ChatGPT as a Java Decompiler

Bradley McDanel
Franklin and Marshall College
bmcdanel@fandm.edu

Zhanhao Liu
Franklin and Marshall College
zliu@fandm.edu

Abstract

We propose a novel approach using instruction-tuned large language models (LLMs), such as ChatGPT, to automatically decompile entire Java classes. Our method relies only on a textual representation of the Java bytecode and corresponding unit tests generated from the bytecode. While no additional domain knowledge or fine-tuning is performed, we provide a single training example of this decompilation process in the model's prompt. To overcome both compilation errors and test failures, we use an iterative prompting approach. We find that ChatGPT-4 is able to generate more human-readable output than existing software-based decompilers while achieving slightly lower pass rates on unit tests. Source code and datasets are available at https://github.com/BradMcDanel/gpt-java-decompiler.

1 Introduction

Decompilation is the process of converting a binary machine language into a corresponding high-level language source code. This technique has numerous applications in fields such as rewriting legacy code, malware analysis, and software vulnerability repair. Unfortunately, existing software-based decompilers are time-consuming to develop and can generate source code that is hard for humans to understand (Hosseini and Dolan-Gavitt, 2022).

Neural Machine Translation (NMT) methods have been recently proposed as an alternative to conventional software solutions to translate between programming languages (e.g., C# to Java) (Wang et al., 2021; Szafraniec et al., 2022). NMT approaches have also been applied to program decompilation, where the source language is a compiled assembly/bytecode representation generated by a compiler and the target language is the original programming language.

The majority of NMT approaches focus on translating a single function with no side effects. We speculate this constraint is due in large part to the limited source and target lengths for Transformer-based translation models. For instance, CodeT5 (Wang et al., 2021) and CodeT5+ (Wang et al., 2023) typically use source and target sequence lengths 1024 or fewer tokens for a variety of code tasks such as writing and translation. This problem is exacerbated when the source sequence is an assembly/bytecode representation that can require 2-8x more tokens than their programming language counterpart.

In this work, we focus on the task of translating the Java bytecode of an entire class file to Java source code. This problem is significantly more challenging than translating a single function for multiple reasons. First, a class can contain tens of methods that, when tokenized, often exceed smaller token limits. Second, fields/methods defined earlier in a class are used in the implementation of other methods, making correct decompilation challenging due to long-term dependencies. Similarly, imported packages, generally defined at the top of the class, are also used throughout the file. Finally, there are many language-specific features that generate more rarely occurring patterns of bytecode (e.g., exceptions, static/final variables, multiple constructors).

With the recent addition of longer context windows for commercial instruction-tuned models[1], we believe it is possible to take on more challenging programming translation tasks, such as entire program decompilation.

To achieve accurate decompilation, we propose an iterative generation loop to guide the model around two types of error conditions: *compilation errors* and *unit test errors*. For compilation errors, we use ChatGPT to determine if the source of the error is due (1) early stopping (i.e., emitting a stop token in the middle of a large class) or (2) invalid Java code. In the case of early stopping, we simply

[1]As of 06/13/2023, OpenAI provides a 16k token GPT-3.5 model and a 8k and 32k token GPT-4 model.

instruct the model to continue generation, while invalid code leads to a retry. Once compiled, we apply unit tests to the code. If any unit test fails, we start over with a new generation pass up to a maximum attempt limit. Our iterative approach achieves a slightly lower test pass rate but leads to higher quality code evaluated by several similarity metrics compared to state-of-the-art software-based decompilers.

2 Related Work

2.1 Software-based Decompilers

Decompilation is the process of converting binary/assembly/bytecode generated by a compiler back to the original high-level language. Decompilation is often more difficult than compilation because much of the information in source file, such as variable names and original control flow, has been removed. Many techniques/heuristics have been developed over time to estimate the original source file with absence of complete information (Cifuentes and Gough, 1995).

We compare our approach against several open-source Java decompilers that have been in development over a long period of time (Benfield, 2022; skylot, 2022; mstrobel, 2022; Storyyeller, 2022; fesh0r, 2022). Harrand et al. provide a detailed analysis of the quality of the source code generated by these decompilers (Harrand et al., 2019). For simple classes, all decompilers are able to provide accurate and readable Java. However, for more complicated class methods (e.g., deeply nested code with complex control flow), they can generate code that, while functionally correct, is often convoluted and not in line with standard Java programming conventions, resulting in code that, though it may execute as intended, is hard for developers to read and understand, and may present challenges in maintenance and integration into existing projects.

2.2 NMT-based Decompilers

Katz et al. framed LLVM-IR (intermediate representation) to C decompilation as a translation problem using a recurrent neural network (Katz et al., 2018). This work constrained the problem to short code snippets (max of 112 binary tokens and 88 source code tokens). DIRE focused on the sub-problem of generating good names for identifiers for x86-64 binary to C decompilation (Lacomis et al., 2019). Coda developed an instruction-aware AST (for C programs) to restrict invalid token generation of an LSTM model (Fu et al., 2019). BTC developed a language agnostic decompiler to generate functions from assembly to many source languages (C/Go/Fortran/OCaml) using a single model (Hosseini and Dolan-Gavitt, 2022).

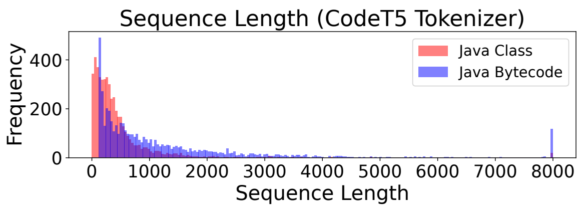


Figure 1: Token sequence length (using the CodeT5 Tokenizer) for 5000 Java classes (in red) and their corresponding bytecode assembly representation (in blue). Sequences longer than 8000 were truncated in the figure. The GPT-3 tokenizer achieves comparable results.

Compared to this prior work, we believe we are the first to tackle full Java class decompilation where both source and bytecode token lengths can be significantly longer than a 1024 token limit (up to 10k tokens per class).

3 Constructing a Java Bytecode Dataset

We extract Java classes from Github repositories indexed by Google BigQuery[2]. In order to generate bytecode, we must be able to compile these files with minimal configuration overhead. Therefore, we discard files with 3rd party imports (anything not starting with `import java.*`). Additionally, we discard files containing multiple classes.

After these preprocessing steps, we split the Java classes into a training and testing set with 150k classes and 20k classes, respectively. When building the training and testing set, we first divide at the Java project level, setting a project to be either in the training or testing set. Additionally, we discard any class that appears multiple times.

For each class, we used the Java 8 compiler to generate bytecode. This bytecode was then disassembled using Krakatau (Storyyeller, 2022) to achieve a human-readable bytecode representation. We use this disassembled bytecode representation as input to our NMT model. Figure 1 shows the sequence length of Java classes and disassembled bytecode representations after being tokenized with the CodeT5 tokenizer (Wang et al., 2021). For any given Java class, the bytecode is often 3-4x longer. Following the same approach as (Roziere et al.,

[2]`https://console.cloud.google.com/marketplace/details/github/github-repos`

Decompiler	pass(%)	chrF	ROUGE	CBS
Krakatau	88.33	0.72	0.71	0.90
Cfr	95.33	0.80	0.83	0.94
Procyon	94.00	0.83	0.85	0.94
Fernflower	95.67	0.78	0.83	0.94
GPT-3(16k)	89.00	0.85	0.78	0.91
GPT-4(8K)	92.33	0.87	0.86	0.94

Table 1: Decompiler evaluation. Pass rate (pass) is the percentage of decompiled classes that pass all tests. chrF, ROUGE (ROUGE-L), CodeBertScore (CBS) measure code similarity between the ground truth and the decompiled Java files.

2021), we generate unit tests for each Java class via fuzz testing using EvoSuite (Fraser and Arcuri, 2011) and keep test with a mutation score larger than 90%. Generating unit tests for all 170k classes took several days on a 32-core server.

While we use only a small subset of the test dataset in this work, we will release the entire dataset for future research into NMT-based Java decompilers.

4 ChatGPT as a Java Decompiler

4.1 Structuring the Prompt

Prompt engineering techniques (Wei et al., 2022; White et al., 2023) have recently shown that the quality of output generated by instruction-tuned LLMs can depend heavily on the structuring of the prompt input to the model. For the task of decompiling Java bytecode, we found it important to add a single training example of the decompilation process with a variety of edge cases critically important to improve the model's chance of correctly solving the task. In a zero-shot setting (with no sample given), the compilation success rate drops 30-40%.

Figure 2 shows the textual representation we use for an example class. See Appendix A for more details on the prompt used. Due to context window limitations, we could only fit a single sample, as it already has several thousand tokens. We found that not adding any

4.2 Iterative Prompting Methodology

Figure 3 presents an overview of our method for using LLMs like ChatGPT as a decompiler. A `test.class` (bytecode) file is converted into a human-readable disassembled text format using Krakatau and used by Evosuite to generate unit tests. This test sample (consisting of Java assembly and unit tests) is passed along with a single training example to be formatted as part of the prompt to ChatGPT.

The prompt is then used as part of an iterative prompting method that will attempt to generate valid Java code that passes all unit tests up to a maximum number of attempts. We define m as the maximum number of attempts allowed for either compilation or testing, A_t as the current test attempt, and A_c as the current compilation attempt for a test attempt. The A_c value is reset to 0 for each test attempt.

ChatGPT generates an output string that is interpreted as a `pred.java` file. We attempt to compile this file using the Java compiler. If the compilation fails, we use another instance of ChatGPT (without message history) to try and diagnose the cause of the failure. We find two general types of failure modes: (1) early stopping and (2) invalid code. Early stopping typically occurs after approximately 1000 tokens (regardless of how much code is left to be generated). We believe this is due to the typical lengths of messages being no more than 1000 tokens during the instruction-tuning process, which makes long generations (e.g., 5000 tokens) improbable to the model. To overcome early stopping, we instruct the model to continue generation. After each generation, we concatenate all prior messages and treat it as a single file. This process can be repeated multiple times up to a set number of compile attempts A_c. Alternatively, invalid code leads us to delete the message history and start over.

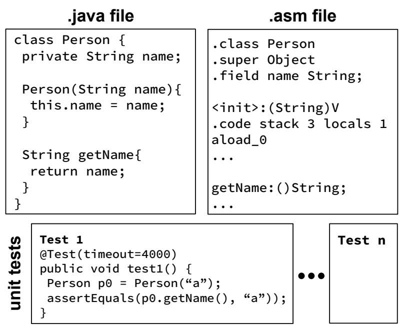
If compilation is successful, we pass the generated `pred.class` to the Evosuite test runner (along with the unit tests generated from the ground-truth `test.class`) to get the number of tests passed by the generated class. If one or more tests fail, we again delete the message history and start over. We do this until $A_t = m$ test attempts, at which point we give up. Once all tests pass, we immediately return the `pred.java` file that was successful.

5 Results

5.1 Evaluation Metrics

As mentioned before, we use unit tests to evaluate the functional correctness of the generated `pred.java` file. We define pass rate as the percentage of samples that pass all unit tests.

Additionally, we use several similarity metrics to estimate how similar the output of a given decom-



```
class Person {
 private String name;

 Person(String name){
  this.name = name;
 }

 String getName{
  return name;
 }
}
```

```
.class Person
.super Object
.field name String;

<init>:(String)V
.code stack 3 locals 1
aload_0
...

getName:()String;
...
```

```
Test 1
@Test(timeout=4000)
public void test1() {
 Person p0 = Person("a");
 assertEquals(p0.getName(), "a"));
}
```

```
Test n
```

Figure 2: An example of the text representations for the bytecode (.asm) and the unit tests used as part of the prompt to generate a source code (.java).

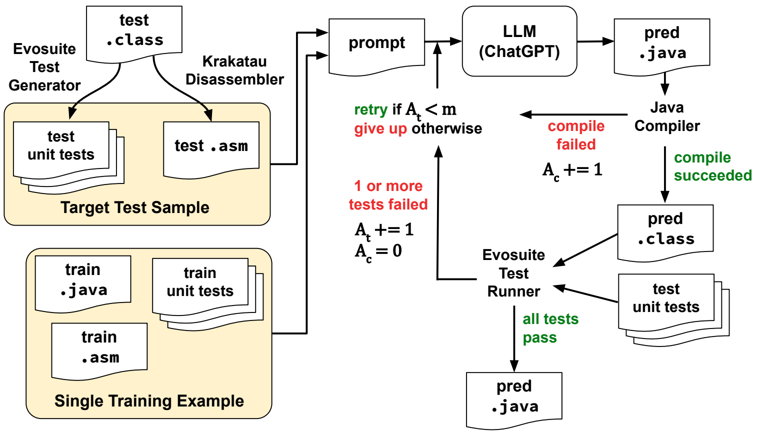


Figure 3: An overview of the proposed iterative prompting method.

piler is compared to the ground-truth Java source file. When measuring code quality, we use a subset of samples that have a pass rate of 1 for all decompilers. Otherwise, we can get skewed results when a decompiler might emit empty strings for samples it miscompiles. Following recommendations in (Evtikhiev et al., 2023) on the quality of code metrics, we use chrF (Popović, 2015) and ROUGE-L (Lin, 2004) to evaluate the decompilers. Additionally, we use CodeBertScore (Zhou et al., 2023) which has recently been shown to achieve stronger correlation with human preferences.

5.2 Comparison to Software Decompilers

We compare our iterative prompting methodology using ChatGPT (with a temperature of 1.0) against 4 software-based decompilers on 300 Java classes, which follow a sequence length distribution similar to Figure 1. Of the software-based decompiler, Fernflower achieves the best pass rate of 95.67%, while Cfr achieves better code quality in terms of both chrF and ROUGE-L. By comparison, our approach using GPT-4 achieves a pass rate of 92.33%. Additionally, it ties or outperforms all software-based decompilers on all code quality metrics. One of the major factors for this improvement is more descriptive variable names for local variables which are not provided in the Java bytecode. In Appendix B, we provide some qualitative comparisons of Java code produced by ChatGPT and the software-based decompilers. In Appendix C, we provide an analysis of why ChatGPT achieves a lower pass rate than software decompilers. In general, the length of a Java class correlates strongly with failure, implying either (1) difficulty with long-range attention between the bytecode in the prompt and the Java code much further away or (2) the model hitting the context limit making decompilation impossible due to losing the bytecode information.

Finally, since our approach detects failed unit tests automatically, it could always fall back to a software-based decompiler (e.g., Procyon) in the case of failures. This would lead to more readable decompiled Java code for the majority of samples (e.g., 92%) but still provide working decompiled code for as many samples as possible.

5.3 Impact of Iterative Prompting

Figure 4 shows the pass rate for the test set as the maximum number of compile attempts and test attempts are varied from 1 to 5. For GPT-3.5 and GPT-4, giving a single attempt for both compilation and passing all unit tests leads to a pass rate of only 65.67% and 82.33%, respectively. Especially for the weaker GPT-3 model, we see a dramatic improvement in performance as both the number of compile and test attempts are increased. This illustrates the usefulness of iterative prompting to improve the success of these stochastic models.

6 Conclusions

We describe an iterative prompting approach using instruction-tuned LLMs such as ChatGPT to perform decompilation of entire Java classes. Compared to existing software-based decompilers, our approach achieves a slightly lower pass rate but more human readable code. We hope the iterative prompting approach can be useful in other domains where an automatic feedback mechanism can be used to attempt additional generations.

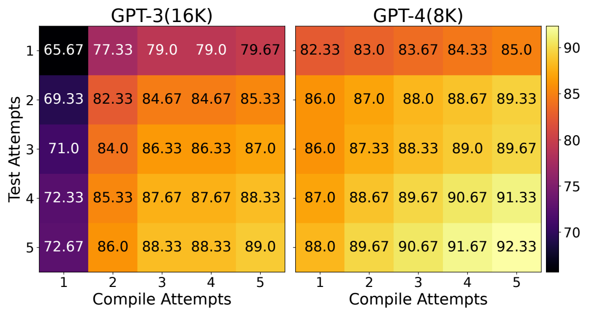


Figure 4: The pass rate (percentage of decompiled samples that pass all tests) for GPT-3 and GPT-4 as the number of compile attempts and test attempts is increased from 1 to 5.

Limitations

Cost Prohibitive Nature of Approach

While ChatGPT can successfully decompile the majority of samples, this often requires multiple attempts on long sequence lengths. This can easily lead to 30k-50k tokens being generated for a single sample. We estimate that we spent approximately $20 and $150 in API costs for GPT-3 and GPT-4, respectively, for only 300 samples. That being said, if an engineer was going to spend multiple hours trying to reverse engineer a Java class, they might benefit from an implementation that is easier to read even at a cost of around 50 cents.

Limited Test Set Size

While we collected a dataset with around 20k test samples, we only evaluated this approach on a random subset of 300 samples. This ties into the previous section on the costs of the approach. In principle, other than the cost, there is not preventing this approach from being applied to a much larger test set.

Lack of Comparison to Open-source Models

Currently, no open-source model trained on code offers a context window long enough to reliably perform decompilation of an entire Java class (especially if we include a training sample). For a shorter context length of 2K tokens, the proposed approach will not work, as our current prompt is already longer than that. We hope that this work provides evidence for the importance of more open-source LLMs trained on longer context windows.

Ethics Statement

The field of decompilation, and specifically the use of neural machine translation (NMT) models for decompilation, raises a number of ethical considerations. In this section, we will discuss some of the key concerns that arise in this context.

Generation of Nefarious or Invalid Code

One unique concern with NMT-based decompilation is that it may generate code that is invalid or malicious in ways that differ from conventional software-based decompilers. For example, a decompiler might produce code that appears syntactically correct, but that has unintended or malicious side effects when executed. This could be a result of the model failing to accurately understand the original code, or it could be due to the model intentionally feeding specific bytecode samples for the purpose of generating malicious code.

To mitigate this risk, it is important to make these types of issues known and to carefully evaluate the code generated by NMT-based decompilers and to use appropriate testing/validation techniques.

Software Reverse Engineering

Another ethical concern with NMT-based decompilation is the potential for it to be used for software reverse engineering. Reverse engineering is the process of taking apart a piece of software in order to understand how it works, or to identify vulnerabilities or other weaknesses. In some cases, reverse engineering may be done for legitimate purposes, such as to identify and fix security vulnerabilities or to develop compatibility or interoperability solutions. However, in other cases, it may be used for nefarious purposes, such as to steal intellectual property or to create competing software products.

While reverse engineering is possible using conventional software-based decompilers, the improved syntactic structure and clearer variables names of NMT-based decompilers like our approach may lower the barrier of entry for many programmers. This could lead to an increase in the number of individuals and organizations engaging in software reverse engineering, which could pose a threat to the intellectual property and competitive advantage of software companies.

To address these ethical concerns, it may be necessary to put measures in place to restrict the use of NMT-based decompilers to only those with legitimate purposes. This could include the implementation of licensing or access controls, as well as educational campaigns to raise awareness about the potential consequences of software reverse engineering. It may also be necessary to address any legal or regulatory issues surrounding the use of these tools, such as clarifying the boundaries of fair use and protecting the rights of software developers. Ultimately, the responsible use of NMT-based decompilers will require a balance between the benefits they offer and the potential risks they pose.

Security and Privacy

Finally, there are also potential security and privacy concerns related to NMT-based decompilation. Decompiling software may reveal sensitive information, such as hardcoded passwords or keys, which could be exploited by malicious actors. In addition, decompiling software may reveal vulnerabilities or weaknesses in the code, which could be exploited to gain unauthorized access or to disrupt the software's functionality. Again, while this is already possible with conventional decompilers, as NMT-based decompilers improve the readability of code, it could become a larger risk.

Summary

In summary, the development and use of NMT-based decompilers raises a number of ethical concerns that should be carefully considered. These include the potential for the generation of nefarious or invalid code, the use of decompilers for software reverse engineering, intellectual property concerns, and issues related to security and privacy. While these concerns are not unique to NMT-based decompilers, the improved capabilities of these tools may make them more appealing to those with malicious intent. Therefore, it is important for researchers and practitioners in this field to carefully consider these ethical implications and to take steps to minimize potential negative consequences. This may include carefully controlling access to these tools, implementing safeguards to prevent the generation of invalid or malicious code, and working with legal and policy experts to ensure that these tools are used responsibly and in compliance with relevant laws and regulations.

References

L. Benfield. 2022. Cfr - yet another java decompiler. Last accessed 25 November 2022.

Cristina Cifuentes and K John Gough. 1995. Decompilation of binary programs. *Software: Practice and Experience*, 25(7):811–829.

Mikhail Evtikhiev, Egor Bogomolov, Yaroslav Sokolov, and Timofey Bryksin. 2023. Out of the bleu: how should we assess quality of the code generation models? *Journal of Systems and Software*, 203:111741.

fesh0r. 2022. Fernflower. Last accessed 25 November 2022.

Gordon Fraser and Andrea Arcuri. 2011. Evosuite: automatic test suite generation for object-oriented software. In *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*, pages 416–419.

Cheng Fu, Huili Chen, Haolan Liu, Xinyun Chen, Yuandong Tian, Farinaz Koushanfar, and Jishen Zhao. 2019. Coda: An end-to-end neural program decompiler. *Advances in Neural Information Processing Systems*, 32.

Nicolas Harrand, César Soto-Valero, Martin Monperrus, and Benoit Baudry. 2019. The strengths and behavioral quirks of java bytecode decompilers. In *2019 19th International Working Conference on Source Code Analysis and Manipulation (SCAM)*, pages 92–102. IEEE.

Iman Hosseini and Brendan Dolan-Gavitt. 2022. Beyond the c: Retargetable decompilation using neural machine translation.

Deborah S. Katz, Jason Ruchti, and Eric Schulte. 2018. Using recurrent neural networks for decompilation. In *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 346–356.

Jeremy Lacomis, Pengcheng Yin, Edward Schwartz, Miltiadis Allamanis, Claire Le Goues, Graham Neubig, and Bogdan Vasilescu. 2019. Dire: A neural approach to decompiled identifier naming. In *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 628–639. IEEE.

Chin-Yew Lin. 2004. Rouge: A package for automatic evaluation of summaries. In *Text summarization branches out*, pages 74–81.

mstrobel. 2022. procyon. Last accessed 25 November 2022.

Maja Popović. 2015. chrf: character n-gram f-score for automatic mt evaluation. In *Proceedings of the tenth workshop on statistical machine translation*, pages 392–395.

Baptiste Roziere, Jie M Zhang, Francois Charton, Mark Harman, Gabriel Synnaeve, and Guillaume Lample. 2021. Leveraging automated unit tests for unsupervised code translation. *arXiv preprint arXiv:2110.06773*.

skylot. 2022. Jadx. Last accessed 25 November 2022.

Storyyeller. 2022. Krakatau. Last accessed 25 November 2022.

Marc Szafraniec, Baptiste Roziere, Hugh Leather Francois Charton, Patrick Labatut, and Gabriel Synnaeve. 2022. Code translation with compiler representations. *arXiv preprint arXiv:2207.03578*.

Yue Wang, Hung Le, Akhilesh Deepak Gotmare, Nghi D.Q. Bui, Junnan Li, and Steven C. H. Hoi. 2023. Codet5+: Open code large language models for code understanding and generation. *arXiv preprint*.

Yue Wang, Weishi Wang, Shafiq Joty, and Steven CH Hoi. 2021. Codet5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation. In *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing*, pages 8696–8708.

Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Ed Chi, Quoc Le, and Denny Zhou. 2022. Chain of thought prompting elicits reasoning in large language models. *arXiv preprint arXiv:2201.11903*.

Jules White, Quchen Fu, Sam Hays, Michael Sandborn, Carlos Olea, Henry Gilbert, Ashraf Elnashar, Jesse Spencer-Smith, and Douglas C Schmidt. 2023. A prompt pattern catalog to enhance prompt engineering with chatgpt. *arXiv preprint arXiv:2302.11382*.

Shuyan Zhou, Uri Alon, Sumit Agarwal, and Graham Neubig. 2023. Codebertscore: Evaluating code generation with pretrained models of code.

A Prompt

The prompt used in our iterative decompilation procedure is shown below.

```
**TASK**: Convert Java Assembly to a Complete Java Class

Your task is to transform the provided Java assembly and corresponding generated Java tests into a complete, syntactically valid Java class.

Please follow the guidelines carefully:

1. **Complete Class**: Ensure your result is a complete Java class, with a properly defined class structure. This can be spaced out across multiple messages if the class is extremely long. In this case, do not mention that you are doing so, simply assume the user will understand and will be able to piece together the class from the messages.
2. **Package Imports**: Incorporate any necessary package imports at the beginning of the class. If you're unsure, you may import any package you deem necessary.
3. **Javadoc Comments**: Every method in your class must be preceded by clear and concise Javadoc comments, outlining the method's purpose, parameters, and return values (if any).
4. **Variable Naming**: In cases where you need to infer variable names, make sure they are meaningful and self-explanatory, adhering to Java's naming conventions.
5. **Avoid Java Assembly Instructions**: Your output should be devoid of any Java assembly instructions such as ldc, invokevirtual, aload, etc. Remember, you're converting assembly code to high-level Java code.
6. **Valid Java Code**: Your final output should be a valid plain text Java code, adhering strictly to Java's syntax and semantic rules. It must be a complete, correct, and executable Java class.
7. **Edge Cases**: Your code must be able to handle edge cases such as empty input, null input, etc. appropriately. Your code will be tested with EvoSuite testing frameworks to ensure it matches exactly the provided Java assembly code for all possible inputs.

**Additional Information**:

- You can only respond with code as it will be compiled directly. Any written text will lead to a compilation error.
- Always initialize variables where necessary.
- Handle exceptions appropriately with try-catch blocks to avoid any unexpected runtime errors.
- Ensure appropriate access specifiers (public, private, protected) are used where necessary.
- You must not end the class early prior to all methods being defined.
  - Example: ... //other methods here ... }
  - This breaks the class structure and will lead to a compilation error.
- Make sure the main method is present if the class is intended to be executable.
- Regularly format and indent your code for better readability.
- You must always respond in plaintext. Do not respond in a codeblock.
- Do not generate any test code as it is already provided. Simply write the class code.

**Example**:
Example Java Assembly Input:
version 52 0
class public super TimeStat
super java/lang/Object
field private starts Ljava/util/Hashtable
field private times Ljava/util/Hashtable

method public <init> : ()V
code stack 3 locals 1
aload_0
invokespecial Method java/lang/Object <init> ()V
aload_0
new java/util/Hashtable
dup
invokespecial Method java/util/Hashtable <init> ()V
putfield Field TimeStat starts Ljava/util/Hashtable
aload_0
new java/util/Hashtable
dup
invokespecial Method java/util/Hashtable <init> ()V
putfield Field TimeStat times Ljava/util/Hashtable
aload_0
invokevirtual Method TimeStat reset ()V
return

end code
end method
... shortened for brevity ...
end code
end method
sourcefile 'TimeStat.java'
end class

Example Java Test Input:
import org.junit.Test;
import static org.junit.Assert.*;
import org.evosuite.runtime.EvoRunner;
import org.evosuite.runtime.EvoRunnerParameters;
import org.evosuite.runtime.System;
import org.junit.runner.RunWith;

@RunWith(EvoRunner.class) @EvoRunnerParameters(mockJVMNonDeterminism = true, useVFS = true, useVNET = true, resetStaticState = true, separateClassLoader = true, useJEE = true)
public class TimeStat_ESTest extends TimeStat_ESTest_scaffolding {

  @Test(timeout = 4000)
  public void test0()  throws Throwable  {
      TimeStat timeStat0 = new TimeStat();
      timeStat0.markStartTime("");
      timeStat0.markEndTime("");
      timeStat0.markStartTime("");
      timeStat0.getTotalMilliseconds("");
      timeStat0.markEndTime("");
      timeStat0.keys();
      String string0 = timeStat0.getAverageSummary();
      assertEquals("Average Summary:\n\n      averaged 0.0 milliseconds. (2 total measurements)\n\n", string0);
```

```
        timeStat0.markEndTime("");
        timeStat0.markStartTime("");
        double double0 = timeStat0.getTotalMilliseconds("");
        assertEquals(0.0, double0, 1.0E-4);
  }
... shortened for brevity ...
  }
}


Example Java Output:
import java.sql.*;
import java.util.*;

public class TimeStat {
  /** Hash that holds a single start time for any single key
      */
  private Hashtable starts = new Hashtable();

  /** Hash that holds all the measurement (duration) times
     for any single key */
  private Hashtable times = new Hashtable();

  /** Sole constructor */
  public TimeStat() {
    reset();
  }

  /** Resets all measurements by deleting all times. */
  public void reset() {
    starts = new Hashtable();
    times = new Hashtable();
  }

  /**
   * Marks the start time for a given measurement,
      identified by a key. Any previous start time in
   * the hash is overwritten.
   */
  public void markStartTime(String key) {
    starts.put(key, new Timestamp((new java.util.Date()).
      getTime()));
  }
... shortened for brevity ...
}

***INPUT JAVA ASSEMBLY CODE***
{input_java_assembly}

***INPUT JAVA TEST CODE***
{input_java_test_code}
```

B Examples

In this section, we present a qualitative analysis of decompiled samples that all software decompilers miscompiled (i.e., had a compilation error) that ChatGPT correctly compiled and passed all tests with. For brevity, we show only the Procyon decompiler's output for these classes. Refer to the supplemental material for more details.

B.1 OptionalsFromEmptyStreams

B.1.1 Procyon

```java
public static void main(final String[]
    array) {
     System.out.println(Stream.empty().
    findFirst());
     System.out.println(Stream.empty().
    findAny());
     System.out.println(Stream.empty().
    max((Comparator<? super Object>)
    String.CASE_INSENSITIVE_ORDER));
     System.out.println(Stream.empty().
    min((Comparator<? super Object>)
    String.CASE_INSENSITIVE_ORDER));
     System.out.println(Stream.empty().
    reduce((s, s2) -> s + s2));
     System.out.println(IntStream.empty()
    .average());
}
```

Listing 1: Pryocon

The above code results in a compilation error due to improper use of Java's generic types. Specifically, the 'max', 'min', and 'reduce' methods are trying to operate on an unknown type.

B.1.2 GPT-4

The GPT-4 model corrected the issue by specifying that the Stream was intended for Strings:

```java
public static void main(String[] args) {
     System.out.println(Stream.empty().
    findFirst());
     System.out.println(Stream.empty().
    findAny());
     System.out.println(Stream.<String>
    empty().max(String.
    CASE_INSENSITIVE_ORDER));
     System.out.println(Stream.<String>
    empty().min(String.
    CASE_INSENSITIVE_ORDER));
     System.out.println(Stream.<String>
    empty().reduce((s1, s2) -> s1 + s2))
    ;
     System.out.println(IntStream.empty()
    .average());
}
```

Listing 2: GPT-4 Solution

Here, GPT-4 correctly specifies that the empty Stream is intended for Strings resolving the previously erroneous operation issues.

B.2 UrlListFactoryBean

B.2.1 Procyon

```java
for (final URL url : this) {
    if (sb != null) {
        sb.append(',');
    } else {
        sb = new StringBuilder();
    }
    sb.append(url.toString());
}
```

Listing 3: Pryocon

This code fails to compile because it attempts to iterate over 'this' with a for-each loop, but 'this' refers to the current object which is not necessarily iterable, thus causing a type mismatch error.

B.2.2 GPT-4

The GPT-4 model corrected the issue by using an iterator:

```java
while (it.hasNext()) {
    final URL url = (URL) it.next();
    if(sb.length() > 0) {
        sb.append(',');
    }
    sb.append(url.toString());
}
```

Listing 4: GPT-4 Solution

Here, GPT-4 uses an Iterator (it) to loop through the collection. By casting the next object to a URL before assigning it to the url variable, it ensures type safety. The ‘final‘ keyword was added to the ‘url‘ variable declaration to maintain the immutability from the original code. This resolves the type mismatch problem in the original code.

B.3 Student

B.3.1 Procyon

```
public static void main(final String[] array) throws FileNotFoundException {
    final HashMap hashMap = new HashMap();
    final Scanner scanner = new Scanner(new File("out/production/Student/tutorials/othersingleexamples/students.txt"), "windows-1251");
    while (scanner.hasNext()) {
        final String[] split = scanner.nextLine().split("\\s*\\|\\s*");
        final String s = split[0];
        final String s2 = split[1];
        final String s3 = split[2];
    }
}
```

Listing 5: Pryocon

This code fails to compile because the FileNotFoundException, HashMap, and Scanner classes were not imported, causing a "cannot find symbol" error.

B.3.2 GPT-4

The GPT-4 model corrected the issue by using specific data types and classes, and assumed necessary imports:

```
public static void main(String[] args) throws FileNotFoundException {
    HashMap<String, ArrayList<Student>> map = new HashMap<>();
    Scanner scanner = new Scanner(new File("out/production/Student/tutorials/othersingleexamples/students.txt"), "windows-1251");

    while (scanner.hasNext()) {
        String line = scanner.nextLine();
        String[] parts = line.split("\\s*\\|\\s*");

        String firstName = parts[0];
        String lastName = parts[1];
        String course = parts[2];
    }
}
```

Listing 6: GPT-4 Solution

Here, GPT-4 specifies the types for the HashMap and the array from the split line. It also replaces the vague variable names with more descriptive ones. The final keyword was omitted to conform to usual Java conventions.

C Failure Modes

Figure 5 shows the cumulative pass rate of samples for each decompiler ordered by the number of tokens in each sample. We see that GPT-4 arguably has the highest pass rate for samples with less than 5000 tokens. As the samples get longer, all decompilers decrease in performance. However, the GPT models decrease at a steeper rate compared to the better software decompilers. We postulate that longer sequences are harder for instruction-tuned models that were mainly trained on shorter tasks to adapt to correctly. Perhaps this could be overcome with sufficient training on longer samples.

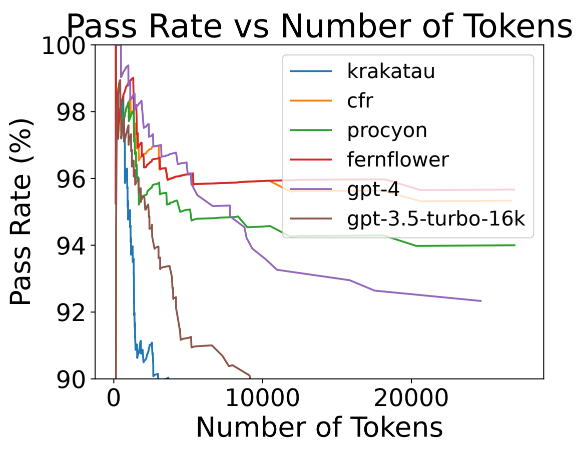


Figure 5: The pass rate (percentage of decompiled samples that pass all tests) for all models ordered by the number of tokens in the Java assembly code plus the generated Java file.