Test Driven Generation:
Optimizing pAIr Programming By Conditioning Generated Code On Unit Tests
(Literature Review)

Matthew Billman
XCS224U / 70 Remsen St #7F, Brooklyn NY 11201
`mgbvox@gmail.com`
`https://github.com/mgbvox/tdg`

1 Introduction

In this literature review, I examine the state of AI-facilitated Pair Programming — pAIr Programming or AI-PP — contrasting it with traditional Pair Programming (PP). I introduce the Test-Driven Generation (TDG) framework, leveraging Test-Driven Development (TDD) principles to address the pitfalls observed in AI-PP and Traditional PP.

2 Traditional Pair Programming

“Two heads are better than one.” “Many hands make light work.” Commonsense aphorisms like these manifest in modern software design practice as Pair Programming (PP), a popular teamwork strategy currently seeing extensive use in the Agile (noa, 2024a) and Extreme Programming paradigms (noa, 2024b).

In traditional PP, one developer takes the role of Driver, focusing on low-level implementation, while the other takes the role of Navigator, focusing on high-level program design and architecture (Ma et al., 2023). These roles are switched often, with each developer contributing their expertise to solving the task. The process is hypothesized to lead to higher-quality output while encouraging knowledge sharing and growth between partners. However, the effectiveness of pair programming from a management perspective varies depending on your measurement criteria, and in general, the answer to whether a team should adopt the practice is “it depends” (Hannay et al., 2009).

3 From Pair Programming to pAIr Programming

With the advent of Large Language Models (LLMs) fine-tuned for code generation - OpenAI’s Codex (noa, b), Meta’s Code-LLAMA (noa, 2023), and GitHub’s Copilot (noa, a) to name a few - AI has started taking the driver’s seat in PP (referred to, punnily, as pAIr Programming, and referred to in this paper as AI-PP). The technology is still quite new, and studies on its effectiveness are limited (Ma et al., 2023), but the field is growing rapidly.

4 A Brief Summary of the Literature

A meta-analysis of traditional PP’s effectiveness (Hannay et al., 2009) indicated moderate improvements in the quality of code (defined as how well a programmed solution meets intended requirements or passes certain quality checks such as test cases), a moderate increase in task duration (defined as the time to complete a given task), and a moderate decrease in developer effort (defined as twice the duration of each individual in a pair). All observed effects were slight, and inter-study variance was high.

A study by GitHub, MIT, and Microsoft on GitHub Copilot’s impact on developer productivity (Peng et al., 2023) finds a 55.8% increase in developer productivity relative to a placebo group not given access to Copilot. This is an impressive increase, though it is important to note confounding factors. This study did not measure code quality, was geographically restricted to participants in India and Pakistan, and only surveyed Copilot’s performance in an educational setting on a fairly low-complexity problem. There are also conflicts of interest - Microsoft owns GitHub.

A meta-analysis of existing research into AI-PP (Ma et al., 2023) observed limited, mixed results in most development experience factors. The authors propose customizing AI pair programmer experience levels to mesh with the knowledge level of the human developer, maximizing learning while minimizing confusion. They are wary of AI over-helping junior-level developers, potentially leading to a loss of self-efficacy in non-pAIr scenarios and loss/lack of knowledge. The authors point out that further research is needed on how AI-PP is deployed to users - perhaps the current paradigm of tab completion in AI-PP Integrated Development

Environment (IDE) plugins is sub-optimal.

An independent study of Copilot (Bird et al., 2022) looked at community response to Copilot, how Copilot was being used, and concerns around its use. Often, Copilot kept users off other code question-answering forums like Stack Overflow, leading to more time spent in IDE; however, users indicated less understanding of how the code works. First-time users were likely to accept completions offered by Copilot but were found to wrestle with the resultant loss of autonomy. Finally, the frequency of completion acceptance was correlated with self-reported productivity. Concerns over security and accidental leakage of PII were expressed; while private information is claimed not to be used to generate code for other users, users have to opt out of data storage, and PII was still found to be leaked in some cases.

A study of code completion accuracy by LLMs (Liu et al.) introduced the new evaluation metrics EvalPlus and HumanEval^{+}, open sourced at `https://github.com/evalplus/evalplus`, as well as a python package for use in evaluating models on this dataset. This new benchmark exposes edge case flaws in generated code that were not accounted for on previous metrics (e.g., HumanEval (Chen et al., 2021)). The paper also introduces a means for automatic test augmentation, which was used to create these benchmarks.

5 Discussion

5.1 AI-PP: An Imperfect Alternative to Traditional PP

Traditional PP does appear to be effective for complex tasks when quality is prioritized but at a much higher cost in effort and duration. With the advent of LLMs, the question arises as to whether an AI pair programmer can address some of traditional PP's flaws.

AI-PP is quite young, and papers investigating its utility universally decry a lack of data for reliable inference of its import. Several trends have been consistently noted, however:

- Productivity in AI-PP does not appear to be as penalized as traditional PP.
- On the whole, AI-PP tends to lead to lower quality code (as measured by passing test cases) or lower efficiency code (as measured by superfluous generated lines) than PP, with the following caveats:
  - For low complexity cases or when in the hands of more Junior-level developers, quality, particularly code correctness, improves.
  - For high complexity cases or when in the hands of more Senior-level developers, efficiency improves.

5.2 The Shifting Role of Human Developers

Several sources have voiced concern about AI's pedagogical impact - junior-level developers are likely to learn less by offloading programming reasoning and logic to an AI service. Indeed, productivity must not come at the cost of diminishing workforce expertise. Conversely, the role of human developers in the AI-PP paradigm does seem to shift more towards a project-manager or architect-type role; it may be that human developers are not wholly replaced but rather that their roles shift to managing higher levels of abstraction while an AI (or several) does the work of implementation. Developers graduate from instruments in an orchestra to conductors.

5.3 Lack of Diversity in AI-PP Tooling

There is a lack of diversity in tools for AI-PP; a very high percentage of analyses and meta-analyses look at just two services, ChatGPT and Copilot, despite a proliferation of open and closed source code generation models (EvalPlus's leaderboard showcases at least 80). I suspect there is a translational barrier of entry here - developing a code LLM is one thing, but building an ergonomic plugin and getting it into widespread use by developers is something else entirely. This is confounded by the fact that one of the most popular IDEs today, VSCode, is owned by Microsoft - which runs GitHub Copilot and is a major stakeholder in OpenAI. There is likely an incentive here to keep VSCode code generation tooling somewhat of a walled garden.

6 Interlude: Test Driven Development

Test Driven Development (TDD) (Beck, 2022) is a stability-oriented development technique roughly outlined as follows:

1. Describe your problem in natural language.
2. Define the desired behavior of your code solution by writing a single, limited unit test.

3. Ensure the unit test fails.
4. Write just enough code to get the unit test passing and no more.
   - If necessary, identify a sub-problem and initiate a TDD cycle to solve it.
5. Refactor existing code if needed.
6. Formulate a new problem and repeat from 1.

TDD encourages simple, efficient design and inspires confidence in the software's functionality. When rigorously applied throughout the production cycle, it can lead to near-perfect test coverage, each test acting as a safety belt ensuring continued functionality of the system under test as more code is added and complexity grows. However, this comes at the cost of productivity; the cognitive overhead required to design efficient and distinct tests is high, and the amount of code one must write is at least doubled.

I hypothesize that TDD can provide additional structure and context to AI-PP by formalizing the Navigator role as code while reducing the cognitive load and productivity cost of TDD and traditional PP.

7 Future Work

7.1 TDG

Code quality has been shown to suffer in AI-PP (Ma et al., 2023). A common metric for code quality is whether or not generated code passes a test suite. Quality can be explicitly optimized by applying TDD to AI-PP, specifically by conditioning generated code on test cases included in the prompt and only accepting generated code snippets that pass these tests. I intend to develop a framework called Test Driven Generation (TDG), wherein a developer can define the external API of a code object (module, class, function) through unit tests, which are then used as metrics against which an LLM generates code.

7.2 Failure-Augmented TDG through Chain-of-Thought Reasoning

Most analytics of generated code quality are one-shot - a problem is posed, code is generated, and it either fails or passes the specified tests. If, instead, we captured failed test tracebacks and forced the generating LLM to reason about them before attempting to solve the problem again, I would expect a substantial gain in quality. This is a form of chain-of-thought generation, an approach that's been quite powerful in other contexts (Wei et al.).

7.3 Proposed Implementation

TDG would be developed as a pytest plugin. A python pseudocode example of the API can be found in Appendix A.

8 Time Permitting

8.1 Test Case Augmentation

EvalPlus's augmented test case generation is intriguing. I want to explore test-case augmentation for TDG, and their methodology seems a good place to start. It would be interesting for TDG to generate supporting unit tests in addition to end-user code.

8.2 Localized Models

Using proprietary closed-source models like GPT4 for code generation is hard to avoid—these models consistently outperform open-source models, if only by their sheer size. However, as the performance of open-source models increases, it is possible that locally running open-source models on a sufficiently powerful machine could become a competitive option, particularly when it comes to cost. While I expect early development of TDG will necessarily use GPT4, I would like to add compatibility for running models locally.

8.3 Freedom of Model Choice

There is a discrepancy between the number of code-generation services seeing widespread use and the number of code-generation models available online. Services offering code generation should provide the means to change which models are used to generate code easily; I would like to explore offering this as part of my service.

9 Conclusion

In late 2022, with the release of ChatGPT, LLMs exploded onto the scene, bringing a paradigm shift in how we all work. In nearly every industry, humans must now learn to work in tandem with AI; this is no more obvious than in software development. As AI becomes our omnipresent partner, we must explore optimal ways to balance the load between humans and AI, preserving as much creative autonomy as possible for developers while ensuring maximal quality of the end product. I look forward to exploring TDG as a solution to this problem.

Appendix A: TDG API

Development Environment

Assume we're developing a package called `my_package`. Here is a hypothetical Poetry package directory structure:

```
└─my_package
  ├─src
  │ └─my_package
  │   └─__init__.py
  ├─tests
  │ ├─__init__.py
  │ └─test_my_package.py
  ├─README.md
  ├─poetry.lock
  └─pyproject.toml
```

Simple Usage

Assume we want `my_package` to implement a `math` module that implements a factorial function. Using tdg, we define a test (or several) that outlines how factorial should behave:

```
# A pseudocode python sketch of the proposed TDG interface

import tdg
import my_package

# Decorate functions that require generation:
@tdg
def test_factorial_ints():
    # reference the function as if it exists
    factorial = my_package.math.factorial

    # define common behaviors
    assert factorial(3) == 3*2*1
    assert factorial(4) == 4*3*2*1
    with pytest.raises(ValueError):
        factorial(-1)
```

Generation flow specifics will be further outlined in the Experimental Protocol section. But in short, after executing `pytest tests` a new `math.py` module should be created at `src/my_package/math.py`, exposing a new function called `factorial` that passes the specified test, with 100% coverage as defined by coverage.py and full static typing as defined by mypy:

```
└─my_package
  └─src
    └─my_package
      ├─__init__.py
      └─math.py <- new
```

Continued next page

Potential contents of `math.py`:

```
def factorial(n: int) -> int:
if n < 0:
    raise ValueError("n␣must␣be␣>=␣0")
return 1 if n == 0 else n * factorial(n - 1)
```

References

a. GitHub Copilot · Your AI pair programmer.

b. OpenAI Codex.

2023. Introducing Code Llama, an AI Tool for Coding.

2024a. Agile software development. Page Version ID: 1212961679.

2024b. Extreme programming. Page Version ID: 1200985304.

Kent Beck. 2022. *Test Driven Development: By Example*. Addison-Wesley Professional. Google-Books-ID: zNnPEAAAQBAJ.

Christian Bird, Denae Ford, Thomas Zimmermann, Nicole Forsgren, Eirini Kalliamvakou, Travis Lowdermilk, and Idan Gazit. 2022. Taking Flight with Copilot: Early insights and opportunities of AI-powered pair-programming tools. *Queue*, 20(6):35–57.

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, Alex Ray, Raul Puri, Gretchen Krueger, Michael Petrov, Heidy Khlaaf, Girish Sastry, Pamela Mishkin, Brooke Chan, Scott Gray, Nick Ryder, Mikhail Pavlov, Alethea Power, Lukasz Kaiser, Mohammad Bavarian, Clemens Winter, Philippe Tillet, Felipe Petroski Such, Dave Cummings, Matthias Plappert, Fotios Chantzis, Elizabeth Barnes, Ariel Herbert-Voss, William Hebgen Guss, Alex Nichol, Alex Paino, Nikolas Tezak, Jie Tang, Igor Babuschkin, Suchir Balaji, Shantanu Jain, William Saunders, Christopher Hesse, Andrew N. Carr, Jan Leike, Josh Achiam, Vedant Misra, Evan Morikawa, Alec Radford, Matthew Knight, Miles Brundage, Mira Murati, Katie Mayer, Peter Welinder, Bob McGrew, Dario Amodei, Sam McCandlish, Ilya Sutskever, and Wojciech Zaremba. 2021. Evaluating Large Language Models Trained on Code. ArXiv:2107.03374 [cs].

Jo E. Hannay, Tore Dybå, Erik Arisholm, and Dag I.K. Sjøberg. 2009. The effectiveness of pair programming: A meta-analysis. *Information and Software Technology*, 51(7):1110–1122.

Jiawei Liu, Chunqiu Steven Xia, Yuyao Wang, and Lingming Zhang. Is Your Code Generated by ChatGPT Really Correct?

Qianou Ma, Tongshuang Wu, and Kenneth Koedinger. 2023. Is AI the better programming partner? Human-Human Pair Programming vs. Human-AI pAIr Programming. ArXiv:2306.05153 [cs].

Sida Peng, Eirini Kalliamvakou, Peter Cihon, and Mert Demirer. 2023. The Impact of AI on Developer Productivity: Evidence from GitHub Copilot. ArXiv:2302.06590 [cs].

Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Brian Ichter, Fei Xia, Ed H Chi, Quoc V Le, and Denny Zhou. Chain-of-Thought Prompting Elicits Reasoning in Large Language Models.