International Journal of Internet, Broadcasting and Communication Vol.16 No.4 194-202 (2024)
http://dx.doi.org/10.7236/IJIBC.2024.16.4.194

IJIBC 24-4-22

A Study on Test-Driven Development Method with the Aid of Generative AI in Software Engineering

Woochang Shin

Professor Dept. of Computer Science, Seokyeong University, Korea
wcshin@skuniv.ac.kr

Abstract

This study explores the integration of Generative AI into Test-Driven Development (TDD) to efficiently produce code that accurately reflects programmers' requirements in software engineering. Using the Account class as an example, we analyzed the code generation capabilities of leading Generative AI models—OpenAI's ChatGPT, GitHub's Copilot, and Google's Gemini. Our findings indicate that while Generative AI can automatically generate code, it often fails to capture programmers' intent, potentially leading to functional errors or security vulnerabilities. By applying TDD principles and providing detailed test cases to the Generative AI, we demonstrated that the generated code more closely aligns with the programmer's intentions and successfully passes specified tests. This approach reduces the need for manual code reviews and enhances development efficiency. We propose a development process that combines TDD with Generative AI, leveraging the strengths of both to efficiently produce high-quality software. Future research will focus on extending this approach to more complex systems and exploring automatic test case generation techniques.

Keywords: *: TDD, Generative AI, Automatic Code Generation, Software Development Methodology, Software Testing*

1. Introduction

The field of software development is undergoing transformative changes due to advancements in artificial intelligence and machine learning. In particular, the emergence of Generative AI has spurred efforts to improve development efficiency in various areas such as automatic code generation, bug detection, and performance optimization [1-3]. These technologies contribute to reducing development time and costs, facilitating the implementation of complex software systems. Additionally, the ability to generate code through natural language allows even non-experts to participate in programming [4][5]. In this context, the introduction of Generative AI is expected to revolutionize the software development process.

However, there are several issues associated with generating source code using Generative AI [6][7]. First, the generated code may not accurately reflect the programmer's requirements, as AI models may not fully comprehend the nuances or context of natural language. Second, there is a high possibility that the generated code may contain security vulnerabilities, since AI may not incorporate the latest security standards or best

---

Manuscript Received: September. 26, 2024 / Revised: September. 30, 2024 / Accepted: October. 5, 2024
† Corresponding Author: wcshin@skuniv.ac.kr
Tel:+82-2-940-7755
Professor, Department of Computer Science, Seokyeong University, Korea

Copyright© 2024 by The Institute of Internet, Broadcasting and Communication. This is an Open Access article distributed under the terms of the Creative Commons Attribution Non-Commercial License (http://creativecommons.org/licenses/by-nc/4.0)

practices due to limitations in training data. Third, additional efforts may be required for debugging and maintenance.

To address these challenges, it is necessary to apply Generative AI within the Test-Driven Development (TDD) methodology. TDD is a development approach that emphasizes writing tests before code implementation, clarifying programmer requirements and enhancing code accuracy and stability [8]. By integrating Generative AI with TDD, we can ensure that automatically generated code is guided to pass predefined tests. This approach allows for accurately reflecting the programmer's intent, minimizing security vulnerabilities, and improving overall development efficiency.

In this paper, we discuss methods and the potential of applying Generative AI in the software implementation stages using the TDD approach. We compare and analyze the code generation capabilities of major Generative AI models and identify the causes of inaccurate code generation. Furthermore, by applying the code generation features of Generative AI to the TDD method, we demonstrate that it is possible to effectively generate code that more closely aligns with the programmer's requirements. Finally, we propose a process for integrating Generative AI functionalities into the TDD methodology."

The main contributions of this paper are as follows:

- Enhancing Development Efficiency by Integrating Generative AI with TDD: This study presents a method for efficiently generating code that accurately reflects programmers' requirements by integrating Generative AI into the TDD process.
- Analysis of Code Generation Capabilities and Limitations of Major Generative AI Models: We conducted a comparative analysis of the code generation capabilities of leading Generative AI models, including OpenAI's ChatGPT, GitHub's Copilot, and Google's Gemini.
- Proposal of a New Development Process Combining TDD and Generative AI: We proposed a development process that combines the strengths of TDD and Generative AI. This process reduces the need for manual code reviews and enhances development efficiency by ensuring that generated code meets the predefined test cases.

The remainder of the paper is organized as follows.

In Section 2, related works are reviewed. In Section 3, we analyze the code generation capabilities of major Generative AI models. Section 4 discusses the method of applying Generative AI within the TDD process. In Section 5, we perform a comparative analysis of the generated code to evaluate the effectiveness of our proposed method. Finally, Section 6 summarizes and concludes this paper.

2. Related Work

Research on code generation using generative AI has been increasingly prominent in the field of software engineering. Chen et al. [1] explored the potential of code generation using large-scale language models like GPT-3. They successfully generated executable code based on natural language descriptions, demonstrating the feasibility of using generative AI for code synthesis. Similarly, Svyatkovskiy et al. [9] developed IntelliCode Compose, which leverages Transformer-based models to enhance programmer productivity by providing intelligent code completions. Their work showed improvements in code completion quality and reductions in development time.

However, several studies have pointed out the challenges and problems associated with code generation using generative AI. Austin et al. [10] highlighted limitations in program synthesis with large language models, expressing concerns about the accuracy and reliability of the generated code. They emphasized that generated code might contain unexpected errors or security vulnerabilities.

Research on software testing using generative AI has also been actively pursued. Tian et al. [11] proposed DeepTest, an automated testing framework for deep neural network-driven autonomous cars. By utilizing deep learning techniques, they were able to automatically generate test cases that improve the robustness and safety of software systems. Ayenew et al. [12] conducted a systematic literature review on automated test case generation using machine learning, providing insights into current trends and future research directions in AI-based testing. Bhatia et al. [13] studied whether Generative AI is effective in generating unit test scripts for Python programs and how the generated test cases compare to those generated by the existing unit test generator, Pynguin.

Other related research includes studies on software refactoring and bug fixes using Generative AI. Tufano et al. [14] proposed a method to automatically generate bug fix patches using deep learning models and demonstrated its effectiveness in real open-source projects. Vasic et al. [15] developed a technique to automatically detect and fix program bugs using neural networks, contributing to improving software quality.

These studies show many possibilities for applying Generative AI in the field of software engineering, but there is still a lack of research applying Generative AI to the TDD method.

3. Analysis of Generative AI's Code Generation Capabilities

This study assumes the application of Generative AI during the code implementation process using the TDD method, in a situation where software design has been somewhat completed.

In this section, we compare and analyze the capabilities of representative Generative AI models (ChatGPT, Copilot, and Gemini) before applying the TDD method. The models of each Generative AI are as follows:

- OpenAI's ChatGPT (GPT-4o model)
- GitHub's Copilot (Based on the GPT-4o model)
- Google's Gemini (Sep. 2024 model)

To analyze the capabilities, we use the *Account* class as an example. The basic requirements and design details of the member functions of this class are shown in Figure 1.

[Requirements-1 for the *Account* class]

- Calculate different interest rates based on the duration of the deposited amount:
  - Less than 3 months: 2%
  - 3 months to less than 6 months: 3%
  - 6 months or more: 4%
- The member function names and their functionalities of the class are as follows:
  - deposit(): Accepts the amount to be deposited (int) and the deposit date (LocalDate) as arguments, and stores them in the account.
  - calculateDeposit(): Calculates and returns the total amount of all deposits, including interest rates based on their deposit dates.

Figure 1. Requirements-1 for the *Account* class

By inputting these requirements into the Generative AIs and requesting source code generation, source codes as shown in Figure 1 were generated.

[ChatGPT]

```
public class Account {
    ...
    private List<Deposit> deposits = new
                                ArrayList<>();
    public void deposit(int amount,
                              LocalDate date) {
        deposits.add(new Deposit(amount,date));
    }
    public double calculateDeposit() {
        double total = 0;
        LocalDate today = LocalDate.now();
        for (Deposit dep : deposits) {
            int months = Period.between(
                    dep.date, today).getMonths();
            double interestRate;
            if (months < 3) {
                interestRate = 0.02;
            } else if (months < 6) {
                interestRate = 0.03;
            } else
                interestRate = 0.06;
            total += dep.amount +
                    (dep.amount * interestRate);
        }
        return total;
    }
}
```

[Copilot]

```
public class Account {
    private int balance;
    private LocalDate depositDate;
    public void deposit(int amount,
                               LocalDate date) {
        this.balance = amount;
        this.depositDate = date;
    }
    public double calculateDeposit() {
        if (depositDate == null) {
            throw new IllegalStateException("No
                              deposit made");
        }
       LocalDate currentDate=LocalDate.now();
        long monthsBetween = ChronoUnit
              .MONTHS.between(depositDate
                              , currentDate);
        double interestRate;
        if (monthsBetween < 3) {
            interestRate = 0.02;
        } else if (monthsBetween < 6) {
            interestRate = 0.03;
        } else {
            interestRate = 0.04;
        }
        return balance + (balance * interestRate);
    }
}
```

[Gemini]

```
public class Account {
    private int balance;
    private LocalDate depositDate;
    public void deposit(int amount,
                                  LocalDate date) {
        balance += amount;
        depositDate = date;
    }
    public double calculateDeposit() {
        LocalDate today = LocalDate.now();
        long months = java.time.temporal
                .ChronoUnit.MONTHS
                .between(depositDate, today);
        double interestRate = 0.0;
        if (months < 3) {
            interestRate = 0.02;
        } else if (months < 6) {
            interestRate = 0.03;
        } else {
            interestRate = 0.04;
        }
        return balance * (1 + interestRate);
    }
}
```

Figure 2. The codes generated by Generative AI for Requirement-1

Despite the same requirements, each Generative AI generated source codes with different functionalities. First, ChatGPT used the List class to generate code that can store multiple deposits and dates, while Copilot generated code that can store only a single deposit. Gemini allowed multiple deposits, but had the issue that the dates of each deposit were not stored separately and were fixed to the date of the last deposit.

When explaining interest rates in Requirement-1, annual interest rates and simple interest rates were considered common sense and were not explained separately. However, all Generative AIs calculated the deposit amount uniformly at interest rates of 2–4% depending on the deposit period, not using the annual interest rate.

Requirement-2 was written by describing the programmer's requirements in more detail in Requirement-1. The additional content written in Requirement-1 is shown in Figure 3, and the code generated by generative AIs according to Requirement-2 is as shown in Figure 4.

[Requirement-2 for the *Account* class (only added part)]

- Interest is applied as simple annual interest and is calculated daily.
- Deposits can be made multiple times, and calculateDeposit() calculates and returns the total amount of multiple deposits.

Figure 3. Requirements-2 for the *Account* class

[ChatGPT]

```
public class Account {
    ...
    private List<Deposit> deposits = new
                              ArrayList<>();
    public void deposit(int amount, LocalDate
                              date) {
      deposits.add(new Deposit(amount, date));
    }
     private double getInterestRate(long
                months) {
        if (months < 3) {
            return 0.02;
        } else if (months < 6) {
            return 0.03;
        } else {
            return 0.04;
        }
    }
    public double calculateDeposit() {
        double totalAmount = 0;
        LocalDate today = LocalDate.now();
        for (Deposit deposit : deposits) {
          long daysBetween = ChronoUnit.DAYS
              .between(deposit.depositDate, today);
          long monthsBetween = ChronoUnit
              .MONTHS
              .between(deposit.depositDate, today);
          double interestRate =
              getInterestRate(monthsBetween);
          totalAmount += deposit.amount +
(deposit.amount * interestRate * daysBetween
/ 365);
        }
        return totalAmount;
    }
}
```

[Copilot]

```
public class Account {
    ...
    private List<Deposit> deposits = new
                                ArrayList<>();
    public void deposit(int amount, LocalDate
date) {
      deposits.add(new Deposit(amount, date));
    }
    public double calculateDeposit() {
        double total = 0.0;
        LocalDate now = LocalDate.now();
        for (Deposit deposit : deposits) {
            long months = ChronoUnit.MONTHS
                  .between(deposit.getDate(), now);
            double interestRate =
                        getInterestRate( months );
            double interest = deposit.getAmount()
                        * interestRate * months / 12;
            total += deposit.getAmount()+interest;
        }
        return total;
    }
    private double getInterestRate(long months)
    {
        if (months < 3) {
            return 0.02;
        } else if (months < 6) {
            return 0.03;
        } else {
            return 0.04;
        }
    }
}
```

[Gemini]

```
public class Account {
     ...
     private List<Deposit> deposits = new
                              ArrayList<>();
     public void deposit(int amount, LocalDate
                              depositDate) {
         deposits.add(new Deposit(amount,
                              depositDate));
     }
     public double calculateDeposit() {
         double totalAmount = 0;
          LocalDate today = LocalDate.now();
         for (Deposit deposit : deposits) {
             long days = today.toEpochDay() –
               deposit.depositDate.toEpochDay();
             double months = (double) days / 30;
             double interestRate;
              if (months < 3) {
                 interestRate = 0.02;
              } else if (months < 6) {
                 interestRate = 0.03;
              } else {
                 interestRate = 0.04;
              }
              totalAmount += deposit.amount *
              (1 + interestRate * (months / 12));
         }
         return totalAmount;
     }
}
```

Figure 4. The codes generated by Generative AI for Requirement-2

The codes generated for Requirement-2 all seem to satisfy the given requirements. However, when running the test cases shown in Figure 5, they all output different result values.

```
@Test
public void testCalculateDeposit() {
    LocalDate now = LocalDate.now();
    Account account = new Account();
    double expDeposit1 = 1000 * (1+ 0.02*30/365);
    double expDeposit2 = 2000 * (1+ 0.03*120/365);
    double expDeposit3 = 3000 * (1+ 0.04*210/365);
    account.deposit(1000, now.minusDays(30));
    assertEquals(expDeposit1, account.calculateDeposit(), 0.01 );
    account.deposit(2000, now.minusDays(120));
    assertEquals(expDeposit1+expDeposit2, account.calculateDeposit(), 0.01 );
    account.deposit(3000, now.minusDays(210));
    assertEquals(expDeposit1+expDeposit2+expDeposit3, account.calculateDeposit(), 0.01 );
}
```

Figure 5. Test cases for calculateDeposit()

In the case of the code generated by ChatGPT, it output the same values as the programmer's expected values in the test cases. However, Copilot and Gemini output values different from the expected values, failing to pass the test function.

4. TDD Method with the Aid of Generative AI

As observed in Section 2, the codes generated by Generative AI often do not adequately provide the functionalities desired by the programmer for various reasons. Reviewing the code generated by Generative AI one by one to check if it deviates from the requirements significantly reduces the effectiveness of Generative AI.

A suitable method to solve this problem is TDD. TDD is a software development methodology that emphasizes writing tests before writing the actual code. TDD is presented as a disciplined approach to software development that can lead to more reliable, maintainable, and robust code [8].

Following TDD discipline, programmers first write test scenarios and test cases and then use Generative AI to generate source code that successfully passes these tests. The test cases written by the programmer can indirectly supplement the requirements for the source code.

Along with requirement-1, which did not sufficiently describe the programmer's intention, the test case shown in Figure 5 was input into the generative AI, and a request was made to create an Account class that can pass the given test case. The codes generated by the generative AI according to this request is as shown in Figure 6.

[ChatGPT]

```
public class Account {
    ...
    private List<Deposit> deposits = new
ArrayList<>();
    public void deposit(int amount, LocalDate
date) {
       deposits.add(new Deposit(amount, date));
    }
    public double calculateDeposit() {
        LocalDate now = LocalDate.now();
        double total = 0.0;
        for (Deposit deposit : deposits) {
            long daysBetween=ChronoUnit.DAYS
                .between(deposit.depositDate, now);
            double interestRate =
                    getInterestRate(daysBetween);
            total += deposit.amount * (1 +
              interestRate * daysBetween / 365.0);
        }
        return total;
    }
    private double getInterestRate(long
                                daysBetween) {
        if (daysBetween < 90) {
            return 0.02;
        } else if (daysBetween < 180) {
            return 0.03;
        } else {
            return 0.04;
        }
    }
}
```

[Copilot]

```
public class Account {
    ...
    private List<Deposit> deposits;
    public Account() {
        this.deposits = new ArrayList<>();
    }
    public void deposit(int amount, LocalDate
date) {
       deposits.add(new Deposit(amount, date));
    }
    public double calculateDeposit() {
        double total = 0.0;
        LocalDate now = LocalDate.now();
        for (Deposit deposit : deposits) {
            long days = ChronoUnit.DAYS
                    .between(deposit.date, now);
            double interestRate =
                        getInterestRate(days);
            double interest = deposit.amount *
                        (interestRate * days / 365);
            total += deposit.amount + interest;
        }
        return total;
    }
    private double getInterestRate(long days) {
        if (days < 90) {
            return 0.02;
        } else if (days < 180) {
            return 0.03;
        } else {
            return 0.04;
        }
    }
```

[Gemini]

```
public class Account {
    ...
    private List<Deposit> deposits = new
ArrayList<>();
    public void deposit(int amount, LocalDate
date) {
       deposits.add(new Deposit(amount, date));
    }
    public double calculateDeposit() {
        double totalDeposit = 0.0;
        LocalDate now = LocalDate.now();
        for (Deposit deposit : deposits) {
            long days = now.toEpochDay() –
                deposit.depositDate.toEpochDay();
            double interestRate =
                calculateInterestRate(days);
            totalDeposit += deposit.amount * (1 +
                    interestRate * days / 365);
        }
        return totalDeposit;
    }
    private double calculateInterestRate(long
days) {
        if (days < 90) {
            return 0.02;
        } else if (days < 180) {
            return 0.03;
        } else {
            return 0.04;
        }
    }
}
```

Figure 6. Source codes generated using Requirement-1 and test cases as input

After reviewing the three source codes, we found that they all showed similar forms, even though they were generated by different generative AIs. Even though the same requirement-1 was entered, unlike the codes shown in Figure 2, we can see that the codes are more suitable for the programmer's intention, such as allowing multiple deposits and calculating the annual interest rate as a percentage of the daily rate.

5. Comparative Analysis

We tested the output values of the calculateDeposit() function of the Account class generated using Requirement-1 as input, the Account class generated using Requirement-2 as input, and the Account class generated using Requirement-1 along with the test cases. The output values from the functions of each class are shown in Table 1. In the table, the shaded areas indicate cases where the output values differed from the expected test case values.

Table 1. Output values of calculateDeposit() generated by generative AI

Input to Generative AI	Deposit period (interest rate)	Principal	Expected output	ChatGPT's output	Copilot's output	Gemini's output
Requirement-1	1M (2%)	1000	1001.64	1020.0	1020.0	1020.0
	4M (3%)	2000	2019.72	2060.0	2060.0	2060.0
	7M (4%)	3000	3069.04	3180.0	3120.0	3120.0
	Total	7000	6090.41	6260.0	N/A	6240.0
Requirement-2	1M (2%)	1000	1001.64	1001.64	1000.0	1001.66
	4M (3%)	2000	2019.72	2019.72	2015.0	2020.16
	7M (4%)	3000	3069.04	3069.04	3060.0	3070.66
	Total	7000	6090.41	6090.41	6075.0	6092.50
Requirement-1 & Test cases	1M (2%)	1000	1001.64	1001.64	1001.64	1001.64
	4M (3%)	2000	2019.72	2019.72	2019.72	2019.72
	7M (4%)	3000	3069.04	3069.04	3069.04	3069.04
	Total	7000	6090.41	6090.41	6090.41	6090.41

The codes generated with Requirement-1 as input all output values different from the expected outputs. This is because Requirement-1 did not include detailed guidelines on interest calculation (annual interest rate, per-diem calculation, simple interest, etc.), and lacked specific information such as the ability to have multiple deposits and calculating 30 days as one month.

The codes generated with the supplemented Requirement-2 as input showed functionalities closer to the programmer's intent than the previous codes. ChatGPT output values identical to the expected values. ChatGPT's generated code can be seen as the closest to what the programmer intended. In the case of Gemini, despite stating in the requirements that interest is calculated on a per-diem basis, the generated code calculated interest on a monthly basis, outputting values smaller than expected. On the other hand, Copilot calculated interest based on the number of months considering one month as 30 days and treated a year as 360 days, resulting in values higher than expected.

Lastly, the codes generated by inputting Requirement-1 along with the test cases showed very similar codes across all three, and all output values identical to the expected values. It can be inferred that Generative AI analyzed the test cases and, in the process of satisfying them, obtained information omitted in Requirement-1, such as the interest rate period calculation, multiple deposit functionalities, and per-diem interest calculation, and used it in writing the Account class code.

We can conclude that if the design deliverables of a specific class have been produced during the software development cycle, then following TDD principles by first writing test scenarios and test cases, and subsequently utilizing Generative AI to assist in coding that class, proves to be a highly effective method of class implementation. The process of the TDD method applying Generative AI is as follows.

Step 1. Prepare Requirements for the Specific Class: Describe in detail the meanings and functionalities of the class name, member variables, and member functions derived from the design phase. Particularly, any rules or exceptions should be written in detail.

Step 2. Write Test Cases for the Class: Write various test cases to determine whether the member functions operate as intended by the programmer. You may also receive assistance from Generative AI when creating test cases.

Step 3. Review Test Cases: Check whether the test cases properly reflect the requirements and the programmer’s intent. If there are aspects that are not adequately represented, supplement the deliverables from Steps 1 and 2.

Step 4. Request Class Implementation from Generative AI: Input the requirements from Step 1 and the test cases from Step 2 into the Generative AI to request the implementation of the class.

Step 5. Verify Whether Test Cases Are Passed: Use the test cases from Step 2 to verify the class implemented in Step 4. If the tests are not passed, return to Step 1.

6. Conclusion

In this study, we explored the potential and effectiveness of integrating Generative AI into the Test-Driven Development (TDD) methodology. To reduce the burden of writing test cases inherent in traditional TDD and to efficiently generate code that meets programmers’ requirements, we introduced Generative AI into the development process. The experimental results demonstrated that providing test cases to Generative AI enabled the generation of code that accurately reflects the programmer's intent, positively impacting development efficiency and code quality.

In particular, we confirmed that the stage of writing test cases in TDD is crucial to compensate for the limitations of Generative AI, which may not fully understand the programmer's requirements when generating code automatically. This suggests that the synergy between Generative AI and TDD can minimize errors and security vulnerabilities that occur during the software development process.

Future research aims to expand the applicability to various programming languages and complex software systems. Additionally, we plan to explore ways to build a more efficient development process by combining improvements in Generative AI models with technologies for automatic test case generation. Such efforts are expected to make significant contributions to rapidly developing high-quality software.

Acknowledgement

This Research was supported by Seokyeong University in 2023.

References

[1] M. Chen, J. Tworek, H. Jun, Q. Yuan, H. P. de O. Pinto, J. Kaplan, and W. Zaremba, "Evaluating Large Language Models Trained on Code," arXiv preprint arXiv:2107.03374, 2021. DOI: https://doi.org/10.48550/arXiv.2107.03374

[2] A. Svyatkovskiy, S. Deng, S. Fu, and N. Sundaresan, "IntelliCode Compose: Code Generation Using Transformer," Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, pp. 1433-1443, 2020. DOI: https://doi.org/10.1145/3368089.3417058

[3] M. Tufano, C. Watson, G. Bavota, M. D. Penta, M. White, and D. Poshyvanyk, "Deep learning similarities from different representations of source code," 2018 IEEE/ACM 15th International Conference on Mining Software Repositories (MSR), pp. 542-553, 2018. DOI: https://dl.acm.org/doi/10.1145/3196398.3196431

[4] I. Goodfellow, J. Pouget-Abadie, M. Mirza, B. Xu, D. Warde-Farley, S. Ozair, A. Courville, and Y. Bengio, "Generative Adversarial Networks," Communications of the ACM, Volume 63, Issue 11, pp. 139-144, 2020. DOI: https://doi.org/10.1145/3422622

[5] T. B. Brown, B. Mann, N. Ryder, M. Subbiah, J. Kaplan, P. Dhariwal, and D. Amodei, "Language models are few-shot learners," Advances in Neural Information Processing Systems, Vol. 33, pp. 1877-1901, 2020. DOI: https://doi.org/10.48550/arXiv.2005.14165

[6] H. Pearce, B. Ahmad, B. Tan, B.D. Gavitt, and R. Karri, "Asleep at the Keyboard? Assessing the Security of GitHub Copilot's Code Contributions," In 2022 IEEE Symposium on Security and Privacy (SP), pp. 754-768, 2022. DOI: https://doi.org/10.1109/SP46214.2022.9833571

[7] N. Perry, M. Srivastava, D. Kumar, and D. Boneh, "Do Users Write More Insecure Code with AI Assistants?," CCS '23: Proc. of the 2023 ACM SIGSAC Conference on Computer and Communications Security, pp. 2785-2799, 2023. DOI: https://doi.org/10.1145/3576915.3623157

[8] K. Beck, Test-Driven Development: By Example. Addison-Wesley Professional, 2003.

[9] A. Svyatkovskiy, S. Deng, S. Fu, and N. Sundaresan, " Intellicode Compose: Code Generation Using Transformer, " Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, pp. 1433–1443, 2020. DOI: http://doi.org/10.1145/3368089.3417058

[10] J. Austin, A. Odena, M. Nye, M. Bosma, H. Michalewski, D. Dohan, E. Jiang, C. Cai, M. Terry, Q. Le, and C. Sutton, "Program Synthesis with Large Language Models, " arXiv preprint arXiv:2108.07732, 2021. DOI: https://doi.org/10.48550/arXiv.2108.07732

[11] Y. Tian, K. Pei, S. Jana, and B. Ray, "DeepTest: Automated Testing of Deep-Neural-Network-driven Autonomous Cars," Proceedings of the 40th International Conference on Software Engineering, pp. 303-314. 2021. DOI: https://doi.org/10.48550/arXiv.1708.08559

[12] H. Ayenew and M. Wagaw, "Software Test Case Generation Using Natural Language Processing (NLP): A Systematic Literature Review," Artificial Intelligence Evolution, pp. 1-10, 2024. DOI: https://doi.org/10.37256/aie.5120243220

[13] S. Bhatia, T. Gandhi, D. Kumar, and P. Jalote. "Unit Test Generation using Generative AI : A Comparative Performance Analysis of Autogeneration Tools," In Proceedings of the 1st International Workshop on Large Language Models for Code (LLM4Code '24). ACM, NY, USA, pp. 54–61, 2024. DOI:https://doi.org/10.1145/3643795.3648396

[14] M. Tufano, C. Watson, G. Bavota, M. Di Penta, M. White, and D. Poshyvanyk, "An Empirical Study on Learning Bug-Fixing Patches in the Wild via Neural Machine Translation," ACM Transactions on Software Engineering and Methodology, vol. 28, no. 4, pp. 1–29, 2019. DOI: https://doi.org/10.48550/arXiv.1812.08693

[15] M. Vasic, A. Kanade, P. Maniatis, D. Bieber, and R. Shingh, "Neural Program Repair by Jointly Learning to Localize and Repair," Proceedings of the 6th International Conference on Learning Representations, 2019. DOI:https://doi.org/10.48550/arXiv.1904.01720