Achieving High-Level Software Component Summarization via Hierarchical Chain-of-Thought Prompting and Static Code Analysis

Satrio Adi Rukmono
Mathematics and Computer Science
Eindhoven University of Technology
Eindhoven, The Netherlands
s.a.rukmono@tue.nl

Lina Ochoa
Mathematics and Computer Science
Eindhoven University of Technology
Eindhoven, The Netherlands
l.m.ochoa.venegas@tue.nl

Michel R.V. Chaudron
Mathematics and Computer Science
Eindhoven University of Technology
Eindhoven, The Netherlands
m.r.v.chaudron@tue.nl

***Abstract*—Comprehension of software systems is key to their successful maintenance and evolution. This comprehension comes at different levels of abstraction: At the low level, one must focus on comprehending functions; while at the high level, one must abstract and comprehend the system's requirements. Diverse Automated Source Code Summarization (ASCS) techniques have been proposed to comprehend systems at the lower level. However, techniques for abstracting higher-level explanations fall short. Research on related fields, such as software architecture recovery, has tried to address system comprehension at the higher level by attempting to detect abstractions of design decisions from source code. Nevertheless, this is an on-going effort and many steps in the process are still unsolved. In this paper, we leverage the emergent abilities of Large Language Models (LLMs) together with the achievements in the ASCS and static code analysis fields to design an approach that produces component-level summaries of software systems. Particularly, we address the unreliability of LLMs in performing reasoning by applying a chain-of-thought prompting strategy, which allows us to emulate inductive reasoning. We follow a bottom-up approach, where we start by comprehending lower-level software abstractions (e.g., functions), and then we compose these findings—in a cascading style—to comprehend higher-level ones (e.g., classes, components). We demonstrate the feasibility of our approach by applying it to the open-source Java project JHotDraw version 5.1. We believe our approach offers a stepping stone in developing robust automated software summarization approaches that can be applied generally across domains and types of software system.**

***Index Terms*—comprehension of software systems, automated source code summarization, static analysis, large language models, chain-of-thought prompting**

I. Introduction

Comprehension of a software system is essential for its effective evolution and maintenance. Biggerstaff et al. [1] stated that understanding a program involves explaining its structure, behavior, effects on its context, and its relation to its domain in terminologies that are qualitatively different from the tokens of the source code. In other words, comprehending a software system necessitates dealing with abstraction levels beyond code constructs such as expressions and statements. Beyond statements, but still at a lower level of abstraction, programmers may need to understand code at function level to properly use it. At a higher level, architects may need to grasp the purpose of software components to address evolving requirements. As software grows, comprehension, particularly at higher abstractions, becomes more challenging [2]. Moreover, as systems evolve, their conceptual integrity decreases—intentions concealed due to noises from legacy and technical debt.

Automated techniques exist for analyzing and summarizing software segments [3]. These code summarization techniques operate at lower abstraction, often relying on static analysis to infer semantics from syntax [4]. While these techniques can help in verifying whether a function fulfills its formal specification (e.g., whether `range(min, max, value)` correctly limits the input value within the given range), they are of limited use for arguing about high-level specifications like "the system must allow performing operations on geometrical shapes" [2]. The unique nature of software demands tailored analyses, hindering the development of high-level summarization techniques that work generally across diverse domains and types of software system. We argue that this higher level analysis requires a fundamentally different approach.

Large Language Models (LLMs) offer robust summarization of text, including source code [3], [4]. Yet, LLMs struggle with reasoning, especially inductive reasoning [5]. For example, OpenAI's ChatGPT is pre-trained to predict the next token in a document, using both publicly available and licensed data [6]. In other words, it maps an input text or instruction (a "prompt") to its most likely continuation or response without any formal reasoning, even if the result seems to have reasons behind it [7]. This hinders its use for proper higher-level summarization. However, a chain-of-thought approach can guide LLMs to emulate reasoning [8].

Our paper proposes a new direction for approaching automated higher-level summarization of software systems: one that combines the findings in the automated source code summarization and static program analysis fields, together with the "pseudo-reasoning" of LLMs. We take software source code, identifiers, and models as inputs, using chain-of-thought prompting to perform hierarchical, bottom-up summarizations. This approach is demonstrated on the JHotDraw 5.1 project. We believe this approach can be a solid foundation for advancing the field of hierarchical software component summarization.

The remainder of the paper is structured as follows: Section II defines terminologies, Section III gives an overview of related fields, Section IV presents our approach, Section V applies it to JHotDraw 5.1, and Section VI concludes the paper.

II. Definitions

For the purpose of our discussion, we provide the following definitions to ensure terminologies remain generic and not tied to a specific programming language or paradigm.

A **software operation** denotes a granular piece of code that possesses well-defined functionality, available for (re)use by other code segments. Examples include subroutines, functions, procedures, or methods.

A **software module** represents a section of code enclosing a collection of software operations, signifying that the operations belong together. Possible instances of modules are classes or files.

A **software component** is a collection software modules, determined by specific criteria to be cohesively working together towards a shared goal. These criteria could arise from design, where an engineer makes decisions about a component's purpose, leading to the implementation of modules directly based on the decisions. Components might emerge from a development perspective, where modules are logically organized into packages or directories. Reverse engineering might also reveal components, as clusters of modules working together, identified through software architecture recovery techniques. In concert, software components interact to form a complete software system.

We employ the terms **parent** and **child** to represent relationships between adjacent abstraction levels. When component A contains modules X and Y, we designate component A as the *parent* of X and Y, while modules X and Y stand as the *children* of component A.

Thus, we present our hierarchy of software abstractions. At the utmost level, a software system comprises components, which house modules, which in turn encapsulate operations.

III. Related Fields of Study

When it comes to generating higher-level summaries of software systems, research in diverse software fields can be used as a scaffold to reach this goal. In this section, we introduce some of these fields and state how they relate to our research.

Automated Source Code Summarization (ASCS): ASCS is the use of computing systems to provide a summary of pieces of code, usually in natural language. These summaries are usually used to increase the understanding and comprehension of a software system, and consequently, facilitate its maintenance [4]. ASCS output can be used to provide a higher-level explanation of software systems. However, a literature review shows that most existing summarization techniques work at the statement, operation, or module level [3]. Only the study by Hammad et al. is reported as being able to summarize at the component level. This summarization is achieved by integrating summaries of (Java) methods belonging to a package [9]. However, this integration primarily takes the form of a list of method descriptions and additional statistical information, such as the count of abstract and final methods within a package; the cohesive higher-level description of the component is missing. The challenge and existing gap in the literature when using ASCS to provide higher-level summaries of a software system is to summarize the overall vision of the system and its components, rather than just focusing on coarse-grained code modules.

Large Language Models (LLMs) and Chain-of-thought Prompting: LLMs are deep learning models with parameter sizes over a hundred billion that can execute a set of natural language processing tasks [5]. These models have been trained on vast amounts of data and are able to predict the next token in a sentence based on self-supervised learning. With the advent of LLMs, different studies have proposed their use on ASCS. When it comes to ASCS, diverse LLMs such as CodeBERT, CodeT5, and ChatGPT have been fine-tuned on code summarization [10]. One of the features that result particularly attractive about the latter is its ability to enable conversations and consider previous chat history for future answers. In particular, there is evidence that complex reasoning can be emulated by providing a chain-of-thought prompting. Chain-of-thought prompting is a strategy where an LLM solves complex problem in a sequence of intermediate steps—in a similar manner to human reasoning [8]. This type of prompting technique can, thus, be used to orchestrate ASCS techniques, supporting the generation of higher-level summarization of software systems.

IV. Proposed Approach

In this section, we present our approach for performing module- and component-level summarization. Our approach implements chain-of-thought prompting using a cascading cycle to provide higher-level summaries. An overview of the approach is depicted in Figure 1. The details of the approach are described in the following subsections.

A. Preparation

The initial step in the preparation stage involves determining the abstraction levels for the software system in question. Once established, hierarchies can be extracted from the source code using static analysis techniques with tools such as Rascal [11] or srcML [12] (both support multiple languages, including C, C++, and Java), CodeOntology [13], or javapers [14] (for Java).

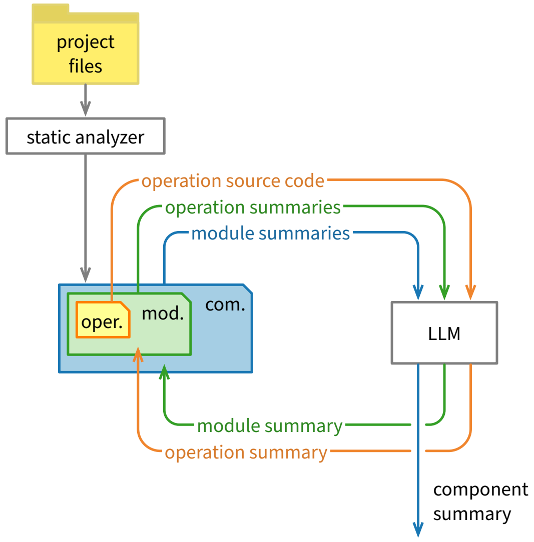

Fig. 1. An overview of the proposed "cyclic" approach.

B. Summarization Cycle

In this stage, we execute a cascading cycle of chain-of-thought prompting [8] on the chosen LLM. The first step is generating summaries of software operations[1]. These operation summaries are then utilized to construct prompts for the LLM to summarize their enclosing modules. This process continues upwards in the hierarchy: module summaries are used to summarize their enclosing components.

There is an interplay between the LLM and static analysis: information extracted from the source code guides the context in which the LLM must respond to the queries. Specifically, this extracted information pertains to identifier names and the system's hierarchy, such as which modules belong to which component.

1) Atomic Summaries: At the "atomic" abstraction level, i.e., software operations, established methods exist for generating summaries [3]. LLMs can generate summaries that exhibit a strong grasp of operation characteristics based on the source code [16].

This initial step of the summarization cycle is illustrated in Figure 2. (The provided prompts in the subsequent figures are illustrative, offering a general idea of what to ask to the LLM. We suggest using language-specific terminologies for the actual prompt. For specific examples, refer to Section V.)

2) Composed Summaries: Our contribution starts with the premise that a software module should inherently constitute a cohesive collection of operations. Therefore, a module summary can be constructed by summarizing its constituent operation summaries, as depicted in Figure 3. This notion extends to higher abstraction levels, such as components being cohesive collections of their modules. This assumption

[1]While documentation comments often contain good summaries, we do not consider them in our approach. The rationale behind this decision is that comments can can be dated or incomplete [15]. Our focus is on what the program code itself reveals about the system.

```
function summarize(operation):
   prompt ← "This is operation
      {operation.name} of module
      {operation.parent.name}:
      ```{operation.sourceCode}```
      Provide a summary for this operation."
   → llm.ask(prompt)
```

Fig. 2. Prompt template for generating a summary of a software operation. Between curly braces we put information extracted from static analysis.

facilitates the application of the chain-of-thought prompting strategy.

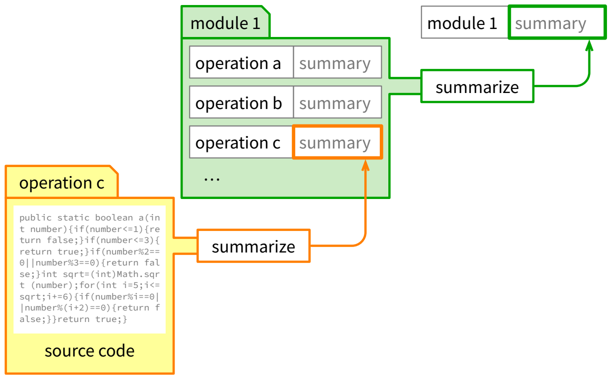

Fig. 3. An illustration of building summary of a module from summaries of its constituent operations.

This step is outlined in Figure 4 for module summarization. To summarize components, replace *modules* with *components* and *operations* with *modules*.

```
function summarize(module):
   op_summaries: list of string
   for each operation in module:
      op_summary ← summarize(operation)
      op_summaries.add("{operation.name}:
         {op_summary}")
   prompt ← "The module {module.name} of
      component {module.parent.name} has the
      following operations:
      {op_summaries}
      Provide a summary for this module."
   → llm.ask(prompt)
```

Fig. 4. Prompt template for generating a summary of a software module from the summaries of its constituent operations.

3) Clustering Modules and Operations: Optionally, a step can be undertaken to cluster operations within a module into intermediate-level categories situated between *modules* and *operations*. This approach enhances summaries by establishing sets of "module responsibilities," each supported by several operations.

As a simplified example, a software module that manages a list of items might feature four operations: (1) "add an item at the front of the list," (2) "add an item at the back of the list,"
```

(3) "remove an item from the front of the list," and (4) "remove an item from the back of the list." Grouping these operations into two responsibilities ("add an item" and "remove an item") could enhance the module's summary.

LLMs excel at text clustering and classification [17]. This capability can be harnessed by listing operations and their summaries, then prompting the LLM to form groups of related operations. The generated operation clusters can become part of the unit summary provided to the LLM in the summarization cycle for generating component summaries.

The step of clustering operations is depicted in Figure 5. Refer to Section V for concrete examples.

```
function cluster(module) → map of
      reponsibilities to lists of operations:
   op_summaries: list of string
   for each operations in module:
      op_summary ← summarize(operation)
      op_summaries.add("{operation.name}:
         {op_summary}")
   prompt ← "The module {module.name} of
      component {module.parent.name} has the
      following operations:
      {op_summaries}
      Group the operations into responsi-
      bilities."
   → llm.ask(prompt)
```

Fig. 5. Prompt template for clustering the operations of a module into responsibilities.

Following the idea of cascading chain-of-thoughts, clustering modules into sub-components can similarly be achieved by substituting *modules* with *components*, *operations* with *modules*, and *responsibilities* with *sub-components*.

V. Application on JHotDraw: A Proof of Concept

To demonstrate the feasibility of our approach, we apply it to the Java project JHotDraw, version 5.1. We choose this project due to our familiarity with it as it often serves as a case study in software engineering research. The chosen abstraction levels are outlined in Table I. We focus on public methods instead of all methods because private methods contain the fine details of how the class works, which we can leave out for a high-level summary. We consider public methods defined within the classes, excluding inherited methods that are not overridden.

TABLE I
Abstraction levels for the JHotDraw project.

Abstraction	**Instance**
Operation	Public methods defined in the module.
Module	Classes, enums, and interfaces.
Component	Packages that contain modules, i.e., excluding packages that only contain other packages.

The JHotDraw project encompasses 11 components, 146 modules, and 942 operations, averaging around 13 modules per component and around 6 operations per module. For detailed distributions of modules in components and operations in modules, refer to Table II.

TABLE II
Descriptive statistics of operations, modules, and components of JHotDraw.

	min	Q_1	Q_2	Q_3	max
# modules per component	1	2	7	19	53
# operations per module	1	2	3	7	52

Throughout this section, we use the terms *package*, *class*, and *method* as instances of our conceptual *component*, *module*, and *operation*, respectively.

We extract the project's hierarchy using javapers [14], yielding a labeled property graph with information that we require: packages, classes, and methods as nodes, and parent-child relationships as edges. The nodes contain necessary properties for summary generation: package, class, and method names, as well as method source code.

Subsequently, we initiate the summarization cycles. We opt for OpenAI's GPT-3.5 model for its known text summarization capability [18], [19]. Specifically, we use the `gpt-3.5-turbo-0613` version via the Chat Completion API with a `temperature` parameter of 0 for focused and deterministic results[2]. We start with method summarization, illustrated by an example in Figure 6.

Prompt:
This is method `'range(int,int,int)'` of class `'CH.ifa.draw.util.Geom'`:

```
```java
public static int range(int min, int max,
    int value) {
  if (value < min) { value = min; }
  if (value > max) { value = max; }
  return value;
}
```
```

Write a 1-sentence documentation comment for this method in imperative mood.

Response:
Limit the given value to be within the specified range.

Actual documentation comment:
Constains [*sic*] a value to the given range.

Fig. 6. An example of a prompt and response for summarizing a method, and the actual documentation comment.

To ensure the summaries are solely based on implementations, we exclude source code comments before constructing prompts. Furthermore, Java class references within code are replaced with fully qualified names (e.g., `List` becomes `java.util.List`) to provide the LLM with context.

[2]https://platform.openai.com/docs/api-reference/chat/create#chat/create-temperature

The prompt instructs the LLM to deliver a one-sentence documentation comment in the imperative mood, constraining length and maintaining style consistency. In the figure, the programmer's actual documentation comment is provided for comparison.

For class-level summarization, we merge summarization and clustering into a single prompt (Figure 7). The prompt solicits a class documentation comment and method grouping into responsibilities. We guide the LLM in a chain-of-thought manner to structure responses in a computer-readable format to help with organizing the results of our experiment. In the example prompt for the `Geom` class, the LLM generates four responsibilities from 15 methods.

Prompt: A Java class `` `CH.ifa.draw.util.Geom` `` contains the following public method(s):

- `` `west(java.awt.Rectangle)` ``: Return the point on the west side of the given rectangle.
- `` `range(int,int,int)` ``: Limit the given value to be within the specified range.
- *...(more methods here, truncated for brevity)*

Write a 1-sentence documentation comment for the class in imperative mood. Group the methods into key responsibilities. Give short names to the responsibilities. Provide the result as JSON string like:

```
{"description": "...", "responsibilities":
[{"name": "...", "description": "...",
"methodsInvolved": [method1Signature,
method2Signature, ...]}, ...]}
```

Only output the JSON string without any additional text.

Response: *(formatted for clarity and truncated for brevity)*

```
{'description': 'Calculate geometric
properties and perform operations on
shapes.',
 'responsibilities': [
  {'name': 'Point Operations',
   'description': ...,
   'methodsInvolved':
    ['west(java.awt.Rectangle)',
     'east(java.awt.Rectangle)', ... ]},
  {'name': 'Value and Length Operations',
   'description': ...,
   'methodsInvolved':
    ['range(int,int,int)',
     'length(int,int,int,int)', ... ]},
  ... ]}
```

Actual documentation comment:
Some geometric utilities.

Fig. 7. An example of a prompt and response for summarizing a class and clustering its methods in one pass, and the actual documentation comment.

Moving to package summarization, we exclude class clustering. Figure 8 provides an example prompt, response, and actual documentation comment.

Prompt:
Given a Java package `` `CH.ifa.draw.contrib` `` containing the following classes:

- class `` `CH.ifa.draw.contrib.PolygonFigure` ``: Represents a polygon figure and provides methods for manipulating and interacting with it.
- *...(more classes here, truncated for brevity)*

Write a 1-sentence documentation comment for the package in imperative mood.

Response:
Provides classes for drawing and manipulating polygon, diamond, and triangle figures.

Actual documentation comment:
Classes that where [*sic*] contributed by others.

Fig. 8. An example of a prompt and response for summarizing a package, and the actual documentation comment.

This section concludes with a side-by-side comparison of generated summaries and actual documentation comments for various JHotDraw packages, classes, and methods in Table V.

VI. Conclusions and Future Work

We have presented an approach for automated software component summarization, combining static analysis with the emerging capabilities of a Large Language Model (LLM). The initial result of our proof-of-concept demonstration on a Java project has produced summaries for software operations (methods), modules (classes), and components (packages). We position this approach as an initial step towards a generally applicable, hierarchical software component summarization.

For the subsequent phases of enhancing software comprehension, our forthcoming efforts are outlined below.

Evaluation of Summaries: Given that the primary objective of automated summarization is to enhance comprehension, we propose evaluating the generated summaries by engaging participants in comprehension tasks based on these summaries. This aligns with the methodology outlined in the study by Stapleton et al. [16]. At the component level, participants can be provided with a list of components along with their descriptions, without divulging the project's context. Subsequently, they could be tasked with activities such as: crafting a one-sentence project description, identifying components relevant for implementing a specific feature or resolving a particular bug, and addressing similar queries.

Selection of Components: Upon closer examination, it becomes evident that several different packages within the case study JHotDraw exhibit similarly worded summaries, featuring terms like "creating," "manipulating," "drawing," and "figures." We contend that this is due to the lack of cohesion in the packages as defined by the original developer of JHotDraw. Some packages contain classes that are intertwined with classes from other packages. For instance, classes in `CH.ifa.draw.contrib` are grouped together

TABLE III

A COMPARISON OF GENERATED SUMMARIES AND ACTUAL DOCUMENTATION COMMENTS OF SEVERAL JHOTDRAW PACKAGES, CLASSES, AND METHODS.

	Generated summary	**Actual documentation comment**
package		
`CH.ifa.draw.contrib`	Provides classes for drawing and manipulating polygon, diamond, and triangle figures.	Classes that where [*sic*] contributed by others.
`CH.ifa.draw.util`	Provides utility classes and interfaces for handling palette buttons, geometric calculations, command choices, palette events, layout management, command menus, fillers, commands, color manipulation, image loading, data storage, animation, clipboard management, text fields, vector iteration, and command buttons.	This package provides generally useful utilities that can be used independent of JHotDraw.
class		
`CH.ifa.draw.contrib. PolygonFigure`	Represents a polygon figure and provides methods for manipulating and interacting with it.	A scalable, rotatable polygon with an arbitrary number of points
`CH.ifa.draw.util.Geom`	Calculate geometric properties and perform operations on shapes.	Some geometric utilities.
method		
`PolygonFigure:: smoothPoints()`	Smooths the points of the polygon figure by removing points that are too close to the line formed by their neighboring points.	Remove points that are nearly colinear with others
`Geom::range(int, int,int)`	Limit the given value to be within the specified range.	Constains [*sic*] a value to the given range.

solely because they are contributed by developers other than the main developer. As another example, the package `CH.ifa.draw.util` has a long summary owing to its inclusion of mainly "miscellaneous" classes. To further enhance our investigation, we intend to experiment with components identified using reverse engineering methods, aiming to assess the quality of summaries when the components express greater cohesion.

Integration into Workflows: The approach we have presented, along with other source code summarization techniques, has the potential to assist programmers in composing and maintaining documentation comments. The logical progression towards adoption would involve developing an Integrated Development Environment (IDE) plugin that incorporates such techniques. This plugin could seamlessly integrate into a software developer's workflow, enabling them to create or update documentation comments as needed, while still retaining the capability to manually compose comments.

REFERENCES

[1] T. J. Biggerstaff, B. G. Mitbander, and D. Webster, "The concept assignment problem in program understanding," in *Working Conference on Reverse Engineering*, 1993, pp. 27–43.

[2] J. Siegmund, "Program comprehension: Past, present, and future," in *IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering*, vol. 5, 2016, pp. 13–20.

[3] Y. Zhu and M. Pan, "Automatic code summarization: A systematic literature review," *ArXiv*, vol. abs/1909.04352, 2019.

[4] C. Zhang, J. Wang, Q. Zhou, T. Xu, K. Tang, H. Gui, and F. Liu, "A survey of automatic source code summarization," *Symmetry*, vol. 14, no. 3, 2022.

[5] Y. Bang, S. Cahyawijaya, N. Lee, W. Dai, D. Su, B. Wilie, H. Lovenia, Z. Ji, T. Yu, W. Chung, Q. V. Do, Y. Xu, and P. Fung, "A multitask, multilingual, multimodal evaluation of ChatGPT on reasoning, hallucination, and interactivity," *ArXiv*, feb 2023.

[6] OpenAI, "GPT-4 technical report," OpenAI, Tech. Rep., 2023.

[7] L. Floridi and M. Chiriatti, "GPT-3: Its nature, scope, limits, and consequences," *Minds and Machines*, vol. 30, pp. 681–694, 2020.

[8] J. Wei, X. Wang, D. Schuurmans, M. Bosma, B. Ichter, F. Xia, E. Chi, Q. V. Le, and D. Zhou, "Chain-of-thought prompting elicits reasoning in large language models," in *Advances in Neural Information Processing Systems*, S. Koyejo, S. Mohamed, A. Agarwal, D. Belgrave, K. Cho, and A. Oh, Eds., vol. 35. Curran Associates, Inc., 2022, pp. 24 824–24 837.

[9] M. Hammad, A. Abuljadayel, and M. Khalaf, "Summarizing services of Java packages," *Lecture Notes on Software Engineering*, vol. 4, pp. 129–132, 05 2016.

[10] W. Sun, C. Fang, Y. You, Y. Miao, Y. Liu, Y. Li, G. Deng, S. Huang, Y. Chen, Q. Zhang *et al.*, "Automatic code summarization via ChatGPT: How far are we?" *ArXiv*, pp. arXiv–2305, 2023.

[11] P. Klint, T. van der Storm, and J. Vinju, "Rascal: A domain specific language for source code analysis and manipulation," in *Ninth IEEE International Working Conference on Source Code Analysis and Manipulation*, 2009, pp. 168–177.

[12] M. L. Collard, M. J. Decker, and J. I. Maletic, "srcML: An infrastructure for the exploration, analysis, and manipulation of source code: A tool demonstration," in *IEEE International Conference on Software Maintenance*, 2013, pp. 516–519.

[13] M. Atzeni and M. Atzori, "CodeOntology: RDF-ization of source code," in *The Semantic Web*, C. d'Amato, M. Fernandez, V. Tamma, F. Lecue, P. Cudré-Mauroux, J. Sequeda, C. Lange, and J. Heflin, Eds. Cham: Springer International Publishing, 2017, pp. 20–28.

[14] S. A. Rukmono and M. R. V. Chaudron, "Enabling analysis and reasoning on software systems through knowledge graph representation," in *IEEE/ACM 20th International Conference on Mining Software Repositories*, 2023, pp. 120–124.

[15] B. Fluri, M. Wursch, and H. C. Gall, "Do code and comments co-evolve? On the relation between source code and comment changes," in *14th Working Conference on Reverse Engineering (WCRE 2007)*, 2007, pp. 70–79.

[16] S. Stapleton, Y. Gambhir, A. LeClair, Z. Eberhart, W. Weimer, K. Leach, and Y. Huang, "A human study of comprehension and code summarization," in *28th International Conference on Program Comprehension*. New York: Association for Computing Machinery, 2020, p. 2–13.

[17] B. Ding, C. Qin, L. Liu, L. Bing, S. Joty, and B. Li, "Is GPT-3 a good data annotator?" *arXiv preprint arXiv:2212.10450*, 2022.

[18] T. Goyal, J. J. Li, and G. Durrett, "News summarization and evaluation in the era of GPT-3," *arXiv preprint arXiv:2209.12356*, 2022.

[19] A. Bhaskar, A. R. Fabbri, and G. Durrett, "Zero-shot opinion summarization with GPT-3," *arXiv preprint arXiv:2211.15914*, 2022.