JavaTestGenie: A Command-Line Tool for AI-Driven Unit Test Generation

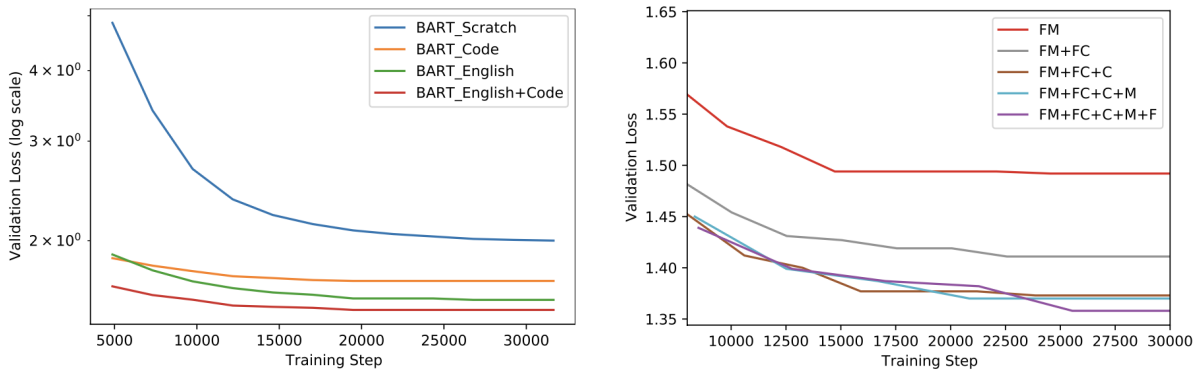
Bora Elci (be2246)
Columbia University
bora.elci@columbia.edu

1 Synopsis

We present JavaTestGenie, a command-line tool for AI-driven unit test generation. It is designed for developers who prefer using built-in text editors (like Vim, Nano, and Emacs) over IDEs. This lightweight solution, written in Python, requires minimal installation, allowing for easy portability and compatibility with multiple systems. It aims to improve code correctness and maintainability. This tool provides significant benefits to software engineers by saving time and reducing human error. Additionally, it is available as open-source software. From our experimental results, readers are expected to gain an understanding of the current state of test generation using publicly available models and datasets. There exist other tools for unit test generation. The authors of [1] describe two of them as follows.

> In particular, [Evosuite] introduces mutants into the software and attempts to generate assert statements able to kill these mutants... [Randoop] is another automated test generation tool that creates assertions with intelligent guessing.

In another paper, it is further explained that "A major weakness and criticism of these approaches is related to the poor readability and understandability of the generated test cases, which clearly appear as machine-generated code" [2]. Although tools like Evosuite and Randoop are useful, pre-trained models have been shown to outperform state-of-the-art (SOTA) techniques in many program understanding and generation tasks [3]. This opens the door for utilizing pre-trained models to generate unit tests. We employ Methods2Test, a publicly available dataset comprising 780,944 instances of test methods mapped to focal methods within classes. We recognize that the size of the dataset is very large to handle with our computational resources. Therefore, we sample a subset of the instances in our experiment. This dataset is organized into various context levels as shown in Figure 7. The authors of [2] conduct experiments with various levels of pre-training as well as focal context levels. In Figure 1a, we observe that pre-training with both English and Code datasets yields the best results. Therefore, we design our experiment to utilize those types of models. In Figure 1b, we observe that the combination of FM+FC+C offers the most favorable balance between validation loss and the number of input tokens. Consequently, we choose this level of context for our experiments. This selection corresponds to incorporating the focal method, class name, and constructors in the input, as illustrated in Figure 7.



(a) Pre-training levels [4] **(b) Focal context levels [4]**

Figure 1: Effects on model performance

2 Research Questions

2.1 RQ1: Which metrics are suitable for measuring model performance?

When examining the individual components of test generation, we find that creating accurate assert statements is particularly challenging. Often, unit tests can have multiple asserts, which makes it difficult to determine the main scope of the test. To address these challenges, previous work [5] adopts an approach where they discard methods with multiple assert statements from their dataset and define the focal method as "method invocation before (or within) the assert statement" [5]. On the other hand, the Methods2Test dataset is created with a better heuristic. The authors implement strategies for matching method names and analyzing which methods are called in a test. This challenge, along with similar ones, raises the question of what defines an "accurate" test. Does it involve calling the correct method, using the appropriate assert, or using correct object signatures? How about all of the above? Furthermore, does being compilable and running mean that a test is "accurate"?

Although we acknowledge that a better analysis is needed to determine the "accuracy" of a test generation model, previous work shows the use of BLEU score for evaluating model performance in program generation tasks. We begin by using BLEU score for the generations obtained from PLBART and GPT-3.5 models, but the results indicate poor performance. The PLBART cases do not seem meaningful while GPT-3.5 cases seem promising and a closer manual inspection indicates that they are accurate. However, the BLEU score is still only around 5%. We investigate how the performance of PLBART differs between BLEU and CodeBLEU scores and observe that CodeBLEU is often slightly higher (only a few points) than BLEU [6]. However, at this point, we begin suspecting that test generation can yield a different trend. We implement CodeBLEU ourselves, with guidance from its original paper [7]. Our implementation parses Java code to get abstract syntax trees (AST) and traverse them to extract tokens (ignoring whitespaces and comments). Then, we compute the BLEU score using the NLTK library based on the extracted tokens, which gives us CodeBLEU. We compare the two scores over 100 samples in Figure 2. CodeBLEU consistently captures when a test is good with a high score and identifies when it is bad by matching a low score, similar to what BLEU provides. This metric allows us to better understand which prediction is actually good or bad, and potentially discard & regenerate.

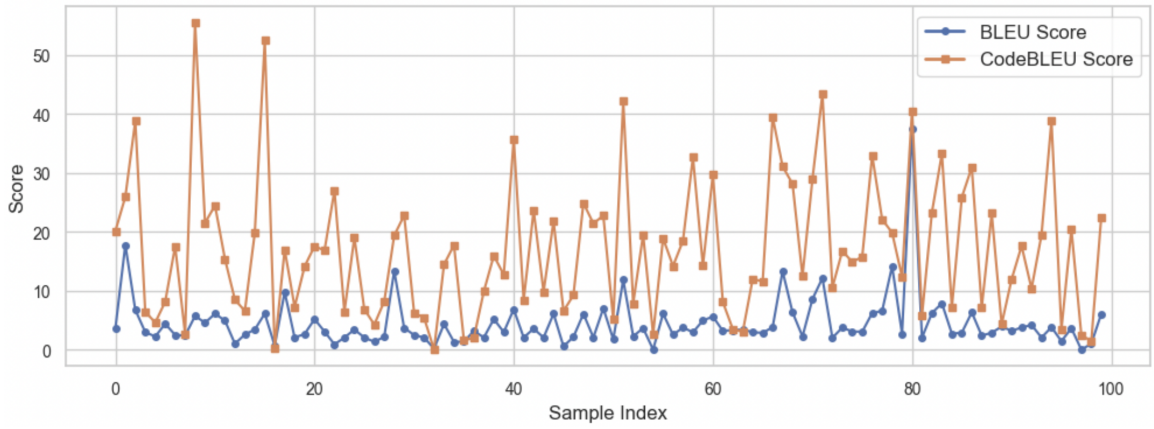


Figure 2: BLEU vs CodeBLEU over 100 samples with GPT3.5

Figure 3 shows a very simple example. The code snippets in the expected output and prediction are undeniably doing the same task, so the generated case is accurate. However, it only receives a BLEU score of around 7% while CodeBLEU is around 38% as shown in Table 1. Since CodeBLEU accounts for code-specific features such as variable names, it better captures the accuracy of the test. Hence, RQ1 is answered with CodeBLEU being the most suitable choice for evaluating the model performance in our experiment.

```
UiLesson implements Serializable {
  public boolean hasDescription() {
    return description != null && description.length() > 0;
  }
}
```

(a) Input

```
@Test public void testTESThasDescription() {
  sut.setDescription("some description");
  boolean actual = sut.hasDescription();
  assertTrue(actual);
}
```

(b) Target (expected output)

```
@Test public void testHasDescription() {
  UiLesson uiLesson = new UiLesson();
  uiLesson.setDescription("This is a description");
  assertTrue(uiLesson.hasDescription());
}
```

(c) Prediction

Figure 3: Example data

BLEU	CodeBLEU
7.52	38.82

Table 1: Scores for example data

2.2 RQ2: How does model choice affect performance?

The authors of [8] fine-tune and compare the performance of PLBART, CodeT5, CodeBERT, and CodeGPT. They find that PLBART outperforms the others. So, we employ this model and begin fine-tuning it. We use *uclanlp/plbart-base* [9] from Hugging Face's transformers library. We begin the training on our local computers, even with a GPU, but soon realize that much more computation power is needed to achieve substantial results. We create a script for running training jobs on AWS SageMaker, but the projected cost exceeds our budget. Finally, we switch to using

Google Colab. The runtime disconnects every few hours while the training definitely requires more than that. We implement checkpoints to frequently save and load intermediate model weights, along with optimizer, scaler, and scheduler states.

To speed up the training process, we implement mixed-precision training, meaning substituting lower-precision data types for some tensor values. To prevent underflows and overflows, we use gradient scaler. Unfortunately, the training time still takes around 30 minutes per epoch for 10,000 samples. We employ pre-tokenization to save tokenized inputs beforehand, but we observe that this causes a massive increase in the size of the input files. For example, the evaluation input file becomes 1.29 GB compared to its original size of 54.3 MB. Despite the increased file size, this strategy still appears promising for accelerating the training process. Finally, we plan to investigate freezing model parameters, as studies show that doing so results in similar performance compared to fine-tuning all the layers. At this point, due to limited time and computational resources, we shift our attention to bringing the command-line tool to an MVP stage first. Therefore, we decide to integrate the tool with OpenAI's GPT models and conduct experiments with them.

We write a script, ClassParser.py, adapted from a similar one used in Methods2Test for parsing Java classes via the tree-sitter library. However, the grammar file provided is not compatible with macOS, where we are running the experiment. Therefore, we compile the Java language file ourselves. The code and instructions to accomplish this can be found in our GitHub repository, although this is not needed to use our tool since we already distribute the grammar file within our releases. We make further adjustments to the scripts to make them compatible with macOS and plan to make an open-source contribution to the Methods2Test repository. Our adapted script brings any Java repository to the same FM+FC+C format that we selected to use. We create another script, evaluation.py, in a similar manner that allows us to evaluate OpenAI's GPT models over the dataset. We sample the first 100 inputs and corresponding outputs from the evaluation set to compare model performances.

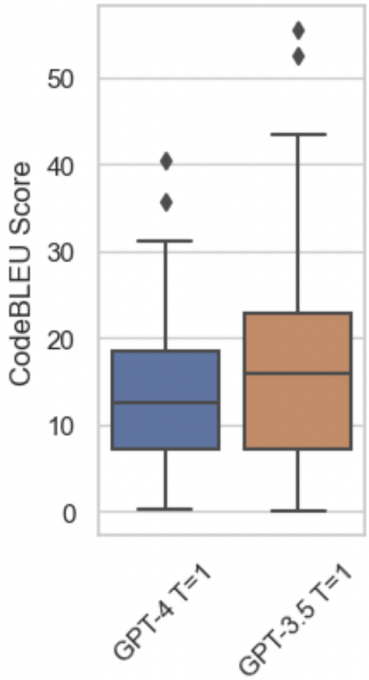


Figure 4: Effect of model choice on performance

We compare GPT-3.5 and GPT-4 with the default temperature setting of 1 and illustrate the results in Figure 4. Surprisingly, GPT3.5 outperforms the newer model by a difference of 3.43% CodeBLEU score as seen in Table 2. We suspect that this is due to the performance fluctuations between different runs since GPT-4 results show a smaller standard deviation. Despite this theory, we do not proceed with GPT-4 in the next experiment and use GPT3.5 instead because of the order of magnitude difference in price between them.

2.3 RQ3: How do hyperparameter settings affect performance?

While fine-tuning PLBART, we reach the conclusion that a learning rate of 1e-3 is too large because the evaluation loss stops converging around 1.14 after a few epochs while smaller learning rates risk taking too many epochs to converge. We find that 1e-4 is suitable in this context for making decent progress towards the loss function's minimum. We explore using a scheduler with a warm-up period to further address this problem. We also observe the effect of batch size for model training. A3Test [8], with a PLBART base, employs a batch size of 32 while the authors of AthenaTest [2] do not specify it. For our GPU, this batch size is unattainable. Therefore, we implement gradient accumulation over 16 steps to achieve an effective batch size of 32 while setting our actual batch size to 2. We see that many pre-trained models for code generation, along with AthenaTest and A3Test, use the Adam optimizer and achieve substantial performance, so we also select that one. Similarly, we use cross-entropy loss.

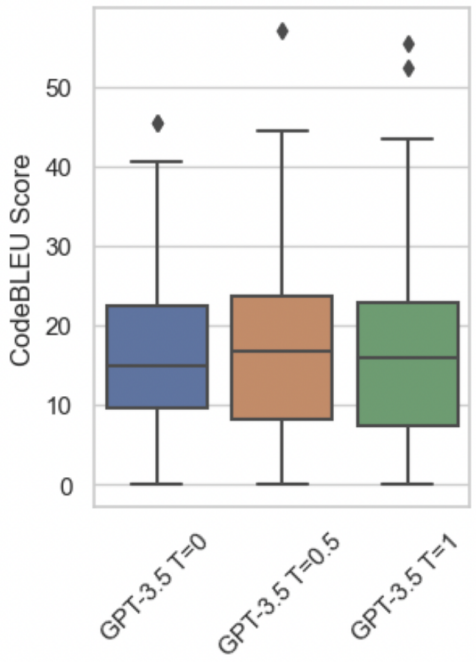


Figure 5: Effect of temperature on model performance

OpenAI's GPT models allow varying temperature and top-p parameters for queries, although it is recommended that only one of them and not both are adjusted in a single query. We select temperature and evaluate how it affects model performance. Figure 5 shows box plots for GPT3.5 models with three temperatures ranging between 0 and 1. A temperature of 0.5 corresponds to the best performance, but it is important to note that the description for temperature states that lower values "make it more focused and deterministic" [10]. This means that the values can significantly

differ between different runs and sample sets. GPT3.5 with a temperature of 0 has the lowest standard deviation and highest lower quartile. To achieve more stability, we select this model to integrate with our tool and evaluate it on 1,000 samples to verify its performance. Table 2 shows an overall summary, where the performance is shown to increase to the levels of other temperature settings. In fact, we see a mean CodeBLEU score of 18.02% and a median of 16.76%, the highest ones achieved in our experiment. We would like to note that sequentially running one thousand samples took around 1 hour and 50 minutes. Our script saved each generated test case, BLEU & CodeBLEU scores, and a performance summary to our repository's evaluation directory.

Model	Mean	Median	SD
GPT-4 T=1	13.63	12.46	8.55
GPT-3.5 T=0	16.42	14.79	9.94
GPT-3.5 T=0.5	16.80	16.73	11.32
GPT-3.5 T=1	17.10	15.89	11.81
GPT-3.5 T=0 S=1K	18.02	16.76	12.15

Table 2: CodeBLEU statistics for GPT models with different settings

2.4 RQ4: What are the benefits of the command-line tool and how can it be improved?

JavaTestGenie is integrated with GPT3.5 to generate complete test classes. However, it often requires the developers to edit various parts of the code such as import statements, model signatures, and (rarely) deleting inaccurately generated methods. We find that this process usually does not take a long time, only around 1-2 minutes per class, which makes this tool a very convenient choice for developers. We also provide an option for querying by each test method rather than the whole class, but this presents the issue of missing imports and package names, as well as lack of complete class definition. It writes the generated test methods back to back to the output file. The software engineer has to figure out what to import, add the class name, and initialize mocks to make it into a compilable file. Despite our clear instructions in the prompt, GPT models still included comments in the code 20% of the time. We use regex statements to mitigate this problem by removing comments. Similarly, we merge multiple space characters into one and remove new lines. We observe that sometimes the queries cannot be completed due to the limit of maximum number of tokens set at 2048. In those cases, we append two closing curly braces to the end for closing the last generated method and the class. It becomes the developer's responsibility to complete or delete the last method.

Although Methods2Test is the only viable dataset in the literature for training a model with the goal of unit test generation, we realized some of its limitations. Despite the rich metadata stored in JSON inputs, package names and import statements are not available. This makes it difficult to create a model that is capable of generating whole classes, although the package name can still be deduced from the source file. AAn important observation to note is that current models lack an understanding of the organizational hierarchy and the relationships between classes. In our experiments, we observed that GPT models make assumptions about signatures and sometimes get them wrong. Let's investigate a scenario where there is an Account model. The AccountService has a method calculateBalance(Account account) {return (account.cash - account.debt)}. We feed our model the class name that is AccountService and the method calculateBalance() in its entirety. How is the testCalculateBalance() in AccountServiceTest supposed to know 1) where to import Account model from and 2) what the signature is to create an instance of that class. Perhaps,

a separate model can be trained to operate on the generated test methods with the purpose of repairing signatures and adding correct imports. We believe this step is essential for creating a holistic tool. A similar issue presents itself in Figure 3, where the prediction involves creating an instance of the UiLesson class. However, the expected output most likely handles this in the setup method, so we recognize that model performance is impacted by the lack of setup knowledge.

We think about how this tool can be improved. Firstly, the generation speed can be increased by leveraging parallel queries. Although GPT3.5 has a limit on the number of tokens per minute, we believe it supports generating at least a few methods at the same time, split into different chat sessions. This would also work for classes, depending on their length. Moreover, several different OpenAI accounts could be used for load balancing. In fact, a backend could be integrated into the tool to allow this kind of functionality and manage API keys in a better way. Currently, our users have to supply their own API keys obtained from OpenAI's website. The future direction of this work should focus on fine-tuning a publicly available model such as PLBART. Deploying a model endpoint on AWS would allow scaling easily to generate entire test suites consisting of many classes and methods seamlessly. The costs associated with this approach should be studied and compared with the cost of using OpenAI models. Secondly, additional programming languages can be supported. We designed the tool specifically to be easily extensible to other languages. We developed reusable classes and methods for this purpose. The tree-sitter library, which we have a dependency on, provides parsers for 113 programming languages. Substituting a different grammar file and adjusting the code should allow both generating tests and evaluating results with CodeBLEU.

3 Deliverables

Developers can quickly download the tool with "pip install java-test-genie". They are expected to navigate to a Java project that they plan to run the tool for and add a configuration file describing the directories & files they want to include & exclude. Figure 6 shows an example configuration for a simple trading service [11], developed with the Spring Boot framework.

```
{
  "include": [
    {
      "parent_dir": "src/main/java/com/ase/restservice",
      "dir_names": ["service"],
      "file_names": []
    }
  ],
  "exclude": [
    {
      "parent_dir": "src/main/java/com/ase/restservice/service",
      "dir_names": ["interface"],
      "file_names": []
    }
  ]
}
```

Figure 6: Example configuration file

The tool can be run with the command "genie". It is open-sourced with an MIT license and the source code is available at [12]. The PyPi releases are distributed at [13]. We documented our

work to make it reproducible. The GitHub repository includes:

- Source code of the command-line tool
- README describing how to install and use it
- Generated tests for each experiment, performance summary, and BLEU & CodeBLEU scores
- Jupyter notebook of visualizations and statistical data presented in this paper
- Project milestones including proposal, progress report, and link to demo slides

4 Self-Evaluation

Through this project, we gained a deep understanding of fine-tuning NLP-PL models using the transformers library. We became familiar with various components that make up hyperparameters and learned about distributed training at scale while exploring AWS SageMaker. We also discovered how to reduce costs by implementing various strategies to accelerate training time and balance computational resources and time allocation to successfully achieve a project's scope. We gained experience working with publicly available datasets and libraries and repurposing scripts to create new software. Additionally, we learned the importance of documenting our work to ensure its reproducibility. Writing this paper taught us how to present experimental data and evaluate it effectively.

Overall, we believe that JavaTestGenie has made significant progress in utilizing AI-driven models for unit test generation. It has also achieved its goal of becoming a lightweight and portable command-line tool for developers who prefer using built-in text editors. The project has successfully addressed the research questions posed and experimented with different models and hyperparameters. However, there is still room for improvement. For instance, the tool could benefit from fine-tuning a publicly available model such as PLBART, which would allow for increased generation speed and scalability. Additionally, support for other programming languages could be added, making the tool more versatile and useful for a broader range of developers.

In terms of the research process, we recognize that the project has encountered some challenges, such as limited computational resources and time constraints. These factors have influenced the decision to shift focus from fine-tuning PLBART to using OpenAI's GPT models. Despite these challenges, the project has managed to produce valuable insights and results. We also initially planned to conduct a user study about the feasibility, adoption, and areas of improvement of the tool. Instead, we conducted an empirical evaluation of model performances to address our research questions. Our rationale was based on the expectation that a user study would be more informative if carried out after transitioning to an approach offering better model ownership, such as hosting a fine-tuned model on AWS, rather than relying on OpenAI.

In conclusion, we are satisfied with the progress made in this project and the development of JavaTestGenie as a practical and useful tool for software engineers. The research conducted has contributed valuable insights to the field of AI-driven unit test generation and has the potential for future improvements and developments.

References

[1] Cody Watson, Michele Tufano, Kevin Moran, Gabriele Bavota, and Denys Poshyvanyk. On learning meaningful assert statements for unit test cases. In *Proceedings of the ACM/IEEE*

42nd International Conference on Software Engineering, pages 1398–1409, 2020. https://arxiv.org/abs/2002.05800.

[2] Michele Tufano, Dawn Drain, Alexey Svyatkovskiy, Shao Kun Deng, and Neel Sundaresan. Unit test case generation with transformers and focal context. *arXiv preprint arXiv:2009.05617*, 2020. https://arxiv.org/abs/2009.05617.

[3] Zhengran Zeng, Hanzhuo Tan, Haotian Zhang, Jing Li, Yuqun Zhang, and Lingming Zhang. An extensive study on pre-trained models for program understanding and generation. In *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 39–51, 2022. https://dl.acm.org/doi/abs/10.1145/3533767.3534390.

[4] Michele Tufano, Shao Kun Deng, Neel Sundaresan, and Alexey Svyatkovskiy. Methods2test: A dataset of focal methods mapped to test cases. In *Proceedings of the 19th International Conference on Mining Software Repositories*, pages 299–303, 2022. https://dl.acm.org/doi/abs/10.1145/3524842.3528009.

[5] Michele Tufano, Dawn Drain, Alexey Svyatkovskiy, and Neel Sundaresan. Generating accurate assert statements for unit test cases using pretrained transformers. In *Proceedings of the 3rd ACM/IEEE International Conference on Automation of Software Test*, pages 54–64, 2022. https://dl.acm.org/doi/abs/10.1145/3524481.3527220.

[6] Wasi Uddin Ahmad, Saikat Chakraborty, Baishakhi Ray, and Kai-Wei Chang. Unified pre-training for program understanding and generation. *arXiv preprint arXiv:2103.06333*, 2021. https://arxiv.org/abs/2103.06333.

[7] Shuo Ren, Daya Guo, Shuai Lu, Long Zhou, Shujie Liu, Duyu Tang, Neel Sundaresan, Ming Zhou, Ambrosio Blanco, and Shuai Ma. Codebleu: a method for automatic evaluation of code synthesis. *arXiv preprint arXiv:2009.10297*, 2020. https://arxiv.org/abs/2009.10297.

[8] Saranya Alagarsamy, Chakkrit Tantithamthavorn, and Aldeida Aleti. A3test: Assertion-augmented automated test case generation. *arXiv preprint arXiv:2302.10352*, 2023. https://arxiv.org/abs/2302.10352.

[9] Uclanlp/plbart-base. https://huggingface.co/uclanlp/plbart-base.

[10] Openai api. https://platform.openai.com/docs/api-reference/chat.

[11] Bora Elci, Joey Lamborn, Raymond Wu, Yigit Ozulku, and Yigit Karakas. Kaiserschmarrn: Team repo for advanced software engineering (coms w4156) / fall 2022 / team kaiserschmarrn. GitHub. https://github.com/boraelci/kaiserschmarrn.

[12] Bora Elci. Javatestgenie: A command-line tool for ai-driven unit test generation. GitHub. https://github.com/boraelci/java-test-genie.

[13] Bora Elci. Javatestgenie: A command-line tool for ai-driven unit test generation. PyPI, 2021. https://pypi.org/project/java-test-genie/.

A Methods2Test Dataset

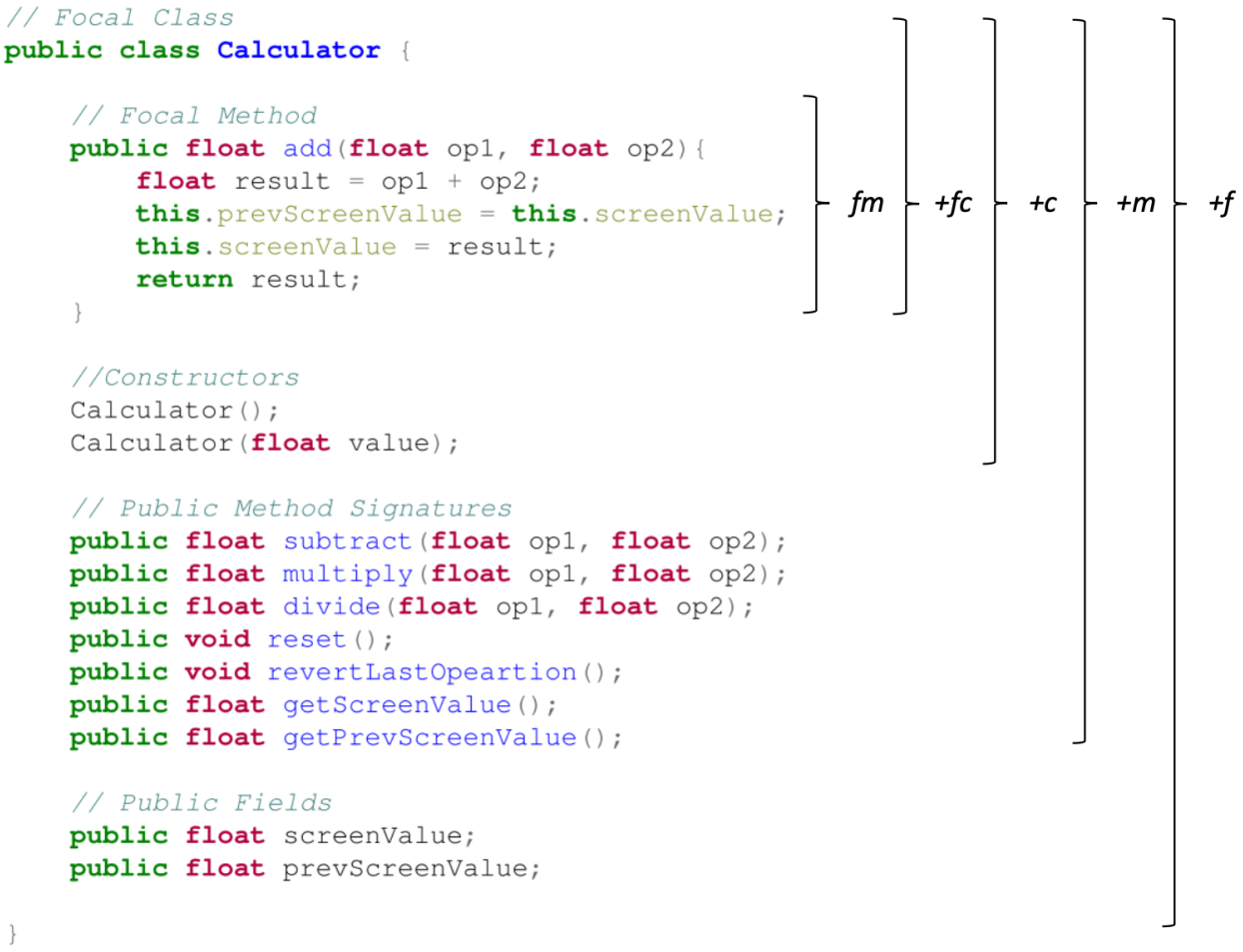


Figure 7: Focal context levels