Naming Methods Automatically after Refactoring Operations

Juan Carlos Recio Abad,
Rubén Saborido Infantes*, and Jose Francisco Chicano García*

Universidad de Málaga
juan.carlos.recio.abad@gmail.com

Abstract. In Object-oriented Programming (OOP), the Cognitive Complexity (CC) of software is a metric of the difficulty associated with understanding and maintaining the source code. This is usually measured at the method level, taking into account the number of control flow sentences and their nesting level. One way to reduce the CC associated to a method is by extracting code into new methods without altering any existing functionality. However, this involves deciding on new names that are representative of the functionality of the extracted code. This work focuses on large language models to automate the process of assigning new methods names after refactoring operations in software projects. We use the OpenAI Chat API with the *textdavinci003* model in order to perform coding tasks. This work studies the capability of this technique for assigning names to new extracted methods during the evolution of a code base. Such evolution comprises continuous extraction operations to study how the method name semantics stability evolves. We found the precision of the model to be highly acceptable, achieving in many cases a level similar to that of a human. However, there are also a few cases in which it fails to provide appropriate names or does not even provide a name inside the indicated standards.

Keywords: Method naming · Extract method refactoring · Natural language processing

1 Introduction

The importance of extract method refactoring operations (among other refactoring techniques) lies on reducing the complexity of a software system. It involves identifying a portion of code that performs a distinct functionality and extracting it into a separate method with a descriptive name. This can improve the overall readability and maintainability of the code, as well as reduce the cognitive effort required to understand it. A novel cognitive complexity metric has been proposed and integrated in the well-known static code analysis tools SonarCloud and SonarQube. These are open-source platforms, respectively, for continuous inspection of code quality, which are extensively used by developers and software

* Advisors

factories today. This cognitive complexity metric has been designed as a measure of how hard the control flow of a method is to understand and maintain [4]. Thus, by reducing the cognitive complexity of a software system, developers can make the software system easier to maintain, modify, and enhance. This can result in a reduced need for costly and time-consuming maintenance, improved software quality, and increased customer satisfaction.

Technically, extract method refactoring operations lead to improve the structure and organization of code, what can lead to modularity and extensibility improvement. In a daily routine, developers usually use an Integrated Development Environment (IDE) that allows to perform extraction operations on selected portions of a code. However, for extract method operations, IDEs normally show up a new window with a default assigned name which does not correspond to what the new method does. It is often based on the original method plus some type of suffix, which gives the developer a new task to focus on and work with: assigning a good informative name for the new method.

After researching several approaches, we decided the most promising for our work is the use of the *textdavinci003* model. This model is described as "Can do any language task with better quality, longer output, and consistent instruction-following than the curie, babbage, or ada models". It is similar to its predecessor *textdavinci002* but it is trained using reinforcement learning instead of supervised training. Also supports inserting completions within text.

The purpose of this work is to find out how precise OpenAI Chat API with the *textdavinci003* model is to predict method names in comparison to the names written by humans. Given that refactorings do not alter functionality, it is expected that method names remain semantically identical at all times, irrespective of the extractions made. Consequently, we question whether the meaning of these names varies with the extractions. Thus, our research question is: "*Are there differences in the semantics of automatically generated method names depending on the number of extractions performed on the same method?*".

The remainder of this report is organized as follows. Sec. 2 summarizes related work. Sec. 3 introduces our methodology for making predictions and calculating similarities between method names. Sec. 4 presents the case of study and summarizes the experimental setting for evaluating our proposal. Sec. 5 provides the results of our experiments. Sec. 6 discusses the benefits of the proposed approach with respect to previous work. Sec. 7 discusses the threats to the validity of our work. Finally, Sec. 8 presents conclusions and future work.

2 Related Work

Recommending or assigning method names automatically is a topic that has been studied with more strength lately due to high rise of artificial intelligence techniques that boosts the results. Liu *et al.* proposed a novel method that is capable of recommending method names [11]. The core of their work is employing an attention mechanism to identify the most important parts of the code by considering both local context (such as the names of variables used in the

method) and global context (such as the names of classes and functions used in the whole codebase) to generate the most appropriate names. They do it by using all the semantic resources (including comments) within a given codebase. The outcome proved the technique outperformed existing state-of-the-art methods for method name recommendation. Approaches like *Code2vec* [2] capture the abstract syntax tree (AST) from the code and generate a fixed-length vector (embedding). Roughly, similar vectors would stand for similar codes, which eventually could be used to provide similar names for new methods. Another area of investigation for code generation that is feasible to infer method names is Code to Sequence (Code2Seq): this is a model that generates method names from their corresponding code snippets [3].

Chowdury *et al.* [6] studied and analyzed various metrics associated with method size and maintainability. Their outcome can be useful for software developers and teams in determining optimal method sizes for better code maintainability. It may provide guidelines or recommendations on preferred method lengths or suggest refactoring techniques to improve code quality and maintainability. This can contribute to the development of tools or approaches that automate the naming of methods based on their intended functionality or purpose.

The technique selected for our work is the OpenAI *textdavinci003* model accessed through the OpenAI Chat API [5]. It is a large-scale machine learning system that uses a Generative Pre-trained Transformer (GPT) natural language model to generate code [1]. It is capable of generating entire functions from natural language prompts and has been trained on a large corpus of code to generate high-quality code that is syntactically and semantically correct. The *textdavinci-003* model works behind the scenes of tools like *GitHub Copilot* [2] or ChatGPT (when prompted for code). The *textdavinci003* model is different from the rest of approaches because most of the reviewed approaches generate method names based on a given codebase or set of codebases, whereas *textdavinci003* generates complete functions based on natural language descriptions. These descriptions (or prompts) can be indefinitely tuned to refine the results as much as we want, providing a higher flexibility and shaping the desired output without changing the algorithm at all.

Feitelson *et al.* [7] diggs into the crucial aspect of selecting suitable names for variables, functions, classes, and other programming elements in software development. It delves into the multitude of factors and considerations that impact the decision-making process of developers when it comes to naming code entities. This is exactly what we want from *textdavinci003* model when predicting names for the extracted methods.

Our work is different from previous papers because *textdavinci003* model relies heavily on Natural Language Processing (NLP) prompts and precise in-

[1] Improving Language Understanding by Generative Pre-training: https://s3-us-west-2.amazonaws.com/openaiassets/researchcovers/languageunsupervised/language understanding paper. pdf

[2] GitHub Copilot: https://github.blog/2021-06-30-github-copilot-research-recitation

structions to predict method names. In contrast, previous works like Liu et al. [11] used various contexts to generalize method names. In our work, we focus only on comments and the specific details of a method, such as its signature, parameters, internal logic, and method calls.

3 Methodology

In order to find out how precise the *textdavinci003* model is to predict method names, we propose a methodology to study the evolution of methods names semantic after applying continuous extract method refactoring operations. The methodology can be decomposed as follows:

- **Input preparation**: Find and prepare a dataset with methods and their corresponding extraction versions of the every method.
- **Prediction of extracted code**: For each method we have to make predictions for all the extraction versions using the *textdavinci003* model. The predictions require specifying a natural language prompt for the model to be able to get the desired method name predictions.
- **Prediction of subject method name**: Once we receive the prediction for every extracted method, we make an additional prediction for the method including the new extracted method name.
- **Similarity computation**: After having received the prediction for the method including the extraction, we need to compute the similarity between semantics of the received method name prediction and the original method name. The output is the obtained similarity. This determines the quality of the extracted method name predictions.

The process can be visualized in Fig. 1. Output/similarities obtained are represented in red color.

In order to answer our research question, we study the evolution of methods names after single extract method refactoring operations. The process is as follows: Given a method, when a new method is extracted out from it, its name is predicted by the model based on the content of the extracted code. Then, a call to the new extracted method name replaces the extracted code inside the original code. So, how good is the predicted name of the extraction? Firstly, we will predict a new method name for the code that contains the new extracted code and we will refer to this as the subject method. Hence, the subject method can be seen as the evolution of the original method after applying extractions and predict names for them including with the new extractions included. Therefore, how good is the predicted name of extraction will depend on how close or far the new name is from the original method name. If they are similar, then the extracted prediction name was good as it reflects with a single method name the same intent as the whole extracted code. Reversely, if the predicted original method name differs too much from the original method name, the extracted method prediction was not accurate.

Let us see it with an illustrating example. We can start with the simple code shown in Listing 1.1, where we highlight in red the part to be extracted.

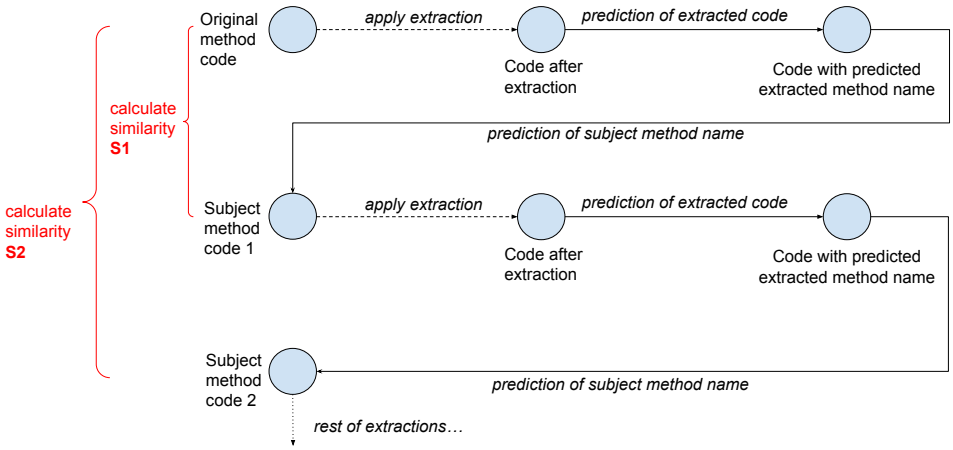

Fig. 1. Flow of extractions and predictions. Black arrows show the flow of mini steps followed to predict extracted method names. Red arrows show the computed similarity between the method without the extraction and the method with the applied extraction.

```
public Tax calculateTaxesPerMonth(Person person) {
    int baseYearlySalary = person.getYearlyBaseSalary();
    int baseMonthlySalary = baseYearlySalary() / 12;

    int bonusSalary = person.getJob().getYearlyAdditionalSalary();
    ...
    ...

    int bonusSalaryPerMonth = bonusSalary / 12;

    return new Tax(bonusSalaryPerMonth + baseMonthlySalary)
}
```

Listing 1.1. Code of a Java method used as illustrating example.

If we extract out a method by a refactoring operation, we might end up with the code in Listing 1.2.

```
public Tax calculateTaxesPerMonth(Person person) {
    int baseYearlySalary = person.getYearlyBaseSalary();
    int baseMonthlySalary = baseYearlySalary() / 12;

    int bonusSalaryPerMonth = extraction1(person);

    return new Tax(bonusSalaryPerMonth + baseMonthlySalary)
}
```

Listing 1.2. Original method with no named extraction.

Then, the challenge is to provide a good method name representative for the extracted code, such that the original method does not get its method name semantics altered. Let us assume, the method name for `extraction1` is set to

`getBonusSalaryMonthly` by then *textdavinci003* model, ending up with the code we show in Listing 1.3.

```
public Tax calculateTaxesPerMonth(Person person) {
    int baseYearlySalary = person.getYearlyBaseSalary();
    int baseMonthlySalary = baseYearlySalary() / 12;

    int bonusSalaryPerMonth = getBonusSalaryMonthly(person);

    return new Tax(bonusSalaryPerMonth + baseMonthlySalary)
}
```

Listing 1.3. Original method with predicted name extraction.

In order to determine the quality of the extracted method name, we predict a method name for the subject calling the extracted code. Then, we compare the new method name with the original one, obtaining a metric of how far the semantic vectors are. That is, we ask *textdavinci003* to provide a name for the method in Listing 1.4.

```
public Tax ???????????(Person person) {
    int baseYearlySalary = person.getYearlyBaseSalary();
    int baseMonthlySalary = baseYearlySalary() / 12;

    int bonusSalaryPerMonth = getBonusSalaryMonthly(person);

    return new Tax(bonusSalaryPerMonth + baseMonthlySalary)
}
```

Listing 1.4. Original method with the original name removed to ask to *textdavinci003* for a new name.

The prediction can be the method name `calculateTaxPerMonth` (see Listing 1.5), which semantically looks pretty similar to the original one `calculateTaxesPerMonth`.

```
public Tax calculateTaxPerMonth(Person person) {
    int baseYearlySalary = person.getYearlyBaseSalary();
    int baseMonthlySalary = baseYearlySalary() / 12;

    int bonusSalaryPerMonth = getBonusSalaryMonthly(person);

    return new Tax(bonusSalaryPerMonth + baseMonthlySalary)
}
```

Listing 1.5. Final version, with predicted extracted method and predicted original method name.

3.1 Similarity between method names

In order to analyse the results (predicted method names) and obtain some metrics to determine if the obtained methods have a similar semantic meaning, we need to be able to compute the similarity between method names. We found two promising NLP approaches for this: *Word2vec* [12] and *SpaCy* [9]. *Word2vec* is a technique used to generate distributed representations of words in a vector space. It is widely used for various NLP tasks such as sentiment analysis, named entity recognition, and machine translation. On the other hand, *SpaCy* is also used for various NLP tasks such as named entity recognition, part-of-speech tagging

and dependency parsing. After preliminary experimenting with both, we found out that *SpaCy* is extremely easy to use and provide excellent results for our purposes: comparing the semantics of two method names.

In order to compute the similarity between method names in *SpaCy*, we first transform the method names in phrases. Right after that, we encode both the subject and predicted names as vectors to calculate its distance. Technically, the similarity between two phrases or sentences can be measured using the cosine similarity. Cosine similarity is a technique used to measure the similarity between vectors or documents by considering their orientations in a multi-dimensional space. When two method names are encoded, each one is represented as a dense vector in a high-dimensional space *Joachims et al.* [10].

According to Java standard notation defined by Oracle, method names should be verbs, in mixed case with the first letter in lowercase, and the first letter of each internal word capitalized. Then, method name tokens can be split by spaces to be turned into sentences with a specific semantic meaning. Such a sentence is encoded into a vector using *SpaCy*. And those will be the vectors we compare. As an example, the methods *getPersonData* and *obtainUserInformation* would be encoded as the sentences "get person data" and "obtain user information".

It is important to note that we did not consider nested extractions. Nested extractions refer to the parts of code that we wish to extract out as new methods, that are contained within each other at different levels of hierarchy. We left nested extractions out of scope because they would add another level of complexity due to the following considerations:

1. The order in which the nested extractions are applied do matter. If we apply first the inner extractions, it is likely the result might be different because the predictions made by the system do not reflect exactly the same as the extracted code.
2. Nested extractions can form a complex hierarchy. Thus, establishing an order or strategy to iterate over the nested extractions and apply them would require further study on what is the best way to proceed.

The listings 1.6 shows method without nested extractions whereas the listings 1.7, 1.8 and 1.9 show an example with nested extractions.

```
calculateHypotenuse(double a, double b) {
    int sumSides = a + b;
    int squareSumSides = sumSides * sumSides;
    int squareSumSidesPerTwo = squareSumSides / 2;
    return squareSumSidesPerTwo;
}

calculateHypotenuse(double a, double b) {
    int sumSides = a + b;
    int squareSumSides = extraction1();
    int squareSumSidesPerTwo = extraction2();
    return squareSumSidesPerTwo;
}
```

Listing 1.6. Original method to illustrate some code base with no nesting extractions. It can be observed that do not interfere with each other.

Code in listing 1.7 shows a code with a simple hierarchy with two nested extractions.

```
public Path calculateMissionTrayectory() {
    Tracker tracker = TrackerFactory.getInstance();
    Place firstPlace = locationProvider.get(1);
    for (int i = 1; i < locationProvider.getPlaces().length) {
        Place targetPlace = locationProvider.getAt(i);
        tracker.setSpot(firstPlace, targetPlace);

        int x = i * targetPlace.getUx3();
        for (int j = 0; j < targetPlace.getReferences()){
            firstPlace.connect(targetPlace.getItem(x));
            targetPlace.assess(firstPlace, Times.Once);

            targetPlace.refresh(firstPlace.getId());
        }

        firstPlace.ensureLocation(targetPlace);
        firstPlace.refresh();
    }

    tracker.sync();
    return tracker.generatePath();
}
```

Listing 1.7. Original method with nested extractions. The extraction highlighted in red (outer extraction) also contains the lines from the extraction highlighted in green (inner extraction).

The listing 1.8 shows the method with inner nested extraction being applied, and listing 1.9 shows the method with outer nested extraction applied.

```
public Path calculateMissionTrayectory() {
    Tracker tracker = TrackerFactory.getInstance();
    Place firstPlace = locationProvider.get(1);
    for (int i = 1; i < locationProvider.getPlaces().length) {
        Place targetPlace = locationProvider.getAt(i);
        tracker.setSpot(firstPlace, targetPlace);

        int x = i * targetPlace.getUx3();
        for (int j = 0; j < targetPlace.getReferences()){
            connectAndAssses(firstPlace, targetPlace)
        }

        firstPlace.ensureLocation(targetPlace);
        firstPlace.refresh();
    }

    tracker.sync();
    return tracker.generatePath();
}
```

Listing 1.8. Original method after applying the inner extraction.

```
public Path calculateMissionTrayectory() {
    Tracker tracker = TrackerFactory.getInstance();
    Place firstPlace = locationProvider.get(1);
    for (int i = 1; i < locationProvider.getPlaces().length) {
        Place targetPlace = locationProvider.getAt(i);
        makeConnections(firstPlace, targetPlace);
    }

    tracker.sync();
    return tracker.generatePath();
```

```
}
```

Listing 1.9. Original method after applying the outter extraction.

3.2 Prompt optimization

We now present one of the biggest challenges we faced during the execution: *prompts*. Prompts in the *textdavinci003* model are so useful that allow you to obtain multiple code responses using natural language. However, the difficulty lies on getting precise responses using the appropriate tokens. In the context of the *textdavinci003* model, tokens refer to the basic units of text that the model processes. A token can be as short as a single character or as long as a word or even a phrase. These tokens are the input and output elements that the *textdavinci003* model uses to understand and generate text. The challenges we identified are the following:

1. Precision in the response. The answer we are looking for is only, and exactly only one single method name. Excluding any other additional information is essential to avoid any noise and complications when handling the study (removing not needed symbols, using regular expressions to extract the data we want, etc.). Additionally, the prompt has to be able to deliver enough information to make the *textdavinci003* model provide the right standard notation for the method name.
2. Brevity, concision and clarity. The amount of tokens we use in our prompt must be minimized. The longer the prompt, the more expensive the query is. Since we need shorter inputs, concision and clarity takes over as we have to effectively communicate the question with less content.
3. Limit of tokens for prediction. The maximum amount of tokens that can be used is 4096, this is a problem as it restricts the size of the methods we can use. We take already 64 tokens to compose the prompt using natural language, we just have 4032 for the code we send, that is the limit for our method contents including comments. A useful guideline to keep in mind is that, on average, one token represents approximately four characters of text in common English language.[3] As a result, this corresponds to roughly three-quarters of a word. Therefore, if we have a count of 100 tokens, it would be equivalent to approximately 75 words. For code though, it is a bit more expensive as camel case always produces additional tokens due to capital letters used in between. Fig. 2 illustrates this fact.
   We use two specific prompts: one to obtain the extracted method names and the other for predicting the original method name. These have been further refined after many iterations until we did not get a better behaviour of the system, that is, we did not get any other prompt which can improve the

[3] OpenAI: What are tokens and how to count them? https://help.openai.com/en/articles/4936856-what-are-tokens-and-how-to-count-them

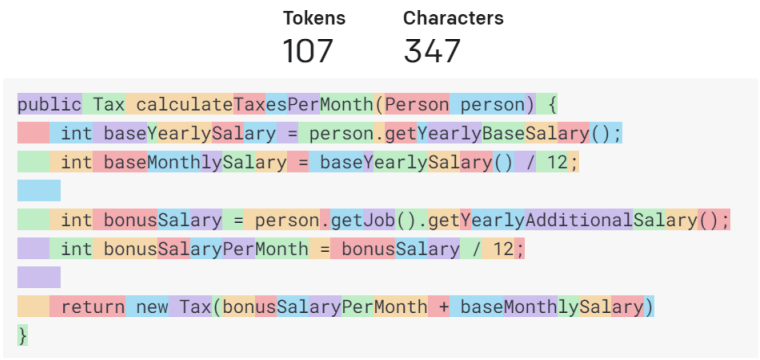

Fig. 2. Token consumption with code. Image auto generated with OpenAI token usage helper. Colors are just random and used to delimit different tokens.

overall similarity.

The content of the prompt used to predict extraction method names is the following:

> *Given the initial java code: {original_code} and its refactored version: {extraction_code} Can you give a name for the extracted method {method_to_compute}? Provide the method name without parentheses. Avoid the extract operation bias for the method name prediction. Use java notation for the method name (for example, do not use _).*

In this prompt we needed to explicitly tell the model to not be biased by the current method placeholder. For example, if we want to predict a proper name for a extraction called `getData_extraction1`, we do not want the model to be incorrectly biased by the tokens *get*, *Data* and *Extraction1*.
The prompt to predict the original method name based on an extraction is as follows:

> *Given the following java code, suggest a name for the method without taking into account the current method name and provide only the method name in the answer without any additional data, symbols or spaces: {code}*

Every time a new extraction prediction is done, we keep a modified version of the original method in which we replace the extraction placeholders (text in braces) by the predicted extractions (see Listings 1.8 and 1.9).

```
public Report obtainDiagnosticReport(double a, double b) {
    Process activeProcess = getCurrentProcessByUser(userId);
```

```
    String processId = activeProcess.getNumber() * 16 - 14;
    Calendar calendar = Calendar.getInstance();
    int dayOfMonth = calendar.get(Calendar.DAY_OF_MONTH);
    processId = processId + dayOfMonth.toString();

    Report report = createReport(userId);
    report.setDay(dayOfMonth);
    report.generate();
    report.setProcess(processId);

    return report;
}
```

Listing 1.10. Original method highlighting in red the extraction to be applied.

```
public Report obtainDiagnosticReport(double a, double b) {
    Process activeProcess = getCurrentProcessByUser(userId);

    String processId = getProcessId(userId);

    Report report = createReport(userId);
    report.setDay(dayOfMonth);
    report.generate();
    report.setProcess(processId);

    return report;
}
```

Listing 1.11. Copy of the method with one extraction applied, highlighting in red the next part of the code to be extracted.

Along with the prompt, the OpenAI Chat API allows the usage of an additional parameter called *temperature*. In the context of GPT NLP models, the *temperature* relates to the generation of text and the level of randomness or creativity in the model output. In a GPT NLP model, the softmax function is employed to transform the model predictions into a probability distribution encompassing the potential next words. The shape of this distribution is controlled by the temperature. When the temperature value is higher, such as 1.0, the distribution becomes more widely dispersed, leading to increased randomness in the generated text. Conversely, a lower temperature value, like 0.5, causes the distribution to be more concentrated and yields text that is more focused and deterministic [8].

4 Experiment

We have performed a study of the applicability of *textdavinci003* model to measure its performance predicting names for new methods that emerge from extract method refactoring operations.

In order to validate the proposed approach, we generated a dataset by processing the extract method refactoring operations obtained in a previous work [13] for a diverse set of 10 open-source Java projects: two popular frameworks for multi-objective optimization, five platform components to accelerate the development of smart solutions, and three popular open-source projects with more

than 10,000 stars and forked more than 900 times in GitHub. These extract method refactoring operations reduce the cognitive complexity of methods when applied. Note that, in some cases, a method requires a sequence of code extractions in order to reduce its cognitive complexity. The processing for generating our dataset is as follows. We select those methods which require more than one extract method refactoring operations to reduce their cognitive complexity (210 methods). Next, we filter all the methods that contain nested extractions. Then, for each method, we create a file with the original source code of the method before any code extraction. Finally, for each code extraction we create a text file containing the resulting refactored method after such extraction. Thus, if we work with a method with N extractions, we end up with N+1 files for each method, N of them containing the modifications to the initial method after each code extraction. From the initial 210 methods, our dataset finally contains code extractions for 115 methods belonging to 8 different projects. The resulting dataset is available in our replication package. It is important to note that extract method refactoring operations do not intersect with the other extractions, and there is not nesting among the extractions.

The experiment has been run in Windows 10, 64bits operating system, running on top of an Intel(R) Core(TM) i7, 2.50Ghz CPU and 8GB of RAM with Python 3. The usage of the OpenAI Chat API allows an additional parameter *temperature*. We set the temperature for the *textdavinci-003* model prompts as zero, as we want the maximum possible control and determinism we can get in the study.

The source code for replication purposes is available in our replication package [1]. It includes:

1. Dataset with JSON files for all the methods and their extractions. Each method has a JSON file with the original method content and one JSON file per extraction.
2. Python script to run the experiment to make predictions.
3. Python script to make REST API calls to the *textdavinci-003* model, including all the necessary prompts and parameters. The prompts are versioned in the previous Python script and it can be parameterized to use different versions and add new prompts to use. The current script contains four versions we used while experimenting. We found out the fourth version is the most accurate. Regarding the temperature of the input: this is a parameter that affects the randomness of the generated output. A higher temperature value, such as 0.85 or 1.0, tends to produce more diverse and creative responses with a certain level of randomness. On the other hand, a lower temperature value, such as 0.2 or 0.5, makes the output more focused and deterministic, with less randomness. We explicitly set it to zero to always obtain the most precise result not giving chance to creativity that could ruin the precision we look for method names with strict constrains. We also need to use an authorized API Key that allows making calls to the REST API.
4. Output folder with all the results obtained from the experiments as JSON files. Each method corresponds with a JSON output file.

5. Python script to display the graphs with the results using the library `matplotlib`.

Python Implementation

The implementation for the experiment is fully done using Python. The main script of the experiment is in charge of guiding all the steps of the experiment defined in the methodology. The functions to make HTTP calls and compute similarity between method names are in additional Python files.

All the methods that are part of the experiment are stored in a Java file each, along with a Java file for each extraction related to the method. The methods are gathered from multiple projects and classes and the implementation organises the methods using the following hierarchy:
`<top-level package>.<folder>.<nested package>.<class>.<method>`

If we analyze the method
`MOEA.src.org.moeaframework.Instrumenter.instrument`, this is structured as follows:

1. MOEA.src.org.moeaframework.Instrumenter.instrument.Original.java: File that contains the original method.
2. MOEA.src.org.moeaframework.Instrumenter.instrument.Extraction1.java: File that contains the method with one extraction applied.
3. MOEA.src.org.moeaframework.Instrumenter.instrument.Extraction2.java: File that contains the method with two extractions applied.

In addition to the main script, we have a Python script that selects only the methods that do not have nested extractions. There is a separate folder only with these methods. Every file in the new separated folder is iterated to compose a Python object to store the hierarchy for project, classes and methods. This structure will also include details to store the results of the experiment.

Reading and storing the methods from the input files to the structured Python object can be considered as the setup for the experiment, so we can start it considering the following structure: a project has a number of classes, a class has a number of methods, and a method has both an original implementation and a number of extractions. Thus, for every method, we iterate each extraction and make a prediction for its extraction method name. Every time we obtain an extracted method name, we also predict the name for the subject method extraction. Once the prediction is made, we compute the similarity between the new subject method name and the original method name.

After each method iteration, a new output file for the method is created. This contains all the relevant information for that method:

- Extraction method names predictions.
- Original method names predictions.
- Similarities between the original method name and the predicted original method names for the subject method.

– Updated versions of the code after having applied the extractions.

The output then will consist of the same number of files as the method in the experiment.

5 Results

The experiment has been performed over 115 methods, each one with their extractions. The average similarity obtained for the whole dataset does not seem to differ at all after applying several extractions which is a good indication that the prediction for the extractions represent very precisely the functionality. Table 1 shows the results obtained from the experiment based on the number of extractions applied. On average, methods exhibit a similarity of approximately 0.81 when considering one to four extractions, as it can be observed in Table 1. Besides, there are some methods in which the similarity obtained is lower than 0.5.

Table 1. It shows the average similarities (second column), the number of samples with similarities under 0.5 (third column) and over 0.5 (fourth column). The fifth column shows the amount of samples that at least contain the applied number of extractions from first column. Some methods are included in some categories, as for instance, from the 111 methods with two extractions, only 27 of them have 3 extractions.

Extractions	Avg. Simil.	Nb. Simil. < 0.5	Nb. Simil. > 0.5	No. Samples
1	0.832	8	103	111 (96.0%)
2	0.805	12	99	111 (96.0%)
3	0.804	2	25	27 (23.0%)
4	0.839	1	7	8 (6.0%)
5	1.000	0	1	1 (0.8%)
6	1.000	0	1	1 (0.8%)

Table 2 shows some particular cases from the experiments classified in four different types. Type one represents an average method from the dataset with a similarity rounding 0.8. Type two represents a method with perfect extracted method name predictions. Type three represents a method which has a problem in some of the extracted method prediction (it can be observed the suffix `Extraction2`). Type four shows some degradation in the similarity due to abbreviations in the original method name (word `mod`).

Figures 3-6 illustrate the similarity obtained after one to four code extractions, respectively. The average similarity is slightly above 0.8 for all methods. Only 2% of the cases obtain a similarity equals to zero. 9% of the methods obtain a similarity less than 0.5 for one and two extractions. As long as we increase the number of extractions, the similarity begins to be slightly more precise.

Table 2. Representative methods of the dataset with the corresponding results of the experiments.

Method type	Original method name	Extracted method predictions	Original method name prediction	Similarities
1	`updateField`	`getStringValue`	`updateFieldValues`	0.872
		`updateLongField`	`updateFieldValues`	0.872
		`updateIntegerField`	`updateFields`	0.905
		`updateWordField`	`updateFieldValue`	0.832
2	`search`	`searchBackward`	`search`	1.000
		`searchForward`	`search`	1.000
3	`getRemediationActionForLeaf`	`getFirewallRule`	`getRemediationActionForLeaf`	1.000
		`getRemediationActionForLeafExtraction`	`getRemediationActionForLeaf`	1.000
		`getRemediationActionForLeafVulnerability`	`getRemediationAction`	0.848
4	`modDetailFeature`	`modDetailFeatureAuditLog`	`modifyInsertMap`	0.576
		`modDetailFeatureAuditLog`	`modifyDetailFeature`	0.842
		`modDetailFeatureErrorHandler`	`modifyMapInsertMap`	0.541
		`modDetailFeatureErrorHandler`	`modifyOrInsertMap`	0.466

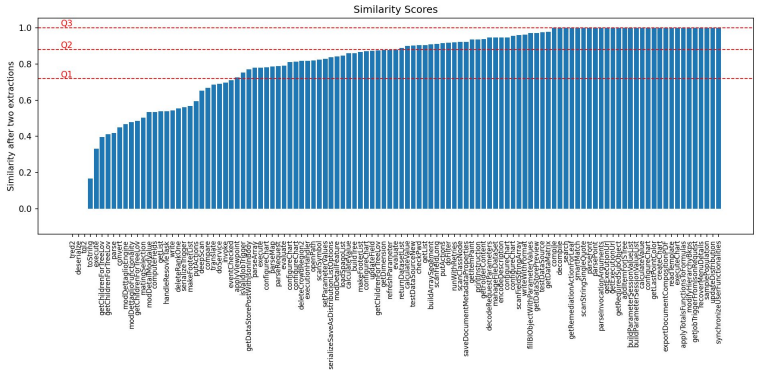

Fig. 3. Similarity scores after predicting one extraction method name. Red lines show percentiles Q1, Q2 and Q3. It can be observed how 25% percentile is around similarity=0.8. The similarities are arranged in ascending order.

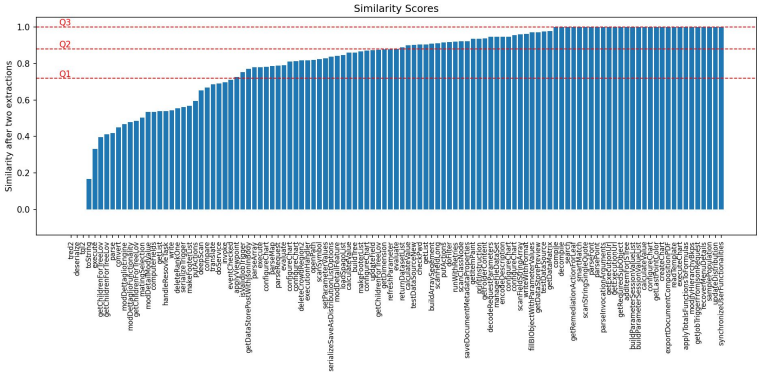

Fig. 4. Similarity scores after predicting two extraction method names. Red lines show quartiles Q1, Q2 and Q3. It can be observed how Q1 is around similarity=0.8. The similarities are arranged in ascending order.

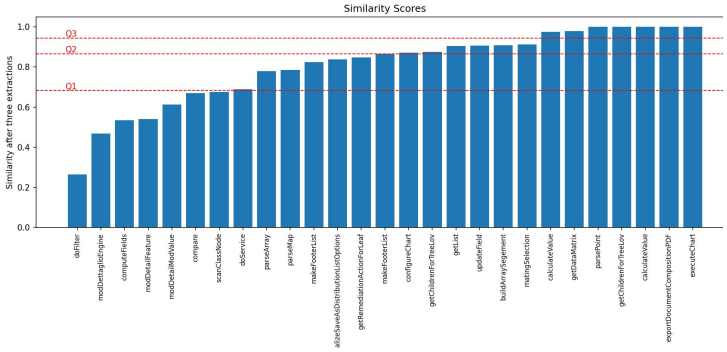

Fig. 5. Similarity scores after predicting three extraction method names. Red lines show percentiles Q1, Q2 and Q3. It can be observed how Q1 is around similarity=0.8. The similarities are arranged in ascending order.

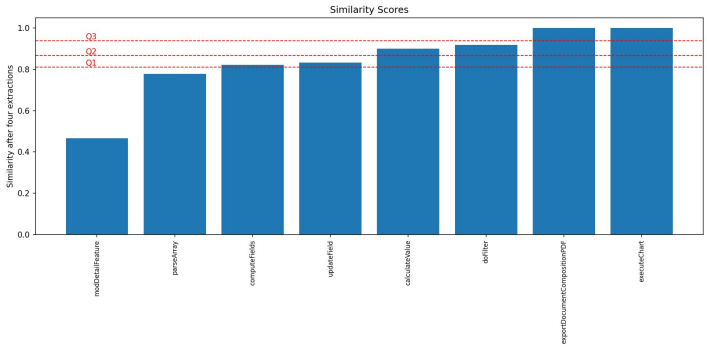

Fig. 6. Similarity scores after predicting four extraction method names. Red lines show percentiles 25%, 50% and 75%. It can be observed how Q1 percentile is around similarity=0.8. The similarities are arranged in ascending order.

6 Discussion

There are some relevant findings from the experiment regarding some method name predictions received. The computed similarity is stable when the number of extractions performed in any code base from the dataset under study varies from one to six. However, it turns out that even having refined the most precise and concise prompt, sometimes we receive method names that do not follow the standard Java convention, which is clearly stated more than once in the prompt. This is an important limitation of the system and it was identified to happen when some of the following circumstances are met:

1. For the code extraction that needs a name, the system is not capable of deducing an informative name (due to lack of descriptive variables, entangled control operations or simply the code is too complex). In those cases the model just returns a generic method name which is meaningless for the extracted code. We observed also that the original method might have a typo. As expected, the *textdavinci003* model finds a similar name without the typo, so similarity computed by *SpaCy* is zero. In Figs. 3 and 4 we can observe that some methods get similarity 0 (no similarity). We illustrate this with some examples from the experiments:
   - `deserialze`. The original method name prediction is `deserialize` which does not have any typo, so *SpaCy* fails to encounter any semantic meaning between these two words.
   - `tql2` and `tred2`. Acronyms make *SpaCy* unable to obtain similarity greater than zero.
2. With the prompt, we state the rules we want the system to follow while doing the prediction, like avoiding any bias from the input. However, in certain cases, we receive a result only partially good because the system decides to prepend information in the prediction method name that is biased from the input we send. For instance, if we ask to predict a name which has an original method name called `deleteCrowdRegion`, we might get the method name `deleteCrowdRegion2_findMaxFitness`. The part after the underscore would be a good name, but the model fails to use the standard Java notation. It is also biased by the original name, which is used as a prefix.
3. With very low frequency, the response we get from the system is not only one method name, but several of them (which again, it is against the prompt we send).
4. We observed also that sometimes the model gives the same name for the original method that is already using for some of the extracted methods, what would end up in a compilation error if they have the same signature.
5. When tokens that compose a method name are not English words, computing the similarity between the original method name and the prediction performs poorly as the methods get transformed into English phrases that barely have meaning.
6. Comments in the code can lead the model to make better predictions, even closer to the original when having applied extractions. This works similarly as the model described by Liu *et al.* [11].

7. The model was also inclined to provide several alternatives, sometimes separated by commas or other unexpected symbols, or even mixed with descriptive text, ignoring the indications in the prompt. We added some instructions in the prompt to avoid those symbols. We noticed that providing examples worked better than explaining with more verbosity.

7 Threats to validity

The validity of the obtained results can be influenced by various threats. In this section, we discuss potential threats to both external and internal validity in the context of the experiment. External validity concerns the generalizability of the results to other Java methods. It refers to the extent to which the results of the experiment can be generalized beyond the specific Java methods used in the study. It involves considering whether the findings can be applied to a broader range of Java methods and codebases. On the other hand, internal validity focuses on the reliability and consistency of the experimental process itself.

Threats to external validity

The dataset used in the experiment is well structured, and all the methods contain appropriate comments and the code follows the standard Java notation. These are the factors that made the prompt be fine-tuned. However, there is no guarantee that different codebases might not produce unexpected consistencies because the prompt did not capture all the necessary information to generalize to all types of codebases.

Threats to internal validity

The study is based on sending an appropriate prompt intended to generalize well for every single method we want to work with. However, this is not always the case. Sometimes the indications in the prompt were not always followed. Some examples is that we received method names not following Java conventions, several method names separated by commas, etc.

This was a challenge since the beginning of the experiment and lead to a well tuned prompt. Initial attempts to get predictions for method names failed in numerous ways:

- The model providing unnecessary text along with the method names, trying to giving a context. That made us trying to add multiple indications in the prompt to just get the desired method name and nothing else. After some iterations, we found out that being too much verbose was also counterproductive.

– The prompt used to predict names for the extracted methods has been iteratively fine-tuned through trial and error until reaching an adequate level of precision. Nonetheless, we cannot ensure that using different set of codebases could not produce unexpected results, not respecting the desired indicated outcome, due to the non-deterministic nature of the *textdavinci003* model.
– The model was also inclined to provide several alternatives, sometimes separated by commas or other unexpected symbols, or even mixed with descriptive text, ignoring often the indications in the prompt.

Methods with abbreviations, codes, and in general everything that is not directly translatable to an English phrase will not perform as desired when calculating similarities.

Rarely, some extractions do not have a complete meaning for the model, which ends up in having no real prediction name, but a placeholder (*extraction*, *extension* and similar ones).

The model, even having sufficient context, sometimes fails to provide compilable code when it proposes the same method name for any (or more) of the extractions, or even for the original method name if they have the same signature. This needs to be handled refining the prompt.

8 Conclusion

Obtaining method names automatically whenever a refactor extraction operation takes place is highly valuable in terms of readability, maintainability, consistency and time saving. Developers might benefit of a reduced cognitive effort after refactor extraction operations. Readability and maintainability are guaranteed, as the semantic of the code remains unchanged after assigning automatically the method names for the extractions.

In this work, we explored how well the *textdavinci003* model can predict method names for extracted methods. The key is how to prompt the model indicating all the necessary requirements needed to infer the method name in an adequate manner. After conducting an experiment to uncover how the model performs, we observed the semantics of the method names resembles to the original method names with a similarity average above 0.8.

As future work, we plan to extend our experiment to support nested method extractions. Also, there is another area of interest in setting temperature values in *textdavinci003* model greater than zero. Other topics we look forward to improve is typo fixing and being able to compute similarities without being impacted by typos.

Bibliography

[1] Juan Carlos Recio Abad. Source code for the experiment in github. URL https://github.com/jcrecio/auto-names.

[2] Uri Alon, Meital Zilberstein, Omer Levy, and Eran Yahav. code2vec: Learning distributed representations of code. 2018. https://doi.org/10.1145/3290353.

[3] Uri Alon, Shaked Brody, Omer Levy, and Eran Yahav. code2seq: Generating sequences from structured representations of code, 2019.

[4] G. Ann Campbell. Cognitive Complexity: An Overview and Evaluation. In *Proceedings of the 2018 International Conference on Technical Debt*, TechDebt '18, pages 57–58, New York, NY, USA, 2018. Association for Computing Machinery. https://doi.org/10.1145/3194164.3194186.

[5] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Pondé de Oliveira Pinto, Jared Kaplan, Harrison 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, Joshua 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. Evaluating large language models trained on code. *CoRR*, abs/2107.03374, 2021. https://doi.org/10.5281/zenodo.4956884.

[6] Shaiful Alam Chowdhury, Gias Uddin, and Reid Holmes. An empirical study on maintainable method size in java. 2022.

[7] Dror G. Feitelson, Ayelet Mizrahi, Nofar Noy, Aviad Ben Shabat, Or Eliyahu, and Roy Sheffer. How developers choose names. *IEEE Transactions on Software Engineering*, 48(1):37–52, jan 2022. https://doi.org/10.1109/tse.2020.2976920.

[8] Alex Graves. Generating sequences with recurrent neural networks. 2014.

[9] Matthew Honnibal and Ines Montani. spaCy 2: Natural language understanding with Bloom embeddings, convolutional neural networks and incremental parsing. 2017. URL https://sentometrics-research.com/publication/72/.

[10] Thorsten Joachims. Text categorization with support vector machines: Learning with many relevant features. 1998. https://doi.org/10.1007/BFb0026683.

[11] F. Liu, G. Li, Z. Fu, S. Lu, Y. Hao, and Z. Jin. Learning to recommend method names with global context. page IEEE Computer Society, May 2022. https://doi.org/10.1145/3510003.3510154.

[12] Tomas Mikolov, Kai Chen, Greg Corrado, and Jeffrey Dean. Efficient estimation of word representations in vector space, 2013.

[13] Rubén Saborido Infantes, Javier Ferrer, Francisco Chicano, and Enrique Alba. Automatizing software cognitive complexity reduction. *IEEE Access 10*, pages 11642,11656, 01 2022. https://doi.org/10.1109/ACCESS.2022.3144743.