Automated Software Engineering (2024) 31:9
https://doi.org/10.1007/s10515-023-00405-w

# Regression test selection in test-driven development

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

Received: 18 March 2023 / Accepted: 12 November 2023 / Published online: 27 December 2023


## Abstract

The large number of unit tests produced in the test-driven development (TDD) method and the iterative execution of these tests extend the regression test execution time in TDD. This study aims to reduce test execution time in TDD. We propose a TDD-based approach that creates traceable code elements and connects them to relevant test cases to support regression test selection during the TDD process. Our proposed hybrid technique combines text and syntax program differences to select related test cases using the nature of TDD. We use a change detection algorithm to detect program changes. Our experience is reported with a tool called RichTest, which implements this technique. In order to evaluate our work, seven TDD projects have been developed. The implementation results indicate that the RichTest plugin significantly decreases the number of test executions and also the time of regression testing despite considering the overhead time. The test suite effectively enables fault detection because the selected test cases are related to the modified partitions. Moreover, the test cases cover the entire modified partitions; accordingly, the selection algorithm is safe. The concept is particularly designed for the TDD method. Although this idea is applicable in any programming language, it is already implemented as a plugin in Java Eclipse.

**Keywords** Software testing · Test-driven development (TDD) · Regression test · Program differencing · Segmentation · Change detection

✉ Seyed-Hassan Mirian-Hosseinabadi
hmirian@sharif.edu

Zohreh Mafi
zohreh.mafi1@sharif.edu

1 Engineering Faculty, Sharif University of Technology, Intl. Campus, Kish Island, Iran

2 Faculty Member of Computer Engineering Department, Sharif University of Technology, Tehran, Iran

## 1 Introduction

Test Driven Development (Beck 2002) is one of the agile defect-reduction practices in which *"unit test cases are incrementally written prior to code implementation. All of the test cases that exist for the entire program must successfully pass before new code is considered fully implemented"* (George and Williams 2004). New tests are written to add/revise the desired features in such a way that the current version of the program fails. Refactoring (Fowler et al. 1999) is one of the key aspects of TDD which improves the software design, code structure quality, and code performance as well as enhances coding standards and principles (Dalton 2019).

Although TDD avoids writing extra code and delivers clean code, however, it increases the number of test cases rapidly. The TDD method has drawn the interest of software developers because of its advantages, including short and simple readable code, high-quality code, reliability, maintainability, and the capability of regression testing (as a result of creating a set of unit tests). Apart from its advantages, TDD also has certain deficiencies (Karac and Turhan 2018) such as higher development time (Khanam and Mohammed 2017). This study aims to resolve one of the drawbacks that has been less considered previously—the large number of test cases and the necessity of repeated executions.

The number of test cases generated in TDD is greater than that of other methods (Erdogmus et al. 2005). As a result, the time required for the regression test increases significantly. On the other hand, it is necessary to re-execute all of the test cases after each modification to ensure that the code remains accurate thereafter. A substantial amount of time is subsequently required in order to execute the test cases in the TDD method.

There are many cost reduction algorithms reducing the number of test cases, which we will discuss in Sects. 2.2 and 2.3. Different techniques may have different performances in different environments. The suitable technique is therefore selected based on methodology, topic, and program conditions. However, none of these methods are specifically designed for TDD. Therefore, this research proposed a test selection algorithm for TDD implemented programs to reduce the regression test execution time in TDD. Our experience is reported with a tool called RichTest, which implements this technique. It is a Java plugin and is available as a GitHub project.[1]

Textual differencing is not based on programming language, but we use a hybrid technique that combines text and syntax program differences to detect code changes, so it is necessary to choose the programming language. Since Java is one of the three most popular languages in the last twenty years[2] and has been widely used, this language was considered as a reference language.

We use a hybrid differencing technique as well as using block concept to divide the program into small trackable elements. Segmentation is defined on two levels.

[1] https://github.com/MafiZo/RichTest.git.

[2] https://www.tiobe.com/tiobe-index/.

High-level blocking considers each method as a block, low-level blocking considers each statement, such as an ***if*** statement, as a block.

After adding a new test case, we run that test case. If the test case passes, then the next test case will be added, but if the test case does not pass, the source code must be modified to pass the new test case.

- RichTest performs code segmentation to track code elements. It creates both code and test blocks.
- RichTest identifies all modified code blocks.
- RichTest connects modified code blocks to the new test case that leads to these changes.
- In the test selection phase, RichTest tracks and selects only those test cases that are related to the modified parts of the code, so instead of running all the test cases, only the selected test cases run.

We measured the number of selected test cases and RT time to compare our work with two types of TDD, as well as another Java plugin. The results showed that our work has an advantage in reducing the number of tests as well as the RT time.

Section 2 discusses the basics and the principles of TDD as well as the regression test, which must be run repeatedly in the TDD cycle. Program differencing as one of the regression test selection methods used in this article is presented in detail and a comparison between different levels of its implementation will be provided. Section 3 introduces related work.

In Sect. 4, our test case selection algorithm will be discussed in detail. Segmentation, segment comparison, and relationship creation algorithm are explained in this section. The RichTest tool, which is developed to implement the foregoing is explicated in Sect. 5. Automatic and manual block segmentation and regression test wizard are explained in this section.

Section 6 presents the evaluation of RichTest using another program that we implemented to access the TDD projects on GitHub to compare the number of executed test cases in TDD and RichTest. Section 7 concludes the paper. Several images of the RichTest tool are illustrated in Appendix A.

## 2 Background

The proposed technique allows for avoiding the execution of some test cases in TDD. This section discusses the basics and principles of TDD as well as its advantages and disadvantages. The regression test must be repeatedly run in the TDD cycle. Previous work on the regression test and the principal approaches for its cost reduction, particularly program differencing from the standpoint of regression test and other software maintenance applications, is presented in detail.

### 2.1 Test driven development (TDD)

In the traditional approach, software development proceeds by first creating the working code and thereafter writing unit tests (Ammann and Offutt 2008). This method is sometimes referred to as test-last development. In several traditional software development models, such as the waterfall model, software testing is one of the last tasks to be performed before the software maintenance phase. On the contrary, in modern and agile software development methods, testing is often adopted as an integrated part of the entire development process. This technique aids developers in finding and fixing bugs starting from the early phases of development. In test-driven development, however, software tests are written before the actual source code (Beck 2002).

The concept of the TDD method was first studied by Beck (Beck 2002). As its name suggests, TDD is a test-first software development approach for building software incrementally allowing test cases to drive the production code development. New test cases are written based on the software requirements and new features that should be considered in the software. If there is any fault or defect in the current version of the program, the test case will detect the problem. Then the developer would write the proper code to fix the failure. As a result, the tests are always written first, and thereafter only a sufficient amount of code is written to fix the failure (Beck 2002; Beningo 2022). Despite its name, TDD is not a test method; it is in fact a new software design and implementation method in which the idea of writing test cases before developing the code is combined with the concept of refactoring.

According to Astels, in the TDD method, the project is first broken into smaller parts using the divide-and-conquer method. The program is developed incrementally, starting from the development of each part by writing a test (Astels 2003). The TDD process proceeds as follows (Beck 2002; Beningo 2022):

1. Add a small test;
2. Run all tests and see if the new one fails (The test might not even compile);
3. Write a minimum amount of code to pass the test;
4. Run all tests and see all of them succeed;
5. Refactor the code to clean them and remove possible duplications.

The development process is thereafter continued by repeating the steps mentioned above.

### 2.2 Regression test (RT)

In the software development and maintenance process, product requirements are modified or corrected because of the addition of new customer requirements. These changes are implemented to match new technologies and environments, fix hidden errors that occur in various stages of development, and fix deficiencies and bugs to improve current features.

RT is an activity that is performed after a change is implemented in the system. Its objective is to reveal the defects that may have been introduced by these changes as a result of software evolution (Riebisch et al. 2012). In view of the large number of test cases, RT is extremely time-consuming. It is therefore an expensive test to validate the modified software. To reduce cost, several techniques may be employed. The four principal cost reduction approaches are (1) RT minimization, (2) RT prioritization, (3) RT optimization, and (4) RT selection (Rosero et al. 2016). The coverage-based RT using program differencing used in this paper can be considered as an RT selection method.

#### 2.2.1 Regression test minimization (RTM)

According to Yoo and Mark (2012), RTM refers to the removal of redundant test cases from the test suite. Minimization is sometimes also called test suite reduction, meaning that the elimination is permanent.

#### 2.2.2 Regression test prioritization (RTP)

Test case prioritization aims to reorder test cases to increase the rate of fault detection during RT. The RTP prioritizes tests based on error detection criteria or code coverage using experimental methods. Thus far, various prioritization strategies have been suggested (Zhang et al. 2013).

#### 2.2.3 Regression test optimization (RTO)

RT techniques are considered from the point of view of multi-objective optimization and Artificial Intelligence (AI). Their main goal is to select test cases through the use of optimization or AI approaches. Some of the RTO techniques are based on fuzzy logic, and some of them are based on heuristics. This technique includes contributions in the line of greedy algorithms, Pareto optimization, and integer linear programming in combination with genetic algorithms (Rosero et al. 2016).

#### 2.2.4 Regression test selection (RTS)

The RTS method chooses some of the test cases and ignores the rest. In this category, the reduction is also present but its strategy focuses on the detection of modified parts of a program that normally runs based on white box static analysis (Rosero et al. 2016).

Safe RTS techniques prove that under certain well-defined conditions, test selection algorithms exclude no tests (from the original test suite) that if executed would reveal faults in the modified software. Under these conditions, the algorithms are safe, and the fault detection abilities are equivalent to those of the retest of all tests. (Rothermel and Mary 1998).

### 2.3 Program differencing

In regression tests, the knowledge of which parts of the program are unmodified can aid in identifying the test cases that do not have to be executed (Apiwattanapong et al. 2007). Considering the fact that the behaviors of preserved components in the new and old versions of a program do not differ at runtime, it is guaranteed that no retest of all cases is necessary, and testing the affected component only is sufficient (Binkley 1992).

Program differencing is also a principal step to solve some of the crucial problems in software maintenance such as locating bugs, introducing changes, tracking code pieces or drawbacks in versions, merging files, and analyzing software evolution (Asaduzzaman et al. 2013). DbRT, a delta-based RT in the context of MDD proposed to propagate the changes from a software specification to testing artifacts in order to preserve consistency after system evolution (Nooraei Abadeh and Mirian-Hosseinabadi 2015). In general, software modification is classified into three levels: textual modification, syntax modification, and semantic or behavioral modification. The previous works are presented in these three categories.

#### 2.3.1 Textual differencing

In the textual approach, regardless of whether the code file is an executable program, the common parts of the two versions are identified using algorithms, e.g., "longest common sub-series algorithm." For instance, diff (Myers 1986) is among the most utilized tools in UNIX that presents the difference between two versions of a program. It generates a report consisting of a series of added or deleted lines between two files after identifying the common parts.

Vokolos and Frankl (1998) developed a tool for textual differencing, named Pythia, which is capable of analyzing large software systems written in C. The results indicate that this technique is considerably fast and can significantly reduce the size of RT suite.

An enhanced language-independent tool, LDiff (Canfora et al. 2009), is developed based on Unix diff and resolves numerous problems encountered by the latter. These include determining if a line has been modified or is a result of additions and deletions, and tracking code blocks that have been moved up or down inside the file.

Another tool that tracks source code lines between two different versions of the file is LHDiff (Asaduzzaman et al. 2013), which takes two different versions of the program as input and uses the Unix diff technique to identify unmodified parts. In order to track the remaining lines, a mixture of context and content similarities is used.

#### 2.3.2 Syntactic differencing

Yang (1991) obtained the difference between the two programs based on grammar and parse trees. This is known as the syntactic difference. Each program is displayed using a parse tree built by the parser. The tree-matching algorithm takes two trees as input and finds a set of pairs of nodes in which each node belongs to one tree and appears maximum in one pair.

Maletic and Collard (2004) presented a syntactic differencing approach to analyze source code differences. The meta-differencing approach attempts to automatically produce some information related to the difference between the two programs. Complex questions on the difference between two versions of a program can be solved by this system. Meta-differencing uses an XML-based language called SrcML to display the two programs and their differences.

Archambault (2009) took the graphs of two versions of a program and merged them based on similar node names to obtain a new graph. In order to reduce the graph size, the concept of MetaNode for collecting the nodes is employed. The betweenness centrality measure is used to determine the difference between the two input graphs. This value is determined for all graph nodes. The small and large values indicate the stability and instability, respectively, as well as the difference among the points.

Goto (2013) considered merging similar programs to increase program maintainability and focus on structural differences. The AST trees for two similar methods are first built using Eclipse JDT; the differences among the trees are then determined. Finally, coherent code pieces are identified as Extract Method (EM) candidates. The FTMPATool is implemented to accomplish this task.

The ChangeScribe (Linares-Vásquez 2015) tool is an Eclipse plugin that considers the textual differences between the new and previous version of the program at commit time and generates messages to automatically explain the modifications. ChangeScribe is currently applicable for Java projects on GitHub. Shen et al. (2016) continued this work by defining four types of changes to describe the code change and include information that explains the reason for the code change.

The LSDiff[3] (Kim and David 2009) tool attempts to answer some of the high-level questions of programmers and present systematic structural differences as logical rules. LSDiff represents each version of the program using a set of predicates that describe code components, their relationships, and their structural dependencies.

Falleri et al. (2014) employed the GumTree tool, which is comprised of two sequential steps, to compute the mappings between two ASTs: (1) top-down greedy algorithm for finding isomorphic subtrees, and (2) bottom–up algorithm to detect corresponding nodes.

The SEGMENT tool (Wang et al. 2011) divides the different parts of the program by adding blank lines to increase the readability of the program. SEGMENT uses the program structure AST tree as well as the name information and identifies meaningful primary blocks with a particular logical operation. In order to identify logical blocks, three main types of blocks are considered: syntactically the same, data flow chain, and extended SWIFT.[4]

[3] Logical Structural Diff.

[4] Statements such as synchronized, do, try, for, if, while, and switch.

### 2.3.3 Semantic differencing

Horwitz (1990) used a program graph representation and a partition operator on these graphs to semantically find differences. His partitioning algorithm is limited to a language with scalar variables, conditional statements, assignment statements, while loops, and output statements.

Binkley (1992) reduced the RT cost by using semantic differences between the two programs. In his work, the limitations of program statements are reduced compared to those in Horwitz (1990). He also included function definitions and function calls. He used a system dependency graph instead of a flow control graph that avoids unnecessary dependencies among the components on a path in a control flow graph. Binkley reduced the complexity of test cases using the program slicing technique.

Neamtiu et al. (2005) proposed a tool to rapidly compare the source code of different versions of C programs and thereafter find semantic differences among program versions based on partial AST matching. The tool can track simple code-level modifications related to changes in global variable names, types, and functions. This tool compares the body of functions with similar names considering that the name of function is not changed throughout the software lifetime.

Apiwattanapong et al. (2007) presented a method to compare object-oriented programs and used an extended control flow graph (ECFG). Görg and Zhao (2009) extended the method proposed in Apiwattanapong et al. (2007) in such a way that it also supports the new concepts introduced by aspect-oriented programs.

The patent in Hsu (1999) presents a technique for identifying the differences between two graphic programs. BinHunt (Debin et al. 2008) is aimed at identifying the semantic differences in the binary code between the two programs that can be used in cases where the program code is not available. BinHunt uses the STP[5] theorem proving and symbolic execution to compare the primary blocks. It is applicable only for minor differences.

Wang et al. (2014) used normalized[6] control dependence trees to represent two versions of the program and improved the traditional metrics-based and graph-based approaches to propose a combinational approach.

Liu et al. (2006) produced a plagiarism detection tool called Gplag. Plagiarized codes are often modified for deception, and identifying such codes is possible by using a suitable and similar code identification tool. This approach represents the program code as program dependence graphs (PDG) and identifies similar code based on the sub-graph isomorphism test.

Nguyen (2011) proposed the iDiff tool as a plugin in Eclipse for identifying program differences. The iDiff can identify changes in classes and methods, track re-ordered, relocated, and renamed classes and methods, and detect internal changes in methods. The iDiff uses JavaModel and ASTParse related to the JDT plugin in order to parse the project for obtaining all information related to the types and limitations of methods.

[5] Simple Theorem Prover.

[6] Code normalization is a semantic-preserving transformation.

#### 2.3.4 Summary of program differencing

Table 1 summarizes the above references related to program differencing according to the type of difference identification (text/ syntax/ semantic) and tool produced. Some of these tools are related to a particular language, developed for multiple languages, and not language-dependent. Some of them normalize the code before identifying the differences and use a limited set of statements for simplification. Most of the tools use graph or tree structures.

Graph-based methodologies consider both syntax structure and data stream as abstraction levels, making those suitable bases for identifying similar code on a semantic level. Sometimes, however, problems, such as code diversity, hinder the identification of similar codes. High computational complexity in graphs limits graph size. Some studies have attempted to resolve this problem by forming meta-nodes and reducing the number of graph nodes (Archambault 2009). A tree, as a special form of graph, reduces computational complexity. In particular, the use of AST trees neglects certain basic differences by considering the syntax structure (Yang 1991; Goto et al. 2013; Falleri et al. 2014; Wang et al. 2011; Neamtiu et al. 2005; Nguyen et al. 2011; Wang et al. 2014). We also use the AST tree as the base of our change detection algorithm.

Each article examined for this research has certain deficiencies. For example, some do not thoroughly discuss language statements (Horwitz 1990), exhibit certain limitations (Linares-Vásquez et al. 2015), or encounter computational problems as the program grows larger and the number of graph nodes increases (Debin et al. 2008). Some do not capable of tracking the relocated code or matching a single line of code with multiple lines with the same meaning (Canfora et al. 2009). Others do not detect the updated code and only detect lines that are either added or deleted (Myers 1986; Vokolos and Frankl 1998). There are those that require a pre-processing phase to normalize code (Asaduzzaman et al. 2013; Horwitz 1990; Wang et al. 2014). Additionally, most of the programs have high time complexities in the order of $O(n^3)$ or $O(n^2)$. The idea presented in this paper overcomes some of these limitations and its time complexity is O(n). Table 2 compares the three types of program differencing (text/ syntax/ semantic).

Textual differencing can be applied to any text file. It indicates detailed changes such as added or deleted or updated lines. Its line-based view does not respect syntactic boundaries. Thus, the differences often do not sufficiently reflect on the real meaning of the changes and often are not readable enough, also relocating the code may be unsupported.

Syntactic differencing is based on grammar and parse trees, therefore it ignores changes to whitespace, comments, and preprocessor statements. Tree-matching algorithms are used to identify unchanged parts of the tree (code) and display the remaining parts as syntactic differences. These algorithms are generally slow and thus do not scale to large systems. Also, sometimes two completely identical structures may be in different situations that show different functionalities and are not semantically the same.

Semantic differencing corresponds to changes in the program functionality and is not related to programing structure or statements. Normalization methods are

**Table 1** Comparison of program differencing references

| References | Type | Language | Limit | Normalize | Representation | Description |
|---|---|---|---|---|---|---|
| diff (Myers 1986) | Text | Independent | X | X | Text | Add and delete only |
| Pythia (Vokolos and Frankl 1998) | Text | C | X | X | Text | Add and delete only |
| Ldiff (Canfora et al. 2009) | Text | Independent | X | X | 1:1 Line | Add, delete, update, and relocate |
| LHDiff (Asaduzzaman et al.2013) | Text | Independent | X | √ | Neighbor Lines | Add, delete, update, and relocate |
| Yang 1991) | Syntax | C | X | X | AST | AST and synchronous pretty-printing |
| Maletic and Collard 2004) | Text and Syntax | C/C++ | X | X | SrcML[a] | High Level |
| Archambault 2009) | Graph | Independent | X | X | Graph | Meta node |
| FTMPATool (Goto et al. 2013) | Syntax | Java | X | X | AST | |
| ChangeScribe (Linares-Vásquez et al. 2015) | Text and Syntax | Java | JGit | X | Code Change NLP Impact Analysis | Add comments add, delete, and update |
| Shen et al. 2016) | Syntax | Java | X | X | JGit System Version Control | Add why and what comments |
| LSDiff (Kim and David 2009) | Syntax | Java | X | √ | Predicate | |
| GumTree (Falleri et al. 2014) | Syntax | Java | X | X | AST | Relocate |
| Segment (Wang et al. 2011) | Syntax | Java | X | X | AST | Basic block detection |
| Horwitz 1990) | Semantic | C | √ | √ | PRG[b] | Safe set |
| Binkley 1992) | Semantic | Independent | √ | X | PDG[c] | More complete than (Horwitz 1990) |
| a tool (Neamtiu et al. 2005) | Semantic | C | X | X | AST | Semantic-preserving transformations |
| JDiff (Apiwattanapong et al. 2007) | Semantic | Java | X | X | ECFG[d] | Difference testing |
| AJDiffer (Görg and Zhao 2009) | Semantic | AspectJ | X | X | ECFG | |
| Hsu 1999) | Semantic | graphical | GUI | X | Matching Matrix | graphical programs |
| Binhunt (Debin et al. 2008) | Semantic | Binary Code | X | X | CFG[e] and CG[f] | Small difference, Low speed |
| Wang et al. 2014) | Semantic | C | X | √ | PDG and AST | Augmented graphs |
| GPlag (Liu et al. 2006) | Semantic | C/C++, Java | X | X | PDG | Plagiarism detection |
| iDiff (Nguyen et al. 2011) | Semantic | Java | X | X | JavaModel | JavaModel, ASTParser attributed graph |
| RichTest (Our Work) | Text & Syntax | Java | SWIFT | X | AST, JSON | add, delete, update, and relocate, Reduce RT |

**Table 1** (continued)

[a]Source Code Markup Language

[b]Program Representation Graph

[c]Program Dependence Graph

[d]Extended Control Flow Graph

[e]Control flow Graph

[f]Control Graph

objective of this study is to identify 20% of the tests that can detect 80% of errors instead of creating an infinite subset of tests that detect 100% of errors.

Different from other safe selective RT methods, this technique limits the number of selected test cases. Results show that the test suite is not safe Results show that the test suite is not safe because 20% of the errors were ignored. The restricting method reduces this problem to a prioritization problem, which chooses 20% of the higher-priority test cases.

Cibulski presented selection techniques based on natural language analysis and dynamic programming via the TestRank tool. TestRank takes a Java program with its test suite as input and requires a pre-processing step, which is considerably time-consuming. As mentioned above, two fundamental problems arise: (1) the test suite is unsafe, and (2) the synchronization of the system with the latest version of the program is considerably time-consuming (up to one day, 24 h).

As another related work, we refer to ChangeScribe (Linares-Vásquez et al. 2015) and iDiff (Nguyen et al. 2011) tools, which are Eclipse plugins similar to our project. These plugins generate comments to explain changes. ChangeScribe only considers the textual differences of the new program from the previous version and generates comments that explain changes. ChangeScribe, however, cannot be used for RTs and is only applicable for Java projects existing on GitHub because it does not have a version manager. The iDiff tool receives two program versions at a time and determines the modified, deleted, or added classes and methods. It does not provide, however, a complete environment that contains all versions created throughout the software evolution process. Also, Eclipse has been considered in Santosh Singh and Kumar (2018) for learning techniques selection.

# 4 Methodology

In the TDD method, any minor changes result in RT. The problem, therefore, is the growing number of tests and the necessity of re-executing these tests. Finding a small subset of the test suite that can be utilized to scrutinize the software with high confidence is thus important.

## 4.1 Add a new phase to three phase TDD cycle to reduce the test re-execution time

As pointed out in Biswas et al. (2011), reducing the time of test execution differs among various software development methodologies, so a TDD-specific approach should be determined to choose test cases that must be re-executed in each iteration of the TDD process.

In pure TDD, the part of the code that each unit test belongs to is precisely determined. The code is developed after writing the test; hence, there is a close relationship between the unit test and the modified code. In every step of the software development process, the modified parts of the code are determined, and only tests that lead to these parts are chosen for re-execution.

**Table 2** Comparison of the three types of program differencing

| Factor | Type | | |
|---|---|---|---|
| | Textual | Syntactic | Semantic |
| Speed | Fast | Not Fast | Not Fast |
| Accuracy | High | Medium | Medium |
| Readability | Low | High | High |
| Scalability | High | Low | Medium |
| Flexibility | High | Language dependent | Language dependent |
| Abstraction Level | Line-based | Statement-based | Module-based (function-based/class-based) |
| Regardless of worthless details | Low | High | High (ignore refinements) |
| Modification Level | Add, delete, (update) | Add, delete, update, relocate | Transformation |
| Representation | Line, Text | AST tree | Graph, Tree |

usually used in order to remove code variations. Module signature modification is considered as a semantic difference.

## 3 Related work

First, previous works on the TDD are examined and different approaches are considered. The various methods that have been suggested are studied to reduce the RT execution time and to propose a suitable method to reduce this time in the TDD method.

As an instance, Continuous Test-Driven Development (CTDD) recommends background testing to reduce this time. CTDD is a recent enhancement of the TDD practice and combines the TDD with continuous testing practice. During the execution of test cases, the developers have to stop the system to execute the test physically, thus increasing the program development time. By using the continuous compile feature in the new IDEs, e.g., Eclipse or Visual Studio that keep the source code in the compiled mode, this goal of reducing execution time will be realized (Madeyski and Marcin 2013).

Madeyski and Kawalerowicz (2018) evaluated the CTDD practice via an empirical study in a real industrial software development project that employs Microsoft.NET. If the developers that use TDD adopt CTDD, it can run slightly faster, thereby leading to slight improvements in coding. Although the idea is to write a code and execute the test in parallel, it does not change the number of test cases and the number of times they run; hence, it does not reduce the amount of load and processing costs. In terms of reducing the number of test cases, our proposed method is thus preferred.

In another instance, Cibulski and Amiram (2011) performed the RT in TDD. A small subset of test suites for each small local change is automatically found. The

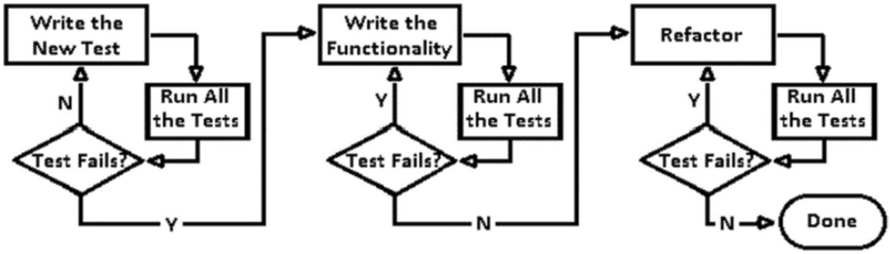


**Fig. 1** TDD activities (Madeyski and Marcin 2013)

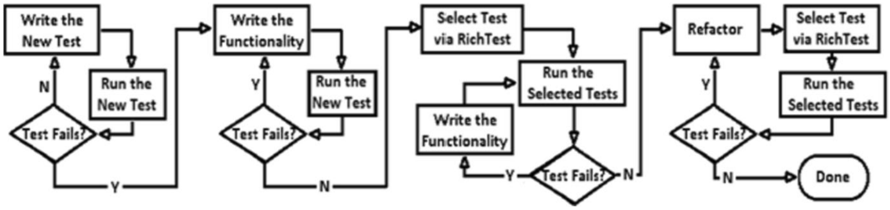


**Fig. 2** Our Improved TDD activities to reduce test cases

Figure 1 illustrates the TDD tasks that are comprised of three steps, which correspond to the three phases of the TDD cycle. In the first step, the new test is written and executed until an error occurs. In the second, the code is written to pass the test. In the third, the refactoring phase occurs.

Figure 2 illustrates the tasks in our improved TDD cycle that are comprised of four steps. The first and second steps are similar to the first two steps illustrated in Fig. 1. In these two steps, however, only "the new test" is executed instead of executing "all test cases". The third step is a new phase added to this figure. In this step, tests that require re-execution are selected and executed using our selection algorithm. The last step in both figures is refactoring. In the refactoring phase of Fig. 2, only tests that are related to the modifications are selected and executed. In the improved TDD, test case execution is limited in all of the given steps, as illustrated in the flowchart in Fig. 2.

## 4.2 Segmentation

First of all, we divide the program into several code blocks based on the Java programming language grammar. Program segmentation has three benefits:

1. The program is divided into small independent components called blocks.
2. Each block has a fixed unique name, so it can be traced. Line tracking is not applicable. Because the program changes and as a result the line number also changes.

3. It is possible to detect changes in the program by detecting changes inside the block. Also, the location of changes in the program is specified precisely.

We desire two levels of granularity for these code blocks: (a) coarse-grained level for whole classes and methods and (b) fine-grained level for language control flow statements. However, structured block information is stored in a database.

By segmenting the program code into blocks and assigning a name to each block, code tractability property is created, so any movement or update in the block content will therefore modify the program code in that block. This determines the location of changes and makes block relocation traceable.

### 4.3 Change detection algorithm

We initially decided to compare the block content textually. Textual-differencing approaches are limited to a line-level granularity. We omitted extra spaces between words and lines, as well as entire comments, then we compare this pre-processed text of each block with its previous version to detect if it has changed. Later, however, we also decide to use an abstract syntax tree to compare the contents of each block. By applying this structure, minor changes can be ignored too. So, we use the combination of text and syntax differencing method. The difference between the two versions of a program is determined by identifying the modified code blocks based on Java grammar as a combination of textual and syntactical difference methods.

Although semantic and behavioral modifications are at a higher level and indicate real changes, the focus of this study is on textual and syntactical modifications. The reason behind this choice is that we have to find all the tests that require re-execution after code modifications. In the case of omitting tests that check the changes in appearance (e.g., change in the name of a variable or method), the set of test cases is not considered safe. Hence, although the modifications are of the refactoring type, the tests should be re-executed to ensure accuracy. Focusing on the textual and syntactical levels may ensure the safety and reliability of the RT.

### 4.4 Relationship between test case and code blocks

After adding any new test case that has encountered errors, new code blocks are created, or existing code blocks are modified. These modifications are implemented to pass the last test; therefore, the last test is related to the modified code block(s). A connection must therefore be automatically established between the modified code blocks and the last test case to be used by the selection algorithm.

Given project $P$, includes a set of code blocks $C$ and a set of test cases $T$. To pass the new test case $t$, some of the code blocks $M \subset C$ will be modified (to $M' \subset C'$) and new code blocks $N$ may be created. So the new version of project $P'$ consists of $C'$ and $T'$ such that:

$$C' = (C - M) \cup (M' \cup N) \tag{1}$$

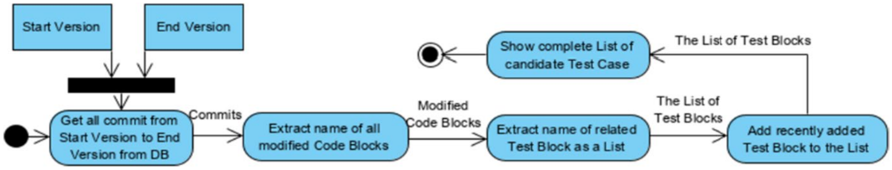


**Fig. 3** improved test case selection process

$$T' = T \cup \{t\} \tag{2}$$

We define ***Link*** relation as follow:

$$\text{Link}: \quad C' \times T' \tag{3}$$

$$\forall c \in (M' \cup N), \text{Link}(c, t) \tag{4}$$

### 4.5 Test case selection

In the TDD method, the code is written or modified only because of test failure. In our proposed concept, however, the failed test is connected with modified code blocks. This task is iteratively executed, and the connections between the code blocks and related unit tests are established and tracked. In order to run the RT, the test cases connected to modified or newly added code blocks are chosen as candidate unit tests for execution.

As a result, the iterative execution of test cases, which are not connected to the modified parts of the code, is avoided, and the number of selected test cases is reduced.

After specifying the '*Start*' and '*End*' versions of the program for RT, the latest commit[7]s and new test cases are identified in this interval. All code blocks related to the new tests are specified, and the tests relevant to these code blocks are introduced as candidate tests. Figure 3 illustrates our improved test selection algorithm. At the first, RichTest identifies the involved commits from the start version to the end version. Then it extracts all the modified code blocks. In the next step, it extracts all the related test blocks. After all, it adds the recently add test block to the list and shows the final complete list of candidate test cases.

As shown in Fig. 3 the RichTest built-in version manager lets the custom start and end version, not necessarily consecutive version, although it is set to the last two versions by default.

Our test case selection algorithm is presented using the following example.

[7] Each copy of the program a developer saves. It is not necessarily a new issue/version of the program.

**Table 3** Relationship of test and code in Example 1

| Commit | Test block | Code block(s) |
|---|---|---|
| c1 | Ta001 | Ca001 |
| c2 | Ta002 | Ca001 |
| c3 | Ta003 | Ca001, Ca002 |
| c4 | Ta004 | Ca003 |
| c4 | Ta005 | – |
| c5 | Ta006 | Ca004 |
| c6 | Ta007 | Ca003, Ca005 |
| c6 | Ta008 | – |
| c7 | Ta009 | Ca001, Ca006 |
| c8 | Ta010 | Ca001, Ca007 |

**Table 4** Modified code blocks in Example 1

| Code block | Test block(s) |
|---|---|
| Ca001 | [Ta001, Ta002, Ta003, Ta009, Ta010] |
| Ca006 | [Ta009] |
| Ca007 | [Ta010] |

### 4.5.1 Test case selection example

Suppose that test cases Ta001–Ta010 are written in sequence. In order to pass each test, code blocks Ca001–Ca007 are added or modified, as listed in Table 3.

Ta005 and Ta008 pass immediately without changing the code, but the rest of the test cases cause changes in some code blocks and a new commit is generated. Commits c1-c8 shows all the saved program copies.

> A question then arises: from the commit related to Ta008, i.e., c6–c8, which test cases are selected for the RT?

It can be observed that Ta008–Ta010 are new tests in this interval that are related to Ca001, Ca006, and Ca007 code blocks, respectively. These code blocks are connected to Ta001, Ta002, Ta003, Ta009, and Ta010 test blocks (as shown in Table 4) that are candidates in the RT.

Although T8 is recently added, its re-execution is unnecessary because this test previously passed without making any code modifications. To ensure safety, however, this test is still considered.

## 4.6 RichTest

RichTest plugin (Rich Software Testing) is based on the Eclipse integrated development environment and is written in Eclipse version 4.8, which is recommended

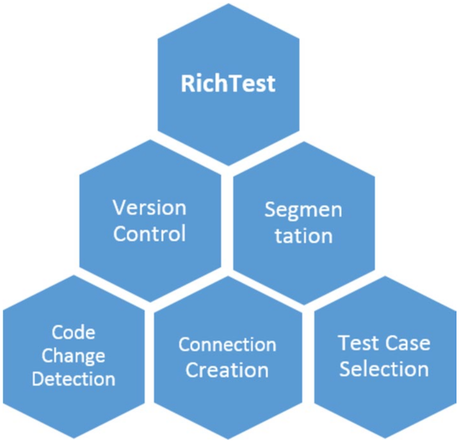


**Fig. 4** The main components of RichTest

for running RichTest. This tool consists of five main components, which are (1) Version Control Manager, (2) Code Segmentation, (3) Code Change Detection, (4) Connection Creation between Code and Test Blocks, and finally, (5) Test Case Selection as shown in Fig. 4.

### 4.7 RichTest algorithm

The algorithms of each of the five modules shown in Fig. 4 are presented separately in Algorithm 1 to Algorithm 5. Algorithms 1 to 4 are executed sequentially after saving the program, while Algorithm 5 is activated by running the regression test wizard.

**Algorithm 1** Version Manager (Trigger: Click the Save button in the Eclipse IDE)

```
Begin
    Static VersionNumber= 1.0.0
    Display the recommended VersionNumber for the program as a three-part number.
    Allow the user to change the VersionNumber.
    Store the program specifications in the database.
    Allow the user to select any VersionNumber of project to view its specification.
End.
```

**Algorithm 2** Code Segmentation (Trigger: Click the Save button in the Eclipse IDE/CTRL+1)

```
Begin
    Static TBlockname=Ta001
    Static CBlockname=Ca001
    foreach file f in the project
        Search the file f to find "@Test" annotation.
        Consider each test case as a test block.
        Assign a unique name to each test block according to the test naming guidelines.
        Insert block information into the database.
        Insert "//Start Of Test Block: + TBlockName" comment at the beginning of each test block.
        Insert "//End Of Test Block: + TBlockName" comment at the end of each test block.
        If (BlockSelection=="Automatic")
            foreach file in the project
                Create the AST tree according to Java programming language grammar (exclude test cases).
                Convert each node to JSON format
                If (CodeGranularity==" FineGrained")
                    Select every new Statement as a block
                If (CodeGranularity=="CoarseGrained")
                    Select every new Class and Method as a block
                Assign a unique name to each new block according to the code naming guidelines.
                Insert block information into the database.
                Insert "//Start Of Code Block: + CBlockName" comment at the beginning of each code block.
                Insert "//End Of Code Block: + CBlockName" comment at the end of each code block.
        If (BlockSelection=="Manual")
          while CTRL+1 buttons clicked by user do
          Consider the section selected by the user as a new code block.
          Assign a unique name to the new code block according to the code naming guidelines.
          Insert block information into the datatbase
          Insert "//Start Of Code Block: + CBlockName" comment at the beginning of the code block.
          Insert "//End Of Code Block: + CBlockName" comment at the end of the code block.
          Specify Child and Parent nodes according to the tree structure
          Store Parent information in the database.
End.
```

**Algorithm 3** Code Change Detection (Trigger: Click the Save button in the Eclipse IDE)

```
Begin
    C= Ø //All AST tree Nodes
    N= Ø //New Nodes
    M= Ø //Modified Nodes
    T= List of the newly added test blocks.
    foreach file f do
        Construct AST tree of file f (considering CodeGranularity)
        Compare the AST tree of this version of the program with the previous version syntactically (exclude test blocks)
        foreach node n in AST
            if n is new
                N=N∪{n}
            else
                C= C∪{n}
        foreach node n in C do
            If (JSON(n) != JSON (n')) // Compare the JSON content of two versions n, n'(previous) textually
                M=M∪{n}
            After identification of the new (N) and modified (M) blocks, store block specifications in the database.
End.
```

**Algorithm 4** Connection Creation (between Test Case and Code Blocks) (Trigger: Click the Save button in the Eclipse IDE)

```
Begin
//use T, M , N List produced from Algorithm 3
    foreach newly added test case t∈T do
        foreach changed code block c∈(N∪M) do
            Link code block c to test block t.
End.
```

**Algorithm 5** Test Case Selection (Trigger: Regression Test Wizard available through RichTest Plugin)

```
Begin
    Input the two version numbers of the program as Start and End.
    C= All the commit from Start to End version (Extract from the database.)
    T= All the test case that has been inserted from the Start version to the End version
    B= { } // affected code blocks
    foreach commit c∈C do
        M_C= List of all the modified code blocks.
        Pc= List of all the Parent of each selected modified block existing in M. //M is available from Algorithm 3
        Bc= Mc∪Pc
        B = B ∪Bc
    foreach modified code block b∈B do
        T_b =List of all the test cases that are linked to code block b
        T= T ∪T_b
    Output T as candidate test cases.
End.
```

### 4.8 RichTest plugin overview

By installing[8] RichTest on Eclipse, the developer will be able to develop TDD projects faster and easier as fewer test cases are selected and executed in the development phase. It also offers several widgets,[9] such as Block Information View, Commit View, Version Manager View, Regression Test View, and Compare View to facilitate the use of RichTest which is explained below.

***BlockInfoView***: It is possible to display the Block List and the relationship between code blocks and test blocks, as well as manage the relationship manually.

***CommitView***: It is possible to show all block creations and modifications and also filter all versions and commits of each block.

***VerssionManagerView***: It is possible to set a new version for the projects.

***RegressinTestView***: It is possible to automatically select candidate test cases, run them to show the time and results (Fail/Pass), and export them to an Excel file format.

[8] Help→Install New Software, and also should set Window→Preferences as Dependency folder address.

[9] Available from Window→Show View→Other→RichTest.

***CompareView***: It is possible to compare two different commits of each block. The code block will be shown in two situations (before/ after) and the differences will be colored and presented on ***CompareResultsView***.

**Preferences**[10] such as Automatic/Manual Block Selection, Code Granularity (Coarse/Fine), and Enable/Disable TDD Mode. Related figures are attached.

Figure 5 is a snapshot of using this plugin as well as its widgets. More additional images are provided in Appendix A.

### 4.9 RichTest plugin working process

RichTest segments the source code and test code into code blocks and test blocks, respectively, during the project development process. It also identifies modified code blocks in each commit, detects the relationship between test blocks and code blocks, and stores them in a database. The main purpose of RichTest is to find candidate test cases for the RT process that are made possible by the connections already made between test blocks and code blocks.

#### 4.9.1 Automatic block segmentation

The segmentation process can be implemented both manually and automatically. In the automatic mode, whenever a file is stored, the plugin segments the file contents into blocks, adds new blocks, and updates modified ones. There are two types of blocks: test block and code block.

1. Test block is in fact a complete test case. It is considered as a block only because of its similarity to the code block.
2. The code block is determined based on the structure of the programming language instructions. Each block represents a node in AST.[11]

Automatic test block segmentation detects the "@*Test*" annotation to identify each test block, and automatic code block segmentation is based on the AST. The code block granularity degree can be chosen from two levels: (a) coarse-grained level for classes and methods and (b) fine-grained level for language control flow statements (SWIFT instructions[12]). The first level produces larger and fewer blocks, and the second level produces smaller and more blocks, especially in large projects. The automatic code block segmentation activity diagram is shown in Fig. 6.

During segmentation, a unique name is automatically assigned to each new block. The block nomination method varies depending on whether the block is a code block or a test block. The names of code and test blocks follow the LNC and LNT regular expressions, respectively.

[10] Available from Window → Preferences → RichTest.

[11] Abstract Syntax Tree.

[12] sw itch, while, if, for, foreach, and try.

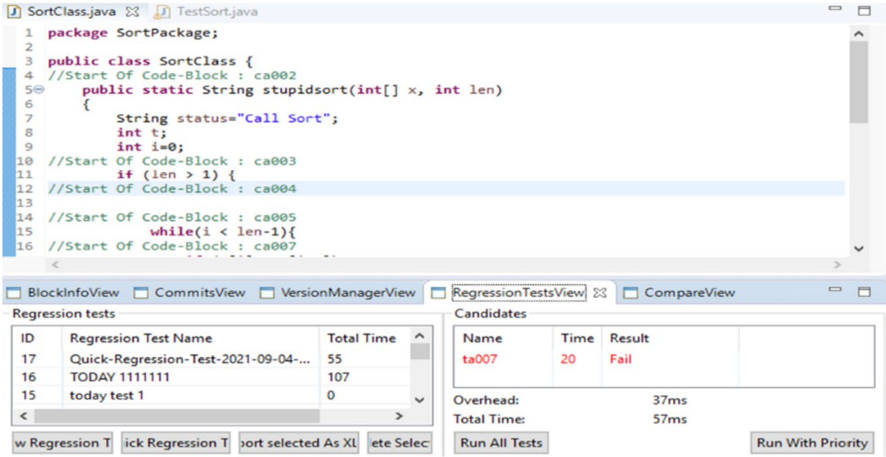


**Fig. 5** Using the RichTest plugin in Eclipse for the sort program

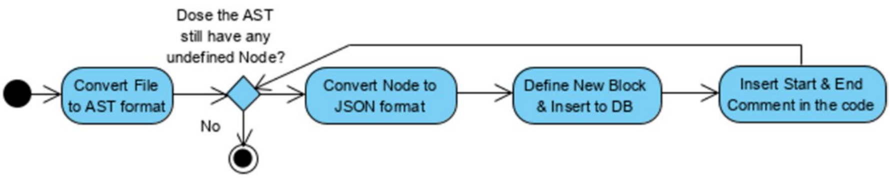


**Fig. 6** Automatic code block segmentation

$$LNC = \text{'}C\text{'}\ lddd \tag{5}$$

$$LNT = \text{'}T\text{'}\ lddd \tag{6}$$

$$l\ ::=\ a|b|c|\ldots|z|A|B|C|\ \ldots\ |Z \tag{7}$$

$$d\ ::= 0|1|2|\ldots|9 \tag{8}$$

### 4.9.2 Manual block segmentation

Segmentation can be manually performed by the developer. Using RichTest, any valid arbitrary part of the code could be specified as a block by simultaneously selecting the desired part of the code and pressing CTRL + 1 Keys. The manual code segmentation activity diagram is shown in Fig. 7.

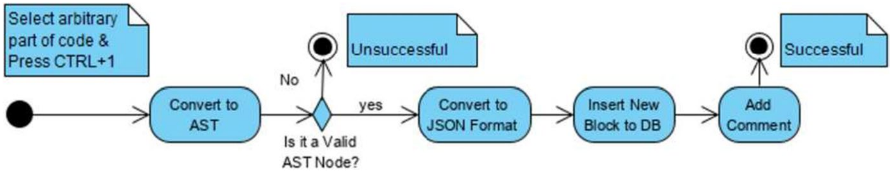


**Fig. 7** Manual code block segmentation

#### 4.9.3 Difference detection algorithm

The RichTest tool transforms each code block into a JSON array. In order to identify the differences in each code block, the elements of the JSON array are compared with those of the previous state. If there is a difference among the array elements, then this block is recognized as a modified block, and the block contents and properties are updated in the database. The JSON is a structured textual format for holding the information that ignores ineffective textual modifications (e.g., adding comments).

The primitive version of the plugin has no programming language limitation and is capable of supporting all languages supported by Eclipse because it uses a text-based difference algorithm. The new version of the plugin, however, is only applicable to the Java programming language because it detects differences using the AST based on Java grammar and stores the syntax information of blocks.

In the new version, the comparison method is a combination of both textual and syntactic differencing methods. Segmentation is performed based on Java syntax, and the block content is stored in the AST model. The data values are compared based on their textual contents.

As emphasized in the literature review, the use of each of the existing methods to find textual and structural differences has advantages and limitations. In this study, these two methods are combined to exploit the following advantages: precision and speed in textual difference, code relocation, and ignoring insignificant modifications in a syntactical structure. The textual difference related to each small modification is considered in the AST to ensure that no related test is ignored in the test case selection process.

#### 4.9.4 Connecting code blocks to test blocks

Each code block can be connected to one (or more) test block(s). In the manual mode, the block relationships can be manually managed using the "Block Information View." In automatic mode, the last test block added is automatically connected to all modified code blocks. In this mode, however, it remains possible to manually manage block connections.

Figure 8 shows an example of the relationship between test and code blocks. A code block may be associated with none, one, or several test blocks. As shown in Fig. 8 the Ta001 test block is first written, then the Ca001 code block is generated as a result of

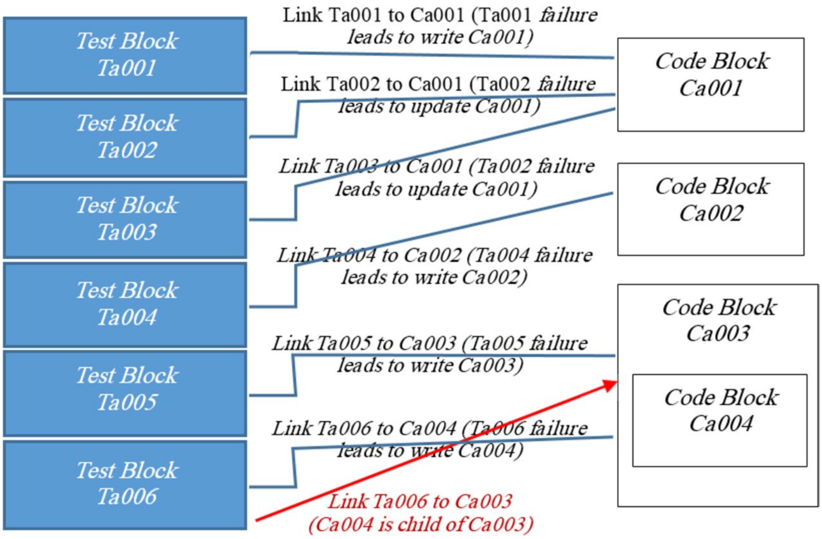


**Fig. 8** Example of n:n relationship between test blocks and code blocks

the Ta001 test failure. Next, the Ta002 test block is written; subsequently, there is a change in block code Ca001. To pass the Ta003 test, block code Ca001 is modified again The Ta004 test block generates the Ca002 code block. The Ca003 code block is created after the Ta005 test block failure.

Inside an existing function, a new loop statement may be added that can be defined as a new code block. In this case, the internal block is a part of the external block, and the test connection to the internal block also extends to the external block. The Ta006 test block is, directly and indirectly, related to Ca004 code block and external Ca003 code block, respectively.

After each newly added test fails, new code block(s) are created, or existing code block(s) are modified. These changes are necessary to pass the last test. Semantically, the given test is relevant to these modified code block(s). A link is therefore created from each of the modified code blocks to the last test; this connection is stored in the database. Figure 9 shows how the connection between code blocks and test blocks is established.

### 4.9.5 Regression test wizard

"Regression Test Wizard" produces a list of candidate test cases between the "Start Version" and "End Version" of the program. The wizard also assigns a name for the list. The last and previous versions are considered as default for the End and Start versions. After specifying the desired Start and End versions, recently added test cases are highlighted, and all test cases associated with the modified code blocks are also nominated. Only candidate test cases are shown. These can be saved and run, as shown in Fig. 10.

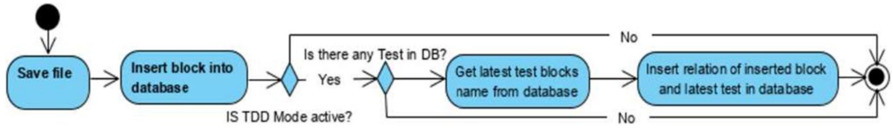


**Fig. 9** Relationship between modified code blocks and new test block

After the execution of test cases, successfully passed and failed test cases are determined. The passed tests are identified in green with a "success" result tag, whereas the failed tests are identified in red with a "fail" result tag. The runtime information of each test case is in milliseconds. Candidate test case information can also be viewed and executed through "Regression Test View."

### 4.10 Empirical evaluation

For the preliminary evaluation, RichTest is employed in three simple examples: exponentiation (power), array selection sorting, and linked list that calculates an integer number raised to the power of a positive integer, sorts array elements in ascending order, and creates and modifies linked lists, respectively. These three programs were written step by step according to TDD kata (Wolfgang 2018) when the RichTest plugin had not yet been implemented by one of the authors. Exactly the same process was re-implemented with RichTest after implementation by another authors.

"Re-implementation" is the same process as implementation, except that it is done in the presence of the RichTest plugin to automatically perform some tasks such as code segmentation, difference detection, relation creation, and test case selection.

Since our goal is to measure the effectiveness of the tool, we kept all the conditions constantly except the implementation environment. For this purpose, we added the same previous test cases one by one and wrote the same previous codes utilizing RichTest. This plugin reduces the number of execution of test cases by selecting some of the test cases. Four large projects are also implemented with and without RichTest tool. Full details are presented in subsequent sections.

### 4.11 Small program development using RichTest

The three small programs—Power, selectionSort, and linkedList—are implemented in the Java programming language using the TDD method twice, with and without utilizing RichTest. Power, selectionSort, and linkedList programs were implemented by five, ten, and nineteen test cases respectively. The two first implementations took five steps, so they have five versions. The last one was implemented in ten steps, so it has ten versions. The implementation results are summarized in Table 5.

It is predictable that the total number of tests performed in the TDD method is more than our method. Because we select some of the test cases, while traditional

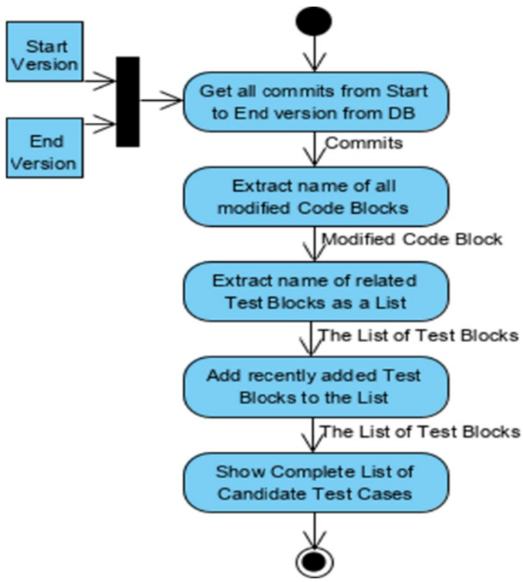


**Fig. 10** RichTest test selection process sequence diagram

**Table 5** Comparison of number of test executions in TDD and RichTest (three simple programs)

| Factor | Program | | | | | |
|---|---|---|---|---|---|---|
| | Power | | selectionSort | | linkedList | |
| | **TDD** | **RichTest** | **TDD** | **RichTest** | **TDD** | **RichTest** |
| Total number of executions (sum) | 61 | 6 | 68 | 9 | 387 | 25 |
| Average= sum/v | 12.2 | 1.25 | 13.6 | 1.8 | 38.7 | 2.5 |
| Number of tests (n) | 5 | | 10 | | 19 | |
| Number of versions(v) | 5 | | 5 | | 10 | |

TDD, executes all of them. But the difference between these two methods is huge. It is trivial that as the program grows larger, the number of commits also increases; consequently, the advantages of RichTest become more evident. The RichTest plugin successfully reduces the selected test cases by reducing the number of test cases and the number of times each test is executed.

## 4.12 Large project development using RichTest

In order to evaluate RichTest with large and real programs and identify projects based on the TDD in GitHub, a survey is conducted using a new program. Similar to the work of Borle et al. (2018), this program searches GitHub for projects that contain created test files before project development or at least one week thereafter.

#### 4.12.1 TDD projects on GitHub

To compare the plain TDD method with the suggested improved technique, some real TDD Java projects are selected from GitHub. Although GitHub provides a code repository for projects, it is not possible to determine the development process of projects. On the other hand, there is no precise definition for TDD projects. It is also not possible to determine with certainty whether the project follows the TDD process using a project repository. Borle et al. (2018) formulated a method for detecting TDD projects on GitHub; however, the names of discovered projects were not disclosed. The authors acknowledge the uncertainty of results with respect to the foregoing problems and attempt to construct a range of code repositories that shows the extent that the TDD process is employed in their projects.

We implemented a Java script program that includes ten asynchronous and normal functions to crawl GitHub repository. First, it creates an asynchronous iterator over all public repositories of GitHub that have Java listed as one of their languages. Then it filters the returned values, limiting them to repositories that have all the following specifications:

1. Primary Programming Language = 'Java'

Size > minSize

No. of Commit > minNoCommit

No. of TestFile > 0

(TestCreateDate < CodeCreateDate) or ((TestCreateDate < 30th CommitDate) and (TestCreateDate < CodeCreateDate + 1 week))

This program is employed to find the TDD projects on GitHub. Within one hour, 89 projects with the above-mentioned properties are identified. Six of these projects, which have a suitable number of lines and commits that could be executed in Eclipse, are chosen for evaluating the RichTest tool. These projects are ScribeJava, Jasmin-Maven Plugin, Metric-Core, Jedis, Commons-Math, and Junit-dataprovider. Table 6 summarizes the properties of these projects.

Scribejava is a simple OAuth library for Java. Jasmine-maven plugin is a Maven plugin for the JavaScript testing framework, Jasmine. The Metric-core is the central library for Metrics that provides basic functionality. Jedis is a client library in Java for Redis. It is driven by a Keystore-based data structure for persistent data and can be used as a database, cache, message broker, etc. Commons-Lang is a package of Java utility classes for the classes that are in java.lang's hierarchy, or are considered to be so standard as to justify existence in java.lang. Commons-Math is a library of lightweight, self-contained mathematics and statistics components addressing the most common problems not available in the Java programming language or Commons-Lang. Junit-dataprovider is a TestNG like dataprovider runner for JUnit with many additional features.

It should be mentioned that the programs selected as TDD projects are not necessarily TDD. These open-source projects, however, have basic TDD specifications with test files besides the code files. Their evolution process can be accessed, and hence, they can be re-implemented as TDD projects.

**Table 6** Properties of selected GitHub Projects

| Project name | LOC | Number of versions (v) | Number of tests ($n$) |
|---|---|---|---|
| ScribeJava[a] | 10,668 | 1134 | 135 |
| Jasmin-Maven Plugin[b] | 3815 | 635 | 103 |
| Metric-Core[c] | 6734 | 240 | 54 |
| Jedis[d] | 30,786 | 1586 | 592 |
| Commons-Lang[e] | 88,775 | 6954 | 3949 |
| Commons-Math[f] | 260,760 | 7010 | 3073 |
| Junit-dataprovider[g] | 22,544 | 586 | 762 |

[a]https://github.com/scribejava/scribejava
[b]https://github.com/searls/jasmine-maven-plugin
[c]https://github.com/avaje-metrics/metrics
[d]https://github.com/xetorthio/jedis
[e]https://github.com/apache/commons-lang
[f]https://github.com/apache/commons-math
[g]https://github.com/TNG/junit-dataprovider

#### 4.12.2 Re-implementing GitHub projects

After finding the appropriate repository, we re-implement each project, step by step. For each repository, we first create an empty project and transfer the first commit of the repository to this project. Then we select the "Save" button. The RichTest performs segmentation and adds start and end comments and inserts block information in the related database. This is the first version of the project.

In the next steps, we have to complete the project incrementally according to the main branch and apply the changes in each commit. We apply test files changes and then we apply code files changes. Then we select the "Save" button again. From the second version onwards, not only automatic block segmentation but also block relationship creation is done and the related information is recorded in database. It is important that in each commit the changes related to the tests are applied first and then the changes related to the code are applied so that the connection between the test cases and the modified code blocks is correctly recognized and recorded. At last, we run the RichTest Regression Test Wizard to select related test cases. Then we store the number of RichTest selected test cases as well as the total number of test cases in two separate table to calculate the total number of the executed test case in each method.

We perform this process for all versions of all projects. The number of versions in each project is extremely high. As a result, it is relatively time-consuming to repeat the process for all versions. Only 100 versions are therefore considered in the first project, and overall, fewer versions are considered in other projects (29, 28, and 15 versions were re-implemented for projects Jasmin-Maven Plugin, Metric-Core, and Jedis, respectively).

Selected projects are not originally written with our plugin; hence, the first version of some projects that have more than one test case, was re-implemented

**Table 7** Number of considered versions for four open-source projects on GitHub

| Project | ScribeJava | Jasmin-Maven Plugin | Metric-Core | Jedis |
|---|---|---|---|---|
| First LOC | 1464 | 111 | 3027 | 220 |
| Last LOC | 2266 | 545 | 4067 | 830 |
| Number of desired versions ($v$) | 100 | 29 | 28 | 15 |
| $t_1$[a] | 51 | 4 | 4 | 10 |
| $t_2$[b] | 81 | 13 | 4 | 93 |

[a]Number of test cases in first version

[b]Number of test cases in our last desired version

manually to establish the connection between code blocks and test blocks. Block segmentation, however, is generally automatically implemented.

Table 7 summarizes the number of versions considered in each project and the number of lines of code (LOC) in the first and last considered versions, as well as the number of test cases in the first and last desired version.

After each modification, the new version is stored, and the Regression Test Wizard is executed. Candidate test cases that are relevant to the modified code blocks are provided by the plugin. The number of candidate test cases is thereafter considered to calculate the number of times the test cases are executed.

Table 8 lists and compares the number of candidate test cases executed in TDD and RichTest plugin for these four selected open-source projects on GitHub. The result indicates that the use of RichTest plugin significantly reduces the number of test case executions by minimizing the number of selected test cases at runtime.

As can be seen in Table 8, the two columns TDD and RichTest have significant differences in all projects. This difference is greater for the first project (ScribeJava). We re-implemented the first project up to the 100th version. As to the other projects, a smaller number of versions were re-implemented. So, the difference between the number of times of test executions of TDD and RichTest in ScribeJava is considerably larger compared to the other projects. This difference is due to the fact that the number of versions in this project is much higher than the others and RichTest ability is more evident in the high number of versions.

To evaluate the improved method, three small programs and four large open-source projects on GitHub are implemented in RichTest. The number of test case executions in the main TDD method and improved method are thereafter calculated and compared. As illustrated in Fig. 11 (obtained from Table 5 (page 16) and Table 8), the RichTest plugin (box crosshatched with *orange* and diagonal lines) significantly reduces the number of test case executions by reducing the number of selected test cases at runtime. This reduction would be more significant in large projects with a larger number of test cases (ScribeJava is an evident example).

**Table 8** Comparison of number of executed test cases in TDD and RichTest for four projects on GitHub

| Factor | Project | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| | ScribeJava | | Jasmin-Maven Plugin | | Metric-Core | | Jedis | |
| | TDD | RichTest | TDD | RichTest | TDD | RichTest | TDD | RichTest |
| Total number of executions (*sum*) | 6056 | 173 | 296 | 54 | 112 | 20 | 730 | 116 |
| Average= *sum*/*v* | 60.56 | 1.73 | 10.21 | 1.86 | 4 | 0.71 | 48.97 | 7.73 |
| Number of test cases (*n*) | (51–81)† | | (4–13) | | (4–4) | | (10–93) | |
| Number of versions (*v*) | 100 | | 29 | | 28 | | 15 | |

[a]Format (m−n) indicates that the start version of project has m test cases and the end version has n ones.

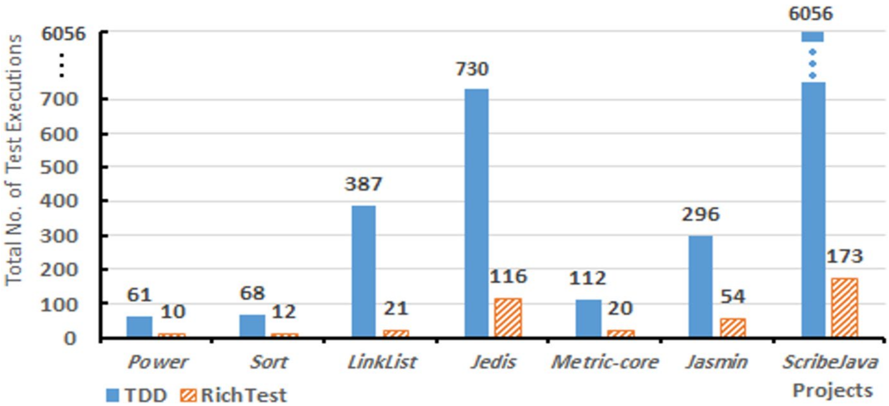


**Fig. 11** Total number of test case executions in TDD vs RichTest

In Fig. 12, the total number of test cases is divided by the number of versions to determine the average number of test cases per iteration. As shown in this figure, in RichTest, the average number is small in all cases but varies according to the number of test cases in the TDD. This figure confirms that the average number of candidate test cases in the improved method is small and is not related to the number of test cases.

The desired versions of ScribeJava are larger compared to the other projects. As illustrated in Fig. 13, the difference in the number of test execution times between the two methods (TDD and RichTest) in this project is more significant. This indicates that the advantages of this approach are more evident in large projects that have a longer production process and when the number of test cases is higher.

Figure 13 illustrates that the number of times that the test runs in RichTest (*orange* dashed line) completely overlaps with the number of test cases (*black* dotted line labeled as "***n***"). The number of times the test runs in the TDD (**blue** line), however, significantly differs from the number of test cases.

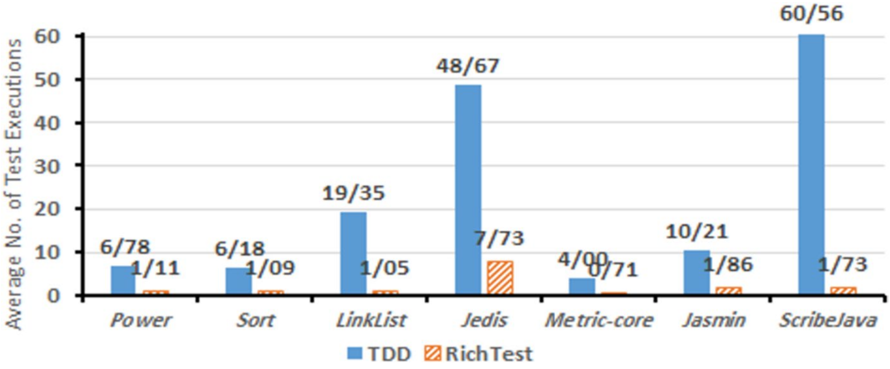


**Fig. 12** Average of test case execution for each iteration in TDD vs RichTest

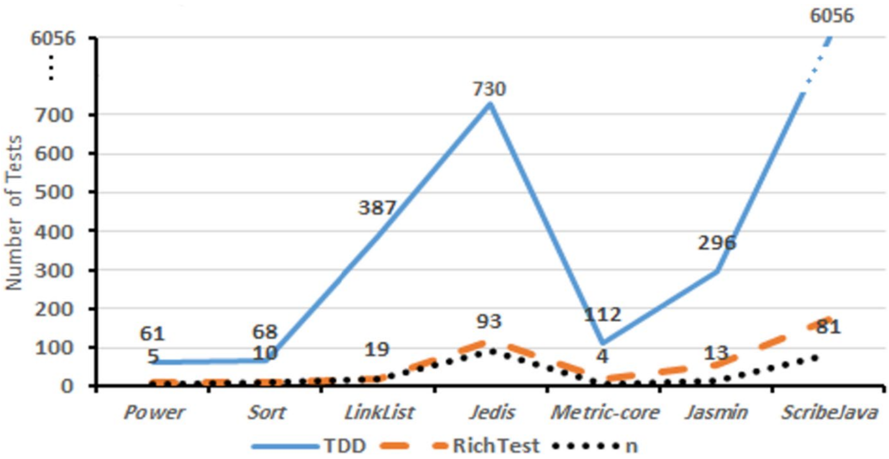


**Fig. 13** Comparison between TDD and RichTest with $n$

### 4.13 Evaluation results

Since some TDD developers' only re-run test cases related to the new class or the new unit, maybe this question arises why we didn't compare our work with it. So we decided to resume our work and compare our approach with such a simpler TDD we called STDD. Therefore considering that there is no standard dataset or projects to compare our method with others' methods, for the baseline we desired two methods, pure TDD and STDD. We did these reviews for five TDD projects on GitHub. The results were recorded in separate tables. The summation of run test cases was calculated. The number of run test cases in TDD, STDD, and RichTest for five projects on GitHub are represented in Table 9. Although the STDD works much better than TDD, our method still performs better than the STDD. Selected [%] columns (5th columns) showing the percentage of selected test case (RichTest) in the ratio

**Table 9** Comparison of the number of Run Test Cases in TDD, STDD, and RichTest for Five Projects on GitHub

| Project | *TDD* | *STDD* | *RichTest* | Selected (%) | Number of test cases | Number of versions |
|---|---|---|---|---|---|---|
| ScribeJava | 4485 | 257 | 157 | 3.5 | (51–66) | 69 |
| Jedis | 5432 | 1106 | 382 | 7.0 | (10–136) | 53 |
| Commons-Lang | 2766 | 418 | 186 | 6.7 | (3–141) | 23 |
| Commons-Math | 10106 | 2061 | 501 | 5.0 | (9–226) | 70 |
| Junit-dataprovider | 12312 | 4437 | 750 | 6.1 | (1–240) | 69 |
| Average | 7020 | 2069 | 395 | 5.6 | (15–162) | 57 |

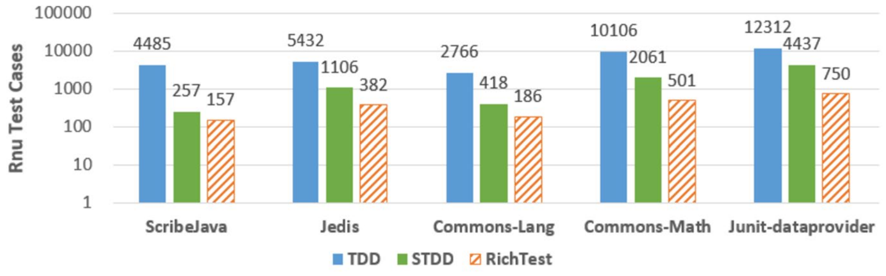


**Fig. 14** Comparison between run test cases in TDD, STDD, and RichTest for five GitHub projects represented in Table 9

of retest all (TDD). As can be seen it is on average 5.4%, minimum 3.5% and maximum 7% of retest all.

Figure 14 compares the total number of run test cases in three methods, TDD (**blue** box), STDD (green box), and RichTest (**orange** dashed box). As shown, RichTest conquers STDD as well as TDD. The logarithmic vertical axis represents that the number of run test cases has improved more than tenfold.

Due to the reduction in the number of run test cases in RichTest, the test execution time will also be reduced in this tool. But in order to accurately calculate the RT time for each project, it is necessary to calculate the overhead time due to the use of this tool and consider it in the calculation of the RT time.

Therefore, we made changes in the RichTest so that all the times related to doing the general tasks, segmentation, and creating connections between code and test blocks are calculated and stored in the project database. For four projects, we calculated and recorded the overhead time in RichTest, then we added these time to the RT time and compared the result with the RT time in the TDD and STDD methods. The final results are presented in Table 10.

The spent time in TDD, STDD, and RichTest for five Projects on GitHub are represented in Table 10. Time [%] columns (8th columns) is showing the percentage of RichTest time in the ratio of retest all (TDD). As can be seen, the average time of RichTest compared to retest all is on average 6.8%, minimum 3.9% and maximum 7.8%. The logarithmic vertical axis in Fig. 15 represents that the time has improved tenfold.

We also compared RichTest RT time (including overhead time) with STDD. It was found that they have slight differences with each other. If there are a few test cases written for each class, the number of selected test cases in both methods is

**Table 10** Comparison of the number of Run Test Cases and RT Time in TDD,STDD, and RichTest (*considering RichTest overhead time)* for Five GitHub Projects

| Project | Number of run test cases | | | RT time | | | | Number of test cases | Number of versions |
|---|---|---|---|---|---|---|---|---|---|
| | TDD | STDD | RichTest | TDD | STDD | RichTest | Time (%) | | |
| ScribeJava | 4485 | 257 | 157 | 12479 | 537 | 484 | 3.9 | (51–66) | 69 |
| Jedis | 5432 | 1106 | 382 | 13706 | 1031 | 1073 | 7.8 | (10–136) | 53 |
| Commons-Lang | 2766 | 418 | 186 | 485 | 64 | 78 | 16.2 | (3–141) | 23 |
| Commons-Math | 10106 | 2061 | 501 | 2517 | 132 | 222 | 8.8 | (9–226) | 70 |
| Junit-dataprovider | 12312 | 4437 | 750 | 42379 | 12821 | 2756 | 6.5 | (1–240) | 69 |
| Average | 7020 | 2069 | 395 | 14313 | 2917 | 923 | 8.6 | (15–162) | 57 |

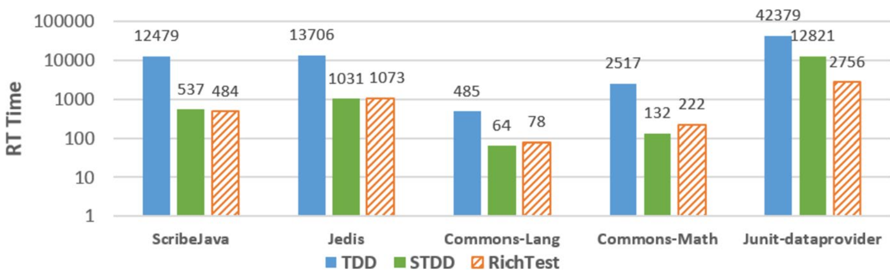


**Fig. 15** Comparison of the RT Time in TDD, STDD, and RichTest for Five GitHub Projects represented in Table 10

almost the same and as a result, STDD is slightly faster than RichTest. But if there are a lot of test cases, our tool selects only the related test cases and will perform better despite the overhead time. Also, results show that RichTest is suitable for large projects. Because in the early versions, the number of selected test cases and RT time does not differ much.

Coverage information is shown in Table 11, Figs. 16, and 17. We assumed TDD code block coverage to be 100% and compared it to STDD and RichTest. Also, we defined the modified code block coverage percentage criterion as the percentage of the selected test cases related to the modified code blocks. RichTest reached 100% coverage of the modified code block and STDD selected on average 61.67% of related test cases. Indirect test cases were not selected in STDD and STDD coverage is lower than RichTest; So RichTest is safer than STDD. TDD exceeded the over-test and we considered it 100% in Fig. 17.

### 4.14 Discussion

To compare our work with other similar plugins, we first decided to compare our work with the plugins listed in Table 1. So, we filtered Java plugins, which were eight, but we found that only JDiff (Apiwattanapong et al. 2007) used the program differencing for the regression testing, which lacked criteria comparable to the criteria of our work and the focus of the article is on finding the optimal modified blocks and has studied four basic issues (Apiwattanapong et al. 2007):

**Table 11** Comparison of the block coverage and modified block coverage in TDD, STDD, and RichTest for Five GitHub Projects

| Project | Block coverage (%) | | | Modified block coverage (%) | | | Number of test cases | Number of versions |
|---|---|---|---|---|---|---|---|---|
| | TDD | STDD | RichTest | TDD | STDD | RichTest | | |
| ScribeJava | 100 | 1.11 | 2.01 | 1296.68 | 92.83 | 100 | (51–66) | 69 |
| Jedis | 100 | 13.33 | 30.86 | 749.94 | 59.93 | 100 | (10–136) | 53 |
| Commons-Lang | 100 | 16.71 | 34.77 | 1321.09 | 66.28 | 100 | (3–141) | 23 |
| Commons-Math | 100 | 9.69 | 26.38 | 875.62 | 42.71 | 100 | (9–226) | 70 |
| Junit-dataprovider | 100 | 14.38 | 39.48 | 579.36 | 46.63 | 100 | (1–240) | 69 |
| Average | 100 | 11.04 | 18.12 | 964.53 | 61.67 | 100 | (15–162) | 57 |

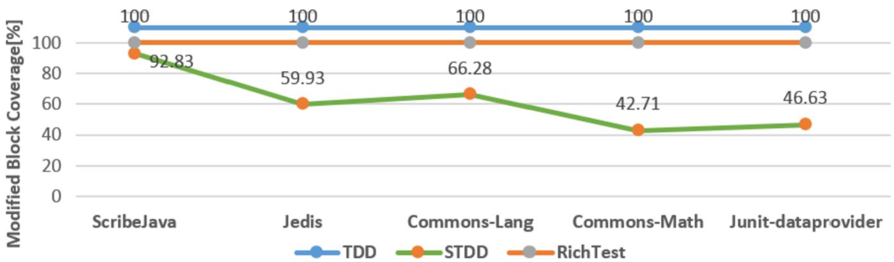


**Fig. 16** Comparison of the block coverage in TDD, STDD, and RichTest for Five GitHub Projects represented in Table 11

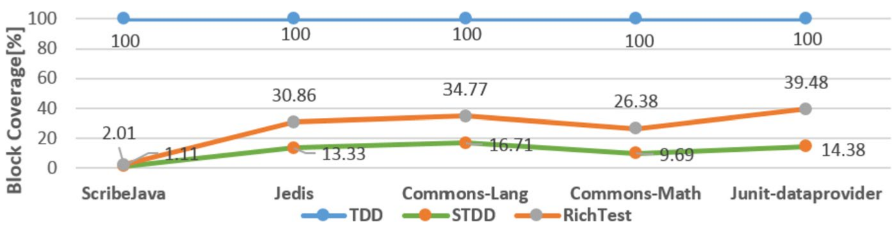


**Fig. 17** Comparison of the modified code block coverage in TDD, STDD, and RichTest for Five GitHub Projects represented in Table 11

1. Object-oriented changes: JDiff (Apiwattanapong et al. 2007) has shown that a large number of changes are object-oriented changes, which were not considered in traditional tools. Like JDiff, RichTest detects all changes, including object-oriented changes, and also identifies indirect changes by specifying parent and child blocks.
2. Optimization similarity threshold: In our article, considering that the name, the beginning and the end of each block are known, the matching block is simply tracked and does not have these parameters. RichTest also uses comparison of AST tree and JSON code to discover differences in similar blocks.
3. The number of matched nodes: The number of matched nodes in our tool is maximum (Same reasons as above).
4. Coverage estimation: In our article, Eclipse environment facilities are used for this purpose. We reached 100% modified block coverage.

**Table 12** Comparison of RichTest and STARTS for two common projects and average of all reviewed projects

| Project | Metric<br>Tool | | | |
|---|---|---|---|---|
| | Selected (%) | | Time (%) (include overhead) | |
| | STARTS (Legunsen et al. 2017) (%) | RichTest (%) | STARTS (Legunsen et al. 2017) (%) | RichTest (%) |
| Commons-Lang | 32 | 6.7 | 73.3 | 16.23 |
| Commons-Math | 28.9 | 5.0 | 30.3 | 8.8 |
| Average of all Reviewed Projects | 35.2 | 5.6 | 81 | 8.6 |

Therefore, we compared our work with the STARTS (Legunsen et al. 2017) which is also reviewed in framework checker (Zhu et al. 2019). STARTS is a Java plugin for RT, selecting the impacted test cases. Legunsen et al. (2017) examined several Java projects with the STARTS and provided three criteria (1) number of selected test cases, (2) the offline time, and (3) the online time (includes time for the a, e, and g phases) similar to our work. Their results show that the number of selected test cases is on average 35.2% of all test cases, the offline time is on average 81% of retest all, and also, the online time is on average 87.6%.

As shown in Table 9, the RichTest selects an average of 5.6% of the tests, while the STARTS selects an average of 35.2% of the tests. Also, as shown in Table 10 the RichTest whole time is on average 8.6% of retest all test cases while STARTS takes 81% time. Table 12 compares RichTest and STARTS tools for the two projects Commons-Math and Commons-Lang as well as for the average of all reviewed projects. The result shows that RichTest has made a great improvement both for the two projects under common comparison and on average in all projects. It seems that the reason for this improvement is the use of the nature of TDD in the test case selection. Therefore, it can be concluded that it is necessary to create special tools for testing TDD programs.

At last, by using Python's AutoRank[13] function (Herbold 2020), we compare the number of test cases and RT time between RichTest with TDD and STDD for four projects (other projects were not completely applicable). Final results are shown in Table 13, 14, 15 and 16.

Table 13 represents the comparison of the number of run test cases and Table 15 represents the RT time for two populations: TDD and Richtest. Table 14 represents the number of run test cases and Table 16 represents the RT time for two populations: RichTest and STDD. The result of AuroRank is provided for all the projects comparing two populations RichTest and TDD and also RichTest and STDD. Below are the results of comparing RichTest and STDD populations for Junit-dataProvider with 68 versions:

The statistical analysis was conducted for 2 populations with 68 paired samples.

The family-wise significance level of the tests is alpha = 0.050.

[13] result = autorank(data, alpha = 0.05, verbose = False).

We rejected the null hypothesis that the population is normal for the populations STDD time ($p = 0.000$) and RichTest time ($p = 0.000$). Therefore, we assume that not all populations are normal.

No check for homogeneity was required because we only have two populations.

Because we have only two populations and both of them are not normal, we use Wilcoxon's signed rank test to determine the differences in the central tendency and report the median (MD) and the median absolute deviation (MAD) for each population.

We reject the null hypothesis ($p = 0.000$) of Wilcoxon's signed rank test that population STDD time (MD = 15.095 ± 6.889, MAD = 7.954) is not greater than population RichTest time (MD = 15.329 ± 8.124, MAD = 8.590). Therefore, we assume that the median of STDD time is significantly larger than the median value of RichTest time with a negligible effect size (gamma = −0.019).

Considering that the initial versions of the projects are also taken into account, RichTest RT time is worse than STDD method, but the number of run test cases in all projects shows the superiority of RichTest. Also, RichTest RT time is better than TDD in every four projects. Magnitude fields shows that RichTest is negligible while TDD and STDD are large.

### 4.15 Research questions

This study focuses on the following five main questions:

**RQ1**: How many (complexity) test cases would be executed in the traditional and improved TDD process? The question is, if $n$ test cases are written during the TDD process, what is the complexity of the number of test cases that will be executed?

#### 4.15.1 Calculation of minimum number of test case execution

In test-driven development, all previous tests should be re-executed in each iteration to ensure that they will perform correctly under new conditions. Among the principal disadvantages of TDD is the necessity of having a large number of test cases that must be repeatedly executed.

For clarity, consider the following example. Suppose that n is the number of test cases, which have been written and passed one by one, during the program development. This means that every time a new test is added, all previous tests, including the first test, are run again. Therefore, the first test will be performed at least n times. The second, third, and nth tests are executed at least (n − 1) times, at least (n − 2) times, and at least once, respectively.

So we add these items to get the minimum number of times that the test cases will have to be executed. As can be seen, the sum is equivalent to the sum of an arithmetic sequence that we have calculated by the formula (9).

$$SUM(1 : n) = n + (n-1) + (n-2) + \cdots + 3 + 2 + 1 = 1/2\, n(n+1) \tag{9}$$

**Table 13** Summary of comparison of number of run test cases in RichTest and TDD (produced by AutoRank)

| Project | Sample | Meanrank | | Median | | Mad[a] | | ci_lower | | ci_upper | | effect_size | | Magnitude | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | RichTest | TDD | RichTest | TDD | RichTest | TDD | RichTest | TDD | RichTest | TDD | RichTest | TDD | RichTest | TDD |
| Junit-dataProvider | 68 | 1/99 | 1/01 | 4 | 201/5 | 3 | 72/5 | 3 | 105 | 8 | 255/00 | 0 | -2/60 | negligible | large |
| Commons-Math | 69 | 1/99 | 1/01 | 6 | 128 | 3 | 89 | 4 | 83 | 9 | 248 | 0 | -1/31 | negligible | large |
| Jedis | 52 | 1/99 | 1/01 | 3 | 111 | 1 | 15/5 | 2 | 100 | 8 | 127 | 0 | -6/63 | negligible | large |
| Commons-Lang | 23 | 1/98 | 1/02 | 2 | 138 | 1 | 3 | 1 | 117 | 20 | 140/00 | 0 | -41/02 | negligible | large |

[a]Median absolute deviation

**Table 14** Summary of comparison of number of run test cases in RichTest and STDD (produced by AutoRank)

| Project | sample | meanrank | | median | | mad | | ci_lower | | ci_upper | | effect_size | | magnitude | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | RichTest | STDD | RichTest | STDD | RichTest | STDD | RichTest | STDD | RichTest | STDD | RichTest | STDD | RichTest | STDD |
| Junit-dataProvider | 68 | 1/96 | 1/04 | 4/00 | 55/50 | 3/00 | 26/50 | 3/00 | 39/00 | 8/00 | 89/00 | 0 | -1/84 | negligible | large |
| Commons-Math | 69 | 1/96 | 1/04 | 6/00 | 24/00 | 3/00 | 12/00 | 4/00 | 15/00 | 9/00 | 37/00 | 0 | -1/39 | negligible | large |
| Jedis | 52 | 1/98 | 1/02 | 3/00 | 15/50 | 1/00 | 9/50 | 2/00 | 8/00 | 8/00 | 26/00 | 0 | -1/25 | negligible | large |
| Commons-Lang | 23 | 1/93 | 1/07 | 2/00 | 14/00 | 1/00 | 12/00 | 1/00 | 2/00 | 20/00 | 38/00 | 0 | -0/95 | negligible | large |

**Table 15** Summary of comparison of RT Time in RichTest and TDD (produced by AutoRank)

| Project | Sample | Meanrank | | Median | | Mad | | ci_lower | | ci_upper | | effect_size | | Magnitude | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | RichTest | TDD | RichTest | TDD | RichTest | TDD | RichTest | TDD | RichTest | TDD | RichTest | TDD | RichTest | TDD |
| Junit-dataProvider | 68 | 1/98 | 1/02 | 16/17 | 713/51 | 9/08 | 249/26 | 10/44 | 352/17 | 29/90 | 877/13 | 0 | -2/67 | negligible | Large |
| Commons-Math | 69 | 1/99 | 1/01 | 2/37 | 27/50 | 1/25 | 21/01 | 1/60 | 19/62 | 3/93 | 63/35 | 0 | -1/14 | negligible | Large |
| Jedis | 52 | 1/98 | 1/02 | 10/51 | 287/92 | 4/35 | 45/29 | 7/26 | 245/18 | 20/74 | 329/84 | 0 | -5/82 | negligible | Large |
| Commons-Lang | 23 | 1/91 | 1/09 | 0/94 | 24/05 | 0/48 | 0/56 | 0/53 | 20/78 | 11/90 | 24/47 | 0 | -29/82 | negligible | Large |

**Table 16** Summary of comparison of RT Time in RichTest and STDD (produced by AutoRank)

| Project | Sample | Meanrank | | Median | | Mad | | ci_lower | | ci_upper | | effect_size | | Magnitude | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | RichTest | STDD | RichTest | STDD | RichTest | STDD | RichTest | STDD | RichTest | STDD | RichTest | STDD | RichTest | STDD |
| Junit-dataPro-vider | 68 | 1/11 | 1/89 | 16/17 | 14/49 | 9/08 | 8/06 | 10/44 | 10/14 | 29/90 | 25/67 | -0/13 | 0 | negligible | Negligible |
| Commons-Math | 69 | 1/12 | 1/88 | 2/37 | 1/14 | 1/25 | 0/55 | 1/60 | 0/82 | 3/93 | 1/85 | 1/85 | 0 | large | Negligible |
| Jedis | 52 | 1/10 | 1/90 | 10/51 | 9/88 | 4/35 | 3/96 | 7/26 | 7/26 | 20/74 | 20/45 | -0/10 | 0 | negligible | Negligible |
| Commons-Lang | 23 | 1/02 | 1/98 | 0/94 | 0/37 | 0/48 | 0/18 | 0/53 | 0/19 | 11/90 | 3/28 | -1/06 | 0 | large | Negligible |

In formula (9) we supposed that the TDD development process starts with only one test, but some of the GitHub projects used in this research have more than one test case in the first version. So, we suppose that the initial number of test cases is *t1* (instead of one test), and the number of final tests is *t2*; hence, there are ***t1, t1+1***, and ***t2*** tests in the first, second, and last turns, respectively. The sum of the number of times the test cases are executed can be calculated by the formula (10).

$$SUM\left(t_1 : t_2\right) = 1/2\left(t_2 - t_1 + 1\right) * \left(t_1 + t_2\right) \tag{10}$$

As presented by formulae (9) and (10), the *minimum* number of times that the test cases are executed is calculated by the sum of an arithmetic sequence formula. So, the total number of test case executions is of $\boldsymbol{O(n^2)}$ complexity, where $n$ is the number of test cases. Actually, we have a quadratic complexity in traditional TDD, but in practice, we reach a linear complexity of executing test cases using RichTest, improved TDD (Fig. 13).

Considering that the number of test cases in the TDD is many times more than those in other methods, the relationship between the number of times that the tests will have to be executed and the second power of the number of test cases is one of the principal problems of TDD.

**RQ2**: How can we reduce the number of times that the test cases are executed without compromising the software reliability of TDD?

#### 4.15.2 Safe test case selection

The main problem in test case reduction methods is the lack of confidence that the reduced test suite can detect errors. If we can ensure that the selected test cases can detect all errors, then the method is safe as well as software quality and reliability are maintained.

For this purpose, we intend to delete only the insignificant test cases. Thus some of the test cases that are less important could be ignored execution in any interval. In this paper, we focus on the differences between the two versions of the program instead of focusing solely on its latest version. As presented in Sect. 2.2.3, the behaviors of unchanged components in the new and old versions of a program do not differ at runtime so, it is guaranteed that no retest of all cases is necessary, and testing the affected component only is sufficient. RichTest skips all the test cases related to the unaffected parts of the program in RT. All test cases related to the modified parts are considered, so we have 100% modified code coverage.

The main problem in test case reduction methods is the lack of confidence that the reduced test suite can detect errors. If we can ensure that the selected test cases can detect all errors, then the method is safe as well as software quality and reliability are maintained.

Rothermel believed that under controlled RT, the modification-traversing tests are a superset of the fault-revealing tests (Rothermel and Harrold 1997). Thus an algorithm that selects every modification-traversing test is also safe.

It should be mentioned that test $t \in T$ is modification-traversing for program P and modified program P' if and only if ET (P(t)) and ET (P′(t)) are nonequivalent.

Execution trace ET (P(t)) for test *t* on program P, consisting of the sequence of statements in P that are executed when P is executed with t.

It should be mentioned that ET (P(t)) is the execution trace for test *t* on program P, consisting of the sequence of statements in P that are executed when P is executed with test *t*. Also, Test $t \in T$ is modification-traversing for program P and its modified program P′ if and only if the execution traces of them are nonequivalent (ET (P(t)) $\neq$ ET (P′ (t))) (Rothermel and Harrold 1997).

What happens in our algorithm? Is it safe or not?

We "link" all the modified code blocks in each step to associated test cases. That is, when program P becomes P′, ET (P(t)) is different from ET (P′ (t)). So we select all modification-traversing test cases.

When a test failure causes code modification, all modified code blocks are then connected to the test. Then all the tests related to these modified blocks are selected. Due to the code change in the block, for all of these selected tests, the sequence of executed instructions will be different at the time of running the test, i.e. ET (P(t)) $\neq$ ET (P′(t)): These tests are all modification-traversing, and because they are a superset of the fault-revealing test suite, the algorithm is safe.

**RQ3**: How can the TDD method aid in selecting test cases?

#### 4.15.3 TDD based test case selection

For the TDD method, the test case is written first, and thereafter the code is written to pass the test; a close relationship between the test case and source code is established. The question is whether the test cases can be selected based on the TDD *nature*.

We used the nature of TDD to model the relationship between test blocks and code blocks as shown in Fig. 18. In the TDD method, each requirement leads to writing a set of test cases. Each test case also leads to creation or modification of source code. So there are some relations between the test cases and modified parts of the code. That's why we use the code segmentation algorithm and save the relationship between test and code blocks.

It should be mentioned that our selection algorithm is based on TDD nature and assumes that developers follow the TDD cycle. It may fail if the developer does not follow the TDD cycle, so another question arises.

**RQ4**: What is the impact of human behavior on this approach?

#### 4.15.4 The impact of human behavior

Our proposed approach assumes that developers always follow the TDD cycle. However, in reality, the order of this cycle is not always observed (Beller et al. 2017). What is the impact of such a human behavior?

We assumed that the developer would not write any code except for passing the test or refactoring the code. Therefore, we connect all modified code blocks to the last test case. If the developer writes the code before writing the test case, RichTest assumes the changes are made to refactor the code. So RichTest connects these changes to the last test case.

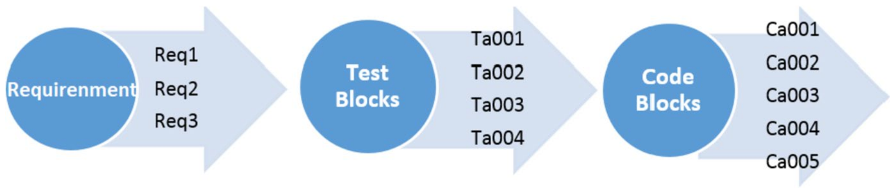


**Fig. 18** Relationship between requirement, test, and code

Although ignoring the refactoring phase is not a problem, late refactoring may cause an unrealistic relationship between the previous code and the new test.

It is important to consider three questions. The first is whether the modified parts of the code are covered 100% or not. Fortunately, the answer is yes, because the modified parts will be connected to the last test case, and the coverage of the modified code is achieved.

The second question is whether the test cases will be selected correctly in the next steps. Unfortunately, the answer is "no". The test case that is mistakenly assigned to the code block may be selected and added to the test suite. The suggested solution is that the developer disconnects the wrong relations manually. This is possible through the BlockInfoView to uncheck and remove the test case relation.

The third question is whether the test suite is complete. Unfortunately, the answer is "no". Since the developer has not written any test case before modification, the test suite is not complete. The only suggested solution is that the developer connects the lost relations manually. This is possible through the BlockInfoView.

**RQ5**: To what extent it is possible to select test cases (semi-)automatically?

#### 4.15.5 Automatic test case selection

One of the main questions of the research is whether an effective model and tool can be considered to select test cases. Can a set of rules and steps that can be automated or semi-automated be defined to perform the task of the test case selection?

As explained in the previous question, RichTest is implemented based on the nature of TDD. In this way, if the new test fails, the programmer will apply enough changes to the code blocks, until passing the new test. Then, RichTest automatically links all the modified code blocks to the new test. Therefore, all test cases involved in creating or modifying any code block are linked to it.

During the development process, whenever each code block changes, RichTest selects all the test cases associated with that code block as a candidate test case. Therefore, any test case that was involved in creating or modifying the code will be selected automatically. Also, the programmer could link/unlink a test block to a code block manually.

## 5 Summary and conclusion

### 5.1 Summary

In this research, the problem of excessive numbers of test cases developed in the TDD and the repetitive execution of test cases are investigated. The results indicate that the complexity of test case execution correlates with the second power of the number of test cases.

The differences between the two program versions while ignoring test cases related to unmodified parts are identified, and some insights to reduce test cases and RT execution time in the TDD are suggested. A combinational difference identification algorithm based on textual and syntactical differencing is thereafter presented to accomplish these tasks. The proposed method to reduce test cases, particularly for the TDD method is presented. Program differencing is not a new approach to test selection, but the innovative aspect of our work is "how" to do it. We select test cases using the nature of TDD. For this purpose, we developed the RichTest tool. Whenever a copy of the program is saved, RichTest considers this version as a commit of the program and automatically monitors new test cases and program differences as new test blocks and modified/ new code blocks, then establishes the relationship between test cases and code blocks, automatically. TDD-based RT selection performs using these connections.

The RichTest plugin is employed to improve and simplify the implementation of TDD projects by reducing the number of run test cases and also reducing the RT execution time. The RT is executed by selecting only the test cases related to modified code blocks.

To evaluate the improved method, three small programs and six large open-source projects on GitHub are implemented in RichTest. The results show the RichTest plugin significantly reduces the number of test case executions by reducing the number of selected test cases at runtime (compared to both TDD and STDD). This reduction would be more significant in large projects with a larger number of test cases. Also, the number of times that the test cases runs in RichTest completely overlaps with the number of test cases. Although we have a quadratic complexity in traditional TDD, but in practice, we reach a linear complexity of executing test cases using RichTest, improved TDD.

The results showed that in the first version of each project, the number of test cases in pure TDD, STDD and RichTest is equal to the total number of test cases, so in the first version the RichTest method has the longest execution time due to the overhead time; but gradually by reducing the number of selected test cases in the next versions, this overhead time will be compensated and the total execution time will be reduced. RichTest RT time (including overhead time) is one tenth of TDD RT time. It was found that RichTest and STDD RT time have slight differences with each other. If there are a few test cases written for each class, the number of selected test cases in both methods is almost the same and as a result, STDD is slightly faster than RichTest. But if there are a lot of test cases, RichTest selects only the related test cases and will perform better despite the overhead time.

### 5.2 Restrictions

RichTest is not a commercial tool and is only the result of student research, so it is not free of problems and limitations. Its limitations are presented below.

1. Our block segmentation algorithm is based on Java programming language grammar, so RichTest limits projects to Java language. Also only some of control flow instructions such as switch, while, if, for, foreach, and try are considered.
2. Our plugin is developed in Eclipse IDE Photon June 2018, so RichTest limits to this development environment and Junit4 Tests.
3. Our method supports only TDD projects that follow the TDD cycle, otherwise, the developer must manually (dis)connect the code blocks (from) to the related tests. Human behavior is explained in Sect. 5.4.4 in more detail.
4. Our plugin doesn't execute test cases properly on the maven projects and gradle projects. Sometimes test case execution encountered a problem and we had to write another program to run the test cases.
5. This plugin is not recommended for projects that have many interfaces, because the number of selected test cases will not decrease significantly. Interface modification propagates to all of its implementations, so all tests related to all those codes should be selected for re-execution.
6. The RichTest tool uses commenting to track each block. For example, the beginning and the end of the first code block are defined by inserting two comments: //Start Of Code-Block: ca001, and //End Of Code-Block: ca001, respectively. RichTest needs these comments to trace code blocks, so in the refactoring phase, it is necessary to keep the comments in place so that the connections between the previous code blocks and related test cases are retained. Removing these comments will disrupt the test case selection process.

### 5.3 Future work

We are going to upgrade the plugin to resolve some of the restrictions, provide execution time reports, and keep the test result history. We will use these reports to prioritize tests. For example, the test face more failures will have a higher priority. We use this result to combine our test selection algorithm with regression rest prioritization. Also, we should investigate another real project as well as start a real project implementation in a laboratory.

## Appendix A

Figure 19 illustrates the Preferences window of RichTest. The developers could set block selection mode and the level of block granularity as well as enable TDD mode there.

Figure 20 until 26 represent RichTest views. These figures are related to selectionSort project. Figure 20 is BlockInfo View and illustrates that test case "ta009"

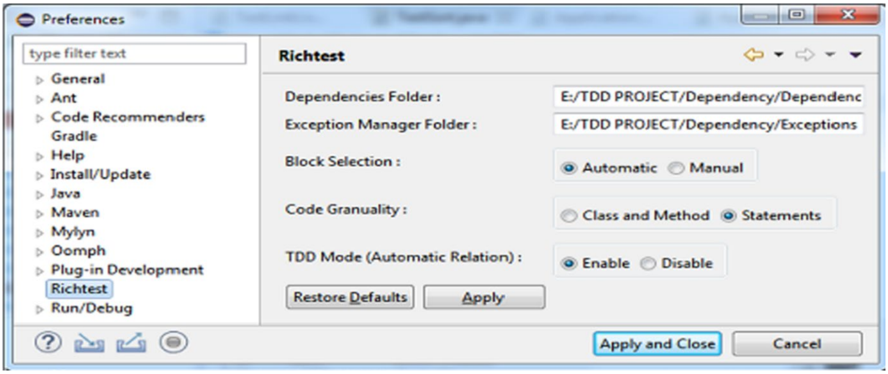


**Fig. 19** Preferences window

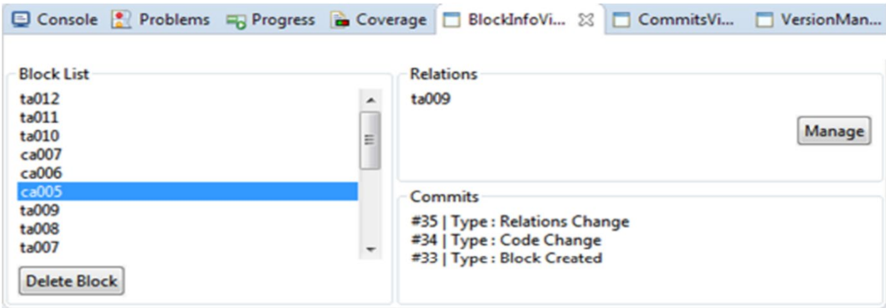


**Fig. 20** Block info view

is the only test block that is connected to the "Ca005" code block. The "Manage" key lets the developers manage the relation manually as seen in Fig. 21. As seen in this figure, the developer can link/unlink each test block to the "Ca005" code block manually.

Figure 22 illustrates the Commit View. The developers can see the type of modification and block content. Figure 23 illustrates the Version Manager View that shows all versions of the project as well as creates a new version. Figure 24 illustrates Regression Test View. Two test cases, 'Ta011' and 'Ta012' have been selected and run successfully (green color, means pass and red color means fail), required time and test results have been shown.

Figure 25 shows Compare View. The developer selects the desired code block (Ca003) to see the history of its changes and then selects two commits for comparison (Commit #21 and Commit #42). Figure 26 shows the comparison result. The orang box shows the changed part from Commit #21 to Commit #42.

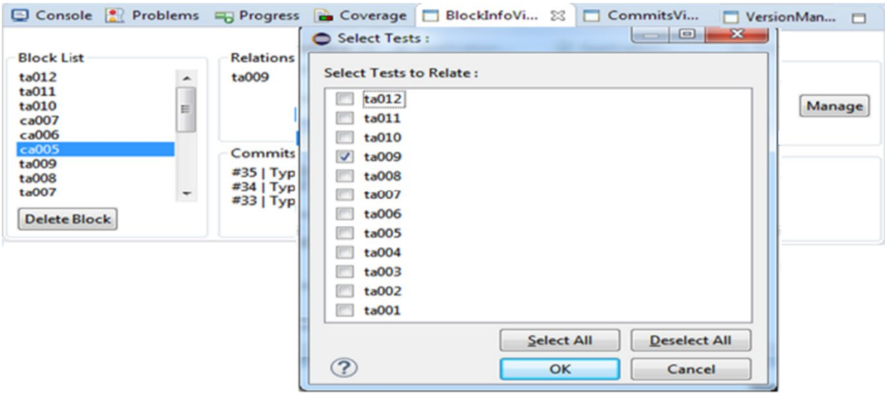


**Fig. 21** Manual Management Window of BlockInfo View

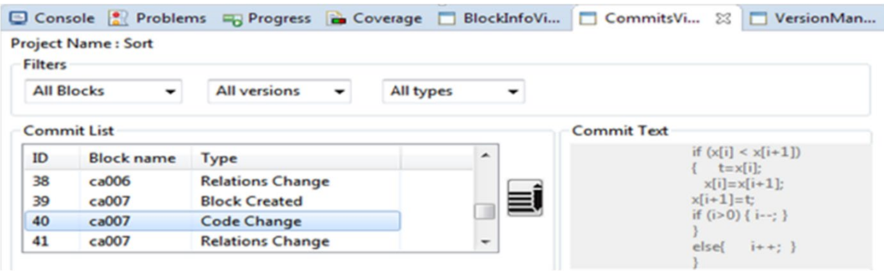


**Fig. 22** Commit view

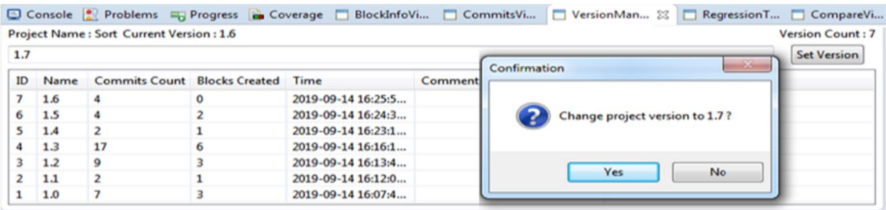


**Fig. 23** Version manager view

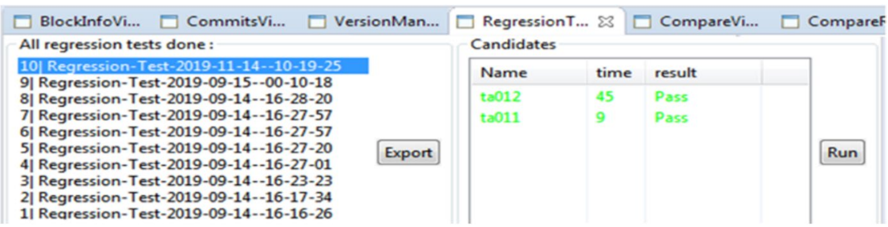


**Fig. 24** Regression test view

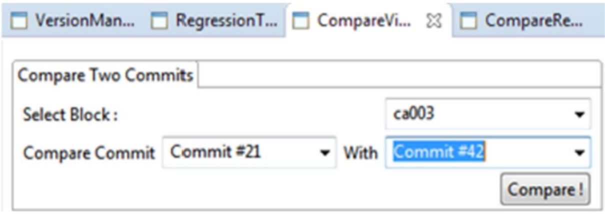


**Fig. 25** Compare view

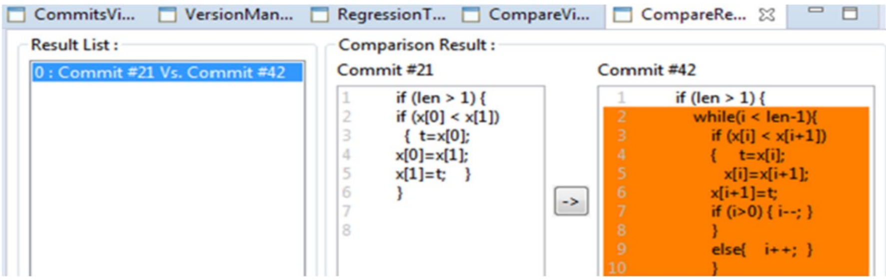


**Fig. 26** Compare view result

**Author contributions** Dear sir, Hi. We have previously submitted an article titled "RichTest: An Improved Test-Driven Development Plugin" to your journal, which was rejected. I have made some corrections to it now, which I hope will attract the opinion of the respected reviewers. Is it possible to resubmit to this journal? Manuscript number: AUSE-D-20-00114 Initial Date Submitted: 14 Oct 2020 Zohreh Mafi and Seyed-Hassan Mirian-HosseinAbadi, both developed programs. Mafi wrote the main manuscript text and Mirian-HosseinAbadi Edited the text.

## Declarations

**Conflict of interests** The authors declare no competing interests.

## References

Ammann, P., Offutt, J.: Introduction to software testing. Cambridge University Press, Cambridge (2008)
Apiwattanapong, T., Orso, A., Harrold, M.J.: JDiff: a differencing technique and tool for object-oriented programs. Autom. Softw. Eng.. Softw. Eng. **14**(1), 3–36 (2007)
Archambault, D.: Structural differences between two graphs through hierarchies. In: Proceedings of Graphics Interface, Kelowna (2009)
Asaduzzaman, M., Roy, C., Schneider, K., Di Penta, M.: LHDiff: a language-independent hybrid approach for tracking source code lines. In: IEEE International Conference on Software Maintenance, Eindhoven (2013)
Astels, D.: Test Driven Development: A Practical Guide. Prentice-Hall/Pearson Education, New Jersey (2003)
Beck, K.: Test Driven Development: By Example. Addison-Wesley, Boston (2002)
Beller, M., Georgios, G., Annibale, P.: Developer testing in the ide: patterns, beliefs, and behavior. IEEE Trans. Softw. Eng.softw. Eng. **45**(3), 261–284 (2017)

Beningo, J.: Testing, verification, and test-driven development. In: Embedded Software Design: A Practical Approach to Architecture, Processes, and Coding Techniques, pp. 197–218. Apress, Berkeley, CA (2022)

Binkley, D.: Using semantic differencing to reduce the cost of regression testing. In: IEEE Conference on Software Maintenance, Orlando (1992)

Biswas, S., Mall, R., Satpathy, M., Sukumaran, S.: Regression test selection techniques: a survey. Informatica **35**(3), 289–321 (2011)

Borle, N.C., Feghhi, M., Stroulia, E., Greiner, R., Hindle, A.: Analyzing the effects of test driven development in GitHub. Empir. Softw. Eng.. Softw. Eng. **23**(4), 1931–1958 (2018)

Canfora, G., Cerulo, L., Penta, M.D.: LDiff: an enhanced line differencing tool. In: 31st International Conference on Software Engineering. IEEE Computer Society, Washington (2009)

Cibulski, H., Amiram, Y.: Regression test selection techniques for test-driven development. In: IEEE Fourth International Conference on Software Testing, Verification and Validation Workshops, Washington (2011)

Dalton, J.: Test-driven development. In: Great Big Agile, pp. 263–264. Apress, Berkeley (2019)

Debin, G., Reiter, M.K., Song, D.: Binhunt: automatically finding semantic differences in binary programs. In: International Conference on Information and Communications Security. Springer, Berlin (2008)

Erdogmus, H., Maurizio, M., Marco, T.: On the effectiveness of the test-first approach to programming. IEEE Trans. Softw. Eng.softw. Eng. **31**(3), 226–237 (2005)

Falleri, J., Floréal, M., Xavier, B., Matias, M., Martin, M.: Fine-grained and accurate source code differencing. In: 29th ACM/IEEE International Conference on Automated Software Engineering. Västerås (2014)

Fowler, M., Beck, K., Brant, J., Opdyke, W., Roberts, D.: In: Gamma, E. (ed.) Refactoring: improving the design of existing code. Pearson Education India, Karnataka (1999)

George, B., Williams, L.: A structured experiment of test-driven development. Inf. Softw. Technol.softw. Technol. **46**(5), 337–342 (2004)

Görg, M., Zhao, J.: Identifying semantic differences in AspectJ programs. In: 18th International Symposium on Software Testing and Analysis (ACM), Chicago (2009)

Goto, A., Yoshida, N., Ioka, M., Choi, E., Inoue, K.: How to extract differences from similar programs? A cohesion metric approach. In: 7th International Workshop on Software Clones (IEEE Press), San Francisco (2013)

Herbold, S.: Autorank: a python package for automated ranking of classifiers. J. Open Sour. Softw. **5**(48), 2173 (2020)

Horwitz, S.: Identifying the semantic and textual differences between two versions of a program. ACM Sigplan **25**(6), 234–245 (1990)

Hsu, R.: Method for detecting differences between graphical programs. U.S. Patent 5,974,254, 26 Oct 1999

Karac, I., Turhan, B.: What do we (really) know about test-driven development? IEEE Softw.softw. **35**(4), 81–85 (2018)

Khanam, Z., Mohammed, N.A.: Evaluating the effectiveness of test driven development: advantages and pitfalls. Int. J. Appl. Eng. Res. **12**(18), 7705–7716 (2017)

Kim, M., David, N.: Discovering and representing systematic code changes. In: IEEE 31st International Conference on Software Engineering, Washington (2009)

Legunsen, O., August, S., Darko, M.: STARTS: STAtic regression test selection. In: 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE) (2017)

Linares-Vásquez, M., Cortés-Coy, L., Aponte, J., Poshyvanyk, D.: Changescribe: a tool for automatically generating commit messages. In: 37th IEEE International Conference on Software Engineering, Florence (2015)

Liu, C., Chen, C., Han, J., Yu, P.S.: GPLAG: detection of software plagiarism by program dependence graph analysis. In: 12th ACM SIGKDD international conference on Knowledge discovery and data mining. New York (2006)

Madeyski, L., Marcin, K.: Continuous test-driven development—a novel agile software development practice and supporting tool. In: Evaluation of Novel Approaches to Software Engineering (ENASE), pp. 260–267

Madeyski, L., Kawalerowicz, M.: Continuous test-driven development: a preliminary empirical evaluation using agile experimentation in industrial settings. Towards Synerg. Combin. Res. Pract. Softw. Eng. **733**, 105–118 (2018)

Maletic, J.I., Collard, M.L.: Supporting source code difference analysis. In: 20th IEEE International Conference on Software Maintenance, Chicago (2004)
Myers, E.W.: AnO (ND) difference algorithm and its variations. Algorithmica **1**(1–4), 251–266 (1986)
Neamtiu, I., Foster, J.S., Hicks, M.: Understanding source code evolution using abstract Syntax Tree Matching. Missouri (2005)
Nguyen, H.A., Nguyen, T.T., Nguyen, H.V., Nguyen, T.N.: iDIFF: interaction-based program differencing tool. In: 26th IEEE/ACM International Conference on Automated Software Engineering, Washington (2011)
Nooraei Abadeh, M., Mirian-Hosseinabadi, S.: Delta-based regression testing: a formal framework towards model-driven regression testing. J. Softw. Evol. Process **27**(12), 913–952 (2015)
Riebisch, M., Farooq, Q., Lehnert, S.: Model-based regression testing: process, challenges and approaches. In: Emerging Technologies for the Evolution and Maintenance of Software Models, Ilmenau, Germany, pp. 254–297. IGI Global (2012)
Rosero, R.H., Gómez, O.S., Rodríguez, G.: 15 Years of software regression testing techniques—a survey. Int. J. Softw. Eng. Knowl. Eng.softw. Eng. Knowl. Eng. **26**(05), 675–689 (2016)
Rothermel, G., Harrold, M.J.: A safe, efficient regression test selection technique. ACM Trans. Softw. Eng. Methodol. (TOSEM) **6**(2), 173–210 (1997)
Rothermel, G., Mary, J.H.: Empirical studies of a safe regression test selection technique. IEEE Trans. Softw. Eng.softw. Eng. **24**(6), 401–419 (1998)
Santosh Singh, R., Kumar, S.: An approach for the prediction of number of software faults based on the dynamic selection of learning techniques. IEEE Trans. Reliab.reliab. **68**(1), 216–236 (2018)
Shen, J., Sun, X., Li, B., Yang, H., Hu, J.: On automatic summarization of what and why information in source code changes. In: 40th Annual Computer Software and Applications Conference (COMPSAC), Atlanta (2016)
Vokolos, F.I., Frankl, P.: Empirical evaluation of the textual differencing regression testing technique. In: IEEE International Conference on Software Maintenance (Cat. No. 98CB36272), Bethesda (1998)
Wang, T., Wang, K., Su, X., Ma, P.: Detection of semantically similar code. Front. Comput. Sci. **8**(6), 996–1011 (2014)
Wang, X., Pollock, L., Vijay-Shanker, K.: Automatic segmentation of method code into meaningful blocks to improve readability. In: 18th Working Conference on Reverse Engineering IEEE, Limerick (2011)
Wolfgang, O: "TDD Kata," 9 12 2018. [Online]. Available: https://www.programmingwithwolfgang.com/tdd-kata/. Accessed 8 May 2021
Yang, W.: Identifying syntactic differences between two programs. Softw. Pract. Exp. **21**(7), 739–755 (1991)
Yoo, S., Mark, H.: Regression testing minimization, selection and prioritization: a survey. Softw. Test. Verific. Reliab. **22**(2), 67–120 (2012)
Zhang, L., Hao, D., Zhang, L., Rothermel, G.: Bridging the gap between the total and additional test-case prioritization strategies. In: Proceedings of the 2013 International Conference on Software Engineering, San Francisco (2013)
Zhu, C., Legunsen, O., Shi, A., Gligoric, M.: A framework for checking regression test selection tools. In: IEEE/ACM 41st International Conference on Software Engineering (ICSE), Montreal, QC, Canada (2019)