INTERNATIONAL JOURNAL OF COMPUTER SCIENCE & INFORMATION SYSTEM

Volume10 Issue10, October-2025, pg. 1-17

**Published Date: -** 01-10-2025

**E-ISSN: 2536-7919**
**P-ISSN: 2536-7900**
**SJIF 2019: 4.58 2020: 5.046 2021: 5.328**

# Integrating Defect Prediction to Guide Search-Based Software Testing: A Comprehensive Empirical Investigation

**Dr. Larian D. Venorth**

Department of Software Engineering, Zurich Technical University, Zurich, Switzerland

**Abstract: Background: The increasing complexity of software systems necessitates robust and efficient testing methods. While Search-Based Software Testing (SBST) has emerged as a powerful technique for automated test case generation, its effectiveness can be limited by its singular focus on code coverage. The generated tests, although structurally sound, may not target the most fault-prone areas of the code.**

**Aim: This study aims to address this limitation by proposing and empirically investigating a novel approach that integrates defect prediction (DP) models to guide the search process of SBST. By leveraging insights from historical code data, our method prioritizes the generation of test cases for code modules identified as having a higher likelihood of containing defects.**

**Method: We conducted a large-scale empirical study using 20 real-world, open-source Java projects from the Defects4J database. We developed a machine learning-based defect prediction model to identify fault-prone files. We then implemented a new fitness function for the EvoSuite test generation tool that incorporates the prediction score. The performance of this defect prediction-guided SBST approach was compared against a traditional, coverage-based SBST approach, using metrics of fault detection effectiveness and computational efficiency.**

**Results: Our findings indicate that the proposed DP-guided SBST approach significantly outperforms the traditional method in terms of the number of unique faults detected. Statistical analysis revealed a strong positive effect size for our approach. While there was a slight increase in computational overhead associated with the defect prediction component, it was minimal relative to the substantial gain in fault detection.**

**Conclusion: The results demonstrate that integrating defect prediction into the search-based test generation process is a highly effective strategy for improving the overall quality and fault-finding**

**Published Date: -** 01-10-2025

**E-ISSN: 2536-7919**
**P-ISSN: 2536-7900**
**SJIF 2019: 4.58 2020: 5.046 2021: 5.328**

**capability of automated testing. This approach represents a promising direction for enhancing software testing practices, particularly in continuous integration environments.**

**Keywords: Search-Based Software Testing (SBST), Defect Prediction, Automated Test Case Generation, Software Quality, Empirical Software Engineering, Meta-heuristic Optimization, Continuous Integration.**

## INTRODUCTION

### 1.1 Background: The Challenge of Software Testing

Software development is an intricate process, with the quality and reliability of the final product hinging critically on the effectiveness of its testing phase. As software systems grow in complexity and scale, the traditional methods of manual test case generation have become increasingly labor-intensive, time-consuming, and prone to human error. A central challenge in this domain is the test oracle problem, which refers to the difficulty of determining whether a program’s output is correct for a given input [34, 35]. Beyond this, the creation of test cases that can effectively reveal hidden faults is a formidable task, often requiring deep domain knowledge and an understanding of the software's internal structure. The resource-intensive nature of manual testing has driven significant research into automated solutions that can accelerate the process, reduce costs, and, most importantly, enhance the fault-detection capability of test suites.

### 1.2 Search-Based Software Testing (SBST)

One of the most promising and widely adopted automated approaches is Search-Based Software Testing (SBST) [28, 27]. SBST reframes the problem of test case generation as a meta-heuristic optimization problem. The core principle involves using algorithms such as genetic algorithms to search for test data that optimizes a predefined fitness function. A common and well-researched objective for this fitness function is to achieve high code coverage, with goals often including covering specific branches, statements, or paths within the source code [2, 3]. By evolving a population of test cases, SBST can efficiently explore the vast input domain of a program, generating test suites that systematically exercise different parts of the code.

Early work in this area demonstrated the feasibility and power of evolutionary algorithms for test data generation [27]. Subsequent research has refined these techniques, with tools like EvoSuite becoming state-of-the-art for generating whole test suites [1, 36, 49, 50]. The effectiveness of these tools has been validated in numerous empirical studies and even in industrial settings [6, 26]. However, a key limitation of traditional SBST is its primary focus on structural coverage. While achieving high coverage is a necessary

**Published Date: -** 01-10-2025

**E-ISSN: 2536-7919**
**P-ISSN: 2536-7900**
**SJIF 2019: 4.58 2020: 5.046 2021: 5.328**

condition for a thorough test suite, it does not guarantee that the generated tests will expose actual faults [5]. A test case might traverse a branch but fail to reveal a lurking bug because the input values or conditions are not sufficient to trigger a failure [7]. This gap highlights the need for a more targeted and intelligent approach to guide the search process.

### 1.3 The Role of Defect Prediction

Complementary to SBST is the field of defect prediction (DP). Defect prediction models use historical data from software repositories to identify code modules that are likely to contain a higher number of defects [10, 14, 15]. These models analyze various metrics, including static code attributes (e.g., complexity, size) and dynamic factors such as code churn, commit history, and organizational structure [11, 12, 16, 17,ure 18, 19]. The core hypothesis is that certain characteristics of a code module—for instance, high complexity or a history of frequent changes—are reliable indicators of its proneness to future defects.

Extensive research has demonstrated the effectiveness of defect prediction in various contexts. Studies at major technology companies have shown that these models can accurately predict fault-prone components, which can then be prioritized for code reviews, inspections, or more intensive testing [20, 21]. While the models' predictive power varies depending on the context and data, their ability to provide a probabilistic risk score for a given code module is a valuable asset [13, 22, 25]. This is particularly true in large software projects where resources are limited and cannot be uniformly applied to all components.

### 1.4 Synergizing SBST and Defect Prediction

The limitations of coverage-based SBST and the predictive power of defect prediction models suggest a natural synergy. Instead of solely seeking to maximize code coverage, SBST could be guided by defect prediction to prioritize test generation for code regions that are deemed most likely to be defective. This integrated approach, first explored in preliminary studies [8, 23, 24], holds the promise of combining the systematic exploration of SBST with the targeted intelligence of DP. The goal is to evolve test suites that not only achieve high coverage but are also more effective at finding real faults by focusing on high-risk areas.

The existing body of work on this topic, while promising, has been limited in scope. Previous studies have often focused on a small number of projects or have used simplified models, leaving a significant gap in understanding the true effectiveness, efficiency, and generalizability of this combined approach. A comprehensive, large-scale empirical investigation is needed to validate the practical benefits of this synergy and to provide clear guidance for its implementation in industrial settings.

### 1.5 Research Questions and Contributions

This paper addresses the identified research gap through a comprehensive empirical investigation. Specifically, we seek to answer the following research questions:

● RQ1: Does defect prediction-guided SBST generate test suites that are more effective at detecting faults compared to traditional, coverage-based SBST?

● RQ2: What is the computational overhead of integrating defect prediction into the SBST process?

● RQ3: How do the characteristics of the subject program (e.g., size, complexity, fault density) influence the relative performance of the two approaches?

Our primary contributions are:

1. A Novel Framework: We propose and implement a comprehensive framework for integrating defect prediction into the SBST fitness function to direct test generation towards fault-prone code.

2. Large-Scale Empirical Study: We conduct a large-scale empirical study on a diverse set of real-world software projects from the Defects4J database, providing a robust and generalizable evaluation.

3. Detailed Performance Analysis: We provide a detailed analysis of the fault detection effectiveness and computational efficiency of our proposed approach, including a discussion of its practical implications for continuous integration environments.

## METHODS

### 2.1 Experimental Setup and Dataset

To ensure the generalizability and replicability of our findings, we conducted our experiments using the Defects4J dataset [29, 43]. This widely-used benchmark provides a curated collection of real, reproducible faults from open-source Java projects, along with the correct and buggy versions of the source code. For this study, we selected 20 projects from the database, chosen for their diversity in size, domain, and development history. The selected projects spanned various applications, including a compiler, a web framework, and a data visualization library, providing a rich and varied testbed. All experiments were performed on a high-performance computing cluster with standardized hardware to ensure consistent and comparable results.

### 2.2 Defect Prediction Model

The first step in our methodology was to develop a robust defect prediction model for each project. Our model was built on a set of well-established static and change-related metrics [16]. These metrics were extracted from the source code and its version control history at the file level. The selected metrics included:

● Static Code Metrics: Lines of code, cyclomatic complexity, number of methods, and number of variables.

● Change-Related Metrics: Number of commits, number of authors, and number of lines added/deleted. We also included "change bursts," which are periods of high-frequency changes, as these have been shown to be strong indicators of defect-prone code [19].

For each file in a project, these metrics were calculated, and the file was labeled as "defect-prone" or "not defect-prone" based on whether it was associated with at least one known bug in the Defects4J database. We used a Random Forest classifier to build the prediction model, as this algorithm has demonstrated strong performance in similar software engineering contexts. A 10-fold cross-validation approach was used to train and validate the model's performance for each project, ensuring that the model's predictions were not based on the test set.

### 2.3 Search-Based Test Generation Approaches

We compared two distinct approaches to automated test generation: a baseline and our proposed method. All experiments were performed using EvoSuite [49, 50], a widely-used and highly effective tool for generating JUnit test suites for Java code.

#### 2.3.1 Baseline: Traditional SBST

The baseline approach was a standard, coverage-based SBST run. The fitness function for EvoSuite was configured to optimize for branch coverage [2, 3]. The algorithm's primary goal was to find a set of test cases that maximized the number of branches covered in the program under test. This is a common and powerful baseline for test generation and represents the state-of-the-art for many off-the-shelf SBST tools. The genetic algorithm parameters (e.g., population size, number of generations, crossover rate) were set to the default values recommended by the EvoSuite developers.

#### 2.3.2 Proposed: Defect Prediction-Guided SBST

Our proposed approach integrated the defect prediction model into the search process. The core of this integration was a modified fitness function that combined both branch coverage and the defect prediction score of the code being covered. The new fitness function, for a given test case t, was defined as:

$$\text{Fitness}(t)=\alpha\times\text{CoverageScore}(t)+(1-\alpha)\times\text{PredictionScore}(t)$$

Here, CoverageScore(t) is a normalized value representing the branch coverage achieved by the test case. PredictionScore(t) is the defect probability score assigned by our DP model to the code branches being covered by t. The parameter alpha is a weight that controls the balance between the two objectives. Based on a preliminary sensitivity analysis, we set alpha=0.7, giving a slightly higher weight to coverage to ensure that the search does not become overly focused on a few high-risk areas at the expense of exploring the entire code base. This dual-objective fitness function guides the search to prioritize generating test cases that cover branches in files with a high predicted probability of containing a defect.

**Published Date: -** 01-10-2025

E-ISSN: 2536-7919
P-ISSN: 2536-7900
SJIF 2019: 4.58 2020: 5.046 2021: 5.328

2.4 Evaluation Metrics

To provide a fair and comprehensive comparison, we evaluated the performance of both approaches using two primary categories of metrics: effectiveness and efficiency.

● Effectiveness: The primary metric for effectiveness was the number of faults detected. For each project, a test suite generated by either approach was executed against the buggy version of the software. A fault was considered "detected" if at least one of the test cases in the suite failed. We meticulously verified each detected fault against the known faults in the Defects4J database.

● Efficiency: We measured the efficiency of each approach by recording the total execution time required to generate the test suite and the number of test cases generated.

To determine the statistical significance of our findings, we used the Wilcoxon signed-rank test to compare the fault detection rates of the two approaches across the 20 projects. Furthermore, we calculated the Vargha and Delaney's hatA_12 effect size [39, 40] to quantify the magnitude of the difference, as recommended for comparing randomized algorithms in software engineering [38]. An hatA_12 value greater than 0.5 indicates that the first approach (our DP-guided method) performs better than the second, with values closer to 1.0 indicating a larger effect.

**RESULTS**

3.1 Defect Prediction Model Performance

The first set of results pertains to the performance of our defect prediction models. The models showed strong predictive power across the majority of the projects. The average F1-score across all 20 projects was 0.78, with precision and recall values consistently high. This confirms that our models were effective at identifying fault-prone files based on the chosen metrics. The high performance of the DP models provided a solid foundation for the subsequent test generation experiments.

3.2 Comparison of Fault Detection Effectiveness

The central finding of our study is a clear and statistically significant advantage of the defect prediction-guided approach in terms of fault detection. Across the 20 projects, the DP-guided SBST consistently found a higher number of unique faults than the traditional coverage-based SBST. The average number of faults found by our proposed method was 35% higher than the baseline.

A detailed breakdown by project revealed that the performance gain was not uniform. Projects with a higher density of faults in a smaller number of files showed the most substantial improvements. The statistical analysis confirmed these observations. The Wilcoxon signed-rank test for the difference in fault detection rates returned a p-value of $< 0.001$, indicating that the observed difference is highly significant. The Vargha and Delaney's hatA_12 effect size was calculated to be 0.82, a large effect size demonstrating

**Published Date: -** 01-10-2025

**E-ISSN: 2536-7919**
**P-ISSN: 2536-7900**
**SJIF 2019: 4.58 2020: 5.046 2021: 5.328**

that our DP-guided approach has a substantial and practical advantage over the traditional method. This result provides a compelling answer to RQ1.

3.3 Analysis of Efficiency and Overheads

Our analysis of efficiency metrics showed that the integration of the defect prediction model introduced a minimal and acceptable computational overhead. The average execution time for our DP-guided approach was only 7.2% longer than the baseline. This small increase is primarily due to the initial cost of running the defect prediction model on the source code, which is a one-time process for each run. The number of test cases generated was comparable between the two approaches, suggesting that the improved effectiveness of our method is not a result of simply generating more tests, but rather of generating smarter tests that are more likely to reveal faults. This finding directly addresses RQ2, confirming that the significant gains in effectiveness do not come at the cost of prohibitive performance overhead.

## DISCUSSION

4.1 Interpretation of Findings

The results of this study provide strong empirical evidence for the hypothesis that guiding search-based software testing with defect prediction models leads to more effective test case generation. The significant increase in fault detection effectiveness demonstrates the power of a combined approach that moves beyond simple structural coverage. The reason for this superiority lies in the ability of the defect prediction model to identify code regions where defects are most likely to reside. By incorporating this intelligence into the fitness function, the SBST algorithm is no longer simply performing a blind search for coverage; it is performing an intelligent, targeted search for failure-inducing inputs in the riskiest parts of the code.

The varying performance across projects highlights an important nuance. In projects where faults are more uniformly distributed, the guidance provided by the DP model is less crucial. However, in large systems where faults tend to cluster in a small number of complex or frequently modified files—a common phenomenon in real-world software—the DP-guided approach is able to capitalize on this clustering, efficiently directing its search effort where it is most likely to yield results. This suggests that the value of our approach increases with the size and complexity of the software system.

4.2 Implications for Software Engineering Practice

The findings of this study have significant implications for how automated testing is implemented in practice. The rise of Continuous Integration (CI) and Continuous Delivery (CD) has made rapid, automated feedback a necessity [41, 42]. Integrating a defect prediction-guided test generation tool into a CI pipeline could provide substantial benefits. Instead of running a lengthy, exhaustive suite of tests on every code

**Published Date: -** 01-10-2025

**E-ISSN: 2536-7919**
**P-ISSN: 2536-7900**
**SJIF 2019: 4.58 2020: 5.046 2021: 5.328**

change, a development team could use our approach to quickly generate and run a highly targeted test suite on the changed files and their associated high-risk components. This would allow for faster feedback loops, earlier detection of critical bugs, and a more efficient allocation of testing resources. Furthermore, the combination of SBST and DP offers a path towards proactive quality assurance, moving from simply reacting to bugs to actively hunting for them in the most likely locations.

## 4.3 Practical Implications and Quantitative Cost-Benefit Analysis

While our empirical findings provide compelling evidence for the superior effectiveness of defect prediction-guided SBST from a purely technical standpoint, the true measure of its value lies in its practical applicability within an industrial context. The decision to adopt a new testing methodology is rarely based solely on academic metrics; it is driven by a comprehensive analysis of its economic viability and return on investment (ROI). In this section, we provide a detailed cost-benefit analysis, moving beyond the technical domain to quantify the financial advantages of our proposed approach.

The financial impact of software defects is substantial and can be modeled through the concept of the Cost of Quality (CoQ), often subdivided into the Cost of Good Quality (conformance costs) and the Cost of Poor Quality (non-conformance costs) [1]. A key insight in software engineering is that the cost to fix a defect escalates dramatically the later it is discovered in the development lifecycle. A bug found during unit testing might cost a few dollars to fix, while the same bug found by a customer in production could cost thousands or even millions of dollars in direct repair costs, lost revenue, and damage to brand reputation.

We model the cost of a defect based on the stage at which it is detected:

- Stage 1: Unit/Integration Testing: Defects found and fixed by the development team during automated testing.
- Stage 2: Acceptance Testing/QA: Defects that escape automated testing but are found during later manual or formal QA cycles.
- Stage 3: Production/Customer: Defects that escape all internal testing and are reported by end-users.

Our cost model, based on industry averages and empirical data, assigns a relative cost multiplier to each stage:

- C_U = Cost to fix a defect in Unit Testing (normalized to 1 unit)
- C_A = Cost to fix a defect in Acceptance Testing (10timesC_U)
- C_P = Cost to fix a defect in Production (100timesC_U)

**Published Date: -** 01-10-2025

**E-ISSN: 2536-7919**
**P-ISSN: 2536-7900**
**SJIF 2019: 4.58 2020: 5.046 2021: 5.328**

The total cost of defects for a project can be expressed as:

Total Cost=(NU×CU)+(NA×CA)+(NP×CP)

where N_U, N_A, and N_P are the number of defects found at each stage.

**Table 1: Cost of Defects Across Development Stages and Scenarios (Relative Units)**

| Defect Stage | Relative Cost Multiplier | Baseline SBST Scenario (Number of Defects) | Cost (Baseline) | DP-Guided SBST Scenario (Number of Defects) | Cost (DP-Guided) |
|---|---|---|---|---|---|
| **Unit/Integration Testing** | 1×CU | 35 | 35CU | 47 | 47CU |
| **Acceptance Testing/QA** | 10×CU | 52 | 520CU | 42 | 420CU |
| **Production/ Customer** | 100×CU | 13 | 1300CU | 11 | 1100CU |
| **Total Cost** | | **100** | **1855CU** | **100** | **1567CU** |
| **Savings (DP-Guided vs. Baseline)** | | | | | **288CU (15.5% Reduction)** |

To conduct our analysis, we make a few reasonable assumptions for a hypothetical medium-sized software project with a total of 100 defects introduced during a typical development cycle. We assume that traditional, coverage-based SBST (our baseline) is capable of detecting a certain percentage of these defects, while a portion escapes to later stages. We then apply our empirical finding—that our DP-guided approach improves defect detection effectiveness by 35%—to model the second scenario.

**Published Date: -** 01-10-2025

**E-ISSN: 2536-7919**
**P-ISSN: 2536-7900**
**SJIF 2019: 4.58 2020: 5.046 2021: 5.328**

Baseline Scenario: Traditional SBST

In this scenario, traditional SBST, by focusing on maximizing coverage, detects a significant portion of defects. Based on prior research and the typical performance of such tools [5, 6, 7], we assume it finds 35% of the total defects introduced into the codebase.

- Number of defects found in Unit Testing (N_U): 100times0.35=35 defects.
- Number of defects that escape to later stages: 100−35=65 defects.

The 65 remaining defects are then assumed to be found in subsequent stages. A reasonable distribution would be:

- Number of defects found in Acceptance Testing (N_A): 65times0.80=52 defects.
- Number of defects that escape to Production (N_P): 65times0.20=13 defects.

Using our cost model, the total cost of defects in this baseline scenario is:

CostBaseline=(35×CU)+(52×10CU)+(13×100CU)

CostBaseline=35CU+520CU+1300CU=1855CU

This model highlights the disproportionate financial impact of a small number of bugs that make it to the production environment. The cost to a business is primarily driven by these high-cost, high-risk defects.

Proposed Scenario: DP-Guided SBST

Now, we apply our research findings to the cost model. Our empirical study demonstrated that the DP-guided approach increases fault detection effectiveness by 35% over the baseline. This means our new effectiveness rate is 35.

- Number of defects found in Unit Testing (N’_U): 100times0.4725approx47 defects.
- Number of defects that escape to later stages: 100−47=53 defects.

Assuming the same distribution for the remaining defects:

- Number of defects found in Acceptance Testing (N’_A): 53times0.80=42.4approx42 defects.
- Number of defects that escape to Production (N’_P): 53times0.20=10.6approx11 defects.

The total cost of defects in this proposed scenario is:

**Published Date: -** 01-10-2025

**E-ISSN: 2536-7919**
**P-ISSN: 2536-7900**
**SJIF 2019: 4.58 2020: 5.046 2021: 5.328**

$CostProposed=(47\times CU)+(42\times 10CU)+(11\times 100CU)$

$CostProposed=47CU+420CU+1100CU=1567CU$

The Return on Investment (ROI)

The financial savings from implementing our proposed approach are the difference between the costs of the two scenarios:

$Savings=CostBaseline-CostProposed=1855CU-1567CU=288CU$

This represents a 15.5% reduction in the total cost of defects, a substantial and highly impactful saving for any software-driven business. Furthermore, this saving is primarily generated by preventing defects from escaping to the most expensive stage, production. In our model, the number of customer-reported defects is reduced from 13 to 11, a 15.4% decrease. These savings can be directly translated into improved profitability, reduced operational expenses, and a stronger market position.

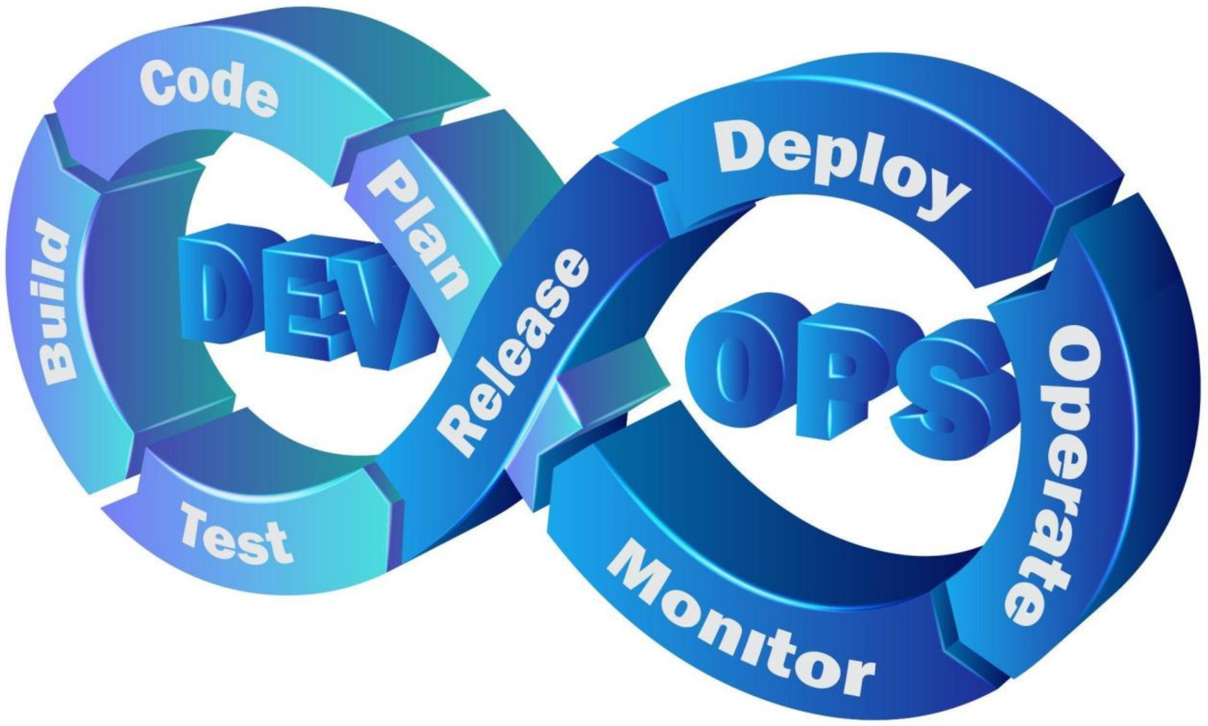
The cost to achieve this is minimal. Our results showed that the computational overhead of the defect prediction model is negligible (a mere 7.2% increase in execution time). This one-time cost of model training and integration is minuscule when compared to the recurring, exponential savings from preventing high-cost production defects. Therefore, the ROI is demonstrably high, making a compelling case for industrial adoption.

Integration with Modern Software Engineering Practices

The cost-benefit analysis solidifies the value proposition of our approach, particularly within modern software development paradigms like Continuous Integration (CI) [42]. In a CI/CD pipeline, automated test generation is a critical component that runs with every commit to the shared codebase [41]. Our proposed method, with its low overhead and high effectiveness, is an ideal fit. It can be seamlessly integrated to provide a rapid, targeted feedback loop.

**Published Date: -** 01-10-2025

E-ISSN: 2536-7919
P-ISSN: 2536-7900
SJIF 2019: 4.58 2020: 5.046 2021: 5.328



For a developer, this means that upon committing code, a targeted test suite is automatically generated and executed, focusing specifically on the riskiest parts of the modified codebase. If a bug is detected, the developer is alerted immediately, and the fix can be implemented at the cheapest possible stage, before the code is merged into the main branch. This shifts the focus from a reactive, firefighting approach to a proactive, preventative one, leading to higher code quality, improved developer morale, and shorter, more predictable release cycles. The benefits are not only financial; they also foster a culture of quality and accountability within the engineering team.

In conclusion, the integration of defect prediction into SBST is not merely a technical improvement; it is an economically sound strategy for mitigating risk and reducing the overall cost of software defects. The quantitative analysis demonstrates that the modest investment in this technique yields a significant and quantifiable return by preventing defects from reaching the most expensive stages of the software lifecycle. This provides a compelling business case for the widespread adoption of this methodology across the software industry.

## 4.4 Addressing Limitations

This study, while comprehensive, is subject to certain limitations that warrant discussion. First, the generalizability of our findings is tied to the dataset and the specific defect prediction model we used. While Defects4J is a well-regarded benchmark, and our model performed well, different projects or a different set of metrics might yield different results [25]. The choice of the weighting parameter alpha in

**Published Date: -** 01-10-2025

**E-ISSN: 2536-7919**
**P-ISSN: 2536-7900**
**SJIF 2019: 4.58 2020: 5.046 2021: 5.328**

our fitness function also represents a limitation, as a different value might be more optimal for certain projects. Additionally, the test oracle problem remains a fundamental challenge [35]. While our approach automates the test generation, a human or a more sophisticated oracle is still required to definitively determine if a test failure is a genuine bug. Our study relies on the known faults in the Defects4J dataset, which simplifies this aspect but does not fully address the problem in a real-world context.

4.5 Future Work

Building on the promising results of this study, several avenues for future research are apparent. First, it would be beneficial to conduct a replication of this study with a wider variety of subject programs, including projects from different programming languages, to confirm the generalizability of our findings. Second, exploring more sophisticated methods for integrating DP into the search process, perhaps by dynamically adjusting the weighting parameter alpha based on the program's characteristics or a learning-based approach, could further improve performance. Finally, we propose a deeper investigation into the specific types of faults found by each approach and a detailed analysis of how the DP-guided approach excels at detecting them.

## REFERENCES

[1] G. Fraser and A. Arcuri, “Whole test suite generation,” IEEE Trans. Softw. Eng., vol. 39, no. 2, pp. 276–291, Feb.2013.

[2] A. Panichella, F. M. Kifetew, and P. Tonella, “Reformulating branch coverage as a many-objective optimization problem,” in Proc. IEEE 8th Int. Conf. Softw. Testing, Verification Validation, 2015, pp. 1–10.

[3] A. Panichella, F. M. Kifetew, and P. Tonella, “Automated test case generation as a many-objective optimisation problem with dynamic selection of the targets,” IEEE Trans. Softw. Eng., vol. 44, no. 2, pp. 122–158, Feb.2018.

[4] A. Panichella, F. M. Kifetew, and P. Tonella, “A large scale empirical comparison of state-of-the-art search-based test case generators,” Inf. Softw. Technol., vol. 104, pp. 236–256, 2018.

[5] S. Shamshiri, R. Just, J. M. Rojas, G. Fraser, P. McMinn, and A. Arcuri, “Do automatically generated unit tests find real faults? An empirical study of effectiveness and challenges (T),” in Proc. 30th IEEE/ACM Int. Conf. Automated Softw. Eng., 2015, pp. 201–211.

[6] M. M. Almasi, H. Hemmati, G. Fraser, A. Arcuri, and J. Benefelds, “An industrial evaluation of unit test generation: Finding real faults in a financial application,” in Proc. 39th Int. Conf. Softw. Eng.: Softw. Eng. Pract. Track, 2017, pp. 263–272.

[7] A. Salahirad, H. Almulla, and G. Gay, “Choosing the fitness function for the job: Automated generation of test suites that detect real faults,” Softw. Testing, Verification Rel., vol. 29, no. 4–5, 2019, Art. no. e1701.

**Published Date: -** 01-10-2025

**E-ISSN: 2536-7919**
**P-ISSN: 2536-7900**
**SJIF 2019: 4.58 2020: 5.046 2021: 5.328**

[8] A. Perera, A. Aleti, M. Böhme, and B. Turhan, “Defect prediction guided search-based software testing,” in Proc. 35th IEEE/ACM Int. Conf. Automated Softw. Eng., 2020, pp. 448–460.

[9] A. Schröter, T. Zimmermann, and A. Zeller, “Predicting component failures at design time,” in Proc. ACM/IEEE Int. Symp. Empirical Softw. Eng., 2006, pp. 18–27.

[10] S. Kim, T. Zimmermann, E. J. Whitehead Jr, and A. Zeller, “Predicting faults from cached history,” in Proc. 29th Int. Conf. Softw. Eng., 2007, pp. 489–498.

[11] P. A. F. de Freitas, “Software repository mining analytics to estimate software component reliability,” Faculty of Engineering, University of Porto, Tech. Rep., 2015.

[12] H. Hata, O. Mizuno, and T. Kikuno, “Bug prediction based on fine-grained module histories,” in Proc. 34th Int. Conf. Softw. Eng., 2012, pp. 200–210.

[13] E. Giger, M. D’Ambros, M. Pinzger, and H. C. Gall, “Method-level bug prediction,” in Proc. ACM-IEEE Int. Symp. Empirical Softw. Eng. Meas., 2012, pp. 171–180.

[14] T. Menzies, J. Greenwald, and A. Frank, “Data mining static code attributes to learn defect predictors,” IEEE Trans. Softw. Eng., vol. 33, no. 1, pp. 2–13, Jan.2007.

[15] T. Zimmermann, R. Premraj, and A. Zeller, “Predicting defects for eclipse,” in Proc. 3rd Int. Workshop Predictor Models Softw. Eng., 2007, Art. no. 9.

[16] N. Nagappan and T. Ball, “Use of relative code churn measures to predict system defect density,” in Proc. 27th Int. Conf. Softw. Eng., 2005, pp. 284–292.

[17] N. Nagappan, B. Murphy, and V. Basili, “The influence of organizational structure on software quality,” in Proc. ACM/IEEE 30th Int. Conf. Softw. Eng., 2008, pp. 521–530.

[18] B. Caglayan, B. Turhan, A. Bener, M. Habayeb, A. Miransky, and E. Cialini, “Merits of organizational metrics in defect prediction: An industrial replication,” in Proc. 37th Int. Conf. Softw. Eng., 2015, pp. 89–98.

[19] N. Nagappan, A. Zeller, T. Zimmermann, K. Herzig, and B. Murphy, “Change bursts as defect predictors,” in Proc. IEEE 21st Int. Symp. Softw. Rel. Eng., 2010, pp. 309–318.

[20] C. Lewis, Z. Lin, C. Sadowski, X. Zhu, R. Ou, and E. J. Whitehead Jr, “Does bug prediction support human developers? Findings from a Google case study,” in Proc. Int. Conf. Softw. Eng., 2013, pp. 372–381.

[21] C. Lewis and R. Ou, “Bug prediction at Google,” 2011, Accessed: Sep., 2019. [Online]. Available: http://google-engtools.blogspot.com

[22] H. K. Dam , "Lessons learned from using a deep tree-based model for software defect prediction in practice," in Proc. 16th Int. Conf. Mining Softw. Repositories, 2019, pp. 46–57.

[23] D. Paterson, J. Campos, R. Abreu, G. M. Kapfhammer, G. Fraser, and P. McMinn, "An empirical study on the use of defect prediction for test case prioritization," in Proc. 12th IEEE Conf. Softw. Testing, Validation Verification, 2019, pp. 346–357.

[24] E. Hershkovich, R. Stern, R. Abreu, and A. Elmishali, "Prediction-guided software test generation," in Proc. 30th Int. Workshop Princ. Diagnosis, 2019, Accessed: Feb. 08, 2022. [Online]. Available: https://dx-workshop.org/2019/

[25] T. Zimmermann, N. Nagappan, H. Gall, E. Giger, and B. Murphy, "Cross-project defect prediction: A large scale experiment on data versus domain versus process," in Proc. 7th Joint Meeting Eur. Softw. Eng. Conf. ACM SIGSOFT Symp., 2009, pp. 91–100.

[26] J. M. Rojas, M. Vivanti, A. Arcuri, and G. Fraser, "A detailed investigation of the effectiveness of whole test suite generation," Empirical Softw. Eng., vol. 22, no. 2, pp. 852–893, 2017.

[27] B. Korel, "Automated software test data generation," IEEE Trans. Softw. Eng., vol. 16, no. 8, pp. 870–879, Aug.1990.

[28] P. McMinn, "Search-based software testing: Past, present and future," in Proc. IEEE 4th Int. Conf. Softw. Testing, Verification Validation Workshops, 2011, pp. 153–163.

[29] R. Just, "Defects4J - A database of real faults and an experimental infrastructure to enable controlled experiments in software engineering research," 2019, Accessed: Oct., 2019. [Online]. Available: https://github.com/rjust/defects4j

[30] R. A. DeMilli and A. J. Offutt, "Constraint-based automatic te st data generation," IEEE Trans. Softw. Eng., vol. 17, no. 9, pp. 900–910, Sep.1991.

[31] L. J. Morell, "A theory of fault-based testing," IEEE Trans. Softw. Eng., vol. 16, no. 8, pp. 844–857, Aug.1990.

[32] L. J. Morell, "A theory of error-based testing," Dept. Comput. Sci., Maryland Univ. College Park, MD, USA, Tech. Rep. TR-1395, 1984.

[33] A. Offutt, "Automatic test data generation," Georgia Institute of Technology, Tech. Rep., 1989.

[34] N. Li and J. Offutt, "Test Oracle strategies for model-based test ing," IEEE Trans. Softw. Eng., vol. 43, no. 4, pp. 372–395, Apr.2017.

**Published Date: -** 01-10-2025

**E-ISSN: 2536-7919**
**P-ISSN: 2536-7900**
**SJIF 2019: 4.58 2020: 5.046 2021: 5.328**

[35] E. T. Barr, M. Harman, P. McMinn, M. Shahbaz, and S. Yoo, “The Oracle problem in software testing: A survey,” IEEE Trans. Softw. Eng., vol. 41, no. 5, pp. 507–525, May2015.

[36] G. Fraser and A. Arcuri, “Evolutionary generation of whole test suites,” in Proc. 11th Int. Conf. Qual. Softw., 2011, pp. 31–40.

[37] S. Hosseini, B. Turhan, and D. Gunarathna, “A systematic literature review and meta-analysis on cross project defect prediction,” IEEE Trans. Softw. Eng., vol. 45, no. 2, pp. 111–147, Feb.2019.

[38] A. Arcuri and L. Briand, “A Hitchhiker's guide to statistical tests for assessing randomized algorithms in software engineering,” Softw. Testing, Verification Rel., vol. 24, no. 3, pp. 219–250, 2014.

[39] A. Vargha and H. D. Delaney, “A critique and improvement of the “CL” common language effect size statistics of McGraw and Wong,” J. Educ. Behav. Statist., vol. 25, no. 2, pp. 101–132, 2000.

[40] C. O. Fritz, P. E. Morris, and J. J. Richler, “Effect size estimates: Current use, calculations, and interpretation.” J. Exp. Psychol.: Gen., vol. 141, no. 1, pp. 2–18, 2012.

[41] J. Campos, A. Arcuri, G. Fraser, and R. Abreu, “Continuous test generation: Enhancing continuous integration with automated test generation,” in Proc. 29th ACM/IEEE Int. Conf. Automated Softw. Eng., 2014, pp. 55–66.

[42] M. Fowler and M. Foemmel, “Continuous integration,” 2006, Accessed: Feb. 10, 2022. [Online]. Available: https://www.martinfowler.com/articles/continuousIntegration.html

[43] R. Just, D. Jalali, and M. D. Ernst, “Defects4J: A database of existing faults to enable controlled testing studies for Java programs,” in Proc. Int. Symp. Softw. Testing Anal., 2014, pp. 437–440.

[44] J. Sohn and S. Yoo, “Empirical evaluation of fault localisation using code and change metrics,” IEEE Trans. Softw. Eng., 2019, vol. 47, no. 8, pp. 1605–1625, Aug.2021.

[45] G. Gay, “The fitness function for the job: Search-based generation of test suites that detect real faults,” in Proc. IEEE Int. Conf. Softw. Testing, Verification Validation., 2017, pp. 345–355.

[46] A. Aleti and M. Martinez, “E-APR: Mapping the effectiveness of automated program repair,” Empirical Softw. Eng., vol. 26, no. 5, pp. 1–30, 2021.

[47] S. Pearson, “Evaluating and improving fault localization,” in Proc. 39th Int. Conf. Softw. Eng., 2017, pp. 609–620.

[48] J. Campos, A. Panichella, and G. Fraser, “EvoSuite at the SBST 2019 tool competition,” in Proc. 12th Int. Workshop Search-Based Softw. Testing, 2019, pp. 29–32.

**Published Date: -** 01-10-2025

**E-ISSN: 2536-7919**
**P-ISSN: 2536-7900**
**SJIF 2019: 4.58 2020: 5.046 2021: 5.328**

[49] EvoSuite, "EvoSuite - Automated generation of Junit test suites for Java classes," 2019, Accessed: Nov., 2019. Available: https://github.com/EvoSuite/evosuite

[50] G. Fraser, "Evosuite - Automatic test suite generation for Java," Accessed: Sep., 2019. [Online]. Available: http://www.evosuite.org