ITRC ICT Research Institute (Iran Telecommunication Research Center)

Journal of Information and Communication Technology

Iranian Society of Information and Communication Technology

Volume 16, Issue 59-60, Spring and Summer 2024, pp. 228-239

Regression Test Time Reduction in Test Driven Development

Zohreh Mafi[1*], Seyed-Hassan Mirian-Hosseinabadi[2]

[1] Computer Engineering Department, Sharif University of Technology, Kish, Iran and Faculty Member of ICT Research Institute, Tehran, Iran

[2] Computer Engineering Department, Sharif University of Technology, Tehran, Iran

Received: 01 May 2023, Revised: 05 June 2023, Accepted: 30 July 2023

Paper type: Research

Abstract

Test-Driven Development (TDD) is one of the test-first software production methods in which the production of each component of the code begins with writing the test case. This method has been noticed due to many advantages, including the readable, regular, and short code, as well as increasing quality, productivity, and reliability. The large number of unit test cases produced in this method is considered as an advantage (increases the reliability of the code), however, the repeated execution of test cases increases the regression test time. The purpose of this article is to present an algorithm for selecting test cases to reduce the time of the regression test in the TDD method. So far, various ideas have been proposed to select test cases. Most of these ideas are based on programming language and software production methods. The idea presented in this article is based on the program difference method and the nature of the TDD method, also a tool is written as a plugin in Java Eclipse. The provided tool consists of five main components: 1) Version Manager, 2) Code Segmentation, 3) Code Change Detection (in each version compared to the previous version), 4) Semantic Connection Creation (between unit tests and code blocks), and finally 5) Test Cases Selection.

Keywords: Test Driven Development (TDD), Unit Test, Regression Test, Program Differencing, Segmentation, Version Control.

[*] Corresponding Author's email: z.mafi@itrc.ac.ir

دوفصلنامه
فناوری اطلاعات و ارتباطات ایران

سال شانزدهم، شماره های ۵۹ و ۶۰، بهار و تابستان ۱۴۰۳، صفحه ۲۲۸ الی ۲۳۹

کاهش زمان آزمون بازگشت در روش تولید آزمون‌رانه

زهره مافی[1*]، سیدحسن میریان حسین آبادی[2]

[1] دانشجوی دکتری دانشگاه صنعتی شریف و عضو هیات علمی پژوهشگاه ارتباطات و فناوری اطلاعات، تهران، ایران

[2] دانشیار دانشگاه صنعتی شریف، دانشکده مهندسی کامپیوتر، تهران، ایران

تاریخ دریافت: ۱۴۰۲/۰۲/۱۱ تاریخ بازبینی: ۱۴۰۲/۰۳/۱۵ تاریخ پذیرش: ۱۴۰۲/۰۵/۰۸

نوع مقاله: پژوهشی

چکیده

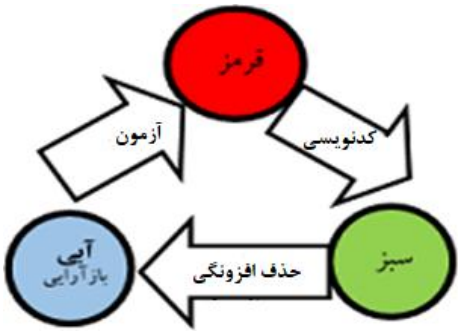
تولید آزمون‌رانه یکی از شیوه‌های تولید نرم‌افزار اول آزمون است که در آن تولید هر جزء از کد با نوشتن آزمون شروع می‌گردد. این شیوه به دلیل مزایای فراوان ازجمله تولید کد خوانا، منظم، کوتاه و همچنین افزایش کیفیت، بهره‌وری و قابلیت اطمینان موردتوجه قرار گرفته است. تعداد زیاد موارد آزمون تولیدشده در این روش به عنوان نقطه قوتی جهت افزایش قابلیت اطمینان مطرح است با این حال اجرای مکرر موارد آزمون، موجب افزایش زمان آزمون بازگشت است. هدف این مقاله ارائه روش انتخاب موارد آزمون جهت کاهش زمان آزمون بازگشت در شیوه تولید آزمون‌رانه است. تاکنون ایده‌های مختلفی برای انتخاب موارد آزمون مطرح شده است. اغلب این ایده‌ها مبتنی بر زبان برنامه‌نویسی و شیوه تولید نرم‌افزار است. ایده ارائه شده در این مقاله مبتنی بر روش اختلاف برنامه و ماهیت شیوه تولید آزمون‌رانه اتخاذ گردیده است و ابزاری به صورت یک پلاگین در محیط Eclipse برای برنامه‌های زبان جاوا نوشته شده است. ابزار ارائه شده از پنج مولفه اصلی ۱) مدیریت نسخه‌های برنامه، ۲) بلاک‌بندی کد تولید شده، ۳) تشخیص بلاک‌های تغییریافته در هر نسخه نسبت به نسخه قبل، ۴) ایجاد ارتباط معنایی بین آزمون‌های واحد و بلاک‌های کد و ۵) انتخاب موارد آزمون تشکیل شده است.

کلیدواژگان: آزمون نرم‌افزار، تولید آزمون‌رانه، آزمون بازگشت، اختلاف دو برنامه، بخش‌بندی، کنترل نسخه.

* رایانامهٔ نویسنده مسؤول: z.mafi@itrc.ac.ir

۱- مقدمه

تولید آزمون‌رانه[1] یکی از روش‌های سبک‌وزن[2] و افزایشی[3] تولید نرم‌افزار و مبتنی بر متدلوژی XP[4] است [۱] که در آن ایده‌ی ساختن آزمون‌ها پیش از تولید کد اصلی با ایده بازآرایی[5] [۲] تلفیق می‌شود. تولید برنامه در این روش طبق چرخه سه مرحله‌ای قرمز/سبز/آبی (بازآرایی) پیش می‌رود که شامل نوشتن آزمون برای رفع نیازمندی، نوشتن کد برنامه تا برطرف شدن خطا و در نهایت حذف افزونگی و بازآرایی کد مطابق شکل ۱ است. مرحله‌ی قرمز، نمایانگر وجود خطا یا کاستی و انگیزه‌ای برای نوشتن کد است. مرحله‌ی سبز به معنای گذرانده شدن آزمون است. پس از گذرانده شدن آزمون، کد نوشته شده نیاز به بهبود و بازآرایی دارد. مرحله‌ی آبی مربوط به انجام بازآرایی و به معنای وضعیت ایده‌آل و آمادگی برای شروع مرحله‌ی بعدی و اضافه کردن آزمون جدید است [۱].



شکل ۱. چرخه سه مرحله‌ای تولید آزمون‌رانه

این روش، میزان کاستی‌ها و خطاها و هم‌چنین هزینه‌ی تولید، عیب‌یابی و نگهداری را کاهش می‌دهد و کدهای با کیفیت بالاتری را تولید می‌کند. البته پیش هزینه‌ها باعث می‌شود که روش آزمون‌رانه در اولین نسخه پروژه، هزینه و زمان بیشتری لازم داشته باشد و مزایای آن تا زمانی که دومین نسخه از پروژه منتشر نشود، آشکار نخواهد شد [۳]. این شیوه به دلیل مزایای آن، از جمله خوانایی، کوتاهی، نظم وسادگی کد، بالا بودن اطمینان کاربران از کد [۴] و [۵] و امکان آزمون بازگشت[6] به دلیل ایجاد مجموعه‌ی بزرگ از موارد آزمون[7] مورد توجه و استقبال تولیدکنندگان نرم‌افزار قرار گرفته است.

اعمال تغییرات و اصلاحات در زمان تولید برنامه ممکن است تاثیرات نامطلوبی بر کیفیت و قابلیت اطمینان نرم‌افزار داشته باشد، به‌طوری‌که رفتار آزمون شده‌ی سیستم نرم‌افزاری پسرفت نماید. نتیجه‌ی این فرایند ایجاد اشکالاتی موسوم به خطای بازگشت خواهد بود. در صورتی که عملکردهای صحیح قبلی دچار اختلال شده باشند، خطای بازگشت اتفاق افتاده است. ازاین‌رو آزمون بازگشت [۶] در فرآیند تولید نرم‌افزار اهمیت پیدا می‌کند. آزمون بازگشت به منظور فراهم نمودن اطمینان از اینکه ویژگی‌های جدید سیستم تحت آزمون، با ویژگی‌های موجود تعارضی ندارند، انجام می‌گیرد.

در روش‌های سنتی هر یک از فازها به‌طور کامل اجرا می‌گردد و فاز آزمون پس از تکمیل فازهای تعریف نیازمندی‌ها، طراحی سیستم و پیاده‌سازی انجام می‌گردد، درحالی‌که در روش‌های چابک فرآیند توسعهٔ نرم‌افزار در مراحل اولیه جزئی است و به تدریج تکامل می‌یابد و در هر چرخه تمام فازها انجام می‌گردد. بنابراین آزمون در متدلوژی‌های سنتی یک بار و در انتهای فاز پیاده‌سازی انجام می‌شود درحالی‌که در متدلوژی‌های چابک در هر چرخه انجام می‌شود و تعداد دفعات آزمون افزایش می‌یابد[۷].

مقدار آزمون‌هایی که در روش آزمون‌رانه تولید می‌شود دو تا سه برابر آزمون با روش‌های دیگر است و این روش ممکن است هزاران آزمون واحد تولید کند [۸]. بنابراین زمان آزمون بازگشت در این روش، افزایش قابل توجهی می‌یابد. از طرف دیگر در این روش، پس از هر تغییر لازم است تمامی موارد آزمون مجددا اجرا گردند تا از درستی کد پس از تغییر، اطمینان حاصل شود. در نتیجه زمان زیادی برای اجرای موارد آزمون در این روش صرف می‌شود. همان طور که در محاسبات انجام شده در تحقیق قبلی [۹] اشاره شده است، تعداد موارد آزمون اجرا شده از مرتبه توان دوی تعداد موارد آزمون می‌باشد. به طور کلی از آنجا که نمی‌توان به ازای تمام ورودی‌ها، آزمون را انجام داد، معیارهای متعددی برای انتخاب برخی موارد آزمون به میان می‌آید [۱۰] که در بخش۲ توضیح داده شده است.

این مقاله روشی برای کاهش زمان آزمون بازگشت مبتنی بر روش تولید آزمون‌رانه با پوشش کد مربوط به اختلاف دو نسخه برنامه[8] ارائه می‌دهد و همان‌طور که ابزارهای خاصی برای شیوه‌های خاص تولید برنامه ارائه شده‌اند، این تحقیق نیز ابزاری پیاده‌سازی می‌کند که به طور خاص به برنامه‌هایی اختصاص دارد که به شیوه آزمون‌رانه تولید می‌شوند.

معیارهای مورد بررسی در این تحقیق عبارتند از تعداد موارد آزمون

[1] Test Driven Development (TDD)
[2] Light Weight
[3] Incremental
[4] eXtreme Programming
[5] Refactoring
[6] Regression Test
[7] Test Case
[8] Program Differencing

انتخابی (یک عدد صحیح مثبت) و زمان آزمون بازگشت که برحسب میلی‌ثانیه بیان می‌گردد.

در ادامه بخش ۲ ادبیات پژوهش در زمینه کاهش زمان آزمون بازگشت است و تمرکز آن بر تحقیقات پیشین به شیوه انتخاب موارد آزمون مبتنی بر روش اختلاف دو نسخه برنامه است. بخش ۳ الگوریتم پیشنهادی و پنج گام اساسی آن را توضیح می‌دهد. بخش ۴ به پیاده‌سازی روش پیشنهادی و معرفی ابزار تولید شده و همچنین تحلیل و ارزیابی آن می‌پردازد. بخش ۵ خلاصه و نتیجه‌گیری است.

۲- ادبیات پژوهش

در روش‌های سنگین‌وزن سنتی کلیهٔ رفتارهای درست در ابتدا کاملاً تعریف می‌گردد اما در متدلوژی‌های سبک‌وزن مانند روش‌های چابک، درستی رفتار نسبت به مجموعهٔ خاصی از آزمون‌ها بازتعریف می‌شود. اگر نرم‌افزار در آزمون‌ها رفتار درستی داشته باشد، درست دانسته می‌شود و درستی به کمک آزمون اعتبارسنجی می‌شود. بنابراین آزمون اهمیت بیشتری دارد. لذا الزامات ذیل به‌وجود می‌آید [7]:

- آزمون‌ها باید خودکار باشند (خودکارسازی پیش‌نیاز تولید آزمون‌رانه است).
- هر آزمون باید شامل اوراکل آزمونی[1] باشد که مشخص کند آیا به‌درستی اجرا می‌شود یا خیر.
- آزمون‌ها جانشین نیازمندی‌ها می‌شوند.
- آزمون‌ها باید باکیفیت و با سرعت اجرای بالا باشند.
- هر زمان که نرم‌افزار تغییر یابد، آزمون‌ها باید مجدد اجرا شوند.

در روش چابک تمرکز آزمون‌ها روی مسیرهای متداول[2] استفاده از نرم‌افزار است و برخی از مسیرهای نادرست مربوط به کاربران ناوارد، متخاصم یا خلاق درنظر گرفته نمی‌شود. بنابراین در طراحی آزمون‌ها بایستی مدل‌سازی نرم‌افزار بر اساس افراز دامنه ورودی، گراف، منطق یا گرامر در نظرگرفته شود و معیارهای پوشش مربوطه برای آن لحاظ گردد [۷].

در متدلوژی‌های سنتی، آزمون بازگشت، در زمان مشخص و توسط گروه آزمون‌گر انجام می‌شد. اما در سال‌های اخیر آزمون بازگشت با افزایش محبوبیت متدلوژی‌های سبک‌وزن از جمله روش‌های چابک شاهد تغییر وضعیت و تاکید بیشتر بوده است. امروزه آزمون بازگشت در روش‌های چابک، به خصوص آزمون‌رانه، نقش محوری در حفظ کیفیت نرم‌افزار پیدا کرده است و به‌طور فزاینده و مداوم و به عنوان یک عمل اساسی در طی فرایند تولید و توسط افراد گروه تولید، انجام می‌گردد. تولید آزمون‌رانه، به‌طورکلی، آزمون و به‌طور خاص آزمون بازگشت را به مرکز تولید نرم‌افزار آورده است و ماهیت آن را تغییر داده است [۱۱]. بنابراین با توجه به افزایش تعداد دفعات آزمون، کاهش زمان آزمون بازگشت اهمیت بیشتری پیدا می‌کند.

چهار رویکرد اصلی جهت کاهش هزینه آزمون بازگشت عبارتند از: ۱) به‌حداقل‌رسانی[3]، ۲) اولویت‌بندی[4]، ۳) بهینه‌سازی[5] و در پایان ۴) انتخاب[6] [۶].

با وجود تنوع روش‌های موجود که به طور جامع در مراجع مروری [۶]و[۱۰] مورد بررسی قرار گرفته است، در ادامه نمونه‌ای از تحقیقات پیشین در خصوص روش مبتنی بر پوشش کد اختلاف دو نسخه برنامه که در این مقاله مورد توجه قرار گرفته است و به عنوان یک روش انتخاب موارد آزمون شناخته می‌شود، ارائه می‌شود.

در این روش، اختلاف دو نسخه برنامه به شیوه‌های مختلف، کشف شده و آزمون‌های مرتبط با بخش مورد اختلاف دو نسخه برنامه مورد توجه قرار می‌گیرد. آزمون کد مربوط به اختلاف دو نسخه برنامه به جای آزمون تمام نسخه جدید از این جهت اهمیت دارد که قسمت‌های تغییر نیافته‌ای که به خوبی آزمون‌ها را پشت سر گذاشته‌اند، نیاز به آزمون مجدد ندارند. درنتیجه برنامه مورد آزمون کوچکتر می‌گردد و تعداد موارد آزمون کاهش می‌یابد.

تحقیقات قدیمی‌تر اختلاف بین دو نسخه برنامه را به صورت متنی بدست می‌آوردند. اما در تحقیقات جدید، اختلاف معنایی و رفتار برنامه مورد توجه قرار گرفته است.

در دیدگاه متنی، بدون توجه به اینکه این فایل کد یک برنامه قابل اجراست، بخش‌های مشترک دو نسخه برنامه مشخص می‌گردد. به عنوان نمونه diff [12] یکی از معروف‌ترین ابزارها در سیستم عامل یونیکس است که پس از تشخیص قسمت مشترک، اختلاف دو نسخه برنامه را به صورت گزارشی از دنباله‌های خطوطی که بین فایل‌ها حذف شده‌اند یا اضافه شده‌اند، ارائه می‌دهد. وکلوس[7] و همکاران یک ابزار تحت سیستم عامل یونیکس به نام Pythia برای تشخیص اختلاف متنی برنامه‌های C تولید کرده‌اند [۱۳].

[1] Test Oracle
[2] Happy Path
[3] Regression Test Minimization
[4] Regression Test Prioritization
[5] Regression Test Optimization
[6] Regression Test Selection
[7] Vokolos

ابزار Ldiff [14] یک ابزار تعیین اختلاف خطی مستقل از زبان است که مبتنی بر ابزار diff در سیستم‌عامل یونیکس ساخته شده است و از الگوریتم پیدا کردن طولانی‌ترین زیردنباله مشترک و الگوریتم پیدا کردن فاصله لونشتاین[1] استفاده می‌کند. این ابزار به متن مستقل از اینکه کد یک برنامه باشد، توجه دارد. ابزار Hdiff [15] نیز از تکنیک diff و تکنیک simhash استفاده می‌کند.

در دیدگاه نحوی، گرامر زبان و درخت پارس[2] مورد توجه قرار می‌گیرد. یانگ[3] [۱۶] هر برنامه را به صورت یک درخت پارس نمایش می‌دهد و با استفاده از الگوریتم تطبیق درخت اختلاف دو نسخه برنامه را تعیین می‌کند. مالتیک[4] [۱۷] از الگوریتم اَبَراختلاف[5] و زبان SrcML[6] مبتنی برXML برای نمایش دو برنامه و اختلاف آنها استفاده کرده است. آرشامبولت[7] [۱۸]، برای برنامه‌های بزرگ، گراف دو نسخه برنامه را که رئوس آنها به شیوه یکسانی نام‌گذاری شده‌اند به عنوان ورودی می‌گیرد و با هم ادغام می‌کند تا اختلاف‌ها را کشف کند. ابزار FTMPATool [19] با ساخت درخت‌های AST[8] تفاوت‌های ساختاری نسخه‌های برنامه را با استخراج توابع کوچک، به‌دست می‌آورد. ابزار ChangeScribe [20] یک پلاگین Eclipse است که تغییرات متنی در کد را در کنار درخت نحو انتزاعی می‌بینند. ابزار LSDiff [21] تغییرات و دلیل ایجاد تغییر را کشف می‌کند و تغییرات ساختاری سیستماتیک را به صورت قواعد منطقی به زبان پرولوگ بیان می‌کند. ابزار GumTree [22] با استفاده از اختلاف درخت‌های AST به دنبال نمایش تغییرات کلی در برنامه است. ابزار مرجع [۲۳] با استفاده از درخت AST برای افزایش خوانایی برنامه، قسمت‌های مختلف برنامه را با اضافه کردن خطوط خالی از هم جدا می‌کند.

در دیدگاه معنایی هورویتز[9] اجزای تغییریافته برنامه را مشخص می‌کند و این اجزا را از نظر نوع تغییر که معنایی[10] بوده است یا متنی[11]، دسته‌بندی می‌کند [۲۴]. محدویت دستورات زبان (شامل تعریف متغیرهای اسکالر، دستور شرطی، دستور مقداردهی، حلقه while و دستور خروجی) در آن وجود دارد به‌طوری‌که بینکلی[12] [۲۵] تعریف تابع و فراخوانی را به آن ([۲۴]) اضافه می‌کند. یولیان[13] [۲۶] گزارش تغییرات و آمار مبتنی بر انطباق درخت AST جزئی در برنامه‌های C را ارائه می‌دهد. ابزار Jdiff [27] نیز با توسعه مقاله هورویتز [۲۴] و استفاده از گراف جریان کنترل توسعه‌یافته اختلاف برنامه‌های شیءگرا را بدست می‌آورد. گرگ[14] [۲۸] با توسعه ابزار [۲۷] برنامه‌های جنبه‌گرا[15] و به طورخاص زبان AspectJ را مدنظر قرار داده است. وانگ[16] [۲۹] با استفاده از ساختار درخت به تشخیص کد معنایی مشابه می‌پردازد. نوگن[17] [۳۰] به منظور نگهداری نرم‌افزار موجودیت‌ها را شناسایی می‌کند و اختلاف کلاس‌ها و متدهای دو نسخه برنامه را با ابزار idiff نشان می‌دهد.

همان‌طور که ملاحظه می‌گردد کشف اختلاف دو نسخه برنامه همواره با اهداف مختلف از جمله آزمون بازگشت و نگهداری نسخه‌های مختلف برنامه و معمولا مبتنی بر زبان و روش تولید برنامه مورد توجه قرار گرفته است.

۳- الگوریتم پیشنهادی

این مقاله به طور خاص به پروژه‌هایی می‌پردازد که به روش آزمون‌رانه تولید شده‌اند. فرض بر آن است که در طول فرایند تولید، چرخه سه مرحله‌ای آزمون‌رانه و همچنین قوانین آن از جمله اینکه هیچ کدی به غیر از نیاز به رفع خطا نوشته نمی‌شود رعایت گردد.

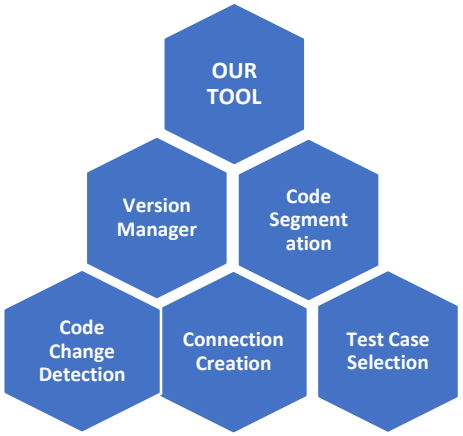
با توجه به ماهیت آزمون‌رانه بودن تولید نرم‌افزار، انتظار می‌رود پس از نوشتن آزمونی که با خطا مواجه می‌شود و گذرانده نمی‌شود، کدنویسی صورت گیرد. در جریان این کدنویسی یا کدهای جدیدی نوشته می‌شود یا بخشی از کدهای موجود تغییر می‌کند. ایده این مقاله این است که اگر بتوان تغییرات کد قبل و بعد از گذرانده شدن هر مورد آزمون و بازآرایی و نهایی شدن کد آن را تشخیص داد، می‌توان ارتباطی بین این بخش از کد و آزمونی که منجر به این تغییرات شده است در نظر گرفت. بنابراین ابزاری متشکل از پنج مولفه مطابق با شکل ۲ پیاده‌سازی شد که به ترتیب در طی پنج گام اساسی الگوریتم را پیش می‌برد.

۳-۱- گام اول: مدیریت نسخه

مدیریت پیکربندی نرم‌افزار یک رویکرد تکاملی جهت کنترل و

[1] Levenhstein
[2] Parse Tree
[3] Yang
[4] Maletic
[5] Meta-Differencing
[6] SouRce Code Markup Language
[7] Archambault
[8] Abstract Syntax Trees
[9] Horwitz
[10] Semantic change
[11] Textual change
[12] Binkley
[13] Iulian
[14] Görg
[15] Aspect Oriented Languages
[16] Wang
[17] Nguyen

مدیریت تغییرات پروژه است. با توجه به لزوم مقایسه نسخه‌ها در این ابزار وجود یک سیستم مدیریت حداقلی شامل شناسایی اشیا، پایگاه داده و کنترل نسخه لازم است.



شکل ۲. پنج مولفه اصلی الگوریتم پیشنهادی و ابزار ارائه شده

این سیستم در زمان ذخیره فایل، به‌طور اتوماتیک و دستی برنامه را با شماره نسخه جدید ذخیره می‌کند و اطلاعات هر نسخه را در پایگاه داده نگهداری می‌نماید. شماره نسخه از سه بخش اصلی تشکیل می‌شود (Major.Minor.Patch). در زمان ذخیره فایل، سیستم شماره پیشنهادی خود را نمایش دهد و امکان تغییر آن توسط کاربر وجود دارد.

۳-۲- گام دوم: بخش‌بندی کد

مهمترین گام در عملی‌سازی ایده مورد نظر بخش‌بندی کد است. روش کار به این صورت است که در فرآیند تولید برنامه در هر زمانی که یک نسخه از برنامه ذخیره می‌گردد، بلافاصله عمل بخش‌بندی کد انجام می‌گردد. این کار با توجه به درخت AST برنامه و دستورات زبان برنامه صورت می‌گیرد. جهت مشخص شدن محدوده هر بلاک و امکان ردیابی آن در صورت جابه‌جایی کد، ابتدا و انتهای هر بخش با دو توضیح[1] یا حاشیه‌نویسی[2] مشخص می‌گردد.

اندازه بخش‌های کد می‌تواند متناسب با اندازه برنامه توسط کاربر تعیین گردد. جدول ۱ حالت‌های مختلف بخش‌بندی کد بر حسب اندازه برنامه در سه سطح دانه‌بندی مختلف را نمایش می‌دهد. هر بخش از کد فارغ از سطح دانه‌بندی یک بلاک نامیده می‌شود.

انتخاب بالاترین سطح توسط کاربر، باعث می‌شود ابزار یک کلاس را به عنوان یک بلاک در نظر بگیرد، سطح دوم هر متد و سطح سوم هر دستورالعمل (مانند دستورشرطی if) را یک بلاک درنظر می‌گیرد.

جدول ۱. سطح‌های مختلف در بلاک‌بندی کد برنامه

سطح	بلاک	توضیحات
۱	کوچک	هر دستورالعمل زبان برنامه‌نویسی یک بلاک درنظر گرفته شود.
۲	متوسط	هر تابع یا متد در برنامه یک بلاک درنظر گرفته شود.
۳	بزرگ	هر کلاس در برنامه یک بلاک درنظر گرفته شود.

با توجه به اینکه در روش آزمون‌رانه آزمون‌ها بخش قابل توجهی از کد را تشکیل می‌دهند، بخش‌بندی برای کد برنامه و کد آزمون انجام می‌شود. بنابراین دو نوع بلاک وجود دارد: بلاک کد و بلاک آزمون.

تشخیص بلاک‌های آزمون از بلاک‌های کد در اغلب زبان‌های برنامه‌نویسی با توجه به حاشیه‌نویسی مربوط به آن زبان در نظر گرفته می‌شود. به عنوان مثال در زبان جاوا بلاک‌های کد با @Test مشخص می‌گردند. در مورد بلاک‌های آزمون سطح دانه‌بندی وجود ندارد و اندازه آن ثابت است و هر مورد آزمون به طور کامل یک بلاک آزمون درنظر گرفته می‌شود.

بلاک‌بندی و نام‌گذاری بلاک‌ها از این جهت اهمیت دارد که در طول فرایند تولید نرم‌افزار امکان ردیابی کد با وجود تغییرات داخلی مختلف و حتی جا به جایی در برنامه وجود داشته باشد.

با توجه به اینکه نیاز است تمام تغییرات در نسخه‌های مختلف برنامه ذخیره گردد، هر بار در زمان ذخیره‌سازی هر یک از فایل‌های پروژه نیاز است عمل بلاک‌بندی تکرار گردد تا بخش‌های جدید، بلاک‌بندی و نام‌گذاری گردند. بلاک‌بندی و نام‌گذاری بلاک‌ها به طور خودکار در زمان ذخیره فایل انجام می‌گردد.

در صورتی که بلاک‌بندی در سطح ۱ مدنظر باشد و داخل یک دستورالعمل، یک یا بیشتر دستورالعمل دیگر باشد، این بلاک به عنوان بلاک پدر برای بلاک‌های داخلی تعریف می‌شود. به عنوان مثال در شکل ۳ داخل یک حلقه تکرار for دو دستور شرطی if قرار دارد. این دو دستور if هر کدام یک بلاک فرزند درنظر گرفته خواهند شد. بلاک for به عنوان پدر دو بلاک داخلی if شناخته می‌شود.

```
for (i=0; i<n;i++) {
    if (a<b)
        a++;

    if (b<a)
        b++;
}
```

شکل ۳. قرارگیری دو بلاک کد درون بلاک کد دیگر

[1] Comment

[2] Annotation

نام بلاک پدر در کنار سایر اطلاعات مربوط به بلاک (شامل نام بلاک، نوع بلاک، آدرس فایل محتوی بلاک، محتوای بلاک به فرمت AST) در یک جدول نگهداری می‌گردد. اهمیت این اطلاعات از این نظر است که در صورت تغییر در بلاک فرزند، در واقع کلاس پدر هم تغییر یافته است. بنابراین تغییر در کلاس پدر نیز بایستی در نظر گرفته شود. در ادامه با استفاده از یک روش نام‌گذاری، نام منحصر به فردی به هر بلاک تخصیص می‌یابد. نام‌گذاری بلاک‌های کد و آزمون به ترتیب از عبارات منظم N_C و N_T مطابق قوانین ۱ تا ۴ پیروی می‌کند.

$$N_C = \text{‘C’ lddd} \quad (۱)$$

$$N_T = \text{‘T’ lddd} \quad (۲)$$

$$l ::= a|b|c|\ldots|z|A|B|C|\ldots|Z \quad (۳)$$

$$d ::= 0|1|2|\ldots|9 \quad (۴)$$

۳-۳- گام سوم: تشخیص بلاک‌های تغییریافته

به طور کلی تغییر در محصول نرم‌افزاری در سه سطح تعیین می‌گردد. تغییرات متنی، نحوی و معنایی. هر چند تغییرات معنایی و رفتاری در سطح بالاتری قرار دارند و مبین تغییرات واقعی هستند، اما در مقاله حاضر، تغییرات متنی و نحوی به صورت ترکیبی مورد توجه قرار گرفته است. علت این امر آن است که هدف این مقاله یافتن تمامی آزمون‌هایی است که با تغییر کد، نیاز به آزمون مجدد دارند. در صورت حذف آزمون‌هایی که تغییرات ظاهری مانند تغییر در نام متغیر یا نام تابع را بررسی می‌کنند، مجموعه موارد آزمون، امن[1] نخواهد بود. یعنی ممکن است برخی موارد آزمون اشتباهاً حذف شود و قدرت تشخیص خطا در برنامه را نداشته باشد.

لازم به ذکر است که هدف اغلب مقالاتی که اختلاف‌های معنایی و سطح بالا را در کد در نظر گرفته اند، نگهداری سیستم و کشف دلیل تغییر، چرایی و چگونگی تغییر بوده است. اما در مورد آزمون بازگشت، حتی اگر تغییر در جهت بازآرایی کد باشد، نیاز به آزمون مجدد دارد تا بتوان از درستی آن اطمینان یافت. بنابراین شناسایی تغییرات متنی و نحوی کد مورد توجه قرار می‌گیرد تا آزمون بازگشت از اطمینان کافی برخوردار باشد.

در هر مرحله که نسخه جدیدی از برنامه ایجاد می‌شود و برنامه ذخیره می‌گردد، ضمن انجام عمل بلاک‌بندی، تشخیص بلاک‌های جدید و تغییریافته هم صورت می‌گیرد. بلاک‌های جدید بلاک‌هایی هستند که قبلا نامگذای نشده‌اند و اخیراً ایجاد شده‌اند. بلاک‌های موجود قبلی به دو دسته بلاک‌های تغییریافته و تغییرنیافته تقسیم می‌شوند.

جهت تشخیص اختلاف در دو نسخه هر بلاک، ابتدا محتوای آن دو نسخه از بلاک با فرمت AST درنظر گرفته می‌شود و سپس به یک آرایه Json[2] تبدیل می‌گردد و در جدول commit ذخیره شود و سپس به صورت متنی با نسخه قبلی مقایسه می‌شود. اگر یکسان نبودند، به عنوان بلاک تغییریافته تشخیص داده می‌شود.

به عنوان مثال در صورت تغییر فقط نام یک تابع، بلاک قابل تشخیص است زیرا نام منحصربه‌فردی که به طور اتوماتیک به بلاک تخصیص یافته است، تغییر نمی‌کند. این بلاک از نظر ساختار نحوی تغییر نکرده است اما از آنجا که روش بکار گرفته شده یک روش ترکیبی است، در مرحله تطابق بعدی بلاک از نظر متنی با بلاک قبلی مقایسه می‌شود که مطابقت ندارد در نتیجه به عنوان بلاک تغییریافته شناسایی می‌گردد.

۳-۴- گام چهارم: ارتباط آزمون و بلاک‌های کد

تمامی تغییرات در نسخه جدید برنامه در راستای گذرانده شدن آزمونی نوشته شده است که در مرحله قبلی رد شده است، بنابراین تمامی بلاک‌های تغییر یافته و بلاک‌های جدید که در هر مرحله ایجاد می‌شوند، به آخرین مورد آزمون که در مرحله قبلی گذرانده نشده بود، مرتبط می‌شود. همچنین تمامی تغییراتی که به دلیل بازآرایی کد روی بلاک‌ها ایجاد می‌گردد و موجب تولید بلاک تغییریافته می‌شوند، به مورد آزمون اخیراً اضافه شده مرتبط خواهند بود. زیرا این تغییرات در جهت گذراندن آن آزمون و بهبود کد مربوطه ایجاد شده است.

بنابراین در این گام بلاک‌های جدید و بلاک‌های تغییریافته شناسایی شده به بلاک آزمون اخیر مرتبط می‌گردند و این ارتباط در یک جدول ذخیره می‌گردد.

فرض می‌شود پروژه P، شامل مجموعه‌ای از بلاک‌های کد C و مجموعه‌ای از موارد آزمون T است (P:CUT). برای گذراندن آزمون t جدید، برخی از بلاک‌های کد اصلاح می‌شوند. اگر مجموعه بلاک‌های تغییر یافته در پروژه P را با M نمایش دهیم، $M \subset C$ است. این مجموعه تغییر می‌یابد و مجموعه بلاک‌های تغییریافته را با M’ نمایش می‌دهیم. ممکن است بلاک‌های جدیدی ایجاد گردد که آنها را با مجموعه N نمایش می‌دهیم. نسخه جدید پروژه را با P’ نمایش می‌دهیم که شامل مجموعه‌ای از بلاک‌های کد C’ و مجموعه‌ای از

[1] Safe

[2] JavaScript Object Notation

موارد آزمون ’T است (’P’:C’UT’). مشخصات نسخه جدید پروژه و نحوه برقراری ارتباط بین بلاک‌های کد و آزمون مطابق قوانین ۵ تا ۸ برقرار خواهد شد.

$$C' = (C - M) \cup (M' \cup N) \quad (۵)$$

$$T' = T \cup \{t\} \quad (۶)$$

رابطه لینک به صورت ذیل تعریف می‌گردد:

$$\textbf{Link}: C' \times T' \quad (۷)$$

$$\forall c \in (M' \cup N), \textbf{Link}\ (c, t) \quad (۸)$$

رابطه لینک مجموعه‌ای از زوج مرتب‌هاست که هر عضو آن ارتباط بین یک بلاک کد و یک آزمون را مشخص می‌کند. همان طور که در قوانین فوق مشخص است، به ازای هر بلاک کد جدید یا تغییر یافته، رابطه لینک با آزمون اخیراً اضافه شده برقرار می‌گردد.

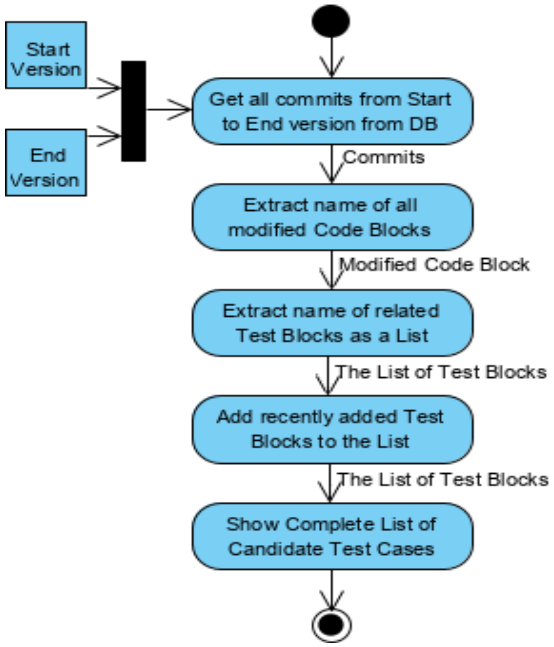
این مجموعه به طور مدام با ایجاد نسخه‌های جدید تکمیل می‌گردد و هرگونه تغییرات در مرحله‌ی بازآرایی و یا تکمیل پروژه موجب تکمیل این مجموعه می‌گردد. اعضای این رابطه در تشخیص آزمون‌های مرتبط در مرحله بعدی مورد استفاده قرار می‌گیرد. هر بلاک کد ممکن است با بلاک‌های آزمون متعددی در ارتباط باشد و این ارتباط از مراحل قبلی تشخیص داده شده باشد.

جدول بلاک‌ها تمامی اطلاعات مربوط به هر بلاک را در خود نگهداری می‌کند. اطلاعات مربوط به رابطه لینک نیز در این جدول ذخیره می‌گردد. یکی از فیلدهای این جدول نام بلاک‌های آزمون مرتبط با بلاک کد است. در این جدول لیستی از بلاک‌های آزمون مربوطه برای هر یک از بلاک‌های کد ذخیره می‌گردد. این لیست نمایانگر آن است که هر بار یک بلاک کد تغییر پیدا کرد، انجام کدام آزمون‌ها ضروری می‌گردد.

۳-۵- گام پنجم: انتخاب موارد آزمون

در هر زمان که نیاز به ایجاد آزمون بازگشت شد، دو نسخه مورد نظر از برنامه به عنوان ورودی درنظر گرفته می‌شود. سپس اختلاف بین دو نسخه مورد نظر مطابق با گام سوم به دست می‌آید.

بلاک‌های جدید (N) و بلاک‌های تغییریافته (M’ یا M) مشخص می‌شوند. بنابراین قسمت‌هایی از کد که دستخوش تغییر شده‌اند (M’UN) بررسی می‌گردد تمام بلاک‌های آزمون مرتبط با تمام بلاک‌های موجود در دو مجموعه M’ و N به عنوان آزمون‌های پیشنهادی برای اجرا معرفی می‌گردند. شکل ۴ نمودار توالی فرایند انتخاب موارد آزمون بازگشت را نمایش می‌دهد.



شکل ۴. نمودار توالی فرایند انتخاب موارد آزمون بازگشت

۴- پیاده‌سازی و تحلیل

جهت بررسی عملی این روش ابزاری تولید و پیاده‌سازی شده است تا گام‌های پنج‌گانه فوق را تسهیل کند.

هر چند ایده مطرح شده محدودیت زبان برنامه‌نویسی ندارد اما در حال حاضر این ابزار به صورت یک پلاگین در محیط Eclipse پیاده‌سازی شده است و فقط برای برنامه‌های جاوا قابل استفاده می‌باشد.

جهت ارزیابی ایده و ابزار تولیدی، سه برنامه ساده با نام‌های توان، مرتب‌سازی و لیست پیوندی به شرح زیر در نظر گرفته شد.

- برنامه توان یک عدد صحیح را به توان یک عدد صحیح نامنفی می‌رساند.
- برنامه مرتب‌سازی یک لیست با تعداد نامشخص می‌گیرد و مرتب می‌سازد.
- برنامه لیست پیوندی اطلاعات مربوط به گره‌ها را می‌گیرد و در یک لیست مرتب قرار می‌دهد و امکان انجام اعمال حذف و اضافه و ویرایش را دارد.

ابتدا این سه برنامه مبتنی بر روش تولید آزمون‌رانه و مطابق با روش ارائه شده در وب‌سایت ولف‌گنگ[۳۱] برای نوشتن برنامه‌های آزمون‌رانه و بدون استفاده از ابزار توسط یک دستیار پژوهشی به زبان جاوا نوشته شد. سپس این سه برنامه توسط دستیار پژوهشی دیگری با الگوگیری از همان سایت و با الگوریتم مشترک و در حضور ابزار پیاده‌سازی شده نوشته شد. تعداد آزمون‌ها در هر دو روش

مساوی درنظر گرفته شد. کد مورد استفاده در برنامه دوم مرحله به مرحله مطابق روش اول با حضور ابزار پیاده‌سازی‌شد. به طوری که کد و آزمون‌ها برای هر برنامه به طور یکسان و هماهنگ در نظر گرفته شد. لازم به ذکر است که با توجه به کوچک بودن اندازه برنامه‌ها، سطح بلاک‌بندی کوچک (سطح ۱) در هر سه برنامه انتخاب شده است.

جدول ۲ نتایج حاصل از این دو روش پیاده‌سازی با ابزار و بدون استفاده از ابزار را با یکدیگر مقایسه می‌کند. ستون TDD مربوط به روش آزمون‌رانه معمولی و بدون استفاده از ابزار و ستون Plugin مربوط به پلاگین و با استفاده از ابزار است.

چهار سطر ابتدایی جدول ۲ مشخصات برنامه‌ها را نمایش می‌دهد و دو سطر آخر سطرهای اصلی مقایسه پلاگین می‌باشند. همان‌طور که در سطر اول جدول ۲ ملاحظه می‌گردد، تعداد خطوط با حضور ابزار پلاگین به واسطه درج مشخصه ابتدا و انتهای بلاک کد به اندازه دو برابر تعداد بلاک‌ها افزایش دارد و تفاوت دیگری ندارند. در روش آزمون‌رانه معمولی بلاک‌بندی انجام نمی‌شود اما در زمان استفاده از پلاگین بلاک‌بندی انجام می‌گردد. سطر ۲ تعداد بلاک هر برنامه را در زمان استفاده از ابزار نمایش می‌دهد. تعداد بلاک‌های کد به ترتیب ۳، ۷ و ۲۴ می‌باشد. تعداد بلاک‌های آزمون نیز به ترتیب، ۵، ۱۰ و ۱۹ آزمون است که در سطر ۳ ارائه شده است.

همان‌طور که در سطر ۴ مشخص شده است، پیاده‌سازی کامل دو برنامه اول در پنج نسخه و برنامه سوم در ده نسخه به پایان رسیده است. تعداد نسخه‌ها در واقع تعداد دفعاتی است که برنامه با نام جدید و به عنوان نسخه جدید ذخیره شده است و آزمون بازگشت اجرا شده است.

جدول ۲. مقایسه نتایج پیاده‌سازی سه برنامه با و بدون ابزار

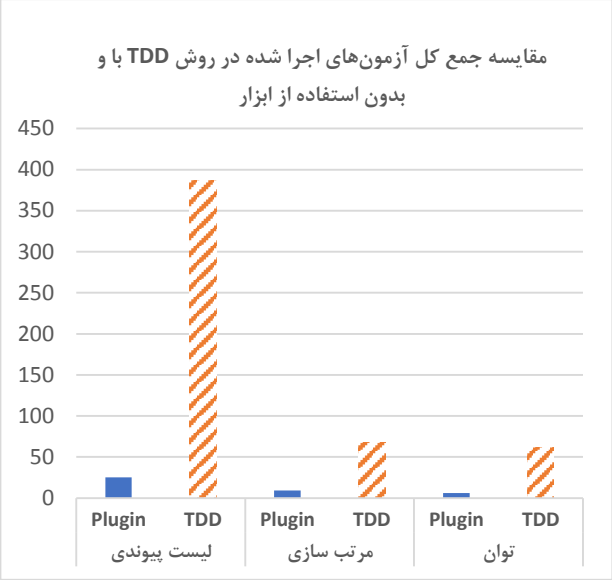
نام برنامه	توان		مرتب‌سازی		لیست‌پیوندی	
پارامترها	TDD	Plugin	TDD	Plugin	TDD	Plugin
تعداد خطوط کد	۱۹	۲۵	۲۵	۳۹	۱۳۳	۱۶۵
تعداد بلاک کد	–	۳	–	۷	–	۲۴
تعداد آزمون	۵	۵	۱۰	۱۰	۱۹	۱۹
تعداد نسخه ها	۵	۵	۵	۵	۱۰	۱۰
جمع آزمون‌های اجراشده	۶۱	۶	۶۸	۹	۳۸۷	۲۵
میانگین تعداد آزمون‌ها	۱۲,۲	۱,۲۵	۱۳,۶	۱,۸	۳۸,۷	۲,۵

جمع آزمون‌های اجرا شده در تمام نسخه‌ها در دو روش آزمون‌رانه معمولی و در کنار استفاده از ابزار در سطر ۵ ارائه شده است. در روش معمولی در هر بار اجرای آزمون بازگشت، تمام آزمون‌های واحد موجود اجرا می‌شوند اما در زمان استفاده از پلاگین تنها آزمون‌هایی اجرا می‌شوند که توسط ابزار انتخاب شده‌باشند. همان‌طور که ملاحظه می‌گردد، جمع کل آزمون‌های اجرا شده در تمام مراحل تولید برنامه در روش آزمون‌رانه و بدون استفاده از ابزار پلاگین با اختلاف قابل‌توجهی بسیار بیشتر از حالتی است که از ابزار استفاده شده است.

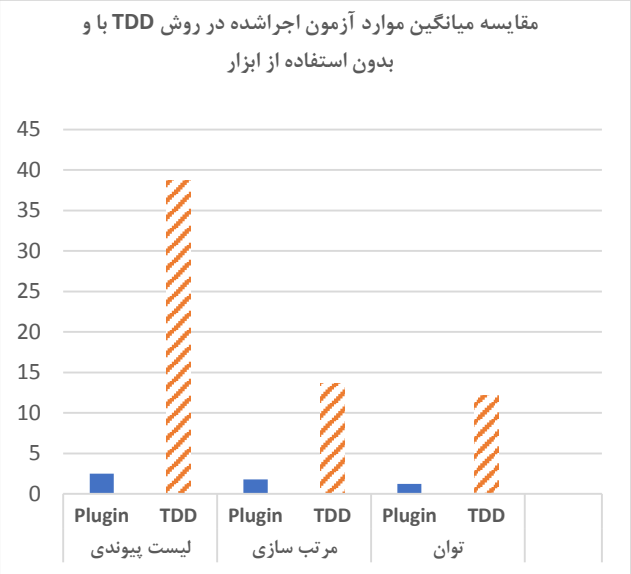
شکل ۵ نمودار مقایسه تعداد آزمون‌های اجراشده برای سه برنامه را نمایش می‌دهد. همان طور که در شکل ۵ ملاحظه می‌گردد، استفاده از ابزار با انتخاب موارد آزمون موثر، تعداد موارد آزمون اجراشده را به شدت کاهش می‌دهد.

جهت مقایسه بهتر این دو حالت، حاصل تقسیم تعداد دفعات اجرا بر تعداد نسخه‌های برنامه به عنوان میانگین در سطر آخر جدول ۲ ملاحظه می‌گردد. این سطر مشخص می‌کند که به طور متوسط در هر نسخه چند آزمون اجرا شده است. شکل ۶ نمودار مقایسه میانگین تعداد آزمون‌های اجراشده در هر نسخه را برای سه برنامه نمایش می‌دهد.

همان‌طور که در دو شکل ۵ و ۶ ملاحظه می‌گردد، در دو برنامه اول تعداد نسخه‌ها برابر با ۵ نسخه است اما تعداد آزمون‌های برنامه اول ۵ و برنامه دوم ۱۰ آزمون می‌باشد. تعداد دفعات اجرای موارد آزمون در روش آزمون‌رانه معمولی از توان دوم تعداد موارد آزمون بیشتر است اما تعداد دفعات اجرای موارد آزمون در روش پیشنهادی و با استفاده از ابزار تقریبا برابر با تعداد موارد آزمون می‌باشد. با افزایش تعداد نسخه‌های برنامه و تعداد موارد آزمون اهمیت استفاده از ابزار بیشتر نمایان می‌گردد. یعنی با افزایش تعداد نسخه‌های برنامه، در برنامه لیست پیوندی نسبت به برنامه‌های دیگر و افزایش تعداد دفعات آزمون، میزان کاهش در اجرای موارد آزمون بیشتر مشخص می‌گردد و تعداد موارد آزمون اجرا شده کاهش چشمگیرتری دارد. اختلاف اعداد به دست آمده در برنامه سوم در این دو روش نسبت به اختلاف اعداد دو برنامه قبلی بیشتر است. این نتیجه قابل پیش‌بینی بود زیرا تعداد دفعات اجرای موارد آزمون در روش آزمون‌رانه با توان دوم تعداد موارد آزمون رابطه دارد و همچنین با تعداد دفعات اجرای آزمون بازگشت و در نتیجه تعداد نسخه‌ها ارتباط مستقیم دارد.



شکل ۵. مقایسه تعداد آزمون‌های اجراشده برای سه برنامه



شکل ۶. مقایسه میانگین موارد آزمون اجراشده برای سه برنامه

موضوع دیگری که نیاز به ارزیابی دارد این است که آیا با وجود کاهش تعداد موارد آزمون، زمان آزمون بازگشت کاهش می‌یابد یا به دلیل سربارهای اضافی ناشی از انجام گام‌های مختلف این روش، زمان آزمون افزایش می‌یابد.

جهت بررسی این موضوع تغییراتی در برنامه اعمال شد تا زمان مربوط به عمل بلاک‌بندی و ایجاد ارتباط بین بلاک‌های کد و آزمون را محاسبه نماید و در پایگاه‌داده ذخیره نماید و سربار زمانی ناشی از انجام گام‌های مربوط به ابزار را محاسبه نماید. سپس برنامه‌ای دیگر به زبان پایتون نوشته شد تا برای هر دو حالت اجرا با ابزار و بدون ابزار در شرایط یکسان تمامی آزمون‌های انتخاب شده را اجرا کند و در زمان استفاده از ابزار، تمامی زمان‌های سربار ابزار ناشی از بلاک‌بندی و ایجاد ارتباط را به زمان آزمون بازگشت اضافه نماید و این دو زمان را با یکدیگر مقایسه نماید. در جدول ۳ زمان اجرای آزمون بازگشت در روش آزمون‌رانه با روش استفاده از ابزار مقایسه شده است. در محاسبه زمان آزمون بازگشت با استفاده از ابزار، سربار زمانی مربوط به استفاده از ابزار نیز در نظر گرفته شده است و با زمان اجرای موارد آزمون جمع شده است. نتایج جدول ۳ نشان می‌دهد که زمان اجرای آزمون بازگشت با استفاده از ابزار با در نظر گرفتن زمان سربار اضافی کمتر از زمان اجرا به روش آزمون‌رانه است.

جدول ۳. مقایسه زمان آزمون بازگشت با و بدون ابزار پلاگین

نام برنامه	توان		مرتب‌سازی		لیست‌پیوندی	
روش	TDD	Plugin	TDD	Plugin	TDD	Plugin
زمان آزمون برحسب ms	۸۹	۱۴	۹۸	۲۲	۴۰۲	۶۷

برنامه‌های ارائه شده در این ارزیابی کوچک هستند و برای ارزیابی این روش به عنوان نمونه ارائه شدند. اما وجود چنین ابزاری جهت کاهش زمان آزمون بازگشت در روش آزمون‌رانه در پروژه‌های بزرگ که زمان آزمون افزایش می‌یابد، اهمیت بیشتری دارد.

پیش‌بینی می‌شود که اختلاف تعداد کل موارد آزمون ایجاد شده و همچنین زمان آزمون بازگشت در پروژه‌های بزرگ بیشتر باشد. لذا در ادامه این تحقیق، تحلیل و بررسی روی پروژه‌های بزرگ صورت خواهد گرفت. از آن‌جا که نوشتن پروژه‌های بزرگ وقت‌گیر است، پروژه‌های آزمون‌رانه موجود در مخزن گیت‌هاب مورد توجه قرار گرفت. لذا با استفاده از زبان سطح بالای بوآ[1] [۳۲] برنامه‌ای برای جستجوی پروژه‌هایی با ویژگی آزمون‌رانه در گیت‌هاب نوشته شد. با توجه به اینکه معیار دقیقی برای تعیین آزمون‌رانه بودن پروژه‌ها وجود ندارد، یکی از معیارهای انتخاب پروژه مشابه کار بورلی[2] [۳۳] وجود فایل آزمون با اختلاف زمانی کمتر از یک هفته با فایل کد در نظر گرفته شد و زبان پروژه به زبان جاوا محدود شد. انتخاب و پیاده‌سازی مجدد این پروژه‌ها با استفاده از ابزار نیازمند پژوهش دیگری است که به‌عنوان کار آتی انجام خواهد شد.

[1] Boa

[2] Borle

۵- جمع‌بندی

در این مقاله، ابتدا شیوه آزمون‌رانه معرفی شد خلاصه‌ای از مزایا (افزایش کیفیت نرم‌افزارهای تولیدی، افزایش قابلیت نگهداری، تولید کدهای ساده، کوتاه، خوانا، منظم و زیبا (به دلیل بازآرایی)، طراحی و معماری بهتر، افزایش اعتماد، افزایش انعطاف‌پذیری، کاهش خطا و کاستی، پوشش کد، امکان آزمون بازگشت) و معایب این روش از جمله تعداد دفعات اجرای موارد آزمون بیشتر و در نتیجه افزایش زمان آزمون بازگشت مطرح گردید.

در این راستا مشکل تعداد دفعات اجرای موارد آزمون مورد بررسی قرار گرفت و توضیح داده شد که مرتبه اجرایی موارد آزمون با توان دوم تعداد موارد آزمون نسبت دارد. از این‌رو ایده‌ای جهت کاهش تعداد دفعات اجرای موارد آزمون متناسب با ماهیت روش آزمون‌رانه مطرح گردید. این ایده مبتنی بر روش پوشش کد تغییر یافته به انتخاب موارد آزمون می‌پردازد. نحوه کار بدین صورت است که بخش تغییر یافته کد با استفاده از رابطه لینک[1] به آزمون مربوطه مرتبط می‌گردد. از رابطه لینک جهت انتخاب آزمون‌های مرتبط با بخش‌های تغییریافته کد استفاده می‌گردد. برای انجام این کار پنج گام اساسی انجام می‌شود. این گام‌ها برای اجرایی شدن ایده به طور کامل توضیح داده شد.

همان‌طور که به عنوان مثال ابزار معرفی شده در مرجع [۲۷] به طور خاص برای شیوه جنبه‌گرا و مرجع [۲۸] به طور خاص برای شیوه شیءگرا ارائه شده‌اند، پلاگین ارائه شده در این مقاله نیز به طور خاص به برنامه‌هایی اختصاص دارد که به شیوه آزمون‌رانه تولید می‌شوند.

باتوجه به اینکه امکان بکارگیری ابزارهای قبلی در روش آزمون‌رانه وجود ندارد، نمی‌توان این ابزارها را از نظر تعداد دفعات اجرای موارد آزمون و زمان اجرای آزمون بازگشت با یکدیگر مقایسه کرد. از این رو این مقایسه بین دو حالت استفاده از ابزار و بدون استفاده از ابزار انجام شد.

جهت محاسبه زمان آزمون بازگشت واقعی در حالتی که از ابزار استفاده می‌شود، سربار زمانی مربوط به انجام گام‌های پنج‌گانه ایده پیشنهادی محاسبه شد و با زمان آزمون بازگشت جمع شد.

نتایج پیاده‌سازی برنامه‌ها نشان می‌دهد استفاده از پلاگین با کاهش تعداد موارد آزمون انتخابی در زمان اجرا، موجب کاهش تعداد دفعات اجرای موارد آزمون و همچنین زمان آزمون بازگشت می‌گردد.

[1] Link

مراجع

[1] K. Beck, Test Driven Development: By Example, Addison-Wesley, 2002

[2] M. Fowler, K. Beck, J. Brant, W. Opdyke and D. Roberts, Refactoring: Improving the Design of Existing Code, 1st ed., E. Gamma, Ed., Pearson Education India, 1999.

[3] L. Madeyski, "Emperical Studies on the impact of test first programming", Technical Report I32/09/, Wroclaw University of Technology, Instituteof Informatics, 2009.

[4] W. Bissi, A. G. Serra-Seca-Neto and M. C. F. Pereira Emer, "The effects of test driven development on internal quality, external quality and productivity: A systematic review", Information and Software Technology, vol. 74, pp. 45-54, 2016.

[5] S. Mäkinen and J. Münch, "Effects of Test-Driven Development: A Comparative Analysis of Empirical Studies", Proceedings of the 6th International Conference Software Quality, vol.6, pp. 155-169, 2014.

[6] R. H. Rosero, O. S. Gómez and G. Rodríguez, "15 years of software regression testing techniques—a survey" ,International Journal of Software Engineering and Knowledge Engineering, vol. 26, no. 5, pp. 675-689, 2016).

[7] P. Ammann, & J. Offutt (2016). Introduction to software testing. Cambridge University Press.

[8] A. Nanthaamornphong and J. C. Carver, "Test-Driven Development in scientific software: a survey", Software Quality Journal, vol. 25, no. 2, pp. 343-372, 2017.

[۹] ز. مافی و س.ج. میریان حسین آبادی، "یک روش تولید آزمون‌رانه بهبود یافته"، کنفرانس ملی فناوری‌های نوین در مهندسی برق و کامپیوتر، اصفهان،۱۳۹۶.

[10] S. Yoo and M. Harman, "Regression testing minimization, selection and prioritization: a survey", Software Testing, Verification and Reliability, vol. 22, no. 2, pp. 67-120, 2012.

[11] D. Parsons, T. Susnjak and M. Lange, "Influences on regression testing strategies in agile software development environments," Software Quality Journal, vol. 22, no. 4, p. 717–739, 2014.

[12] E. W. Myers, "An O(ND) Difference Algorithm and Its Variations," Algorithmica, vol. 1, no. 1-4, pp. 251-266, 1986.

[13] F. I. Vokolos and P. Frankl, "Empirical evaluation of the textual differencing regression testing technique" , IEEE International Conference on Software Maintenance (Cat. No. 98CB36272), 1998.

[14] G. Canfora, L. Cerulo and M. D. Penta, "LDiff: an Enhanced Line Differencing Tool" , 31st International Conference on Software Engineering. IEEE Computer Society, 2009.

[15] M. Asaduzzaman, C. Roy, K. Schneider and M. Di Penta, "LHDiff: A language-independent hybrid approach for tracking source code lines" , IEEE International Conference on Software Maintenance, 2013.

[16] W. Yang, "Identifying syntactic differences between two programs" , Software: Practice and Experience, vol. 21, no. 7, pp. 739-755, 1991.

[17] J. I. Maletic and M. L. Collard, "Supporting Source Code Difference Analysis" , 20th IEEE International Conference on Software Maintenance, 2004.

[18] D. Archambault, "Structural differences between two graphs through hierarchies" , Proceedings of Graphics Interface , 2009.

[19] A. Goto, N. Yoshida, M. Ioka, E. Choi and K. Inoue, "How to extract differences from similar programs? A cohesion metric approach" , 7th International Workshop on Software Clones (IEEE Press), 2013.

[20] M. Linares-Vásquez, L. Cortés-Coy, J. Aponte and D. Poshyvanyk, "Changescribe: A tool for automatically generating commit messages" , 37th IEEE International Conference on Software Engineering, 2015.

[21] M. Kim and N. David, "Discovering and representing systematic code changes" , IEEE 31st International Conference on Software Engineering, 2009.

[22] J.-R. Falleri, M. Floréal, B. Xavier, M. Matias and M. Martin, "Fine-grained and Accurate Source Code Differencing" , 29th ACM/IEEE international conference on Automated software engineering, 2014.

[23] X. Wang, L. Pollock and K. Vijay-Shanker, "Automatic segmentation of method code into meaningful blocks to improve readability" , 18th Working Conference on Reverse Engineering IEEE, 2011.

[24] S. Horwitz, "Identifying The Semantic and Textual Differences Between Two Versions of a Program " , ACM, vol. 25, no. 6, pp. 234-245, 1990.

[25] D. Binkley, "Using semantic differencing to reduce the cost of regression testing" , IEEE Conference on Software Maintenance , 1992.

[26] N. Iulian, F. S. Jeffrey and M. Hicks, "Understanding Source Code Evolution Using Abstract Syntax Tree Matching" , ACM SIGSOFT Software Engineering Notes, vol. 30, no. 4, pp. 1-5, 2005.

[27] T. Apiwattanapong, A. Orso and M. J. Harrold, "JDiff: A differencing technique and tool for object-oriented programs" , Automated Software Engineering, vol 14, pp. 3-36, 2007.

[28] M. Görg and J. Zhao, "Identifying semantic differences in AspectJ programs" , 18th international symposium on Software testing and analysis (ACM), 2009.

[29] T. Wang, K. Wang, X. SU and P. MA, "Detection of semantically similar code" , Frontiers of Computer Science, vol. 8, no. 6, pp. 996-1011, 2014.

[30] H. A. Nguyen, T. T. Nguyen, H. V. Nguyen and T. N. Nguyen, "idiff: Interaction-based program differencing tool" , 26th IEEE/ACM International Conference on Automated Software Engineering, 2011.

[31] O. Wolfgang, [Online]. Available: TDD Kata, https://www.programmingwithwolfgang.com/tdd-kata/ [Accessed April 28, 2024]

[32] R. Dyer, N. Hoan Anh, R. Hridesh and N. N. Tien, "Boa: A language and infrastructure for analyzing ultra-large-scale software repositories" , IEEE, 35th International Conference on Software Engineering (ICSE), 2013.

[33] N. C. Borle, M. Feghhi, E. Stroulia, R. Greiner and A. Hindle, "Analyzing the effects of test driven development in GitHub" , Empirical Software Engineering, vol. 23, no. 4, pp. 1931-1958, 2018.