TASK CONTEXT: A Tool for Predicting Code Context Models for Software Development Tasks

Yifeng Wang
Zhejiang University
Hangzhou, China
yifeng.wang@zju.edu.cn

Yuhang Lin
Zhejiang University
Hangzhou, China
lin_yh@zju.edu.cn

Zhiyuan Wan*
Zhejiang University
Hangzhou, China
wanzhiyuan@zju.edu.cn

Xiaohu Yang
Zhejiang University
Hangzhou, China
yangxh@zju.edu.cn

***Abstract*— A code context model consists of code elements and their relations relevant to a development task. Previous studies found that the explicit formation of code context models can benefit software development practices, e.g., code navigation and searching. However, little focus has been put on how to proactively form code context models. In this paper, we propose a tool named TASK CONTEXT for predicting code context models and implement it as an Eclipse plug-in. TASK CONTEXT uses the abstract topological patterns of how developers investigate structurally connected code elements when performing tasks. The tool captures the code elements navigated and searched by a developer to construct an initial code context model. The tool then applies abstract topological patterns with the initial code context model as input and recommends code elements up to 3 steps away in the code structure from the initial code context model. The experimental results indicate that our approach can predict code context models effectively, with a significantly higher F-measure than the state-of-the-art (0.57 over 0.23 on average). Furthermore, the user study suggests that our tool can help practitioners complete development tasks faster and more often as compared to standard Eclipse mechanism.**

Demo video: https://youtu.be/3yEPh6uvHI8

Repository: https://github.com/icsoft-zju/Task_Context

***Index Terms*—Context Models, Task, Interaction, Context Prediction**

I. INTRODUCTION

As software developers perform development tasks, they spend substantial time searching and navigating through code in software systems. In the course of understanding the relevant code, they form, in their minds, implicit *code context models* consisting of source code elements and relations between those elements relevant to the tasks [1]. When a portion of such models can be made explicit, the information in the models can benefit software developers and software development projects. Prior studies show that making code context models explicit in software tools can support code recommendations [2], [3] and improve the quality of code changes [4].

Previous studies proposed tools to help developers explicitly capture code context models, e.g., Concern Graphs [5], Code Bubbles [6], and Code Basket [7]. Concern Graphs enable the developers to manually capture the code elements and their relations in the code context models [5]. Code Bubbles allow developers to create views of code fragments relative to development tasks being performed [6]. Code Basket provides a canvas on which developers can organize code context models to externalize their mental models [7]. These tools help developers create code context models by themselves after developers identify or navigate relevant code elements for work being performed.

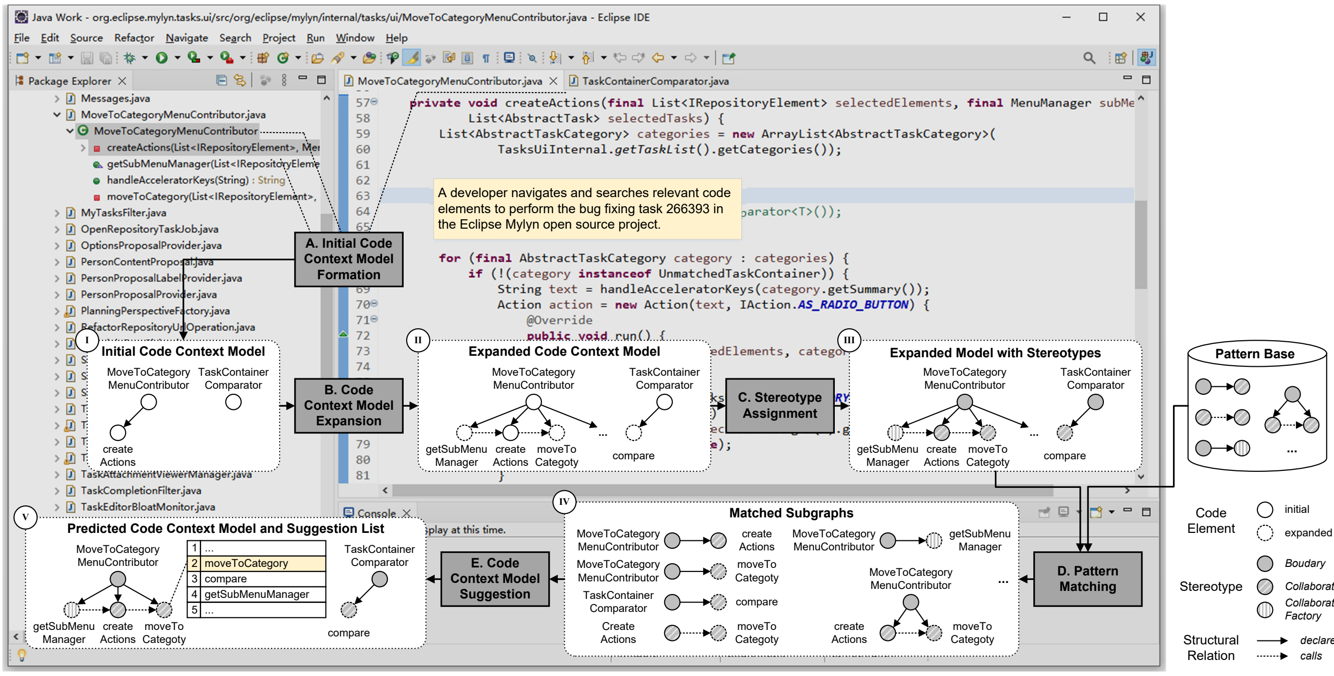
*Corresponding author.

Despite the promise of improving software tools with explicit code context models, little focus has been put on how to proactively form code context models, i.e., to predict code elements in code context models. The proactive formation of code context models would represent a collection of other code elements and relationships a developer is likely to need to draw on to complete the task, beyond a recommendation of what is the next code element for the developer to consider. Researchers used a variety of information to enable the proactive formation of code context models, including the structural information of source code (e.g., *Suade* [8]) and the history data of development tasks [9]–[11].

In a recent study, Wan et al. [12] proposed an approach that integrates structural information of source code and historical data of tasks for the proactive formation of code context models. The approach first constructs code context models with user interaction histories collected as developers work at different points of time in the development of a system. From the code context models, the approach learns abstract topological patterns of how developers investigate structurally connected code elements. Finally, the approach applies the learned patterns to enable a *d-step* prediction of future code context models, where the predicted code elements are up to d steps away in the structure from the code elements of interest as navigated and searched by a developer.

In this paper, we strengthen the approach [12] by implementing a tool as an Eclipse plug-in named TASK CONTEXT. The tool captures the code elements a developer navigates and searches when performing a development task, and constructs an initial code context model. Taking the initial code context model as input, the tool applies the abstract topological patterns in a pattern base and recommends code elements relevant to the current task.

We evaluate TASK CONTEXT with the interaction histories of bug fixing tasks created and stored as part of the Eclipse Mylyn open source project. The experimental results indicate that our approach can predict code context models effectively,


Java Work - org.eclipse.mylyn.tasks.ui/src/org/eclipse/mylyn/internal/tasks/ui/MoveToCategoryMenuContributor.java - Eclipse IDE
File Edit Source Refactor Navigate Search Project Run Window Help
Package Explorer
Messages.java
MoveToCategoryMenuContributor.java
MoveToCategoryMenuContributor
createActions(List<IRepositoryElement>, Mer
getSubMenuManager(List<IRepositoryEleme
handleAcceleratorKeys(String) : String
moveToCategory(List<IRepositoryElement>,
MyTasksFilter.java
OpenRepositoryTaskJob.java
OptionsProposalProvider.java
PersonContentProposal.java
PersonProposalLabelProvider.java
PersonProposalProvider.java
PlanningPerspectiveFactory.java
RefactorRepositoryUrlOperation.java
TaskAttachmentViewerManager.java
TaskCompletionFilter.java
TaskEditorBloatMonitor.java
MoveToCategoryMenuContributor.java
TaskContainerComparator.java
Console
A developer navigates and searches relevant code elements to perform the bug fixing task 266393 in the Eclipse Mylyn open source project.
A. Initial Code Context Model Formation
I
Initial Code Context Model
MoveToCategory MenuContributor
TaskContainer Comparator
create Actions
B. Code Context Model Expansion
II
Expanded Code Context Model
MoveToCategory MenuContributor
TaskContainer Comparator
getSubMenu Manager
create Actions
moveTo Category
compare
C. Stereotype Assignment
III
Expanded Model with Stereotypes
MoveToCategory MenuContributor
TaskContainer Comparator
getSubMenu Manager
create Actions
moveTo Category
compare
Pattern Base
D. Pattern Matching
IV
Matched Subgraphs
MoveToCategory MenuContributor
create Actions
MoveToCategory MenuContributor
getSubMenu Manager
MoveToCategory MenuContributor
moveTo Category
TaskContainer Comparator
compare
MoveToCategory MenuContributor
Create Actions
moveTo Category
create Actions
moveTo Category
E. Code Context Model Suggestion
V
Predicted Code Context Model and Suggestion List
MoveToCategory MenuContributor
TaskContainer Comparator
getSubMenu Manager
create Actions
moveTo Category
compare
Code Element
initial
expanded
Stereotype
Boudary
Collaborator
Collaborator-Factory
Structural Relation
declares
calls

```
private void createActions(final List<IRepositoryElement> selectedElements, final MenuManager subMe
        List<AbstractTask> selectedTasks) {
    List<AbstractTaskCategory> categories = new ArrayList<AbstractTaskCategory>(
            TasksUiInternal.getTaskList().getCategories());

                                                              parator<T>());

    for (final AbstractTaskCategory category : categories) {
        if (!(category instanceof UnmatchedTaskContainer)) {
            String text = handleAcceleratorKeys(category.getSummary());
            Action action = new Action(text, IAction.AS_RADIO_BUTTON) {
                @Override
                public void run() {
```

1	...
2	moveToCategory
3	compare
4	getSubMenuManager
5	...

Fig. 1. Running example and overall framework of TASK CONTEXT.

with a significantly higher F-measure than the state-of-the-art approach *Suade* [8] (0.57 over 0.23 on average). In addition to *1-step* prediction, which is supported by *Suade*, our approach supports *2-step* and *3-step* predictions. Furthermore, we conduct a user study with 12 software practitioners to investigate the effectiveness of TASK CONTEXT in practical use. The user study suggests that practitioners, with additional support from our tool, complete development tasks faster and more often than those using standard Eclipse mechanism.

II. APPROACH

Fig. 1 illustrates the overall framework of TASK CONTEXT with a running example of the bug fixing task 266393 in the Mylyn project. The overall framework of our approach consists of five steps: A. initial code context model formation, B. code context model expansion, C. stereotype assignment, D. pattern matching, and E. code context model suggestion, as described in the following subsections.

A. Initial Code Context Model Formation

Our approach first captures the code elements accessed by a developer and forms an initial code context model. We consider four types of code elements, *interface*, *class*, *method*, and *field*, as well as four structural relations among them, *inherits*, *implements*, *declares*, and *calls*. The code elements and their structural relations correspond to vertices and edges in the code context model, respectively. Fig. 1(I) illustrates an example initial code context model, which consists of two *class* elements, `MoveToCategoryMenuContributor` and `TaskContainerComparator`, and one *method* element `CreateActions`.

B. Code Context Model Expansion

In this step, our approach expands the initial code context model to include likely accessed code elements. Specifically, our approach expands the initial model along its structural relations with the prediction step d as the depth. The expanded code context model includes additional structurally connected code elements and their structural relations, as compared to those in the initial model. With regard to the running example, the expanded code context model includes multiple additional code elements, e.g., a *method* element `compare` connected with the *class* element `TaskContainerComparator`, as shown in Fig. 1(II).

C. Stereotype Assignment

Our approach assigns *stereotypes* to code elements to abstract from specific code elements. Stereotypes represent the roles code elements play in a system and their behavioral aspects and design intents. Specifically, our approach uses *JStereoCode* [13] to assign stereotypes to the code elements in the expanded code context model. As a result, our approach forms an expanded model with stereotypes. As for the running example, the *class* elements are assigned with the *Boundary* stereotype, while the *method* elements are assigned with the *Collaborator* and the *Collaborator-Factory* stereotypes, as shown in Fig. 1(III).

D. Pattern Matching

Our approach relies on abstract topological patterns of how developers investigate structurally connected code elements during development tasks. Previous work [12] formed a pattern base that includes such patterns mined from the Mylyn open source project (see Section III-A for details). Our approach first takes the pattern base and the expanded code context model with stereotypes as input. Our approach then iterates over each pattern in the pattern base and compares it with the subgraphs in the expanded model with respect to the stereotypes of their vertices. Once a match occurs, our approach records the matched subgraph. Fig. 1(IV) shows six matched subgraphs after pattern matching for the running example.

E. Code Context Model Suggestion

In this step, our approach suggests code elements in the future code context model for the development task. Specifically, our approach merges all the matched subgraphs and calculates the confidence value for each code element in the predicted code context model. The confidence value of each code element indicates the frequency of occurrences across the matched subgraphs. Our approach suggests the predicted code elements with a list sorted by the confidence value for the running example, as shown in Fig. 1(V).

III. Task Context Plug-in

We implement our tool in the form of an Eclipse plug-in. The following subsections describe how we form the pattern base and implement the plug-in.

A. Pattern Base

Task Context relies on abstract topological patterns of how developers investigate structurally connected code elements during development tasks to predict code context models. To obtain such patterns, we collected 1,219 interaction histories of bug fixing tasks from the Mylyn project and broke each interaction history into two-hour working periods. For each working period, we identified the code elements and retrieved the history commit in the corresponding code repository as the code snapshot(s). With the code snapshot(s) as input, we extracted structural relations among the code elements identified by *Doxygen* [14] and formed a code context model for each working period. As a result, we compiled a dataset of 1,887 code context models. Based on the dataset, we used *JStereoCode* to assign stereotypes to the code elements in the code context models and further adopted the *gSpan* algorithm[1] to mine abstract topological patterns in the models. Each pattern represents a frequently occurring abstract graph with stereotypes as its vertices. As a result, we obtained a pattern base with 142 abstract topological patterns.

B. Tool Implementation

The plug-in consists of two views, *Task Context* and *Suggestions*, as shown in Fig. 2 and Fig. 3. The plug-in captures the code elements accessed by a developer in Eclipse and constructs an initial code context model, which is displayed in the *Task Context* view. The *Suggestions* view displays a list of code elements recommended by our approach, with the information of their stereotypes and confidence of the prediction. We describe the implementation of our tool according to the overall framework of the approach as follows:

[1]gspan-mining v0.2.2, https://pypi.org/project/gspan-mining

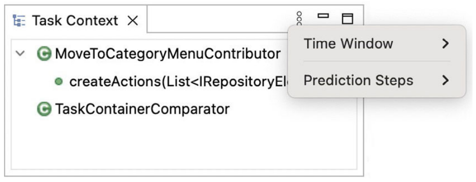
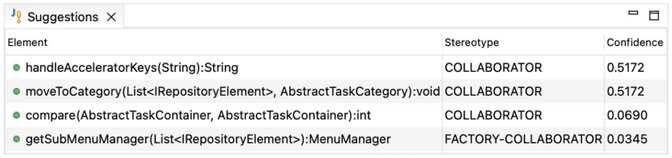

Task Context
MoveToCategoryMenuContributor
createActions(List<IRepositoryEl
TaskContainerComparator

Fig. 2. Initial code context model in the *Task Context* view.


Suggestions

Element	Stereotype	Confidence
● handleAcceleratorKeys(String):String	COLLABORATOR	0.5172
● moveToCategory(List<IRepositoryElement>, AbstractTaskCategory):void	COLLABORATOR	0.5172
● compare(AbstractTaskContainer, AbstractTaskContainer):int	COLLABORATOR	0.0690
● getSubMenuManager(List<IRepositoryElement>):MenuManager	FACTORY-COLLABORATOR	0.0345

Fig. 3. Results of *1-step* prediction in the *Suggestions* view.

Initial Code Context Model Formation. Our tool first uses the `org.eclipse.ui.ISelectionListener` class to record the code elements from the navigation and searching events a developer performs during a development task. Note that our tool supports recording the code elements within a time window, which can be customized by developers. To form the initial code context model, our tool uses the `getChildren` method of the `org.eclipse.jdt.core.IType` class to connect each *class* element with its declared *method* and *field* elements. Fig. 2 shows the tree structure of the initial code context model in the *Task Context* view as for the running example.

Code Context Model Expansion. Our tool uses breadth-first searching to expand the initial code context model along various relations (i.e., *declares*, *inherits*, *implements*, and *calls*), and with the prediction step d as the depth. By default, our tool expands the code context model 1 step away from the initial code context model to enable *1-step* prediction. The developers are allowed to specify the prediction step, i.e., *1-step*, *2-step*, and *3-step*, which are supported by our tool. For the *declares* relation, our tool uses the `IType`, `IMethod`, and `IField` classes in the `org.eclipse.jdt.core` package to search for structurally connected code elements. In terms of the *inherits*, *implements*, and *calls* relations, our tool uses the class hierarchy and call graph obtained from the abstract syntax trees of the code elements.

Stereotype Assignment. After the completion of code context model expansion, our tool bootstraps the `StereotypeIdentifier` class in *JStereoCode* to identify stereotypes of code elements in the expanded code context model. The resulting stereotypes are recorded in the `StereotypedElement` class. Thus, our tool leverages the information recorded in the class and relates each code element in the expanded model with its stereotype.

Pattern Matching. For each pattern in the pattern base, our tool uses the *VF3* algorithm [15] to find the subgraphs in the expanded coed context model with stereotypes that match the pattern. As a result, our tool generates a predicted code context model by merging all matched subgraphs.

Code Context Model Suggestion. With the predicted code context model, our tool first excludes the code elements in the initial code context model and then calculates the confidence values for the rest. The confidence value of a code element denotes the ratio of its occurrence frequency to the total number of matched subgraphs. Fig. 3 shows that the *Suggestions* view displays the predicted code elements in reverse order of their confidence values. The developers can click the predicted code element and navigate to its source code in the Eclipse Editor.

IV. Evaluation

A. Quantitative Evaluation

To evaluate the effectiveness of our approach, we use the dataset we collected from the interaction histories of the Mylyn open source project from the year 2007 to 2011. The dataset includes 1,887 code context models. We divide the dataset into a training set and a test set. Specifically, given the sequential nature of our dataset, we select the 1,254 code context models from the year 2007 to 2009 (84%) as the training set and the remaining 231 code context models from the year 2010 to 2011 (16%) as the test set. We mine abstract topological patterns from the training set. Based on the test set, we simulate the scenario when a developer starts a development task, and our approach proactively forms the code context model for the task.

We adopt the commonly used metric F-measure to measure the effectiveness of prediction. The results show that our approach achieves an average F-measure of 0.57 for *1-step* prediction, which is significantly higher than that of the state-of-art approach *Suade* (0.23). In addition, our tool supports *2-step* and *3-step* predictions.

B. User Study

To investigate if our tool can help in practical use, we conduct a user study with 12 participants, with 6 in the control group and 6 in the experimental group. The participants in the control and experimental groups are asked to perform three real-world programming tasks in Eclipse, using standard Eclipse mechanism and with additional support from our tool, respectively. We compare the completion rates and time of tasks between the two groups and collect feedback from our participants through a post-study survey.

Participants. We recruit 12 participants (3 female, 9 male) from our university and split them into two groups, i.e., control and experimental. Each group has 4 undergraduate students and 2 master students, with an average of 2.8 (control) and 2.3 (experimental) years of development experience.

Tasks. We select three real-world programming tasks with three levels of difficulty (i.e., easy, medium, and hard) from the bug fixing tasks in the Mylyn project by referring to their fixing time and number of edits recorded in the interaction histories. We further make some minor adjustments to the three tasks, ensuring they are easy to understand.

- ***Task 1: sort list*** (easy, bug fixing task 266393) Modify the existing comparator `TaskContainerComparator`. Use it to sort the list of task categories.

TABLE I
Results of user study.

	Control Group		**Experimental Group**	
	# Complete	Avg. Time	# Complete	Avg. Time
Task 1 (easy)	2	10:40	4	8:55
Task 2 (medium)	2	14:57	3	14:27
Task 3 (hard)	2	17:51	4	14:45

- ***Task 2: adjust views*** (medium, bug fixing task 278485) Create a new layout in the `EditorUtil` class. Apply the new layout to two views.
- ***Task 3: change sorting method*** (hard, bug fixing task 213901) Implement a function of getting the activation date of a task in the `AbstractTask` class. Make the history list of tasks sorted by activation date.

Protocol. The participants in the control group are allowed to use standard Eclipse mechanism, including searching and navigation features in Eclipse. For the experimental group, the participants can use our tool for additional support during the tasks. We set the time limits at 15, 20, and 25 minutes for *Task 1*, *Task 2*, and *Task 3*, respectively. Note that the participants have 5 minutes to get familiar with the code repository before starting *Task 1*. After a participant submits the code for tasks within time limits, we inspect the code to confirm whether the task is correctly done. In addition, we perform a post-study survey with the participants in the experimental group to collect feedback on our tool.

Results. As shown in Table I, participants assisted by our tool complete development tasks more often and with less time. On one hand, only 2 participants in the control group complete each task, while 4, 3, and 4 participants in the experimental group complete *Task 1*, *Task 2*, and *Task 3*, respectively. On the other hand, it takes the participants in the experimental group 16.4%, 3.3%, and 17.4% less time to complete the three tasks.

The post-study survey uses Likert-scale questions to collect feedback on our tool from the participants. The majority of the participants in the experimental group agree or strongly agree that our tool *is easy to use* (4 out of 6), *can reduce their time of browsing and understanding code* (6 out of 6), and *the predicted code elements are helpful* (5 out of 6).

V. Conclusion and Future Work

In this paper, we present Task Context, a tool implemented as an Eclipse plug-in to improve the proactive formation of code context models. The tool leverages the abstract topological patterns mined from interaction histories and proactively forms code context models to help developers perform software development tasks. Future work can extend Task Context to support more programming languages and take into account the experience level or other background information of developers when predicting code context models.

VI. Acknowledgements

The authors kindly acknowledge support from the National Key R&D Program of China (No. 2020YFB1005400) and National Science Foundation of China (No. 62102358). This is a demonstration paper that supplements the full research paper [12].

References

[1] T. Fritz, D. C. Shepherd, K. Kevic, W. Snipes, and C. Bräunlich, "Developers' code context models for change tasks," in *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ser. FSE 2014. New York, NY, USA: Association for Computing Machinery, 2014, p. 7–18.

[2] M. Kersten and G. C. Murphy, "Using task context to improve programmer productivity," in *Proceedings of the 14th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ser. SIGSOFT '06/FSE-14. New York, NY, USA: Association for Computing Machinery, 2006, p. 1–11.

[3] R. Robbes and M. Lanza, "Improving code completion with program history," *Automated Software Engineering*, vol. 17, 06 2010.

[4] D. Cubranic and G. Murphy, "Hipikat: recommending pertinent software development artifacts," in *25th International Conference on Software Engineering, 2003. Proceedings.*, 2003, pp. 408–418.

[5] M. P. Robillard and G. C. Murphy, "Concern graphs: Finding and describing concerns using structural program dependencies," in *Proceedings of the 24th International Conference on Software Engineering*, ser. ICSE '02. New York, NY, USA: Association for Computing Machinery, 2002, p. 406–416.

[6] A. Bragdon, R. Zeleznik, S. P. Reiss, S. Karumuri, W. Cheung, J. Kaplan, C. Coleman, F. Adeputra, and J. J. LaViola, "Code bubbles: A working set-based interface for code understanding and maintenance," in *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, ser. CHI '10. New York, NY, USA: Association for Computing Machinery, 2010, p. 2503–2512.

[7] B. Biegel, S. Baltes, I. Scarpellini, and S. Diehl, "Codebasket: Making developers' mental model visible and explorable," in *Proceedings of the Second International Workshop on Context for Software Development*, ser. CSD '15. IEEE Press, 2015, p. 20–24.

[8] M. P. Robillard, "Topology analysis of software dependencies," *ACM Trans. Softw. Eng. Methodol.*, vol. 17, no. 4, aug 2008.

[9] M. Kersten and G. C. Murphy, "Using task context to improve programmer productivity," in *Proceedings of the 14th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ser. SIGSOFT '06/FSE-14. New York, NY, USA: Association for Computing Machinery, 2006, p. 1–11. [Online]. Available: https://doi.org/10.1145/1181775.1181777

[10] R. Robbes and M. Lanza, "Improving code completion with program history," *Automated Software Engineering*, vol. 17, no. 2, pp. 181–212.

[11] T. Zimmermann, A. Zeller, P. Weissgerber, and S. Diehl, "Mining version histories to guide software changes," *IEEE Transactions on Software Engineering*, vol. 31, no. 6, pp. 429–445, 2005.

[12] Z. Wan, G. C. Murphy, and X. Xia, "Predicting code context models for software development tasks," in *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE '20. New York, NY, USA: Association for Computing Machinery, 2021, p. 809–820.

[13] L. Moreno and A. Marcus, "Jstereocode: automatically identifying method and class stereotypes in java code," in *2012 Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering*, 2012, pp. 358–361.

[14] D. Van Heesch, "Doxygen: Source code documentation generator tool," *URL: http://www. doxygen. org*, 2008.

[15] V. Carletti, P. Foggia, A. Saggese, and M. Vento, "Challenging the time complexity of exact subgraph isomorphism for huge and dense graphs with vf3," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 40, no. 4, pp. 804–818, 2018.