[4] Azeem
[5] kent beck

باید به‌طور مداوم توسط توسعه‌دهندگان تغییر کنند تا الزامات جدید را پیاده‌سازی کنند، ویژگی‌های موجود را بهبود بخشند، یا اشکالات مهم را برطرف کنند [۳۵].

کدهای مشکوک (به‌عنوان‌مثال نشانه‌های وجود انتخاب‌های طراحی یا پیاده‌سازی ضعیف در کد منبع)، یکی از جدی‌ترین اشکال بدهی فنی را نشان می‌دهد. در سال‌های گذشته و اخیر، جامعه پژوهشی در این زمینه بسیار فعال بوده است [۳۶]. همچنین بسیاری از مطالعات تجربی با هدف درک موارد زیر انجام شده است [۲۳]:

(۱) چه موقع و چرا کدهای مشکوک معرفی می‌شوند.
(۲) تکامل و طول عمر آنها در پروژه‌های نرم‌افزاری چگونه است.
(۳) شناخت کدهای مشکوک تا چه حد برای توسعه‌دهندگان مناسب است.

۴- سلسه‌مراتب و تعاریف مشکوک بودن

در ادبیات مربوطه، انواع متعددی از بوی‌های بد[1] وجود دارد. در مجموع، ۲۲ مفهوم در زمینه مهندسی نرم‌افزار وجود دارد که به بخش خاصی از بوهای کد می‌پردازند. قبل از فهرست کردن آنها، به یک مشکل خاص در تعریف بوی بد اشاره خواهد شد.

در حال حاضر، بوی بد به‌صورت مترادف برای موضوع موردبحث (مانند بوی بد معماری[2]، بوی بد کد[3] (کد مشکوک بد)) یا به‌عنوان مخفف (مانند بوی کد (کد مشکوک)، بوی طراحی) استفاده می‌شود. همچنین مفهوم ضد الگو[4] نیز وجود دارد که گاهی به‌طور مترادف با بوی‌های بد استفاده می‌شود و گاهی اوقات تفاوتی آگاهانه بین آنها وجود دارد [۱۴].

با بازگشت به این ۲۲ مفهوم، آنها سلسله‌مراتب بوی بد نامیده شده و از اصطلاح «بوهای بد» به عنوان یک اصطلاح چتر استفاده می‌شود که همه اصطلاحات خاص را از هر سلسله‌مراتبی در بر می‌گیرد. برای روشن شدن و حفظ اصطلاحات رایج مورداستفاده در ادبیات، هر سلسله‌مراتب را می‌توان به‌عنوان یک کل نام برد؛ به‌عنوان مثال، بوهای بد کد و به‌طور خلاصه، بوهای کد (کدهای مشکوک). برای جلوگیری از ابهام در تعاریف، ضد الگوها را از بوهای بد متمایز می‌کنیم [۱۴].

جالب توجه است که ضد الگوها توصیفی دارند که به نظر می‌رسد اکثریت قریب به اتفاق با آن موافق هستند: «یک ضد الگو راه‌حل بدی برای یک مشکل طراحی تکرارشونده است که تأثیر منفی بر کیفیت طراحی سیستم دارد» [۳۴]. در حالی که بوی بد به دلیل عدم وجود تعریف دقیق و کامل، دچار مشکل است. با بازگشت به جایی که در آن تعریف شد، در سال ۱۹۹۹، فاولر در کتاب خود بیان کرد که «بوی کد (کد مشکوک) یک نشانه سطحی است که معمولاً مربوط به یک مشکل عمیق‌تر در سیستم است» [۳۰].

فهرست این سلسله‌مراتب (بدون کد مشکوک) به شرح زیر است:

۱) بوهای معماری: مجموعه‌ای از تصمیمات طراحی معماری که بر ویژگی‌های چرخه حیات سیستم تأثیر منفی می‌گذارند (قابلیت درک، توسعه‌پذیری، قابلیت استفاده مجدد، آزمون‌پذیری) [۳۷].
۲) بوهای طراحی[5]: تکرار انتخاب‌های ضعیف طراحی [۳۸].
۳) بوهای پیاده‌سازی[6]: زیرمجموعه‌ای از بوهای کد که دارای ویژگی گسترش[7] «در درون» و سایر جزئیات خاص براساس مقاله شارما در سال ۲۰۱۷ است [۳۹]. لازم به ذکر است که ویژگی گسترش، مشخص می‌کند آیا بوها در فضای یک کلاس هستند یا برای تشخیص، دامنه وسیع‌تری لازم است؛ به عبارت دیگر، بوها بین کلاس‌ها وجود دارند [۱۴].
۴) بوهای توضیحات۸: توضیحاتی که می‌توانند کیفیت نرم‌افزار را کاهش دهند یا توضیحاتی که از نظر درک کد کمک چندانی به خوانندگان نمی‌کنند [۴۰, ۴۱].
۵) بوهای زبانی۹: بوی‌های مرتبط با ناهماهنگی‌ها بین امضاهای متدها، مستندات و رفتار سیستم و ناهماهنگی‌ها بین نام‌ها، نوع‌ها۱۰ و توضیحات [۴۲].
۶) بوهای انرژی: انتخاب‌های پیاده‌سازی که باعث کاهش بهره‌وری انرژی اجرای نرم‌افزار می‌شوند [۴۳].
۷) بوهای عملکرد۱۱: اشتباهات متداول عملکردی که در معماری‌ها و طراحی‌های نرم‌افزار یافت می‌شود [۴۴, ۴۵].
۸) بوهای آزمون۱۲: آزمون‌های تعریف‌شده ناکافی؛ وجود آنها بر درک و نگهداری مجموعه‌های آزمون تأثیر منفی می‌گذارد [۴۶].
۹) بوهای UML: بوی‌های مدل و بازسازی مدل قابل اعمال در

---

[1] Bad Smells
[2] Bad Architecture Smell
[3] Bad Code Smell
[4] Antipatterns
[5] Design Smells
[6] Implementation Smells
[7] Expanse
[8] Comments Smells
[9] Linguistic Smells
[10] Types
[11] Performance Smells
[12] Test Smells

مراحل اولیه توسعه نرم‌افزار مبتنی بر مدل که باعث نقض اهداف آن می‌شوند؛ صحت[1]، کامل‌بودن[2]، سازگاری[3]، قابل درک بودن[4]، محدود بودن[5]، تغییرپذیری[6] [۴۷, ۴۸].

۱۰) بوهای مرور کد۷: نقض مجموعه‌ای از بهترین شیوه‌ها و قوانین استاندارد که هم پروژه‌های متن‌باز و هم شرکت‌ها روی آن‌ها همگرایی دارند که باید رعایت شوند [۴۹].

۱۱) بوهای جامعه۸: الگوهای سازمانی و فنی - اجتماعی غیربهینه در ساختار سازمانی جامعه نرم‌افزاری [۵۰].

۱۲) بوهای فرآیند ردیابی اشکال۹: مجموعه‌ای از انحرافات از بهترین شیوه‌هایی که توسعه‌دهندگان در طول فرآیند ردیابی اشکال دنبال می‌کنند [۵۱].

۱۳) بوهای پیکربندی۱۰: به دو بخش بوی پیکربندی طراحی و بوی پیکربندی پیاده‌سازی تقسیم می‌شوند. مواردی که کیفیت پیکربندی را زیر سؤال می‌برند (اصول نام‌گذاری، سبک، قالب‌بندی، تورفتگی، طراحی یا ساختار) [۵۲].

۱۴) بوهای محیط۱۱: بوی‌هایی که باعث کاهش راحتی کار می‌شوند، به‌عنوان‌مثال، نیاز به گام‌های بیشتر از آنچه باید باشد برای انجام عملیات خاص [۴۱].

۱۵) بوهای ارائه۱۲: دستورالعمل‌هایی برای ایجاد ارائه‌های بهتر [۵۳].

۱۶) بوهای صفحه‌گسترده۱۳: بوی‌های داخلی کد، اما برای برنامه‌نویسان کاربران نهایی کاربرگ [۵۴].

۱۷) بوهای پایگاه‌داده: ضد الگوها از نظر طراحی پایگاه‌داده منطقی، طراحی پایگاه‌داده فیزیکی، پرس‌وجوها و توسعه برنامه [۵۵].

۱۸) بوهای قابلیت استفاده۱۴: شاخص‌های طراحی ضعیف در رابط کاربری یک برنامه، با پتانسیل این که نه‌تنها قابلیت استفاده، بلکه نگهداری و تکامل آن را نیز مختل کند [۵۶].

۱۹) بوهای اندروید: نقض اصول و شیوه‌های استاندارد که بر کیفیت، عملکرد، درک و نگهداری برنامه‌های کاربردی تلفن همراه تأثیر می‌گذارد [۵۷].

۲۰) بوهای امنیت[15]: اشتباهات امنیتی که ممکن است امنیت و حریم خصوصی را به خطر بیندازند و شناسایی آسیب‌پذیری‌های قابل اجتناب [۵۸].

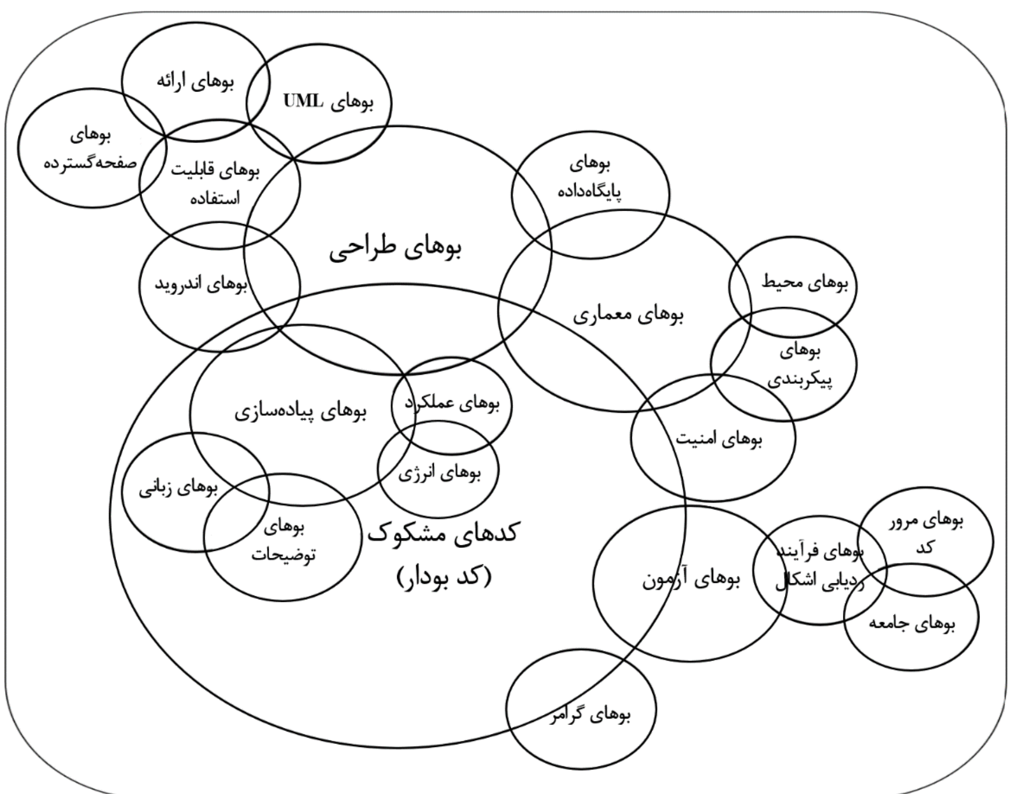
۲۱) بوهای گرامری با دسته‌بندی‌های دوسطحی تودرتو که از آنها به‌عنوان گروه‌های زیربو یاد می‌شود (که برای وضوح درک می‌تواند فقط به‌عنوان گروه‌بندی نامیده شود) [۵۹]. مانند:

- بوهای سازمان: ۱. بوهای قرارداد، ۲. بوهای نماد، ۳. بوهای تجزیه‌وتحلیل[16]، ۴. بوهای تکرار
- بوهای ناوبری[17]: ۱. بوهای اسپاگتی (مشابه فرآیند برنامه‌نویسی پاستا)، ۲. بوهای کمبود[18]، ۳. بوهای ترکیب[19]
- بوهای ساختار: ۱. بوهای پروکسی، ۲. بوهای وابستگی، ۳. بوهای پیچیدگی

تعلق به یک سلسله‌مراتب خاص، یک ویژگی از یک بوی بد است. بوهای بد می‌توانند به یک یا چند سلسله‌مراتب تعلق داشته باشند و لزوماً فقط به یک سلسله‌مراتب وابسته نیستند (به مثال نمودار ون در شکل ۲ مراجعه کنید [۶۰]). به‌عنوان مثال، از «ویژگی حسادت[20]» به عنوان بوی کد [۳] و بوی طراحی [۶۱] یاد می‌شود. تفاوت در چیست؟ مانند معماری نرم‌افزار می‌توان بر اهمیت دیدگاه تأکید کرد. این تعیین می‌کند که از چه زاویه‌ای می‌توان یک بوی بد خاص را مشاهده کرد. این روش، مشکل اصطلاحات نامشخص را حل می‌کند و از دانش مقالات فعلی پشتیبانی می‌کند. به‌عنوان مثال، «نام‌های غیر قابل ارتباط[21]» را می‌توان تنها از طریق خود کد مشاهده کرد اما «ویژگی حسادت» ممکن است هنگام نگاه کردن از هر دو منظر طراحی و کد قابل درک باشد [۶۰].

با خلاصه کردن تمام اطلاعات مذکور، تعریف زیر پیشنهاد می‌شود: «بوی بد، نشان‌دهنده مشکلی در سیستم است که ممکن است در قابلیت نگهداری، توسعه‌پذیری، قابل درک بودن یا قابلیت استفاده آن مشکلاتی ایجاد کند». این به ضد الگوها اجازه می‌دهد تا با بوهای بد درهم‌آمیخته شوند و بنابراین، ادبیات علمی فعلی سازگار خواهد بود [۱۴].

[1] Correctness
[2] Completeness
[3] Consistency
[4] Comprehensibility
[5] Confinement
[6] Changeability
[7] Code Review Smells
[8] Community Smells
[9] Bug Tracking Process Smells
[10] Configuration Smells
[11] Environment Smells
[12] Presentation Smells
[13] Spreadsheet Smells
[14] Usability Smells
[15] Security Smells
[16] Parsing Smells
[17] Navigation Smells
[18] Shortage Smells
[19] Mixture Smells
[20] Feature Envy
[21] Uncommunicative Names



شکل (۳): نمونه نمودار ون از سلسله‌مراتب بوی بد [۶۰]

۵- انواع کدهای مشکوک

تحقیقات متعددی در زمینه کدهای مشکوک انجام گرفته است که هر کدام از آن‌ها نسخه‌ای متفاوت از آرایه‌شناسی آنها را ارائه می‌دهند. چند مورد جدید از این آرایه‌شناسی‌ها عبارت‌انداز:

• آرایه‌شناسی ساده‌شده مانتیلا[1] [۶۲] در یک مرور نظام‌مند در سال ۲۰۲۰ درباره کدهای مشکوک و بازآرایی [۱۶] و مرور ادبیات نظام‌مند درباره فنون تشخیص کدهای مشکوک در یادگیری ماشین [۶۳]. همچنین، اشاره شده است که موارد مشکوک را می‌توان درون کلاس‌ها و بین کلاس‌ها نیز طبقه‌بندی نمود.

• طبقه‌بندی توسعه‌یافته مانتیلا [۶۴] در مقاله مرور ادبیات نظام‌مند درباره رابطه بین کدهای مشکوک و ویژگی‌های کیفیت نرم‌افزار در سال ۲۰۲۰ [۶۵] و در اولویت‌بندی مقاله مروری بر کدهای مشکوک در سال ۲۰۲۱ [۶۶]، اشاره شده است. مطلب قبلی همچنین به طبقه‌بندی موارد مشکوک درون کلاسی و بین کلاسی نیز اشاره می‌کند.

• مرور نظام‌مند سال ۲۰۱۹، که در آن نویسندگان، کدهای مشکوک پیشگیری‌کننده‌های تغییر و غیرضروری‌ها را حذف کرده و همزمان سوءاستفاده‌کنندگان قوانین طراحی و سوءاستفاده‌کنندگان لغوی را معرفی کردند [۶۷].

در حال حاضر، دو نوع گروه‌بندی اصلی وجود دارد که کدهای مشکوک براساس آنها تقسیم می‌شوند. متداول‌ترین آرایه‌شناسی، توسط مانتیلا ارائه شده است که مبتنی بر کدهای مشکوک تعریف‌شده توسط فاولر می‌باشد [۶۴]. این آرایه‌شناسی به یک شکل وجود ندارد و نسخه‌ها یا ترکیب‌های مختلفی از آن استفاده می‌شوند. گاهی اوقات، گروه محصورکننده‌ها[2] که در پیشنهاد اصلی از پایان‌نامه کارشناسی ارشد مانتیلا در سال ۲۰۰۳ ظاهر شده است، کنار گذاشته می‌شود. کدهای مشکوک این زیرگروه به زیرگروه‌های غیرضروری‌ها و سوءاستفاده‌کنندگان شی‌گرایی منتقل می‌شوند [۶۲].

عناصر منفرد و جداگانه در زیرگروه‌های مختلفی ظاهر می‌شوند؛ مانند سلسله‌مراتب موازی ارث‌بری[3] که می‌توان آن را در بین سوءاستفاده‌کنندگان شی‌گرایی [۱۶, ۶۴, ۶۷] و پیشگیری‌کننده‌های تغییر [۶۲, ۶۸] مشاهده کرد که ممکن است نشان‌دهنده این باشد که کدهای مشکوک با گروه‌بندی متناظر خود رابطه‌ای دوجانبه ندارند و ممکن است درهم‌تنیده باشند [۶۰].

در این مقاله، با بررسی و مرور مقالات پیشین، ۵۸ کد مشکوک در ۸ دسته‌بندی بررسی و معرفی شده‌اند که در

[1] Mantyla

[2] Encapsulators

[3] Parallel Inheritance Hierarchies

شکل‌های (۱) و (۲) نشان داده شده است. برخی از نام‌ها ممکن است با نام‌هایی که در ادبیات موضوع، به‌خوبی شناخته شده‌اند، متفاوت باشند. این تغییر نام به دلیل معرفی نسخه به‌روزتر، مانند کدهای مشکوک عنصر تنبل[1] یا معامله داخلی[2] است که قبلاً با نام‌های کلاس تنبل[3] و صمیمیت نامناسب[4] شناخته می‌شدند، اما در اخرین کتاب فاولر [۳] به‌روز شده‌اند.

دو کد مشکوک خاص که در ادبیات فعلی وجود داشته و بیشترین بحث را به همراه دارند، توسط دو کد مشکوک جدید جایگزین شده‌اند. اول، کد مشکوک توضیحات[5] که صحت آن باید روشن شود؛ زیرا همه توضیحات مشکوک نیستند. دومین کد مشکوک، کلاس داده[6] است. کلاس داده به مشکل اساسی در اصل برنامه‌نویسی شیءگرا می‌پردازد که می‌گوید داده‌ها باید نزدیک به منطقی باشند که روی آن عمل می‌کنند [۱۴]. موارد جایگزین برای کدهای مشکوک توضیحات و کلاس داده به ترتیب توضیح "چه‌چیزی"[7] و سرنوشت از عمل[8] می‌باشد که در ادامه توضیح داده خواهند شد.

۵-۱- متورم‌ها

کدهای متورم[9] شامل کدها، متدها و کلاس‌هایی هستند که به حدی بزرگ شده‌اند که کار کردن با آن‌ها دشوار است [۶۹]. به‌طور کلی، متورم‌ها دسته‌ای از کدهای مشکوک را نشان می‌دهند که به دلیل اندازه یا حجمشان غیرقابل کنترل می‌شوند. هنگامی که ویژگی‌های جدید به یک سیستم موجود اضافه می‌شوند، آنها در یک کد معرفی می‌شوند [۷۰]. این دسته شامل نه کد مشکوک به شرح زیر است.

۵-۱-۱- انفجار ترکیبی

انفجار ترکیبی[10] حالتی است که بسیاری از کدها تقریباً کار مشابه را انجام می‌دهند، اما با تغییرات کوچک در داده‌ها یا رفتار. این یک شکل ظاهراً ملایم‌تر است، اما بسیار شبیه به کد تکراری است، جایی که چندین قطعه کد عملکرد یکسانی را اجرا می‌کنند، اما در اشیاء از نوع‌های مختلف [۱۶].

۵-۱-۲-گروه‌های داده

گروه‌های داده[11] یعنی ساختارهای داده‌ای که همیشه با هم ظاهر می‌شوند و زمانی که یکی از آیتم‌ها وجود نداشته باشد، کل مجموعه معنای خود را از دست می‌دهد [۳].

۵-۱-۳- متد طولانی

متد طولانی به معنای متد/تابع بسیار بزرگ است و در نتیجه درک، توسعه و اصلاح آن دشوار است. احتمالاً این متد بیش از حد مسئولیت دارد و این باعث نقض یکی از اصول طراحی شیءگرا مناسب می‌شود (SRP[12]: اصل مسئولیت واحد [۷۱]) [۳].

۵-۱-۴- کلاس بزرگ

کلاس بزرگ، کلاسی است که دارای مسئولیت‌ها، متدها، متغیرهای نمونه و کد تکراری بیش از حدی است که منجر به بزرگ‌شدن کلاس می‌شود [۱۸].

۵-۱-۶- لیست پارامترهای طولانی

لیست پارامترهای طولانی[13]، یک متد یا یک تابع با آرگومان‌ها یا پارامترهای بسیار زیاد است [۱۸]. لیست پارامترهای گسترده، که درک آن را دشوار می‌کند و معمولاً نشان‌دهنده این است که این متد مسئولیت‌های زیادی دارد. این کد مشکوک با متد طولانی رابطه قوی دارد [۱۶].

۵-۱-۷- بررسی پوچ بودن

بررسی پوچ بودن[14] یعنی رویدادهایی که به‌طور مکرر ظاهر می‌شوند و مقادیر تهی اشیاء را تأیید می‌کنند [۱۶]. وقتی که برنامه‌نویسان برای جلوگیری از خطاهای نشانگر تهی[15]، به‌طور مکرر بررسی کنند که آیا یک متغیر پوچ است یا نه، این روش را بررسی پوچ (نال) بودن می‌نامند. این می‌تواند منجر به کدهای تکراری و پراکنده شود و خوانایی کد را کاهش دهد.

۵-۱-۸- راه‌حل عجیب‌وغریب

راه‌حل عجیب‌وغریب[16] زمانی اتفاق می‌افتد که دو راه برای حل یک مشکل مشابه در یک سیستم وجود دارد که معمولاً نشانه ظریف کد تکراری است [۷۲].

۵-۱-۹- وابستگی به نوع‌های ابتدایی

وابستگی به نوع‌های ابتدایی[17] نشان‌دهنده وضعیتی است که در آن برنامه‌نویسان برای نگهداری اطلاعات، به‌طور مکرر از نوع‌های

[1] Lazy Element
[2] Insider Trading
[3] Lazy Class
[4] Inappropriate Intimacy
[5] Comments
[6] Data Class
[7] "What" Comment
[8] Fate over Action
[9] Bloaters codes
[10] Combinatorial Explosion
[11] Data Clumps
[12] Single Responsibility Principle
[13] Long Parameter List
[14] Null Check
[15] Null pointer
[16] Oddball Solution
[17] Primitive Obsession

ابتدایی مانند رشته‌ها و اعداد صحیح به‌جای استفاده از کلاس‌ها و انتزاعات مناسب استفاده می‌کنند [۳]. به‌طور کلی، یعنی استفاده شدید از انواع داده‌های اولیه به جای اشیاء کوچک.

۵-۱-۱۰- کد راه‌اندازی یا خاتمه موردنیاز

در برنامه‌ها، برخی از کلاس‌ها یا متدها ممکن است نیاز به تنظیمات یا عملیات پایانی داشته باشند. یعنی چندین خط کد نیاز باشد تا: ۱. آن را به درستی تنظیم کند، ۲. محیط به اقدامات خاصی قبل یا بعد از استفاده نیاز دارد، ۳. اقدامات پاکسازی موردنیاز است. وقتی که این تنظیمات و عملیات‌ها به‌طور مکرر و در چندین قسمت از کد تکرار می‌شوند، می‌توان آنها را به‌عنوان کد راه‌اندازی یا خاتمه موردنیاز[1] شناخت [۶۰]. همچنین، ممکن است سطح انتزاع مشکوک را نشان دهد.

۵-۲- سوءاستفاده‌کنندگان

سوءاستفاده‌کنندگان به سه دسته سوءاستفاده‌کنندگان شیءگرایی[2]، سوءاستفاده‌کنندگان لغوی[3] و سوءاستفاده‌کنندگان عملکردی[4] تقسیم می‌شوند.

۵-۲-۱- سوءاستفاده‌کنندگان شیءگرایی

سوء استفاده‌کنندگان شیءگرایی، کدهای مشکوکی هستند که اصول برنامه‌نویسی شیءگرا را نقض می‌کنند [۷۳]. به عبارتی، بدون بررسی اصول یک طراحی شیءگرای مناسب، راه‌حل‌های موقتی هستند که در کد استفاده می‌شوند [۷۱]. این دسته شامل هفت کد مشکوک است.

- **گزاره‌های Switch/ Switchهای تکراری**

کد مشکوک گزاره‌های Switch/ Statementهای تکراری[5] یعنی وجود یک گزاره Switch پیچیده یا دنباله‌ای از گزاره‌های شرطی if.

این عناصر با توجه به تعریف لزوماً کدهای مشکوک نیستند؛ اما وقتی که به‌طور گسترده استفاده می‌شوند، معمولاً نشانه‌ای از مشکلات هستند؛ به‌خصوص زمانی که برای شناسایی رفتار یک شیء براساس نوع آن استفاده می‌شوند [۱۶].

- **کلاس‌های جایگزین با رابط‌های مختلف**

کلاس‌های جایگزین با رابط‌های مختلف[6] حالتی است که یک کلاس از کلاس‌های مختلف پشتیبانی می‌کند، اما رابط[7] آنها متفاوت است [۳]. به عبارتی، دو کلاس مختلف عملکردهای مشابهی را با امضاهای متد متفاوت انجام می‌دهند.

- **کلاس پایه با وابستگی به زیرکلاس**

رایج‌ترین دلیل برای تقسیم مفاهیم به کلاس‌های پایه و مشتق این است که مفاهیم کلاس پایه سطح بالاتر می‌توانند مستقل از مفاهیم کلاس مشتق سطح پایین‌تر باشند؛ بنابراین، وقتی می‌بینیم که کلاس‌های پایه نام مشتقات خود را ذکر می‌کنند، معمولاً به مشکلی مشکوک می‌شویم. به‌طورکلی، کلاس‌های پایه نباید چیزی در مورد مشتقات خود بدانند. البته استثناهایی هم در این قاعده وجود دارد. گاهی اوقات تعداد مشتقات کاملاً ثابت است و کلاس پایه کدی دارد که از بین مشتقات انتخاب می‌کند [۴۱].

کلاس پایه با وابستگی به زیرکلاس[8] زمانی رخ می‌دهد که کلاس پایه (کلاس اصلی) به‌شدت وابسته به ویژگی‌ها و عملیات زیرکلاس‌ها می‌شود؛ لذا باعث می‌شود که تغییرات در زیرکلاس‌ها، به‌طور غیرمنتظره‌ای تغییراتی در کلاس پایه ایجاد کنند [۴۱]. این امر می‌تواند منجر به کد ناپایدار و شکننده شود. کد مشکوک مذکور ارتباط نزدیکی با کد مشکوک راهبرد شاتگان دارد.

- **پیچیدگی شرطی**

پیچیدگی شرطی[9] بیان می‌کند که اگرچه ساختارهای شرطی به‌خودی‌خود مشکلی ندارند، اما استفاده بیش از حد از آن‌ها مورد مشکوکی است که باید با آن مقابله شود [۱۶].

- **ایستا (استاتیک) بودن نامناسب**

قاعده کلی که در مورد کد مشکوک ایستا (استاتیک) بودن نامناسب[10] توسط مارتین ارائه شده این است که در صورت تردید، باید متدهای غیرایستا را به متدهای ایستا ترجیح داد. بهترین راه برای بررسی اینکه آیا یک متد باید ایستا باشد یا خیر، این است که فکر کنیم آیا باید چندریختی عمل کند یا خیر. یک مثال عالی از روش ایستا ارائه‌شده توسط مارتین math.max (double a, double b) است - فکر کردن به رفتار چندریختی برای یک تابع max دشوار است. ازسوی دیگر، HourlyPayCalculator.calculate_pay(employee, overtime_rate) مشکوک است؛ زیرا ممکن است الگوریتم‌های مختلفی برای محاسبه پرداخت وجود داشته باشد و بنابراین باید یک متد غیراستاتیک در کلاس Employee باشد [۴۱]. بااین‌حال، باید از کد مشکوک کلی‌نگری بیش از اندازه آگاه بود و احتیاط کرد. در حال حاضر، زمانی که هنوز هیچ الگوریتم متفاوتی درخواست یا برنامه‌ریزی نشده است، این یک عملیات

---

[1] Required Setup or Teardown Code
[2] Object-oriented Abusers
[3] Lexical Abusers
[4] Functional Abusers
[5] Switch Statements/Repeated Switches
[6] Alternative Classes with Different Interfaces
[7] Interface
[8] Base Class depends on Subclass
[9] Conditional Complexity
[10] Inappropriate Static

بدون حالت است که برای متدهای استاتیک قابل قبول است.

• **رد وراثت**

وقتی یک زیرکلاس بخشی از ویژگی‌ها و متدهای والدین خود را به ارث می‌برد و تنها از چند مورد از متدها و ویژگی‌های آن استفاده می‌کند، رد وراثت[1] در یک کد رخ می‌دهد. این عدم اجرای وراثت همیشه نشان‌دهنده مشکلات جدی طراحی است؛ زیرا سلسله‌مراتب به خوبی تعریف نشده است و کلاس هم‌سطح را برای کلاس والد (پایه) ایجاد می‌کند [۷۰].

• **فیلد موقتی**

فیلدهای موقتی[2] کد مشکوکی هستند که به وجود فیلدهایی در یک کلاس اشاره دارد که فقط برای مدت زمان کوتاهی استفاده و سپس دور ریخته می‌شوند. فیلدهای موقت درک و نگهداری کد را سخت‌تر می‌کنند. آنها همچنین خطر ایجاد اشکالات و خطاها را افزایش می‌دهند؛ زیرا تغییرات در فیلد ممکن است بر رفتار کد تأثیر بگذارد. فیلدهای موقت اغلب نتیجه فقدان انتزاع یا ناتوانی در تشخیص الگوهای رایج در کد هستند [۷۳].

۵-۲-۲- سوءاستفاده‌کنندگان لغوی

فاولر و همکاران، کد مشکوک توضیحات کد را تعریف کردند که وقتی توضیحات حاوی اطلاعات متناظر با کد منبع و رفتار آن نباشد، مشکوک هستند که موها و همکاران بعداً آن را به‌عنوان سوء استفاده‌کنندگان لغوی گزارش کردند، اگر با رفتار داخلی کد مطابقت نداشته باشند [۶۷]. این دسته شامل شش کد مشکوک به شرح زیر است.

• **توضیح نادرست**

توضیحات با اکثر نحوهای موجود در زبان‌های برنامه‌نویسی متفاوت است؛ زیرا اجرا نمی‌شوند. این ممکن است منجر به شرایطی شود که پس از بازنگری کد، توضیحات اطراف آن بدون تغییر باقی مانده و دیگر با آنچه که توصیف می‌کردند، مطابقت نداشته باشند. توضیحات خوب از گروه توضیح "why" در معرض این وضعیت قرار نمی‌گیرند. اگر توضیحات، "چه‌چیزی اتفاق افتاده" را شرح داده باشند، تا زمانی که کدی که توضیح می‌دهد بدون تغییر باقی بماند، توضیحات مربوطه مناسب خواهد بود و در غیراین‌صورت کد مشکوک توضیح نادرست[3] وجود خواهد داشت [۱۴]. البته، توضیحات "چه‌چیزی" به تنهایی کد مشکوک هستند که در ادامه معرفی خواهند شد.

• **نام نادرست متد**

کد مشکوک نام نادرست متد[4]، به دلیل ایجاد متدها یا توابعی با نام‌گذاری و عملکرد متناقض به وجود می‌آید. در طول سال‌ها، برنامه‌نویسان ارتباطاتی را بین کلمات خاص و متد ایجاد کرده‌اند که برنامه‌نویسان باید آنها را با هم پیوند دهند. به‌عنوان‌مثال، بر خلاف انتظارات منطقی، ایجاد یک تابع مانند getSomething که مقداری را برنمی‌گرداند، گیج‌کننده و نادرست است [۱۴].

• **نابینایی دودویی**

در جامعه هسکل[5]، یک سؤال مشهور (ناشناخته) در مورد تابع filter وجود دارد – آیا گزاره filter به معنای TAKE است یا DROP؟

```
data Bool = False | True
filter :: (a -> Bool) -> [a] -> [a]
--
data Keep = Drop | Take
filter :: (a -> Keep) -> [a] -> [a]
```

شکل (۴): مثالی از کد مشکوک نابینایی دودویی [۶۰]

کد مشکوک نابینایی دودویی[6] در شرایطی اتفاق می‌افتد که یک تابع یا متدی که روی bool-s عمل می‌کند، اطلاعات مربوط به آنچه boolean نشان‌دهنده آن است را از بین می‌برد. بهتر است در این موارد یک معادل قابل بیان از نوع boolean با نام‌های مناسب داشته باشیم. برای تابع filter، می‌تواند از نوع Keep تعریف شده باشد: Keep = Drop | Take [۶۰].

• **نام‌های ناسازگار**

کد مشکوک نام‌های ناسازگار[7] زمانی ایجاد می‌شود که یک نام در مکان‌های مختلف، برای اهداف مختلف استفاده شود [۱۶]. به‌عبارت‌دیگر، شما یک اصطلاح برای چیزی در یک مکان پیدا می‌کنید و یک نام متفاوت برای همان چیز در جایی دیگر. برای مثال، ممکن است add()، store()، put()، place() و غیره را برای همان ویژگی اصلی ببینید [۷۲].

• **عدد جادویی**

مقادیر عددی هستند که عمداً در کد ظاهر می‌شوند و هیچ‌گاه تغییر نمی‌کنند. این کد مشکوک، به استفاده از اعداد ثابت (بدون توضیحات مناسب) در برنامه اشاره دارد [۷۲].

اصطلاح عدد جادویی[8]، فقط به اعداد مربوط نمی‌شود. این اصطلاح برای هر عنصری که دارای یک مقدار است که خودش را توصیف نمی‌کند، قابل استفاده است [۴۱].

• **نام غیرقابل ارتباط**

نام‌های غیر قابل ارتباط یعنی نام‌هایی در عناصر نرم‌افزار استفاده می‌شوند (معمولاً ویژگی‌ها و متغیرهای محلی) که به اندازه کافی نام/قصد خود را بیان نمی‌کنند، مانند x یا value1. این موضوع هنگامی که در متدها و کلاس‌ها استفاده می‌شود، بسیار مهم‌تر

[1] Refused Bequest
[2] Temporary Fields
[3] Fallacious Comment
[4] Fallacious Method Name
[5] Haskell community
[6] Boolean Blindness
[7] Inconsistent Names
[8] Magic Number

است [۱۶].

۵-۲-۳- سوءاستفاده‌کنندگان عملکردی

این دسته شامل سه کد مشکوک به شرح زیر است.

- **حلقه‌های امری**

حلقه‌های امری[1] جایگزین جدیدی برای کد مشکوک حلقه‌ها می‌باشد [۶۰].
فاولر این احساس را دارد که حلقه‌ها یک مفهوم قدیمی هستند. او قبلاً آنها را به‌عنوان یک مشکل در اولین ویرایش کتاب خود ذکر کرد، اگرچه در آن زمان جایگزین‌های بهتری وجود نداشت [۳۰]. در حال حاضر، زبان‌های برنامه‌نویسی یک جایگزین را در اختیار قرار می‌دهند، یعنی خطوط‌لوله[2]. فاولر، در نسخه ۲۰۱۸ کتاب خود، پیشنهاد می‌دهد که حلقه‌های قدیمی و منسوخ باید با عملیات‌های خط‌لوله مانند filter، map یا reduce جایگزین شوند [۳].
در واقع، گاهی اوقات حلقه‌ها می‌توانند به سختی خوانده شوند و مستعد خطا باشند. جرزیک و مادیسکی[3] بیان می‌کنند که ممکن است این مورد تأیید نشده باشد، اما ما در وجود برنامه‌نویسی که حداقل یک بار قبلاً IndexError را تجربه نکرده باشد، شک داریم. رویکرد توصیه‌شده این است که از حلقه‌های تکرار صریح اجتناب و از حلقه forEach یا for-in/for-of-like استفاده شود که مراقبت از اندیس‌گذاری یا لوله‌های جریان را برعهده دارند [۱۴].

جرزیک و مادیسکی همچنین عنوان کردند که ما از مشخص کردن همه حلقه‌ها به‌عنوان کد مشکوک خودداری می‌کنیم. حلقه‌ها همیشه جزء اساسی برنامه‌نویسی بوده و احتمالاً هنوز هم خواهند بود. زبان‌های مدرن رویکردهای بسیار خوش‌ساختی را برای حلقه‌ها ارائه می‌دهند و حتی مواردی مانند ترکیب لیست در هسکل یا پایتون. بخش اندیس‌گذاری مشکل اصلی نگرانی است. البته، حلقه‌های طولانی یا حلقه‌های دارای اثرات جانبی نیز همین‌طور هستند، اما اینها فقط بخشی از کدهای مشکوک متد طولانی یا اثرات جانبی هستند [۱۴].

- **داده‌های قابل تغییر**

داده‌های قابل تغییر[4] مضر هستند؛ زیرا می‌توانند به‌طور غیرمنتظره‌ای قسمت‌های دیگر کد را خراب کنند. این یک منبع غنی از خطاها است که سخت می‌توان به آن‌ها پی برد؛ زیرا ممکن است در شرایط خاصی رخ دهند. به‌طورکلی، تغییر در داده‌ها اغلب می‌تواند منجر به پیامدهای غیرمنتظره و خطاهای پیچیده شود [۳].

- **اثرات جانبی**

اولین یا دومین اصل اساسی برنامه‌نویسی تابعی، این است که هیچ‌گونه اثرات جانبی[5] وجود نداشته باشد. برنامه‌نویسی شیءگرا نیز می‌تواند این قانون را با مزایای زیادی اعمال کند. در یک سناریوی کامل، وقتی به مجموعه‌ای از فراخوانی‌های متد در سطح بالاتر نگاه می‌کنیم، حتی یک تماشاگر تازه‌کار می‌تواند به‌طور نسبی بفهمد چه اتفاقی می‌افتد. نام‌های متدها و توابع باید نشان‌دهنده کاری که انجام می‌دهند باشند و فقط آن کاری را انجام دهند که مورد انتظار است تا درک کد را بیشینه کنند. توجه داشته باشید که توسعه‌دهندگان باید این مشکل را با حذف اثرات جانبی و جداکردن آنها در متدهای جداگانه و فعال‌سازی آن‌ها به‌صورت مجزا رفع کنند، به‌طوری که اصل مسئولیت واحد را نقض نکنند [۶۰].

۵-۳-پیشگیری‌کننده‌های تغییر

پیشگیری‌کننده‌های تغییر[6] مجموعه پیچیده‌ای از کدهای مشکوک هستند که به‌راحتی توسط نرم‌افزار قابل نگهداری نیستند و اصلاح نرم‌افزار به زمان و منابع زیادی نیاز دارد [۶۹]. این نوع از کدهای مشکوک، زمانی که کلاس‌ها یا متدها بیش از یک ویژگی یا عملکرد مسئولیت داشته باشند، در یک سیستم قابل شناسایی هستند. این کدهای مشکوک اصل مسئولیت واحد را نقض می‌کنند که نشان می‌دهد کلاسی که قابلیت اتصال به پایگاه داده را فراهم می‌کند تنها زمانی تغییر می‌کند که نوع پایگاه داده تغییر کند [۷۰]. این دسته شامل هفت کد مشکوک به شرح زیر است.

۵-۳-۱- جهنم بازگشتی

جهنم بازگشتی[7] شبیه کد مشکوک پیچیدگی شرطی است، این کد مشکوک، مشابه پیچیدگی شرطی است که در آن تورفتگی‌ها به‌طور عمیق ایجاد می‌شوند و براکت‌های بسته به شکلی متوالی و پی‌درپی ظاهر می‌شوند، تقریباً مانند آبشار نیاگارا. Callback تابعی است که به‌عنوان آرگومان به تابع دیگری ارسال می‌شود که قرار است بعداً اجرا شود. یکی از محبوب‌ترین Callbackها می‌تواند addEventListener در جاوا اسکریپت باشد. وقتی به‌تنهایی و جدا از یکدیگر باشند، هیچ مشکلی را ایجاد یا نشان نمی‌دهند. بلکه، لیست طولانی از فراخوانی‌های گروه‌بندی‌شده چیزی است که باید مراقب آن بود. این مورد ممکن است به‌طور خلاصه با نام سلسله‌مراتبی از فراخوانی‌ها شناخته شود، اما

[1] Imperative Loops
[2] pipelines
[3] Jerzyk and Madeyski
[4] Mutable Data
[5] Side Effects
[6] Change Preventers
[7] Callback Hell

خوشبختانه قبلاً یک نام جالب و قابل تشخیص برای آن استفاده شده است. راه‌حل‌های زیادی برای این مشکل وجود دارد که عبارت‌اند از: توابع Promises، غیرهمگام یا تقسیم تابع بزرگ به متدهای جداگانه [۶۰].

۵-۳-۲- تغییر واگرا

کد مشکوک تغییر واگرا[1] اصل مسئولیت واحد را نقض می‌کند؛ زیرا دلایل زیادی برای تغییر دارد. تغییر واگرا درک، اصلاح و آزمون کد را سخت‌تر می‌کند. تغییر واگرا اغلب نتیجه طراحی ضعیف یا عدم وجود انتزاع در کد است [۷۳]. به‌طورکلی، کلاسی که متدهای نامربوط زیادی دارد به عنوان کلاسی با تغییرات واگرا شناسایی می‌شود [۷۰].

۵-۳-۳- انتزاع مشکوک

استیو اسمیت[2] در درس خود از اصطلاح "سطوح انتزاع ناسازگار[3]" استفاده می‌کند. جرزیک و مادیسکی بیان می‌کنند که ممکن است فقط با یادآوری معنی از طریق عنوان آن، اشتباه تفسیر شود و مشکوک است که این می‌تواند شرایطی را ایجاد کند که در جایی، حداقل در یک مجموعه کد، شخصی ممکن است در برابر یک فرد غیرکنجکاو برنده یک بحث شود؛ بنابراین، سطوح انتزاع ثابت اما همیشه اشتباه باقی می‌مانند. به همین دلیل، تصمیم گرفتند نام آن را به انتزاع مشکوک[4] تغییر دهند و مستقیماً به علت بالقوه مشکوک بودن پرداخته شود تا در مورد کدی که شخصی نوشته است، فکر کنند. فاولر می‌گوید که «هیچ راهی برای خروج از یک انتزاع نابجا وجود ندارد و این یکی از سخت‌ترین کارهایی است که توسعه‌دهندگان نرم‌افزار می‌توانند انجام دهند و وقتی اشتباه می‌کنید، راه‌حل سریعی وجود ندارد.» انتزاع مشکوک قرار است در اسرع وقت این سؤال را برانگیزد - "آیا مشکوک است؟" یک ثانیه وقت صرف کنید تا به کد موجود فکر کنید و سپس ادامه دهید یا اگر چیزی مشکوک به نظر می‌رسد، فوراً اصلاح کنید: آیا ابزار واقعاً این پیام را جستجو می‌کند؟ یا دستگاه اتصال این کار را انجام می‌دهد؟ [۱۴].

۵-۳-۴-آرگومان پرچم

مارتین فاولر، آرگومان پرچم[5] را به‌عنوان «نوعی آرگومان تابعی که به تابع می‌گوید بسته به مقدارش، عملیات متفاوتی را انجام دهد» تعریف می‌کند. دو دلیل برای مشکوک بودن آن وجود دارد. اول از همه می‌تواند یک کاندید برای کد مشکوک نابینایی دودویی باشد. مشکل دوم این است که این ممکن است یک فاز پیله، قبل از تبدیل شدن به یک پیچیدگی شرطی کامل و زیبا باشد [۳].

آرگومان‌های پرچم زشت است. انتقال یک مقدار دودویی به یک تابع عملیات بسیار بدی است. این عمل بلافاصله امضای متد را پیچیده می‌کند و با صدای بلند اعلام می‌کند که این تابع بیش از یک کار را انجام می‌دهد. اگر پرچم درست باشد یک کار انجام می‌دهد و اگر پرچم نادرست باشد یک کار دیگر! آنها گیج‌کننده هستند و باید حذف شوند [۴۱].

۵-۳-۵- سلسله‌مراتب موازی ارث‌بری

این کد مشکوک زمانی اتفاق می‌افتد که یک درخت ارث‌بری از طریق ترکیب به درخت ارث‌بری دیگری وابسته است و برای ایجاد یک زیرکلاس برای یک کلاس، فرد متوجه می‌شود که باید برای کلاس دیگری یک زیرکلاس بسازد. فاولر تصریح کرد که این یک مورد خاص از کد مشکوک راهبرد شاتگان است [۳].

۵-۳-۶- راهبرد شاتگان

کد مشکوک راهبرد شاتگان[6] در مقابل تغییر واگرا است؛ زیرا زمانی که یک تغییر اتفاق می‌افتد، چندین کلاس مختلف باید تغییر کنند [۳]. به‌طورکلی، باید هنگام ایجاد یک تغییر در یک سیستم، بسیاری از کلاس‌ها را تغییر دهیم. به‌عنوان مثال، هنگام تغییر پایگاه‌داده از یک فروشنده به فروشنده دیگر، چندین کلاس را تغییر دهیم.

۵-۳-۷- نمونه خاص

ویک به وضعیت پیچیده شرطی به‌عنوان کد مشکوک نمونه خاص[7] با دو علامت اشاره می‌کند: عبارات پیچیده if یا بررسی مقادیر خاص قبل از انجام کار واقعی نظیر کد راه‌اندازی یا خاتمه موردنیاز (مخصوصاً مقایسه با ثابت‌ها یا شمارش‌ها) [۷۲].

۵-۴- غیرضروری‌ها

غیرضروری‌ها[8] کد منبعی است که بیهوده و زائد است [۶۹]. به‌عبارتی، آنها کدهای مشکوکی هستند که به وجود کد اضافی یا غیرلازم اشاره دارند که نبود آن، کد را موثرتر می‌کند [۷۳]. این دسته شامل شش کد مشکوک به شرح زیر است.

۵-۴-۱-کد تکراری

کد تکراری[9] نشانه‌ای است که زمانی ایجاد می‌شود که بلوک‌های

[1] Divergent Change
[2] Steve Smith
[3] Inconsistent Abstraction Levels
[4] Dubious Abstraction
[5] Flag Argument
[6] Shotgun Surgery
[7] Special Case
[8] Dispensables
[9] Duplicated Code

کد یکسان یا مشابه را در قسمت‌های مختلف پایگاه کد پیدا کنیم. کد تکراری، تغییر و آزمون کد را سخت‌تر می‌کند؛ زیرا ممکن است تغییرات در یک نمونه از کد تکراری برای نمونه‌های دیگر اعمال نشود. کد تکراری اغلب در نتیجه کپی کردن کد، فقدان انتزاع یا شکست در تشخیص الگوهای رایج در کد است [۷۳].

۵-۴-۲-کد مرده

کد مرده[1]، به وجود کدی اشاره دارد که دیگر استفاده یا اجرا نمی‌شود. کد مرده درک و نگهداری کد را سخت‌تر می‌کند. همچنین زمان موردنیاز برای ساخت، کامپایل و اجرای کد را افزایش می‌دهد و کارایی آن را کاهش می‌دهد. کد مرده اغلب در نتیجه تغییرات کد، بازآرایی یا حذف ویژگی ایجاد می‌شود که پس از حذف کد بدون استفاده، باقی می‌ماند [۷۳].

۵-۴-۳- عنصر تنبل

یک عنصر تنبل، عنصری است که کار کافی انجام نمی‌دهد. اگر یک متد، متغیر یا کلاس کار کافی را برای خود انجام نمی‌دهد، به اندازه‌ای که بتواند ارزش‌افزوده کافی را برای پروژه فراهم کند، باید با یک موجودیت دیگر ترکیب شود [۳].

۵-۴-۴-کلی‌نگری بیش از اندازه

کلی‌نگری بیش از اندازه[2] قطعه کدهایی است که برای پشتیبانی از رفتار آینده نرم‌افزار که هنوز موردنیاز نیست، طراحی شده است [۳]. مثلا یک کلاس انتزاعی که استفاده نشده است، اما در نسخه‌های آینده سیستم استفاده خواهد شد.

۵-۴-۵- توضیح "چه‌چیزی"

توضیح "چه‌چیزی" جایگزین جدیدی برای کد مشکوک توضیحات می‌باشد [۶۰].
تشخیص همه توضیحات به‌عنوان کد مشکوک بحث‌برانگیز است و چندین نظر متفاوت را ایجاد می‌کند. به همین دلیل، جرزیک و مادیسکی یک زیرمجموعه مشخص از توضیحات به نام توضیح "What" را تعریف کردند که به‌وضوح فقط این توضیحات را تعریف می‌کند که در اکثریت قریب به اتفاق به چیز مشکوکی اشاره خواهد کرد. قانون ساده است: اگر یک توضیح آنچه را که در یک بخش خاص از کد اتفاق می‌افتد را توصیف می‌کند، احتمالاً سعی می‌کند کد مشکوک دیگری را بپوشاند. این تعریف جایی را برای نظرات «چرا» که قبلاً توسط ویک[3] در سال ۲۰۰۴ تعریف شده بود و مفید در نظر گرفته شده بود، باز می‌کند. ویک همچنین خاطرنشان می‌کند توضیحاتی که الگوریتم‌های غیرواضح را ذکر می‌کنند نیز قابل قبول هستند. همان‌طور که اشاره شد، مشکل این است که توضیحات به‌طورکلی مشکوک هستند؛ چرا که آنها به‌عنوان یک ضد کد مشکوک برای کدهای مشکوک دیگر عمل می‌کنند. آنها همچنین ممکن است با گذر زمان به‌سرعت فرسوده شده و به یک دسته دیگر از توضیحات یعنی توضیحات نادرست، تبدیل شوند که یک زیردسته فاسد و گمراه‌کننده از توضیحات "چه‌چیزی" هستند [۱۴].

۵-۴-۶-کلاس داده

کلاس‌هایی که فقط فیلدها، متدهای گرفتن و تنظیم‌کردن دارند به‌عنوان کلاس داده در نظر گرفته می‌شوند. فیلدهای داده مورد استفاده در این نوع کلاس‌ها معمولاً توسط کلاس‌های دیگر دستکاری می‌شوند. این کلاس‌ها هیچ ویژگی ندارند و قادر به فعالیت بر روی فیلدهایی که در اختیار دارند، نیستند [۷۰].

۵-۵- جفت‌کننده‌ها

جفت‌کننده‌ها[4] گروهی هستند که به جفت‌شدن بیش از حد بین کلاس‌ها کمک می‌کنند که اعتبارسنجی اجرا را پس از اصلاح کد دشوار می‌کنند [۶۹]. این دسته شامل شش کد مشکوک به شرح زیر است.

۵-۵-۱- ترس از شکست

ترس از شکست[5] کد مشکوکی است که از ترس رایج (حداقل در بین دانش‌آموزان) از شکست الهام گرفته شده است که به‌طور حرفه‌ای آتیچیفوبیا[6] نامیده می‌شود - ترس از شکست. ترس از اعتراف به شکست (اینکه چیزی اشتباه پیش رفته است) یک ویژگی روان‌شناختی کاملاً مرتبط است و مواجهه با آن ترس برای همه مفید خواهد بود. این ایده خوبی نیست که امیدوار باشیم کسی از این کار خلاص شود. بدون شک، شاید حتی در بیشتر مواقع این‌طور باشد، اما هرچه چیزهای بیشتری در این عدم صداقت جمع شود، در نهایت اگر کشف شود، ضربه سخت‌تر خواهد شد [۱۴].

در برنامه‌نویسی، این رفتار باعث آشفتگی کد می‌شود؛ زیرا بعد از فراخوانی یک متد یا تابع، کد اضافی برای بررسی اعتبار یک کد وضعیت، علامت‌گذاری یک پرچم بولین یا عدم وجود مقدار None برگردانده شده، نیاز است - و همه اینها خارج از دامنه متد قرار می‌گیرند. اگر انتظار می‌رود متدی شکست بخورد، باید با ایجاد یک استثناء[7] یا در غیر این صورت، یک شیء خاص

---
[1] Dead Code
[2] Speculative Generality
[3] Wake
[4] Couplers
[5] Afraid To Fail
[6] Atychiphobia
[7] Exception

از نوع None/Null در کلاس موردنظر (به دنبال الگوی شیء Null)، نه خود null، با شکست مواجه شود. به‌عنوان‌مثال، اگر یک شیء موردانتظار نمی‌تواند دریافت یا ایجاد شود، در عوض، یک نشانگر وضعیت برگشت داده شود (که باید پس از تکمیل متد بررسی شود) و کدهای مشکوکی که ایجاد می‌کند عبارت‌اند از ترس از شکست و کد خاتمه. به جای آن، کد باید با رعایت اصل "شکست سریع"، یک خطا ایجاد کند [۱۴].

۵-۵-۲- عملگر باینری در نام

کد مشکوک عملگر باینری در نام[1] ساده است: نام‌های متد یا تابعی که دارای عملگرهای بیتی باینری مانند AND و OR هستند، کاندیدهای آشکاری برای نقض‌کنندگان اصل مسئولیت واحد هستند. اگر نام متد شامل «AND» باشد و سپس دو کار متفاوت انجام دهد، ممکن است بپرسید که چرا برای انجام این دو کار متفاوت به دو قسمت تقسیم نشده تا این دو کار به‌صورت جداگانه انجام شوند؟ علاوه‌براین، اگر نام متد شامل «OR» باشد، نه‌تنها دو کار متفاوت انجام می‌دهد، بلکه به احتمال زیاد، یک آرگومان پرچم نیز دارد که کد مشکوک دیگری است. این مورد نه‌تنها ممکن است در نام متدها رخ دهد، بلکه در متغیرها نیز وجود دارد؛ اگرچه در بیشتر موارد محل بررسی این نوع کد مشکوک، در نام متدها است [۱۴].

۵-۵-۳- سرنوشت از عمل

سرنوشت از عمل جایگزین جدیدی برای کد مشکوک کلاس داده می‌باشد [۶۰].

توسعه‌دهندگان نباید از خود کلاس داده بترسند؛ زیرا در حال حاضر، این کلاس‌های داده معمولاً اعتبارسنجی‌های مفید و اضافی را به همراه می‌آورند. همچنین می‌توان اشاره کرد که کلاس‌های داده ابزاری سریع و مستقیم برای بسته‌بندی داده‌های خاص در یک انتزاع کوچک هستند تا علائم کد مشکوک گروه‌های داده را مهار کنند (گاهی زبان‌های برنامه‌نویسی برای این منظور ابزارهایی ارائه می‌دهند، مانند رابط). بااین‌حال، نباید فراموش کرد که عملکرد را نزدیک به داده‌ها نگه داشت که در حال حاضر علائم کد مشکوک کلاس داده به آن اشاره می‌کنند. جرزیک و مادیسکی کد مشکوک سرنوشت از عمل را پیشنهاد کردند که ایده فعلی را حفظ می‌کند که نشان می‌دهد مشکل مربوط به خود کلاس‌های داده نیست، بلکه در موقعیت‌هایی است که کلاس‌ها یا توابع خارجی عمدتاً فیلدهای یک شیء را تغییر می‌دهند [۱۴].

[1] Binary Operator in Name

۵-۵-۴- ویژگی حسادت

ویژگی حسادت یک متد است که بیشتر از استفاده از داده‌های خود، به داده‌های شیء‌های دیگر دسترسی دارد. برنامه‌نویسی شیءگرا از امکاناتی برای مرتبط کردن داده‌ها و عملیات بر روی آن‌ها پشتیبانی می‌کند. اما ویژگی حسادت با دسترسی به ویژگی‌های خارجی آن قانون را نادیده می‌گیرد. ویژگی حسادت یک کد مشکوک در سطح متد است که بر انتزاع داده و جفت‌شدگی تأثیر می‌گذارد [۷۴]. به‌طورکلی، وقتی متدی به اعضای کلاس‌های دیگر بیشتر از خود علاقه‌مند است، نشانه واضحی است که در کلاس اشتباه قرار دارد [۳].

۵-۵-۵- آشکارسازی ناشایست

آشکارسازی ناشایست[2] زمانی اتفاق می‌افتد که کلاینت‌ها به کلاس‌هایی که استفاده می‌کنند دسترسی زیادی داشته باشند. این موجب افزایش غیرضروری پیچیدگی سیستم می‌شود [۱۶]. یا زمانی که متدها یا کلاس‌ها بدون دلیل موارد داخلی خود را بیرونی می‌کنند [۷۵].

۵-۵-۶- نوع نهفته در نام

نوع نهفته در نام[3] یعنی قرار دادن نوع‌ها در نام متدها [۷۵]. نام‌هایی که معمولاً با تکرار تعریف می‌شوند، مانند schedule.addCourse(course) به جای schedule.add(course). این دسته، همچنین شامل استفاده از نمادگذاری مجارستانی و متغیرهایی است که نوع آنها را در نقطه مقابل هدف یا عملکرد آنها منعکس می‌کند [۷۲].

۵-۶- مبهم‌کننده‌ها

مبهم‌کننده‌ها[4] کدهایی هستند که باعث پنهان‌سازی و درک نامناسب کد می‌شوند. این دسته شامل هفت کد مشکوک به شرح زیر است.

۵-۶-۱-کد هوشمند

جرزیک و مادیسکی این‌گونه بیان کردند که ما در حال ایجاد تمایز آگاهانه بین نیت مبهم و کد هوشمند[5] هستیم. اگرچه نیت مبهم به ابهام پیاده‌سازی می‌پردازد و بر غیرقابل درک بودن یک قطعه کد تأکید می‌کند، از سوی دیگر، کد هوشمند می‌تواند گیج‌کننده باشد، حتی اگر قابل درک باشد. مواردی که در این کد مشکوک قرار می‌گیرند کدهایی هستند که کار عجیبی انجام می‌دهند. این را می‌توان با استفاده از پیچیدگی تصادفی یک زبان

[2] Indecent Exposure
[3] Type Embedded in Name
[4] Obfuscators
[5] Clever Code

مشخص یا ویژگی‌های پنهان آن دسته‌بندی کرد، و بالعکس، با استفاده از متدها و مکانیسم‌های آن زمانی که راه‌حل‌های آماده/ساخته‌شده در دسترس هستند [۱۴].
نمونه‌هایی از هر دو را می‌توان در اولین مثال ارائه‌شده یافت. این کد چرخه محاسبه طول یک رشته را دوباره اختراع می‌کند که یکی از موارد کد مشکوک کد هوشمند است. علاوه بر این، استفاده از length - = - 1 برای افزایش طول شمارنده نمونه دیگری از کد هوشمند است [۱۴].

```
message = 'Hello World!'
def get_length_of_string(message: str) -> int:
    length = 0
    for letter in message:
        length - = - 1
    return length
message_length = get_length_of_string(message)
print(message_length) # 12
# Solution
message_length = len(message)
print(message_length) # 12
```

شکل (۵): مثالی از کد هوشمند [۶۰]

متداول‌ترین وضعیت می‌تواند مربوط به هر کد پیاده‌سازی مجدد باشد (به‌عنوان‌مثال، ناشی از کد مشکوک کلاس کتابخانه ناقص). در نهایت، مواردی مانند if not game.match.isNotFinished() (منفی دوتایی) وجود دارد که بی‌دلیل بار شناختی موردنیاز برای پردازش آن را افزایش می‌دهد. می‌توان آن را در دسته کد هوشمند قرار داد، اما با تعریف کدهای مشکوک عبارت منطقی پیچیده و عملگر باینری در نام بیشتر مطابقت دارد [۶۰].

۵-۶-۲- عبارت منطقی پیچیده

کد مشکوک عبارت منطقی پیچیده[1] زمانی رخ می‌دهد که کد، شرط‌های پیچیده‌ای دارد که شامل «and»، «or» و «not» است. کد ممکن است پیچیده شروع شده باشد، یا ممکن است شرایط اضافی را در طول مسیر انتخاب کرده باشد. برای این کار می‌توان [۷۲]:

- از قوانین دمورگان استفاده کرد که می‌تواند برای ساده کردن عبارت به کار رود.
- ممکن است متوجه شوید که برخی از متغیرها اگر نامشان را تغییر دهند تا حس معکوس خود را نشان دهند، بهتر ارتباط برقرار می‌کنند.

▫ با معرفی متغیر توضیح‌دهنده، هر بند را واضح‌تر کنید.
▫ از بندهای نگهبان برای از بین بردن شرط‌های خاص استفاده کنید؛ بندهای باقیمانده ساده‌تر می‌شوند.

به‌طورکلی، یک تابع یا متد با نام مناسب، نسبت به هر عبارت منطقی که در آن استفاده می‌شود، در یک نگاه قابل فهم‌تر است.

۵-۶-۳- عبارت باقاعده پیچیده

دو کار ناپسند را می‌توان انجام داد که جرزیک و مادیسکی از آن به‌عنوان عبارت باقاعده پیچیده[2] یاد کردند. اول از همه، باید از استفاده غیرضروری از عبارات منظم (باقاعده) برای کارهای ساده خودداری کرد. عبارات منظم در همان چالش عبارات بولی پیچیده قرار می‌گیرند، با تنها تفاوتی که جمعیت انسانی که تحت‌تأثیر آن قرار می‌گیرد، بسیار بیشتر است. افراد بیشتری به سرعت معنای منطق بولی را درک می‌کنند، اما تعداد بسیار کمتری می‌توانند یک عبارات منظم را مثل یک کتاب بخوانند. اگر ضروری نیست، یا به عبارت ساده‌تر، اگر مجموعه کدی که می‌تواند یک رشته را تأیید کند، زمان بیشتری برای درک دیگران نسبت به معادل آن با عبارات باقاعده نیاز دارد، باید از آن اجتناب کرد [۱۴].

نکته دوم این است که ما دوست داریم در همه سطوح انتزاع، چیزهای قابل توضیحی داشته باشیم. این بدین معناست که ترجیحاً بهتر است یک کلاس با نام مناسب و با متدهایی که به‌درستی نام‌گذاری شده‌اند، داشته باشیم و همچنین رشته‌های طولانی با استفاده از متغیرهایی که به‌درستی نام‌گذاری شده‌اند، ترکیب شوند. این تغییر جزئی باعث افزایش قابلیت درک می‌شود، اگرچه احتمالاً امکان کپی - پیست کردن عبارات منظم را در یکی از ابزارهای آنلاین برای تجزیه عبارات منظم، قربانی می‌کند. توسعه‌دهندگان می‌توانند با افزودن خروجی «کامپایل‌شده» عبارات منظم در یک توضیح یا رشته مستند، این مشکل را کاهش دهند (اما پس از آن باید همراه با متد به‌روزرسانی شود که مشکوک است) [۱۴].

همچنین باید در نظر داشت که عبارات باقاعده و طولانی قابل‌توجهی وجود دارد که می‌توان آنها را پس از جستجوی سریع از اینترنت پیدا و کپی کرد. دریافت یک عبارت منظم استاندارد و تأییدشده اشکالی ندارد، اما اگر کسی تمایل دارد که برای نیازهای خاص خود عبارت منظم خود را ایجاد کند، باید مراقب باشد که آن را به‌خوبی تجزیه کند؛ به‌طوری‌که هیچ توسعه‌دهنده دیگری نیازی به اشکال‌زدایی مجموعه‌ای از کاراکترهای فشرده را نداشته باشد [۱۴].

۵-۶-۴- سبک ناسازگار

همان چیزی که در نام‌های ناسازگار در مورد قالب‌بندی کلی و سبک کد استفاده شده اتفاق می‌افتد، در پروژه نیز صدق می‌کند. مرور کد باید شبیه به خواندن یک مقاله یا یک کتاب خوب، یکسان و زیبا باشد. در پروژه، قالب کد نباید به‌طور ترجیحی یا به صورت تصادفی تغییر کند؛ بلکه باید یکنواخت باشد تا شکل

[1] Complicated Boolean Expression

[2] Complicated Regex Expression

مورد انتظار کد در خطوط بعدی را به هم نزند. نمونه دیگری از کد مشکوک سبک ناسازگار[1] می‌تواند ناهماهنگی توالی در ترتیب پارامترهای درون کلاس‌ها یا متدها باشد. پس از تعریف، ترتیب باید در گروه تمام انتزاعات مرتبط با همان موضوع خاص حفظ شود [۶۰].

۵-۶-۵- نیت مبهم

کد باید تا حد امکان رسا باشد. عبارات طولانی، نماد مجارستانی و اعداد جادویی، همگی نیت نویسنده را مبهم می‌کنند. این مورد کد مشکوک نیت مبهم[2] نامیده می‌شود. به‌عنوان‌مثال، در ادامه تابع overTimePay را مشاهده می‌کنید:

```
public int m_otCalc() {
  return iThsWkd * iThsRte +
    (int) Math.round(0.5 * iThsRte *
      Math.max(0, iThsWkd - 400)
    );
}
```

شکل (۶): مثالی از کد مشکوک نیت مبهم [۴۱]

هرچند ممکن است کوچک و فشرده به نظر برسد، اما تقریباً مبهم است. ارزش دارد که وقت بگذاریم تا نیت کد خود را برای خوانندگان قابل مشاهده کنیم [۴۱].

۵-۶-۶- متغیر وضعیت

متغیرهای وضعیت[3] از نوع‌های اولیه قابل تغییر هستند که قبل از یک عملیات برای ذخیره برخی اطلاعات براساس عملیات، مقداردهی اولیه می‌شوند و بعداً به‌عنوان سوئیچ برای برخی اقدامات استفاده می‌شوند. متغیرهای وضعیت را می‌توان به‌عنوان یک کد مشکوک مشخص شناسایی شوند؛ اگرچه آنها فقط یک سیگنال برای پنج کد مشکوک کد هوشمند، حلقه‌های امری، ترس از شکست، داده‌های قابل تغییر و نمونه خاص هستند. آنها در انواع و اشکال مختلف ظاهر می‌شوند؛ اما نمونه‌های متداول عبارت‌اند از success: bool = False-s قبل از اجرای یک بلوک عملیات یا i: int = 0 قبل از یک عبارت حلقه [۶۰].

۵-۶-۷- جداسازی عمودی

آخرین کد مشکوک در دسته مبهم‌کننده‌ها، جداسازی عمودی[4] نام دارد.

متغیرها و تابع باید نزدیک به محل استفاده تعریف شوند. متغیرهای محلی باید درست بالاتر از اولین استفاده خود تعریف شوند و باید دامنه عمودی کوچکی داشته باشند. ما نمی‌خواهیم که متغیرهای محلی صدها خط از محل استفاده خود فاصله داشته باشند. توابع خصوصی باید درست زیر اولین استفاده از آنها تعریف شوند. توابع خصوصی به دامنه کل کلاس تعلق دارند، اما ما همچنان می‌خواهیم فاصله عمودی بین فراخوان‌ها و تعاریف را محدود کنیم. پیدا کردن یک تابع خصوصی باید به‌سادگی با اسکن‌کردن به سمت پایین‌تر از اولین استفاده انجام شود [۴۱].

۵-۷- واسطه‌های داده

دسته واسطه‌های داده[5] شامل شش کد مشکوک به شرح زیر است.

۵-۷-۱- داده‌های سراسری

داده‌های سراسری[6] می‌توانند از هر جای کد تغییر یابند و هیچ سازوکاری برای کشف قسمتی از کد که آن را تغییر داده است، وجود ندارد. به عبارتی، هر کسی از هر جایی می‌تواند داده‌های سراسری را تغییر دهد و این امر یک مشکل محسوب می‌شود [۳].

این متغیرهای دامنه سراسری نامطلوب هستند؛ زیرا مستقیماً باعث کد مشکوک وابستگی‌های پنهان و کد مشکوک بسیار بد داده‌های قابل تغییر می‌شوند.

۵-۷-۲- وابستگی‌های پنهان

وابستگی‌های پنهان[7] وضعیتی است که در آن، در داخل یک کلاس، متدها به‌طور خاموش و بی‌صدا وابستگی‌ها را حل می‌کنند و این رفتار را از فراخواننده پنهان می‌کنند. وابستگی‌های مخفی می‌توانند باعث بروز استثناء در زمان اجرا شوند، زمانی که فراخواننده قبل از تنظیم محیط مناسب، آنها را فراخوانی کند که به تنهایی نشانه‌ای از وجود کد مشکوک دیگری است: کد راه‌اندازی/خاتمه موردنیاز [۶۰, ۷۶].

۵-۷-۳- معامله داخلی

کلاس‌ها باید تا حد امکان کمتر از یکدیگر اطلاعات داشته باشند. همان‌طور که فاولر گفته است: «کلاس‌ها زمان زیادی را صرف آشکارسازی بخش‌های خصوصی یکدیگر می‌کنند». این کد مشکوک در سال ۱۹۹۹ با نام صمیمیت نامناسب معرفی شد و بعد از آن در نسخه ۲۰۱۸ کتاب، با این نام ذکر نشده است. به‌طورکلی، کد مشکوک معامله داخلی یعنی مشکلات اتصال[8] ناشی از تبادل داده‌ها بین ماژول‌ها [۳].

[1] Inconsistent Style
[2] Obscured Intent
[3] Status Variables
[4] Vertical Separation
[5] Data Dealers
[6] Global Data
[7] Hidden Dependencies
[8] Coupling

۵-۷-۴- زنجیره پیام

یک زنجیره پیام[1] مجموعه‌ای از فراخوانی متدها بر روی اشیاء مختلف است. زنجیره‌های پیام اصل کپسوله‌سازی را نقض می‌کنند، زیرا ساختار داخلی کد را بیش از حد آشکار می‌کنند. زنجیره‌های پیام؛ درک، اصلاح و آزمون کد را سخت‌تر می‌کنند. زنجیره‌های پیام اغلب نتیجه طراحی ضعیف یا فقدان انتزاع در کد هستند [۷۳].

فرض کنید که کلاس A به داده‌هایی از کلاس D نیاز دارد، اما برای دریافت این داده‌ها، باید به‌طور متوالی فراخوانی‌های غیرضروری با کلاس B و سپس C برقرار کند. این توالی توابع، کد مشکوک زنجیره پیام نامیده می‌شود.

۵-۷-۵- دلال

اگر کلاسی تنها کاری که انجام می‌دهد انتقال فراخوانی به کلاس دیگری است، دیگر نیازی به این کلاس وجود نخواهد داشت. این مورد تحت عنوان کد مشکوک دلال[2] نامیده می‌شود. این برعکس زنجیره پیام است [۳].

۵-۷-۶- داده‌های بدون استفاده

کد مشکوک داده‌های بدون استفاده[3] بسیار شبیه به کد مشکوک زنجیره پیام است، با این تفاوت که در آنجا، به‌طورمثال تأمین‌کننده پیتزا، مجبور شد از طریق یک زنجیره طولانی از فراخوانی‌های متد عبور کند. در اینجا، تأمین‌کننده پیتزا علاوه بر آن که از طریق زنجیره طولانی از فراخوانی‌های متد عبور می‌کند، با پیتزا در هر یک از پارامترهای متد حاضر است. این نشان‌دهنده انتزاع مشکوک است. داده‌هایی که از طریق زنجیره‌های طولانی از فراخوانی‌ها عبور می‌کنند، احتمالاً حداقل در یکی از سطوح، با انتزاعی که توسط هر یک از رابط‌های روتین ارائه می‌شود، سازگار نیستند [۶۰, ۷۷].

۵-۸- سایر

در این دسته یک کد مشکوک مورد بررسی قرار می‌گیرد.

۵-۸-۱-کلاس کتابخانه ناقص

زمانی که کتابخانه‌ای آماده می‌شود، بالاخره روزی می‌رسد که این کتابخانه نیازهای پروژه را رفع نمی‌کند و نیاز به توسعه خواهد داشت. ولی ازآنجایی‌که کتابخانه‌ها به‌صورت فقط خواندنی در اختیار پروژه‌ها قرار می‌گیرند، درصورتی‌که توسعه‌دهنده اصلی از توسعه کتابخانه سر باز بزند، مشکلاتی به وجود خواهد آمد. در این حالت کد مشکوک کلاس کتابخانه ناقص[4] وجود خواهد داشت.

به‌طورکلی در این حالت، نرم‌افزار از یک کتابخانه استفاده می‌کند که کامل نیست، و به همین دلیل نیاز به افزونه‌هایی برای آن کتابخانه وجود دارد [۱۶].

۵-۸-۲- بازآرایی

با شناسایی انواع کدهای مشکوک، می‌توانیم اقداماتی را برای رفع آن‌ها انجام دهیم، مانند بازآرایی کد برای پیمانه‌ای کردن آن یا کاهش پیچیدگی کد برای بهبود قابلیت نگهداری و عملکرد [۷۳]. فرآیند بازآرایی برای از بین بردن کدهای مشکوک مضر و بهبود نگهداری و کیفیت نرم‌افزار بسیار حیاتی است [۲].

فاولر، بازآرایی را به‌عنوان یک راه‌حل اصلاحی پیشنهادی تعریف می‌کند که هدف آن بهبود کیفیت کد با اصلاح ساختار داخلی آن بدون تغییر رفتار اصلی کد است [۷۸].

بازآرایی، فرآیندی برای بهبود سیستم‌های نرم‌افزاری با اِعمال تبدیل‌هایی است که باید رفتار قابل مشاهده آنها را حفظ کند. یکی از چالش‌های اساسی در حوزه مهندسی نرم‌افزار، ارائه راهبردهایی برای تعیین اینکه کدام بازآرایی باید اعمال شود و زمان مناسب برای اعمال آنها است [۱۶].

برای اعمال بازآرایی، باید مواردی که قابلیت بازآرایی و بهبود دارند (فرصت‌های بازآرایی) را از کد منبع شناسایی کرد تا مشخص شود که بازآرایی کجا باید اعمال شود و سپس یکی از فنون بازآرایی پیشنهادی را برای اصلاح موارد شناسایی‌شده انتخاب کرد. خودکارکردن فرآیند تشخیص و تصحیح کد مشکوک، به‌ویژه در سیستم‌های نرم‌افزاری بزرگ ضروری است [۷۸].

هر بازآرایی می‌تواند از مجموعه‌ای از مراحل اولیه ساده تشکیل شود. اگر یک توسعه‌دهنده نمی‌داند از کجا شروع کند یا احساس می‌کند سردرگم شده است، این مراحل اولیه، راه خوبی برای شروع هستند. گاهی اوقات این مجموعه مراحل ابتدایی به‌عنوان یک فرآیند به نام فرآیند بازآرایی در نظر گرفته می‌شود [۱۶].

منس و توروه[5] [۷۹] یک فرآیند مشترک را شناسایی کردند که در آن عملیات‌های بازآرایی انجام می‌شوند:

۱) شناسایی قطعات کد با فرصت‌های بازآرایی.
۲) تعیین اینکه کدام بازآرایی می‌تواند در قطعه کد انتخاب‌شده اعمال شود.
۳) حصول اطمینان از اینکه بازآرایی انتخاب‌شده رفتار را حفظ می‌کند.

---
[1] Message Chain
[2] Middle Man
[3] Tramp Data
[4] Incomplete Library Class
[5] Mens and Tourwé

۴) اعمال بازآرایی انتخابی در مکان‌های مربوطه.
۵) ارزیابی اثر بازآرایی بر خصوصیات کیفی نرم‌افزار.
۶) حفظ سازگاری بین نمونه بازآرایی‌شده و سایر اجزای نرم‌افزار.

۶- نتیجه‌گیری

با پیشرفت علوم رایانه و زبان‌های برنامه‌نویسی، روزبه‌روز تعداد نرم‌افزارهایی که افراد در زندگی روزمره‌شان استفاده می‌کنند، افزایش یافته است و نرم‌افزارها در همه‌جا حضور دارند.

برای پاسخ به دغدغه‌های موجود، نرم‌افزارها باید ویژگی‌های مشخصی داشته باشند. قابلیت انعطاف‌پذیری، نگهداری و دارابودن ساختاری منسجم، ویژگی‌های اصلی یک نرم‌افزار خوب هستند. برای اینکه یک نرم‌افزار دارای چنین ویژگی‌هایی باشد، لازم است که برنامه‌نویسان در همان مراحل اولیهٔ توسعه نرم‌افزار، طبق اصول شیءگرایی عمل کنند. متأسفانه به‌خاطر کمبود وقت، سهل‌انگاری، کمبود نیرو و مشکلات مالی، در نوشتن بسیاری از نرم‌افزارها به این اصول توجه کافی نمی‌شود. همین بی‌توجهی منجر به ایجاد یک نرم‌افزار غیرمنسجم با ساختاری اشتباه یا پیچیده و یا گاه مشکل‌زا در آینده می‌شود. در علوم کامپیوتر این ساختارهای غیرمناسب تحت نام کد مشکوک معروف هستند.

توسعه‌دهندگان نرم‌افزار درصددند با شناسایی کدهای مشکوک بتوانند بازآرایی نرم‌افزار را انجام داده و نابسامانی کد را به حداقل برسانند. تشخیص کدهای مشکوک، یک حوزه پژوهشی پرطرفدار در تکامل نرم‌افزار است که در حال حاضر مورد توجه فراوانی قرار گرفته است.

در این مقاله، با ارائه یک آرایه‌شناسی جدید به بررسی انواع کدهای مشکوک در کد برنامه نرم‌افزاری پرداخته شد. آرایه‌شناسی مذکور، شامل ۵۸ کد مشکوک است که در هشت دسته‌بندی مورد بررسی قرار گرفت.

۷- مراجع

[1] Y. Zhang, C. Ge, H. Liu, and K. Zheng, "Code smell detection based on supervised learning models: A survey," Neurocomputing, vol. 565, p. 127014, 2024, https://doi.org/10.1016/j.neucom.2023.127014

[2] A. Abdou and N. Darwish, "Severity classification of software code smells using machine learning techniques: A comparative study," J. Softw. (Malden), vol. 36, no. 1, p. e2454, 2024, https://doi.org/10.1002/smr.2454

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

[4] F. A. Fontana, M. V. Mäntylä, M. Zanoni, and A. Marino, "Comparing and experimenting machine learning techniques for code smell detection," Empir. Softw. Eng., vol. 21, pp. 1143-1191, 2016, http://dx.doi.org/10.1007/s10664-015-9378-4

[5] C. Marinescu, R. Marinescu, P. Mihancea, D. Ratiu, and R. Wettel, "iPlasma: An Integrated Platform for Quality Assessment of Object-Oriented Design.[C]," IEEE Int. Conf. Softw. Maint. Ind. Tool Vol., 2005: DBLP.

[6] F. Palomba, R. Oliveto, and A. De Lucia, "Investigating code smell co-occurrences using association rule learning: A replicated study," 2017 IEEE Workshop Mach. Learn. Tech. Softw. Qual. Eval. (MaLTeSQuE), 2017: IEEE, pp. 8-13, https://doi.org/10.1109/MALTESQUE.2017.7882010

[7] A. Garg, M. Gupta, G. Bansal, B. Mishra, and V. Bajpai, "Do bad smells follow some pattern?," Proc. Int. Congr. Inf. Commun. Technol. (ICICT), 2016: Springer, pp. 39-46, http://dx.doi.org/10.1007/978-981-10-0767-5_5

[8] J. Wang, J. Chen, and J. Gao, "ECC Multi-Label Code Smell Detection Method Based on Ranking Loss," J. Comput. Res. Dev., vol. 58, no. 1, pp. 178-188, 2021, https://dx.doi.org/10.7544/issn1000-1239.2021.20190836

[9] K. Kaur and S. Jain, "Evaluation of machine learning approaches for change-proneness prediction using code smells," Proc. 5th Int. Conf. Front. Intell. Comput. (FICTA), 2017: Springer, pp. 561-572, http://dx.doi.org/10.1007/978-981-10-3153-3_56

[10] R. Wieman, "Anti-pattern Scanner: an approach to detect anti-patterns and design violations." LAP Lambert Academic Publishing, 2011.

[11] T. F. M. Sirqueira, A. H. M. Brandl, E. J. P. Pedro, R. de Souza Silva, and M. A. P. Araujo, "Code smell analyzer: a tool to teaching support of refactoring techniques source code," IEEE Lat. Am. Trans., vol. 14, no. 2, pp. 877-884, 2016, https://doi.org/10.1109/TLA.2016.7437235

[12] G. Szőke, C. Nagy, L. J. Fülöp, R. Ferenc, and T. Gyimóthy, "FaultBuster: An automatic code smell refactoring toolset," 2015 IEEE 15th Int. Work. Conf. Source Code Anal. Manip. (SCAM), 2015: IEEE, pp. 253-258, https://doi.org/10.1109/SCAM.2015.7335422

[13] A. AbuHassan, M. Alshayeb, and L. Ghouti, "Software smell detection techniques: A systematic literature review," J. Softw. (Malden), vol. 33, no. 3, p. e2320, 2021, https://doi.org/10.1002/smr.2320

[14] M. Jerzyk and L. Madeyski, "Code Smells: A Comprehensive Online Catalog and Taxonomy," in Developments in Information and Knowledge Management Systems for Business Applications: Volume 7: Springer, 2023, pp. 543-576.

[15] S. Dewangan, R. S. Rao, A. Mishra, and M. Gupta, "A novel approach for code smell detection: an empirical study," IEEE Access, vol. 9, pp. 162869-162883, 2021, https://doi.org/10.1109/ACCESS.2021.3133810

[16] G. Lacerda, F. Petrillo, M. Pimenta, and Y. G. Guéhéneuc, "Code smells and refactoring: A tertiary systematic review of challenges and observations," J. Syst. Softw., vol. 167, p. 110610, 2020, https://doi.org/10.1016/j.jss.2020.110610

[17] R. Sandouka and H. Aljamaan, "Python code smells detection using conventional machine learning models," PeerJ Comput. Sci., vol. 9, p. e1370, 2023, https://doi.org/10.7717/peerj-cs.1370

[18] N. Vatanapakorn, C. Soomlek, and P. Seresangtakul, "Python code smell detection using machine learning," 2022 26th Int. Comput. Sci. Eng. Conf. (ICSEC), 2022: IEEE, pp. 128-133, https://doi.org/10.1109/ICSEC56337.2022.10049330

[19] A. S. Abdou and N. R. Darwish, "Early prediction of software defect using ensemble learning: A comparative study," Int. J. Comput. Appl., vol. 179, no. 46, pp. 29-40, 2018, http://dx.doi.org/10.5120/ijca2018917185

[20] E. V. de Paulo Sobrinho, A. De Lucia, and M. de Almeida Maia, "A systematic literature review on bad smells–5 w's: which, when, what, who, where," IEEE Trans. Softw. Eng., vol. 47, no. 1, pp. 17-66, 2018, https://doi.org/10.1109/TSE.2018.2880977

[21] Y. Tian, K. Li, T. Wang, Q. Jiao, G. Li, Y. Zhang, and H. Liu, "Survey on Code Smells. Ruan Jian Xue Bao," J. Softw., vol. 34, no. 1, pp. 150-170, 2023, http://dx.doi.org/10.13328/j.cnki.jos.006431

[22] F. L. Caram, B. R. D. O. Rodrigues, A. S. Campanelli, and F. S. Parreiras, "Machine learning techniques for code smells detection: a systematic mapping study," Int. J. Softw. Eng. Knowl. Eng., vol. 29, no. 02, pp. 285-316, 2019, https://doi.org/10.1142/S021819401950013X

[23] M. I. Azeem, F. Palomba, L. Shi, and Q. Wang, "Machine learning techniques for code smell detection: A systematic literature review and meta-analysis," Inf. Softw. Technol., vol. 108, pp. 115-138, 2019, https://doi.org/10.1016/j.infsof.2018.12.009

[24] S. S. Rathore and S. Kumar, "A study on software fault

[44] C. U. Smith and L. G. Williams, "More new software performance antipatterns: Even more ways to shoot yourself in the foot," Comput. Meas. Group Conf., 2003: Citeseer, pp. 717-725.
[45] C. U. Smith and L. G. Williams, "New software performance antipatterns: More ways to shoot yourself in the foot," Int. CMG Conf., 2002, pp. 667-674.
[46] M. Tufano, F. Palomba, G. Bavota, M. Di Penta, R. Oliveto, A. De Lucia, and D. Poshyvanyk, "An empirical investigation into the nature of test smells," Proc. 31st IEEE/ACM Int. Conf. Autom. Softw. Eng., 2016, pp. 4-15, https://doi.org/10.1145/2970276.2970340
[47] H. Mumtaz, M. Alshayeb, S. Mahmood, and M. Niazi, "A survey on UML model smells detection techniques for software refactoring," J. Softw. (Malden), vol. 31, no. 3, p. e2154, 2019, https://doi.org/10.1002/smr.2154
[48] T. Arendt and G. Taentzer, "UML model smells and model refactorings in early software development phases," Universitat Marburg, 2010.
[49] E. Doğan and E. Tüzün, "Towards a taxonomy of code review smells," Inf. Softw. Technol., vol. 142, p. 106737, 2022, https://doi.org/10.1016/j.infsof.2021.106737
[50] F. Palomba, D. A. Tamburri, A. Serebrenik, A. Zaidman, F. A. Fontana, and R. Oliveto, "Poster: How Do Community Smells Influence Code Smells?," 2018 IEEE/ACM 40th Int. Conf. Softw. Eng. Companion (ICSE-Companion), 2018: IEEE, pp. 240-241.
[51] K. A. Qamar, E. Sülün, and E. Tüzün, "Towards a taxonomy of bug tracking process smells: A quantitative analysis," 2021 47th Euromicro Conf. Softw. Eng. Adv. Appl. (SEAA), 2021: IEEE, pp. 138-147, https://doi.org/10.1109/SEAA53835.2021.00026
[52] T. Sharma, M. Fragkoulis, and D. Spinellis, "Does your configuration code smell?," Proc. 13th Int. Conf. Min. Softw. Repos., 2016, pp. 189-200, https://doi.org/10.1145/2901739.2901761
[53] T. Sharma, "Presentation smells: How not to prepare your conference presentation," ed, 2016.
[54] F. Hermans, M. Pinzger, and A. Van Deursen, "Detecting and visualizing inter-worksheet smells in spreadsheets," 2012 34th Int. Conf. Softw. Eng. (ICSE), 2012: IEEE, pp. 441-451, https://doi.org/10.1109/ICSE.2012.6227171
[55] B. Karwin, "SQL Antipatterns: Avoiding the pitfalls of database programming," Pragmat. Bookshelf, pp. 15-155, 2010.
[56] D. Almeida, J. C. Campos, J. Saraiva, and J. C. Silva, "Towards a catalog of usability smells," Proc. 30th Annu. ACM Symp. Appl. Comput., 2015, pp. 175-181, https://doi.org/10.1145/2695664.2695670
[57] A. Carette, M. A. A. Younes, G. Hecht, N. Moha, and R. Rouvoy, "Investigating the energy impact of android smells," 2017 IEEE 24th Int. Conf. Softw. Anal. Evol. Reeng. (SANER), 2017: IEEE, pp. 115-126, https://doi.org/10.1109/SANER.2017.7884614
[58] M. Ghafari, P. Gadient, and O. Nierstrasz, "Security smells in android," 2017 IEEE 17th Int. Work. Conf. Source Code Anal. Manip. (SCAM), 2017: IEEE, pp. 121-130, https://doi.org/10.1109/SCAM.2017.24
[59] M. Stijlaart and V. Zaytsev, "Towards a taxonomy of grammar smells," Proc. 10th ACM SIGPLAN Int. conf. Softw. Lang. Eng., 2017, pp. 43-54, https://doi.org/10.1145/3136014.3136035
[60] N. Kryvinska, M. Greguš, and S. Fedushko, "Developments in Information and Knowledge Management Systems for Business Applications: Volume 7." Springer Nature, 2023.
[61] K. Alkharabsheh, Y. Crespo, E. Manso, and J. A. Taboada, "Software design smell detection: a systematic mapping study," Softw. Qual. J., vol. 27, pp. 1069-1148, 2019, https://doi.org/10.1007/s11219-018-9424-8
[62] M. V. Mäntylä and C. Lassenius, "Subjective evaluation of software evolvability using code smells: An empirical study," Empir. Softw. Eng., vol. 11, pp. 395-431, 2006, http://dx.doi.org/10.1007/s10664-006-9002-8
[63] A. Al-Shaaby, H. Aljamaan, and M. Alshayeb, "Bad smell detection using machine learning techniques: a systematic literature review," Arab. J. Sci. Eng., vol. 45, no. 4, pp. 2341-2369, 2020, https://doi.org/10.1007/s13369-019-04311-w
prediction techniques," Artif. Intell. Rev., vol. 51, pp. 255-327, 2019, https://doi.org/10.1007/s10462-017-9563-5
[25] S. Jain and A. Saha, "Improving performance with hybrid feature selection and ensemble machine learning techniques for code smell detection," Sci. Comput. Program., vol. 212, p. 102713, 2021, https://doi.org/10.1016/j.scico.2021.102713
[26] M. Tufano, F. Palomba, G. Bavota, R. Oliveto, M. Di Penta, A. De Lucia, and D. Poshyvanyk, "When and why your code starts to smell bad," 2015 IEEE/ACM 37th IEEE Int. Conf. Softw. Eng., 2015, vol. 1: IEEE, pp. 403-414, https://doi.org/10.1109/ICSE.2015.59
[27] A. Gupta, B. Suri, and S. Misra, "A systematic literature review: code bad smells in java source code," 17th Int. Conf. Comput. Sci. Appl. (ICCSA), 2017: Springer, pp. 665-682, http://dx.doi.org/10.1007/978-3-319-62404-4_49
[28] D. Mahalakshmi, P. Kasinathan, D. Elangovan, C. R. Bhat, M. Balamurugan, and S. Sivakumar, "Code Smell Detection using Hybrid Machine Learning Algorithms," 2023 5th Int. Conf. Inventive Res. Comput. Appl. (ICIRCA), 2023: IEEE, pp. 633-638, https://doi.org/10.1109/ICIRCA57980.2023.10220911
[29] S. Subedi, "Intelligent Code Smell Detection System Using Deep Learning," Pulchowk Campus, 2021.
[30] M. Fowler, "Refactoring: Improving the Design of Existing Code." Addison Wesley, 1999.
[31] F. Palomba, G. Bavota, M. Di Penta, F. Fasano, R. Oliveto, and A. De Lucia, "On the diffuseness and the impact on maintainability of code smells: a large scale empirical investigation," Proc. 40th Int. Conf. Softw. Eng., 2018, pp. 482-482, https://doi.org/10.1145/3180155.3182532
[32] A. Tahir, S. Counsell, and S. G. MacDonell, "An empirical study into the relationship between class features and test smells," 2016 23rd Asia-Pacific Softw. Eng. Conf. (APSEC), 2016: IEEE, pp. 137-144, https://doi.org/10.1109/APSEC.2016.029
[33] N. Tsantalis and A. Chatzigeorgiou, "Identification of move method refactoring opportunities," IEEE Trans. Softw. Eng., vol. 35, no. 3, pp. 347-367, 2009, https://doi.org/10.1109/TSE.2009.1
[34] N. Moha, Y.-G. Guéhéneuc, L. Duchien, and A.-F. Le Meur, "Decor: A method for the specification and detection of code and design smells," IEEE Trans. Softw. Eng., vol. 36, no. 1, pp. 20-36, 2009, https://doi.org/10.1109/TSE.2009.50
[35] D. A. Tamburri, F. Palomba, A. Serebrenik, and A. Zaidman, "Discovering community patterns in open-source: a systematic approach and its evaluation," Empir. Softw. Eng., vol. 24, pp. 1369-1417, 2019, https://doi.org/10.1007/s10664-018-9659-9
[36] F. Khomh, M. D. Penta, Y.-G. Guéhéneuc, and G. Antoniol, "An exploratory study of the impact of antipatterns on class change-and fault-proneness," Empir. Softw. Eng., vol. 17, pp. 243-275, 2012, http://dx.doi.org/10.1007/s10664-011-9171-y
[37] J. Garcia, D. Popescu, G. Edwards, and N. Medvidovic, "Toward a catalogue of architectural bad smells," Archit. Adapt. Softw. Syst., 2009: Springer, pp. 146-162, https://doi.org/10.1007/978-3-642-02351-4_10
[38] G. Suryanarayana, G. Samarthyam, and T. Sharma, "Refactoring for software design smells: managing technical debt." Morgan Kaufmann, 2014.
[39] T. Sharma, M. Fragkoulis, and D. Spinellis, "House of cards: Code smells in open-source c# repositories," 2017 ACM/IEEE Int. Symp. Empir. Softw. Eng. Meas. (ESEM), 2017: IEEE, pp. 424-429, https://doi.org/10.1109/ESEM.2017.57
[40] E. Jabrayilzade, O. Gürkan, and E. Tüzün, "Towards a taxonomy of inline code comment smells," 2021 IEEE 21st Int. Work. Conf. Source Code Anal. Manip. (SCAM), 2021: IEEE, pp. 131-135, https://doi.org/10.1109/SCAM52516.2021.00024
[41] R. C. Martin, "Clean code: a handbook of agile software craftsmanship." Pearson Education, 2009.
[42] V. Arnaoudova, M. Di Penta, G. Antoniol, and Y.-G. Guéhéneuc, "A new family of software anti-patterns: Linguistic anti-patterns," 2013 17th Eur. Conf. Softw. Maint. Reeng., 2013: IEEE, pp. 187-196, https://doi.org/10.1109/CSMR.2013.28
[43] A. Vetro, L. Ardito, G. Procaccianti, and M. Morisio, "Definition, implementation and validation of energy code smells: an exploratory study on an embedded system," Third Int. Conf. Smart Grids Green Commun. IT Energy-aware Technol., 2013, pp. 34-39.

[64] M. Mantyla, “Bad smells in software-a taxonomy and an empirical study,” PhD thesis, Helsinki University of Technology, 2003.

[65] A. Kaur, “A systematic literature review on empirical analysis of the relationship between code smells and software quality attributes,” Arch. Comput. Methods Eng., vol. 27, no. 4, pp. 1267-1296, 2020, https://doi.org/10.1007/s11831-019-09348-6

[66] A. Kaur, S. Jain, S. Goel, and G. Dhiman, “WITHDRAWN: Prioritization of code smells in object-oriented software: A review,” ed: Elsevier, 2021. https://doi.org/10.1016/j.matpr.2020.11.218

[67] F. Sabir, F. Palma, G. Rasool, Y. G. Guéhéneuc, and N. Moha, “A systematic literature review on the detection of smells and their evolution in object-oriented and service-oriented systems,” Softw. Pract. Exp., vol. 49, no. 1, pp. 3-39, 2019, https://doi.org/10.1002/spe.2639

[68] R. Marticorena, C. López, and Y. Crespo, “Extending a taxonomy of bad code smells with metrics,” Proc. 7th Int. Workshop Object Oriented Reeng. (WOOR), 2006: Citeseer, p. 6.

[69] P. Khamkhiaw, C. Doungsa-ard, and P. Phannachitta, “The Source Code Maintenance Time Classifications from Code Smell,” Int. Conf. Emerg. Internetwork. Data Web Technol., 2023: Springer, pp. 22-32, https://doi.org/10.1007/978-3-031-26281-4_3

[70] M. S. Haque, J. Carver, and T. Atkison, “Causes, impacts, and detection approaches of code smell: a survey,” Proc. ACMSE 2018 Conf., 2018, pp. 1-8, https://doi.org/10.1145/3190645.3190697

[71] M. Martin and R. C. Martin, “Agile principles, patterns, and practices in C.” Pearson Education, 2006.

[72] W. C. Wake, “Refactoring workbook.” Addison-Wesley Professional, 2004.

[73] G. Saranya, D. Mishra, V. Srikar, C. Abhilash, and S. Dooda, “Code Smell Detection Using a Weighted Cockroach Swarm Optimization Algorithm,” 2023 14th Int. Conf. Comput. Commun. Netw. Technol. (ICCCNT), 2023: IEEE, pp. 1-8, https://doi.org/10.1109/ICCCNT56998.2023.10306683

[74] S. Jain and A. Saha, “Rank-based univariate feature selection methods on machine learning classifiers for code smell detection,” Evol. Intell., vol. 15, no. 1, pp. 609-638, 2022, https://doi.org/10.1007/s12065-020-00536-z

[75] L. Bamizadeh, B. Kumar, A. Kumar, and S. Shirwaikar, “Design and Implementation of a Web-Based Application for Code Smells Repository,” Tehnički glasnik, vol. 15, no. 3, pp. 371-380, 2021, https://doi.org/10.31803/tg-20210207102610

[76] Z. Yu and V. Rajlich, “Hidden dependencies in program comprehension and change propagation,” Proc. 9th Int. Workshop Program Comprehension (IWPC) 2001: IEEE, pp. 293-299, https://doi.org/10.1109/WPC.2001.921739

[77] S. McConnell, “Code complete.” Pearson Education, 2004.

[78] M. Alharbi and M. Alshayeb, “A Comparative Study of Automated Refactoring Tools,” IEEE Access, 2024, https://doi.org/10.1109/ACCESS.2024.3361314

[79] T. Mens and T. Tourwé, “A survey of software refactoring,” IEEE Trans. Softw. Eng., vol. 30, no. 2, pp. 126-139, 2004, https://doi.org/10.1109/TSE.2004.1265817