SoCCMiner: A Source Code-Comments and Comment-Context Miner

Murali Sridharan
murali.sridharan@oulu.fi
M3S, University of Oulu
Oulu, Finland

Mika Mäntylä
mika.mantyla@oulu.fi
M3S, University of Oulu
Oulu, Finland

Maëlick Claes
maelick.claes@oulu.fi
M3S, University of Oulu
Oulu, Finland

Leevi Rantala
leevi.rantala@oulu.fi
M3S, University of Oulu
Oulu, Finland

ABSTRACT

Numerous tools exist for mining source code and software development process metrics. However, very few publicly available tools focus on source code comments, a crucial software artifact. This paper presents SoCCMiner (Source Code-Comments and Comment-Context Miner), a tool that offers multiple mining pipelines. It is the first readily available (plug-and-play) and customizable open-source tool for mining source code contextual information of comments at different granularities (Class comments, Method comments, Interface comments, and other granular comments). Mining comments at different source code granularities can aid researchers and practitioners working in a host of applications that focus on source code comments, such as Self-Admitted Technical Debt, Program Comprehension, and other applications. Furthermore, SoCCMiner is highly adaptable and extendable to include additional attributes and support other programming languages. This prototype supports the Java programming language.

CCS CONCEPTS

• **Software and its engineering → Software libraries and repositories**; *Software maintenance tools.*

KEYWORDS

Mining Software Repositories, Source Code Comments, Comment Context, Python

ACM Reference Format:
Murali Sridharan, Mika Mäntylä, Maëlick Claes, and Leevi Rantala. 2022. SoCCMiner: A Source Code-Comments and Comment-Context Miner. In *19th International Conference on Mining Software Repositories (MSR '22), May 23–24, 2022, Pittsburgh, PA, USA*. ACM, New York, NY, USA, 5 pages. https://doi.org/10.1145/3524842.3527998

1 INTRODUCTION

Software Documentation is a critical aspect of software that impacts multiple elements of Software Maintenance from Program Comprehension [23] to Technical Debt[17] Detection. In the past, Hartzman et al. [10] in their experience report recommends software documentation maintenance as a separate maintenance activity owing to its complex and critical nature. They urged continuous monitoring and revision of Inline documentation (source code comments) as they act as a preamble for actual code logic. Later, De Souza et al. [7] conducted an extensive survey with 50+ software practitioners and revealed that Source Code and Source Code Comments are the two critical documentation artifacts for both structured and object-oriented paradigms. There have been several studies in the past that focused on the different facets of source code comments such as i) Quality [12, 13, 22], ii) Program Comprehension [14, 16, 19], and iii) Self-Admitted Technical Debt (SATD) Detection [1, 11, 15, 17, 21].

Most previous studies have focused on specific comments such as block or line comments, with isolated contexts, such as class level or method level comments lacking a holistic perspective. For example, Steidl et al.[22] manually annotated the comments into Copyright, Header (Class and Method) and Inline (comments inside a method) for analyzing and assessing its contribution to understanding the source code. The previous works on SATD detection using source code comments [1, 6, 8, 15, 17] relied mainly on the source code comment text without its context such as the location of the source code comment, or preceding and succeeding source code. Understanding the context of the comment enables a deeper understanding of the source code comment.

Past studies reveal a pattern where metrics and code comments are studied at varying isolated granularities. For example, Zazworka et al. [25] utilized metrics such as Lines of Code, McCabe's Cyclomatic Complexity, Density of Comments, and Sum of Maximum Nesting of all Methods in a Class in addition to Code Smells for Technical Debt (TD) identification. In their work, the fetched metrics are at the module level. Bavota et al.[2] evaluated metrics such as Buse and Weimer Readability, Coupling Between Objects (CBO), and Weighted Method Complexity (WMC) and identified that there is little correlation with those metrics and SATD. Here, the computed metrics are at the class level. Zampetti et al.[24] evaluated metrics such as LOC, Coupling, McCabe's Cyclomatic Complexity, and the number of expressions, variables, identifiers, and comments of previously identified SATD instances to recommend SATD during source code development. In their work, the captured metrics are at the method level.

Moreover, researchers often spend time developing a customized package for data procurement. Often, they extract source code comments at specific source code granularities using `srcML` [4] tool. These indicate the lack of a publicly available reusable tool/library that provides a mechanism to mine comments and comprehensive comment attributes at different granularities.

To this effect, we propose `SoCCMiner` (Source Code-Comments Comment-Context Miner, available online [20]) as a reusable and extendable python package that uses `srcML` [4] for parsing source code and extends it with four data mining pipelines, including i) Comment Meta Attributes pipeline, ii) Comprehensive Comment Attributes pipeline, iii) Source Code Meta attributes pipeline and iv) Program Miner (combine all the previous pipelines). To the best of our knowledge, we are the first to advocate and develop a reusable and extendable tool/package that offers source code mining pipelines to mine comprehensive comment contextual information at different granularities.

2 TOOL DESCRIPTION AND APPLICATIONS

`SoCCMiner` [20] is a Python-based tool that empowers researchers and practitioners with readily available mining pipelines. These pipelines allow easier integration with AI data pipelines and other applications. It utilizes LXML [3] to parse the Abstract Syntax Tree (AST) representation generated by `srcML` [4], to mine comments, comment context, and source code attributes at various granularities. Figure 1 shows the overview of the `SoCCMiner` functionality.

The dedicated `SoCCMiner` tool page in the Description section of the tool repository [20] documents all the attributes of each pipeline along with their samples. The following are the pipelines offered by `SoCCMiner` with its applications:

CommentsMetaAttribute Pipeline: Mines only the essential comment attributes: Comment Content, Line Number, and Source File Name (a total of 3 attributes).

Applications: This pipeline enables researchers and practitioners to perform off-the-shelf granular-level comment retrieval. The source code comments mined at various granular levels using this pipeline can be passed as input to AI pipelines for classification tasks such as Sentiment Analysis, detecting various SATD instances, etc. Such analysis enables deeper insights with granular level sentiment details and SATD instances.

ComprehensiveCommentsAttribute Pipeline: Mines comprehensive comment context attributes: preceding code, succeeding code, parent trace of the comment, comment type (for `Java`, Line or Block), category (Header or Non-Header) of comment and other attributes (a total of 17 attributes including the three from the previous pipeline).

Applications: This pipeline's novel feature of mining source code contextual information of each comment can benefit both researchers and practitioners in various applications. They include studying/applying SATD prediction or Forecasting models, Program Comprehension models, Automatic Comment Generation model, Program/Source Code Summarization models, enabling Automatic

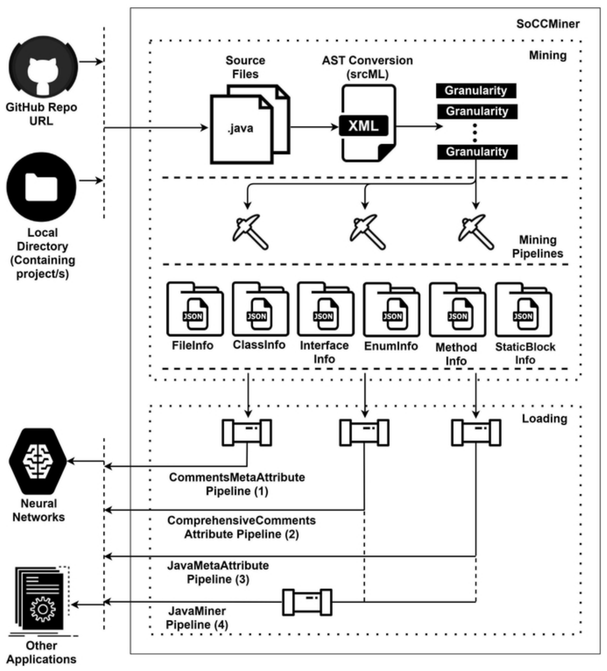

Figure 1: SoCCMiner - An Overview

Comment Scoping, and Source Code Characterization of SATD instances.

JavaMetaAttribute Pipeline: Primarily mines project granularity meta attributes and standard source code metrics such as granular LOC (Class/Method/Interface and other granular LOC), Type, Specifier, and other metrics (a total of 30 unique attributes). The `Java` project granularities include Package, File, Class, Interface, Method, Enumerator (Enum), and Static block.

Applications: This pipeline is instrumental in associating the mined comments with its respective granular unit through its granular signature attribute along with parent identifier and parent trace attributes. This allows researchers to study comment scoping, preparing oracle for comment generation, and correlating SATD or Positive/Negative sentiment granular units with source code metrics from `SoCCMiner`. Additional source code metrics from other third-party tools can also be associated with the granular units through the signature attribute.

JavaMiner Pipeline: Mines all the attributes from previous pipelines (a total of 47 attributes are mined).

Applications: This pipeline combines all the applications discussed for all the previous pipelines.

3 EVALUATION

For evaluation, we run our tool on five different, widely used Java open-source repositories with varying project sizes (i.e., source file count and Lines of Code) for all the pipelines `SoCCMiner` has to offer. Table 1 contains the evaluation project details along with the size of the project's source code in KLOC (1 Kilo Lines of Code is 1000 LOC). The source code file count of each project varies from 900 to 10,000. Overall, a total of 21,618 source code (.java) files amounting to approximately 2.5 million LOC (Lines of Code) are utilized for evaluation.

Table 1: Evaluation Project Repositories

Project Name	Version	Source File Count	Size (in KLOC)
Mockito	4.1.0	942	89.67
JUnit5	5.8.1	1,263	148.80
Apache Log4J2	2.14.1	2,296	293.76
Sonarqube	9.2.0	6,738	783.83
Hibernate-ORM	5.2.0	10,379	1165.47
Total		**21,618**	**2481.53**

Table 2, Table 3, and Table 4 contains the brief evaluation details for CommentsMetaAttribute, ComprehensiveCommentsAttribute, and JavaMetaAttribute pipelines respectively. Considering the space restrictions, only the evaluation details of the core capabilities are listed and discussed.

It appears from Table 2 that file level comments are the most frequent. It might be due to the presence of file header comments. The number of Method comments in all the projects is higher than the Class/Interface comments. It might be because most of the application logic bound to change frequently resides in the methods compared to other granularities. The total number of comments

Table 2: Mined Comments by Granularity Level

Granularity	Project				
	Mockito	JUnit5	Apache Log4J2	Sonarqube	Hibernate-ORM
File	1,318	2,526	4,522	8,138	20,474
Class	668	1,462	4,474	3,970	8,313
Interface	256	359	681	1,193	3,620
Method	1,966	4,379	4,474	5,332	14,878
Enum	12	75	205	118	438
StaticBlock	5	7	95	8	20
Total Comments	**4,225**	**8,808**	**14,401**	**18,759**	**47,743**

across all projects under evaluation shows that the higher the number of files, the higher the comments. Perhaps, this could vary with other projects. The Hibernate-ORM project has 47,743 comments with 10,379 source code files against the lowest 4,225 comments in the Mockito project with 942 source code files.

Table 3 lists the preceding and succeeding source code context of comments from only the smallest and largest project under evaluation due to space restriction. The project Mockito and Hibernate have file header comments for almost all the files with NA, i.e., nothing preceding the comment for 942 and 10,106 instances respectively, almost the entire project files.

The Control statements include conditional statements (`if`, `else`, `switch`), looping statements (`while`, `for`, `do`) and control jump statements (`break`, `continue`). The Exception statements include `try`, `throw`, `catch`, and `finally`. All other type of statements such as `return`, `synchronized`, and others are categorized as Others in Table 3. Overall, the header comments such as class header, method header and other granular headers are more with total number of

Table 3: Mined Comment Context Source Code

Type of Source Code	Mockito		Hibernate	
	# instances succeeding a comment	# instances preceding a comment	# instances succeeding a comment	# instances preceding a comment
Package	992	42	11,132	811
Class	279	87	8,230	992
Interface	98	89	1,130	748
Method	647	1,127	6,568	7,710
Enum	5	4	158	100
StaticBlock	0	4	1	13
Declaration stmts	518	418	5,121	4,717
Expression stmts	1,201	742	6,632	5,206
Control stmts	16	74	468	2,259
Exception stmts	43	129	454	902
End of File (EOF)	39	0	1,038	0
Beginning of File (NA)	0	942	0	10,106
Others	387	567	6,811	14,179
Total	**4,225**	**4,225**	**47,743**	**47,743**

instances succeeding a comment 2,021 and 27,219 being the highest. The most frequent Java project granularity that succeeds/follows a source code comment is Package for both Mockito and Hibernate with 992 and 11,132 instances respectively. The granularity that often precedes a source code comment is Method for both Mockito and Hibernate projects. This can be attributed to the high frequency of the Method granularity as in Table 4.

Table 4: Mined Project Granularity instances

Granularity	Project				
	Mockito	JUnit5	Apache Log4J2	Sonarqube	Hibernate-ORM
Package	118	128	186	546	1372
File	942	1,263	2,295	6,736	10,379
Class	1,695	2,253	2,755	6,958	13,647
Interface	247	177	265	685	1,221
Method	6,356	9,938	17,129	51,793	84,165
Enum	10	37	65	232	313
StaticBlock	13	10	48	15	46
Total Granularities	**8,439**	**12,543**	**20,448**	**60,229**	**1,00,764**

Table 4 shows the frequency of different code granularities. Comparing Tables 4 and 2 suggests that the higher the absolute number of files/classes/methods and other granularities, the higher the number of comments. The higher the number of source code files, the higher the number of File comments. Method, Class, and Interface exhibit a similar pattern, with projects with a higher number of granularities and a higher number of comments. The lower number of Enum and StaticBlock granularities explains the lower number of enum level and static block level comments as observed in Table 2.

4 PERFORMANCE

The computational efficiency of each mining pipeline varies. Table 5 lists the consolidated (of all the five projects that include approximately 2.5 Million Lines of Code) time and efficiency of each mining pipeline. The evaluation happened in an Ubuntu environment running an Intel Core i7-6500 with a clock frequency of 2.50 GHz and 8 GB memory. The AST creation time remains the same with minimal variance, while the Attribute Mining reflects the core mining performance. Although Table 5 provides a preliminary estimate of the performance and throughput based on five projects, one must be aware that these are not constants and will vary depending on the LOC and the number of comments in a source code file.

Table 5: SoCCMiner Performance

Pipeline	Mining Time (seconds)			Mining Efficiency	
	A - AST Creation	B - Attribute Mining	Total, (A + B)	Files / second	KLOC / second
Comments Meta Attribute	527.17	2537.52	3065.69	7.05	0.81
Comprehensive Comments Attribute	530.78	5575.58	6106.35	3.54	0.41
JavaMeta Attribute	519.53	15075.60	15595.13	1.39	0.16
Java Miner	533.78	18033.15	18566.93	1.16	0.13

5 LIMITATIONS

In this prototype, SoCCMiner supports only Java-based project repositories, a fundamental limitation of SoCCMiner. Another limitation with this prototype is the mining efficiency. For example, JavaMiner pipeline mines a meager 0.13 KLOC per second, although it mines a total of 47 attributes. SoCCMiner immediately serializes the mined information to reduce memory consumption while mining all attributes for massive projects with over 10,000 KLOC with limited memory. This trade-off (serialization) has a penalizing effect on the mining speed. However, mining all attributes (JavaMiner pipeline) for a massive project should be a one-off task compared to mining primary comment attributes (CommentsMetaAttribute pipeline), beyond which the source files containing incremental code changes will be comparatively lesser.

6 RELATED WORK

D Schreck et al. [18] developed a tool that focused on the quality of Java source code documentation through readability, completeness, and quantity. Khamis et al. [12] developed the JavaDocMiner tool to assess the source code documentation (inline comments) using a set of heuristics. In their analysis of source code comments, Steidl et al. [22] differentiate the comments into seven categories manually: Copyright, Header, Member, Inline, Section, Code, and Task comments. SoCCMiner automatically retrieves the source code comments at different granularity levels without extra effort for annotation. In addition, SoCCMiner mines and retrieves the preceding and succeeding source code context and other location attributes for all the comments, a unique feature that is not available in any existing open-source tools to the best of our knowledge.

MA de Freitas Farias et al. [5] developed the ExComment tool that could parse Java source code and fetch source code comments in order to detect Technical Debt through source code comment analysis. Liu et al. [15] developed a SATD detector tool that could parse Java source code comments for Technical Debt detection. The discussed tools lack the granular composition of SATD instances, while SoCCMiner used with a classifier provides the granular composition of detected SATD instances.

AlOmar et al. [1] developed SATDBailiff tool that could mine and track source code comments from Github repositories at commit level. In addition to working with Github repositories, SoCCMiner also operates on the source code repositories stored in local directories. It can be integrated with several AI applications through the different mining pipelines. SATDBailiff is specific to Java programs. The tools discussed in the literature are specific to Java, while SoCCMiner can be extended to `C, C++ or C#`, and other programming languages since at the core it works on converting the source code to AST representation using srcML.

Srijoni Majumdar et al., [16] recently proposed a semantic approach called COMMENT-MINE. It mines syntactically and semantically similar comments using a knowledge graph. The knowledge graph was initially developed based on human annotation. Their approach uses similar comments with the objective of program comprehension using a knowledge graph. On the other hand, SoCCMiner does not depend on prior annotation for understanding the context that leads to program comprehension. Its context attributes enable an improved understanding of the scope and context of the source code comment. It can support the automatic oracle creation for similar use cases.

7 CONCLUSION

Several studies in the past either focus on a specific type of source code comment or the entire project comments without source code context information and other comment attributes. The lack of a holistic data mining mechanism is a limiting factor in researching various use cases associated with source code comments. We utilize the AST representation of srcML [4] tool and develop a reusable and customizable tool that offers four different mining pipelines to extract extended comment attributes, its associated context information, and other source code attributes.

The proposed SoCCMiner tool prototype is versatile for handling GitHub repository URLs directly or accepting local project directories containing project repositories. The source code is available online [20] , and the package "soccminer" is available at the official PyPI (Python Package Index) [9] repository. The developed SoCCMiner data mining pipelines are evaluated with five different, widely used open-source Java repositories. However, it can be used to mine other Java software repositories. Based on the literature, we believe that SoCCMiner can benefit the software research community that focuses on source code comments.

For future work, we intend to extend SoCCMiner to accommodate other programming language project repositories such as `C, C++, C#` repositories. In addition, we plan to use SoCCMiner for studying SATD at various granularity and characterizing SATD source code instances which would lead to correlation analysis of source code with code/design smells and other source code metrics.

We acknowledge the Academy of Finland (grant ID 328058) for the financial support.

REFERENCES

[1] Eman Abdullah AlOmar, Ben Christians, Mihal Busho, Ahmed Hamad AlKhalid, Ali Ouni, Christian Newman, and Mohamed Wiem Mkaouer. 2021. SATDBailiff-mining and tracking self-admitted technical debt. *Science of Computer Programming* (2021), 102693.

[2] Gabriele Bavota and Barbara Russo. 2016. A large-scale empirical study on self-admitted technical debt. In *Proceedings of the 13th International Conference on Mining Software Repositories*. 315–326.

[3] Stefan Behnel, Martijn Faassen, and Ian Bicking. 2005. lxml: XML and HTML with Python.

[4] Michael L Collard, Michael John Decker, and Jonathan I Maletic. 2013. srcml: An infrastructure for the exploration, analysis, and manipulation of source code: A tool demonstration. In *2013 IEEE International Conference on Software Maintenance*. IEEE, 516–519.

[5] Mário André de Freitas Farias, Manoel Gomes de Mendonça Neto, André Batista da Silva, and Rodrigo Oliveira Spínola. 2015. A contextualized vocabulary model for identifying technical debt on code comments. In *2015 IEEE 7th international workshop on managing technical debt (MTD)*. IEEE, 25–32.

[6] Mário André de Freitas Farias, Manoel Gomes de Mendonça Neto, Marcos Kalinowski, and Rodrigo Oliveira Spínola. 2020. Identifying self-admitted technical debt through code comment analysis with a contextualized vocabulary. *Information and Software Technology* 121 (2020), 106270.

[7] Sergio Cozzetti B de Souza, Nicolas Anquetil, and Káthia M de Oliveira. 2005. A study of the documentation essential to software maintenance. In *Proceedings of the 23rd Annual International Conference on Design of Communication: Documenting & Designing for Pervasive Information*. 68–75.

[8] Jernej Flisar and Vili Podgorelec. 2019. Identification of self-admitted technical debt using enhanced feature selection based on word embedding. *IEEE Access* 7 (2019), 106475–106494.

[9] Python Software Foundation. 2003. *Python Package Index - PyPI*. https://pypi.org/

[10] Carl S Hartzman and Charles F Austin. 1993. Maintenance productivity: Observations based on an experience in a large system environment. In *Proceedings of the 1993 conference of the Centre for Advanced Studies on Collaborative research: software engineering-Volume 1*. 138–170.

[11] Walid M Ibrahim, Nicolas Bettenburg, Bram Adams, and Ahmed E Hassan. 2012. On the relationship between comment update practices and software bugs. *Journal of Systems and Software* 85, 10 (2012), 2293–2304.

[12] Ninus Khamis, René Witte, and Juergen Rilling. 2010. Automatic quality assessment of source code comments: the JavadocMiner. In *International Conference on Application of Natural Language to Information Systems*. Springer, 68–79.

[13] Douglas Kramer. 1999. API documentation from source code comments: a case study of Javadoc. In *Proceedings of the 17th Annual International Conference on Computer Documentation*. 147–153.

[14] Alexander LeClair, Sakib Haque, Lingfei Wu, and Collin McMillan. 2020. Improved code summarization via a graph neural network. In *Proceedings of the 28th International Conference on Program Comprehension*. 184–195.

[15] Zhongxin Liu, Qiao Huang, Xin Xia, Emad Shihab, David Lo, and Shanping Li. 2018. Satd detector: A text-mining-based self-admitted technical debt detection tool. In *Proceedings of the 40th International Conference on Software Engineering: Companion Proceeedings*. 9–12.

[16] Srijoni Majumdar, Shakti Papdeja, Partha Pratim Das, and Soumya Kanti Ghosh. 2020. Comment-Mine—A Semantic Search Approach to Program Comprehension from Code Comments. In *Advanced Computing and Systems for Security*. Springer, 29–42.

[17] Aniket Potdar and Emad Shihab. 2014. An exploratory study on self-admitted technical debt. In *2014 IEEE International Conference on Software Maintenance and Evolution*. IEEE, 91–100.

[18] Daniel Schreck, Valentin Dallmeier, and Thomas Zimmermann. 2007. How documentation evolves over time. In *Ninth international workshop on Principles of software evolution: in conjunction with the 6th ESEC/FSE joint meeting*. 4–10.

[19] Yusuke Shinyama, Yoshitaka Arahori, and Katsuhiko Gondow. 2018. Analyzing code comments to boost program comprehension. In *2018 25th Asia-Pacific Software Engineering Conference (APSEC)*. IEEE, 325–334.

[20] Murali Sridharan, Mika Mäntylä, Maëlick Claes, and Leevi Rantala. 2021. *SoCCMiner: A Comprehensive Granular Level Natural Language Text Miner*. M3S, University of Oulu, Oulu, Finland. https://github.com/M3SOulu/soccminer/

[21] Murali Sridharan, Mika Mäntylä, Leevi Rantala, and Maëlick Claes. 2021. Data Balancing Improves Self-Admitted Technical Debt Detection. In *18th IEEE/ACM International Conference on Mining Software Repositories, MSR 2021, Madrid, Spain, May 17-19, 2021*. IEEE, 358–368. https://doi.org/10.1109/MSR52588.2021.00048

[22] Daniela Steidl, Benjamin Hummel, and Elmar Juergens. 2013. Quality analysis of source code comments. In *2013 21st International Conference on Program Comprehension (ICPC)*. IEEE, 83–92.

[23] Scott N Woodfield, Hubert E Dunsmore, and Vincent Yun Shen. 1981. The effect of modularization and comments on program comprehension. In *Proceedings of the 5th International Conference on Software Engineering*. 215–223.

[24] Fiorella Zampetti, Cedric Noiseux, Giuliano Antoniol, Foutse Khomh, and Massimiliano Di Penta. 2017. Recommending when design technical debt should be self-admitted. In *2017 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 216–226.

[25] Nico Zazworka, Rodrigo O Spínola, Antonio Vetro', Forrest Shull, and Carolyn Seaman. 2013. A case study on effectively identifying technical debt. In *Proceedings of the 17th International Conference on Evaluation and Assessment in Software Engineering*. 42–47.