amAssist: In-IDE Ambient Search of Online Programming Resources

Hongwei Li*†§, Xuejiao Zhao‡, Zhenchang Xing‡, Lingfeng Bao¶, Xin Peng*†, Dongjing Gao*†, Wenyun Zhao*†

*School of Computer Science, Fudan University, Shanghai, China
†Shanghai Key Laboratory of Data Science, Fudan University, Shanghai,China
‡School of Computer Engineering, Nanyang Technological University, Singapore
§School of Computer Information Engineering, Jiangxi Normal University, Nanchang, China
¶College of Computer Science, Zhejiang University, Hangzhou, China

{lihongwei,pengxin,wyzhao,gaodj14}@fudan.edu.cn {zcxing,xjzhao}@ntu.edu.sg
lingfengbao@zju.edu.cn

***Abstract*—Developers work in the IDE, but search online resources in the web browser. The separation of the working and search context often cause the ignorance of the working context during online search. Several tools have been proposed to integrate the web browser into the IDE so that developers can search and use online resources directly in the IDE. These tools enable only the shallow integration of the web browser and the IDE. Some tools allow the developer to augment search queries with program entities in the current snapshot of the code. In this paper, we present an in-IDE ambient search agent to bridge the separation of the developer's working context and search context. Our approach considers the developers' working context in the IDE as a time-series stream of programming event observed from the developer's interaction with the IDE over time. It supports the deeper integration of the working context in the entire search process from query formulation, custom search, to search results refinement and representation. We have implemented our ambient search agent and integrate it into the Eclipse IDE. We conducted a user study to evaluate our approach and the tool support. Our evaluation shows that our ambient search agent can better aid developers in searching and using online programming resources while working in the IDE.**

***Index Terms*—Context Sensing, Context-Aware Search, Contextual Search Results Annotation**

I. Introduction

As online programming resources contributes a fast-growing body of software development knowledge, it has become a common practice to interleave coding, web search, and learning during software development [1], [2], [3]. Developers work in the Integrated Development Environment (IDE), but search online resources in the web browser. As a result, search engines are unaware of the specifics of the developer's working context due to the separation of working and search context. Several tools [2], [4], [5], [6] have been proposed to integrate the web browser into the IDE. These tools allow developers to search and use online resources directly from within the IDE. These tools make two simplistic assumptions.

First, they assume that the developers' working context can be extracted from the current snapshot of the program (e.g., a program entity or exception that the developers currently select). Our empirical study [7] of developers' online search behavior shows that the developers' working context usually include a set of related program entities at different time periods. Furthermore, the developers often augment program context with task context (e.g., technology being used) and preference (e.g., searching for API specification or code example). This suggests that the dynamics of developers' working context and the task context and preference should be captured, modeled and used to improve online search.

Second, existing tools assume that integrating the IDE and the web browser can effectively bridge the separation of working and search context. They simply present search results in an IDE view in the same way as in the web browser. But more researcher's want to change this status, like Ponzanelli et al. [8] and Bacchelli et al. [4], they try to integrating online resources to IDE in their works. However, the IDE is crowed with many views of program information. As a result, the search results view can use only a small portion of the display. Previous study [2] shows that such a small search results view often becomes inefficient when the developer has to search and explore many online resources (e.g., to learn an unfamiliar API). Furthermore, a shallow integration of the IDE and the web browser do not effectively use the working context in customizing search and exploring search results. Context-aware custom search and exploration techniques are needed to enable the deep integration of working and search context.

In this paper, we present an in-IDE ambient search agent (called *amAssist*) to bridge the separation of the developer's working and search context. *amAssist* considers the developer's working context in the IDE as a time-series stream of programming event observed from the developer's interaction with the IDE over time. It unobtrusively monitors the developer's programming event in the IDE and determines the developer's working focus over time. *amAssist* deeply integrates the developer's working context in the online search process. The developer can use the context to augment his search query or refine the search results. *amAssist* uses the context to tweak the ranking of the search results. It also uses the context to annotate search results and web pages to help the developer assess and browse the search results.

We have implemented our ambient search agent and inte-

978-1-4799-8469-5/15/$31.00 © 2015 IEEE

grate it with the Eclipse IDE. We conducted a user study to evaluate our *amAssist* tool. The study involved 10 developers using our *amAssist* tool and standard Eclipse/Webbrowser respectively. The task was to fix a bug in an existing Eclipse editor plugin and extend the plugin with file save and content statistics features. Our study shows that *amAssit* can help the developers formulate more specific queries with working context information. As such, the developers using *amAssist* can find and integrate relevant online programming resources more quickly with less search queries.

The remainder of the paper is organized as follows. Section II describes the design of ambient search agent. Section III presents the *amAssist* tool. Section IV evaluates our ambient search approach and the *amAssist* tool. Section V reviews related work. Section VI concludes the work and discusses our future plan.

II. The Approach

amAssist unobtrusively monitors the developer's programming activity in the IDE and determines the developer's working focus over time. It visualizes the API entities that are potentially of the most interest to the developer in an interactive visualization [9] (e.g., `foam tree` [1]). The developer can formulate a search query by combining the API entities in the foam tree with his own keywords. *amAssist* integrates Google Custom Search Engine to search popular programming websites. It tweaks the ranking of the search results based on the interest values of API entities in the search query and the developer's preference for websites. The developer can select other API entities in the foam tree to refine the search results. To help the developer assess the relevance of search results, *amAssist* clusters the search results based on the topics mined from the search results. It augments the Google search results entries with the API entities in the foam tree that are mentioned in the corresponding web pages. To help the developer browse a particular web page, *amAssist* highlights the API entities in the foam tree that are mentioned in the opened page.

A. Time-Series Working Context Sensing

Our definition of the developer's working context in the IDE is not a set of program entities at the current snapshot of the code. Instead, the working context is a time-series stream of programming event observed from the developer's interaction with the IDE over time. Table I summarizes observable programming event that *amAssist* tracks. A programming event consists of a tuple (t, a, P, L). t is an index of the event. It increases by one as a new event occurs. a is a category value describing an observable action performed by the developer. P is a set of source-code program entities directly involved in an action. L is a set of library or framework API entities that source-code entities extend and use.

Modern IDE supports various actions to access and update program entities in the IDE. We abstract these concrete actions into 5 types of observable actions. $select$ refers to the selection of a program entity in various views (e.g., Package Explorer, Outline, Type Hierarchy, Search Results, Compilation Problems, Stack Trace) and code editor of the IDE. $reveal$ refers to the action (e.g., open, select, scroll, switch) that makes a program entity visible or partially visible in code editor. $select$ action may $reveal$ a program entity in an opened code editor. In such cases, *amAssist* records only $reveal$ action. $save$ refers to saving code edits made by the developer in a compilation unit. $debug$ refers to debug actions such as set breakpoint, suspend execution, and step execution. $execute - exception$ refers to the program throwing an unhandled exception while the developer executes the program.

A source-code program entity $p_{src} \in P$ can be a compilation unit, class (including interface), method, field, and statement in the source code that the developer acts on. *amAssit* retrieves the program entity involved in an observation action using the appropriate IDE APIs (e.g., Eclipse JDT IJavaElement interfaces and Java DOM/AST classes). If the developer attaches source code to the library API entities, he can *select*, *reveal*, and *debug* API entities in the same way as source-code entities. *amAssist* tracks the events involving API entities with source code in the same ways as the real source-code entities. It adds the source-code-attached API entities in the set P of an event to the event's API entity set L, while it does not do so for the real source-code entities.

A library API entity $p_{api} \in L$ can be an API class, method, and field that a source-code program entity extends and uses. Online programming resources describe how to extend and use certain library or framework APIs. Thus, the application-specific source-code entities are meaningless for online search. *amAssist* uses the appropriate IDE APIs to retrieve the set of library API entities L (may be entity) extended or used by the set of source-code entities P of an event. Developers can use these library API entities to augment their search queries. If an API entity cause compilation errors or runtime exceptions, *amAssist* annotates the API entity accordingly.

TABLE I. Definition of Working Context

Action (a)	Src Entity (P)	Library API Entities (L)
Select Reveal	Class	Inherited API classes
	Method	Overridden methods
		Inherited API classes of parameter type
		Inherited API classes of return type
	Field	Inherited API classes of field type
Select	Statement	APIs used in the statement
Save	Compilation Unit	Inherited API classes of the edited classes
		Overridden methods of the edited methods
		APIs used in the edited statements
Suspend SetBreakpoint Step	Method	Overridden methods of the debugged methods
		Inherited API classes of parameter type
		Inherited API classes of return type
	Statement	APIs used in the debugged statement
Execute -Exception	Method	Overridden methods of the immediate exception-throwing method
		Inherited API classes of parameter type
		Inherited API classes of return type
	Statement	APIs used in the exception-throwing statement
	Exception (Runtime Object)	API exception classes
		Immediate exception-throwing API method

Table II shows an example of a stream of programming events observed from the developer's interaction with the IDE and the program. The developer double-clicks the class $MyEditPart$ to open the class. $MyEditPart$ extends the Eclipse class $EditorPart$. The developer

[1] http://carrotsearch.com/foamtree-overview

then selects the method $doSave(IProgressMonitor)$ in the outline view to reveal the method. This method overrides $EditorPart.doSave(IProgressMonitor)$. Next, the developer edits the method $doSave()$ to use $FileDialog$ APIs and saves the file. The developer runs the program. The program throws a Java exception $IOException$. The immediate exception-throwing API method is $FileWriter.FileWriter(File)$. The developer selects in the stack trace the code statement (calls $FileWriter.FileWriter(File)$. The developer sets a breakpoint at the code statement (calls $FileDialog.open()$) just before the exception-throwing statement (calls $FileWriter.FileWriter(File)$). The developer runs the program. The program suspends at the breakpoint statement (calls $FileDialog.open()$). The developer realizes that he needs to use absolute path to call $FileWriter.FileWriter(File)$. He finally edits and saves the file.

B. Working Context Summarization and Visualization

As the programming events occur, *amAssist* encodes the developer's working context into a Degree-of-Interests (DOI) model of library API entities. It visualizes the API entities that are potentially of the most interest to the developer.

1) Computing the DOI Model of API Entities: A design issue of *amAssist* is content variance [10]. If the working context rarely changes, developers will stop looking over because they have already seen the information. On the other hand, if the working context changes too often it could be a distraction, or relevant information could be swapped out while the developer is trying to view it. Thus, *amAssist* should detect important focus shifts in the working context. Meanwhile, it should remain relative stable so that the contextual information has time to be used.

The *amAssist* DOI model of API entities is inspired by the Fisheye view [11] and the Read-Edit-Wear model [12]. This DOI model associates an interest value with each distinct API entity in the working context. The initial interest value of an API entity of an event is assigned based on the intrinsic interest level of the action and the API entity of the event. The interest value of the API entity is then decayed as new events arrive in the working context.

We define four interest levels of the action: $select < reveal < save$ or $debug < execute - exception$. We denote the interest level of an action a using an integer value l_a ($1 \leq l_a \leq 4$). We define three interest levels of the API entity: $normal < has-compile-error < cause-exception$. We denote the interest level of an API entity p_{api} using an integer value l_{api} ($1 \leq l_{api} \leq 3$). Given an event $e < t, a, P, L >$ and an API entity $p_{api} \in e.L$, the initial interest value of p_{api} (i.e., $v_{init}(p_{api}, e)$ is computed as $\gamma^{l_a} \times l_{api}$, where γ is an integer constant value ($\gamma > 1$). According to this definition, the API entity that causes exception during program execution has the highest initial interest value, while the normal API entity that the developer selects has the lowest initial interest value.

We assume that the older an event $e < t, a, P, L >$ is, the less interests the developer has in $p_{api} \in e.L$. Thus, the current interest value of p_{api} in the event $e < t, a, P, L >$ (i.e., $v(p_{api}, e)$) is computed as $v_{init}(p_{api}, e)/\gamma^{t_m - t}$ where t_m is the current event index. An API entity p_{api} can be involved in several events in the working context. We assume that the more times an API entity p_{api} is used in the working context, the more interests the developer has in p_{api}. Thus, the interest value of p_{api} in the working context at the current event index t_m (i.e., $v(p_{api}, t_m)$) is computed as $\sum_{1 \leq i \leq t_m} f(p_{api}, e(i))$. The function $e(i)$ returns the event at the index i. The function $f(p_{api}, e(i))$ returns $v(p_{api}, e(i))$ if $p_{api} \in e(i).L$, otherwise returns 0.

When a new event $e < t_m + 1, a, P, L >$ occurs, *amAssist* updates the interest value of all the distinct API entities in the working context. The updated interest value of p_{api} in the working context at the latest event index $t_m + 1$ is computed as $v(p_{api}, t_m + 1) = v(p_{api}, t_m)/\gamma + f(p_{api}, e(t_m + 1))$. That is, the updated interest value of p_{api} is the sum of the decayed interest value of p_{api}'s previous interest value and the current interest value of p_{api} in the new event $e(t_m + 1)$.

Table III presents an example of DOI computation of the API entities collected in the example of dynamic working context in Table II . We can see that the developer's initial working focus is on $EditorPart$ related APIs, such as $EditorPart.doSave()$, $EditorPart.getSite()$, $EditorPart.getShell()$. As he works on implementing $doSave()$ method, his working focus shifts to $FileDialog$ related APIs, such as $FileDialog.open()$, $FileDialog.setFileName(String)$, and $File.getAbsolutePath()$. Note that once the exception occurs during program execution, the relevant API entities (such as the exception class $IOException$ and the exception-throwing method $FileDialog.open()$) are ranked high in the DOI model, because the exception is an unusual incident that should highly likely be attended first. However, these exception related API entities will not dominate the DOI model forever. Their interest levels gradually decay over time. Other API entities (such as $FileDialog.setFileName(String)$ and $File.getAbsolutePath()$) can still enter the top positions in the DOI model.

2) Visualizing the DOI Model of API Entities: The DOI model of API entities captures the developer's working focus over time. *amAssit* ranks the API entities by their interest values. It can use intuitive visualization to present the top N (e.g., 10) highest-interest-value API entities to the developer. For example, foam tree or tag cloud [13] are widely used visual representations of weighted list of text data. Such visualization can provide a quick overview of the API entities that are potentially of the most interest to the developer. *amAssit* uses two strategies to update the visualization of the top N highest-interest-value API entities. First, it updates the visualization immediately if the top N API entities change. Second, it updates the visualization at regular time interval T (e.g., 10 seconds) if the top N entities remain the same but their interest values change.

C. Ambient Search and Exploration

amAssist uses the developer's working context in the entire online search process, from query formulation, custom search, to search results refinement and representation.

1) Context-Aware Custom Search: The developer can interactively select one or more API entities in the visualization

TABLE II. AN EXAMPLE OF DYNAMIC WORKING CONTEXT THAT THE *amAssist* TOOL TRACKS

t	Action	SrcEntity	API Entities	Remark
1	Reveal	MyEditorPart (Class)	EditorPart	Inherited API classes
2	Select to Reveal	doSave(IProgressMonitor) (Method)	EditorPart.doSaveAs(IProgressMonitor) IProgressMonitor	Overridden methods Inherited API classes of parameter type
3	Save	FileDialog,fd =new FileDialog (this.getSite().getShell()) fd.setFileName("test.txt") String path = fd.open() FileWriter filewriter = new FileWriter (new File(path)) (Compilation Unit)	FileDialog.FileDialog(Shell) EditorPart.getSite() EditorPart.getShell() FileDialog.open() FileDialog.setFileName(String) FileWriter.FileWriter(File)	APIs used in the edited statements Overridden methods
4	Execute Exception	java.io.IOException (Exception)	java.io.IOException FileWriter.FileWriter(File)	API exception classes Immediate exception-throwing API method
5	Select	FileWriter filewriter = new FileWriter (new File(path)) (Statement)	FileWriter.FileWriter(File)	APIs used in the statement
6	Set Breakpoint	String path = fd.open() (Statement)	FileDialog.open()	API used in the debugged statement
7	Suspend	String path = fd.open() (Statement)	FileDialog.open()	API used in the debugged statement
8	Save	getAbsolutePath() (Method)	File.getAbsolutePath()	API used in the edited statement

TABLE III. AN EXAMPLE OF DOI COMPUTATION

API Entities	Action/Type	l_a	l_{api}	t1	t2	t3	t4	t5	t6	t7	t8
EditorPart	Reveal/Normal	2	1	4	2	1	0.5	0.25	0.125	0.0625	0.03125
EditorPart.doSaveAs(IProgressMonitor)	Select to Reveal/Normal	2	1	—	4	2	1	0.5	0.25	0.125	0.0625
IProgressMonitor	Select to Reveal/Normal	2	1	—	4	2	1	0.5	0.25	0.125	0.0625
FileDialog.FileDialog(Shell)	Save/Normal	3	1	—	—	8	4	2	1	0.5	0.25
EditorPart.getSite()	Save/Normal	3	1	—	—	8	4	2	1	0.5	0.25
EditorPart.getShell()	Save/Normal	3	1	—	—	8	4	2	1	0.5	0.25
FileDialog.open(String)	Save/Normal	3	1	—	—	8	4	2	—	—	—
	Set Breakpoint/Normal	3	1	—	—	—	—	—	9	—	—
	Suspend/Normal	3	1	—	—	—	—	—	—	12.5	6.25
FileDialog.setFileName(String)	Save/Has Compile Error	3	2	—	—	16	8	4	2	1	0.5
FileWriter.FileWriter(File)	Save/Normal	3	1	—	—	8	—	—	—	—	—
	Execute Exception/Cause Exception	4	3	—	—	—	52	—	—	—	—
	Select/Cause Exception	1	3	—	—	—	—	32	16	8	4
java.io.IOException	Execute Exception/Cause Exception	4	3	—	—	—	48	24	12	6	3
File.getAbsolutePath()	Save/Normal	3	1	—	—	—	—	—	—	—	8

of the top N highest-interest-value API entities as keywords in his search query. He can customize the selected API entity keywords and augment these API entity keywords with his own search keywords (e.g., task context). The developer can instruct *amAssist* to tweak the ranking of the search results using the interest values of the selected API entities.

amAssist integrates the `Google Custom Search API` [2] to search online program resources. It allows the developer to customize the Google Custom Search Engine to search his preferred websites. The developer can attach one or more category labels to these websites. By default, *amAssist* searches the following popular programming-oriented web sites as summarized in Table IV, such as technical blogs, code examples, discussion forums, and Q&A websites. The developer can select website category labels to inform Google Custom Search Engine his preference for certain categories of websites (e.g., code examples websites).

amAssist programmatically customizes the Google Custome Search Engine in two ways. First, if the developer indicates that he wants to use the interest values of the selected API entities to tweak the ranking of the search results, *amAssist* normalizes the interest values of the selected API entities as the weight of the search keywords for tweaking the ranking of the search results. It promotes the search results containing the higher-interest-value API entities in the results ranking. Second, if the developer indicates their preferred website categories, *amAssist* promotes the search results from the websites of the preferred categories in the results ranking. *amAssist* uses the BOOST mode of Google Custom Search Engine. That is, it promotes the websites of the developer's preferred categories without excluding other sites.

[2]https://developers.google.com/custom-search/

TABLE IV. WEB SITE CATEGORIES

Category Label	Web Sites Match String
Technical Blogs (TB)	*.iteye.com/blog/* blog.sina.com.cn/* blog.163.com/* www.360doc.com/content/*
Code Examples (CE)	blog.csdn.net/* *.iteye.com/blog/* *.code.google.com/* *.grepcode.com/* *.codeproject.com/*
Discussion Forum (DF)	bbs.csdn.net/* zhidao.baidu.com/* *.stackoverflow.com/* *.superuser.com/* *.stackexchange.com/* *.serverfault.com/*
Q&A web site (QA)	zhidao.baidu.com/* *.stackoverflow.com/* *.superuser.com/* *.stackexchange.com/* *.serverfault.com/*

2) Search Results Refinement: The developer's online search often returns a large number of search results. For the

reason of ease recognizing those search results that programmers want to explorer, It need to clustering those search results, like the way of topics, based on their snippets or contents. *amAssist* supports two ways for the developer to refine the search results.

First, the developer can select one ore more API entities (those not used as query keywords) in the visualization of the the top N highest-interest-value API entities. *amAssist* uses the selected API entities to refine the search results based on Google Custom Search's "Refine Search" feature.

Second, *amAssist* uses semantic clustering technique to cluster the search results. Semantic clustering reveals topics of search results by grouping search results that use similar vocabulary. *amAssit* can display the clusters of search results in a list view. Each list item represents a cluster. It can show the topics of the cluster and the number of search results in the cluster. It can also shows the number of search results from different category of web sites. The developer can filter the search results by the cluster topics and the category of web site he is interested in.

3) Search Results Representation and Browsing: The developer's online search does not end with presenting a list of relevant web pages [14]. The developer must also be able to recognize which web pages meets their particular need and make use of their content.

To help the developer access the relevance of search results, *amAssist* annotates the search-results entries with the working-context API entities mentioned in the corresponding web pages (see the *Search Results* view in Figure 1). For example, the annotation of the first search results entry shows that this web page contains not only the API entities used as keyword (i.e.,IWorkbenchPage.openEditor(IEditorPart)) but also other API entities in the working context foam tree (e.g., PartInitException, IOException) . Such API entity annotations provide a context-aware augmentation of the document surrogate [15]. To support responsive user interaction, *amAssist* can download and search for the working-context API entities in the search-results pages using multi-thread. It can update the search-results entry once the analysis of the corresponding web page is complete.

To help the developer make use of a particular web page, *amAssist* summarizes the working-context API entities mentioned in the opened web page in a tree view. Each tree node represents a mentioned API entity. Expanding the tree node will lists the places where the mentioned API entity appears in the web page. *amAssist* extracts a short excerpt surrounding such places (e.g., 5 words before and after the mentioned API entity). This overview of the mentioned API entities in the opened page can help the developer quickly locate and navigate to the content of the page he is interested in.

III. Tool Support

We have developed an *amAssist* tool and integrate the tool with the Eclipse IDE. The *amAssist* tool listens to the Eclipse workbench's selection, change, and runtime events to monitor the developer's programming activities in the IDE. It uses the Eclipse JDT IJavaElement APIs and Java DOM/AST APIs to resolve the program entities involved in the events. The *amAssist* tool visualizes the DOI model of the API entities using the interactive foam tree provided by the `Carrot2 Search` [3] (An open-source search results clustering engine).

The *amAssist* integrates the Google Custom Search Engine to support custom search and refinement of search results. To cluster the search results, *amAssist* retrieves the snippet of the top 50 (can be configured by the developer) search results from the search engine. A snippet is a small sample of web page content that the search engine returns with each search result to give search users an idea of what is the web page. *amAssist* applies the Lingo algorithm [16] provided by `Carrot2 Search` for semantic clustering of search results based on their snippets. Lingo algorithm reverses the traditional order of cluster discovery by first finding good, conceptually varied cluster labels and then assigning documents to the labels to form clusters [16]. It can generate longer, often more descriptive labels than other topic mining algorithms. This characteristic can help the developer better understand and select group of relevant search results.

Figure 1 presents the user interface of the *amAssist* tool in the Eclipse IDE. The *DOI Model* view display the top 10 (can be configured by the developer) highest-interest-value API entities that the *amAssist* tool summarizes in an interactive foam tree. The *Ambient Search* view contain a search box. The developer can enter their own keywords (e.g., task context) and select API entities in the foam tree as keywords. The developer can tweak the ranking of the search results using the interest values of the selected API entities.

The developer can select his preferred website categories in the list. The *Ambient Search* displays the search-results clusters in a list view as described in Section II-C2. The developer can select a cluster to filter the search results. The *Search Results* view shows the search results. *amAssist* annotates the search results entries as described in Section II-C3. The developer can select one or more search results clusters in the *Ambient Search* view to filter the search results. The developer can open a web page in the embedded Eclipse web browser. The API entities in the working context foam tree that are mentioned in the opened web page are summarized in the *Webpage Overview* view (not visible in the Figure 1). The developer can use this view to quickly locate and navigate to the parts of the web page he is interested in.

IV. Evaluation

Our *amAssit* tool aims to deepen the integration of the developers' working context in the IDE with their online search. To evaluate if the *amAssist* tool achieves this goal, we conducted a user study to investigate the following three research questions:

- Q1 Can the *amAssit* tool help developers formulate more specific queries and locate relevant online resources faster?
- Q2 How do developers use the *amAssist* tool during a software development task?
- Q3 How does the *amAssist* tool change the developers' behaviors during a software development task?

[3] http://project.carrot2.org

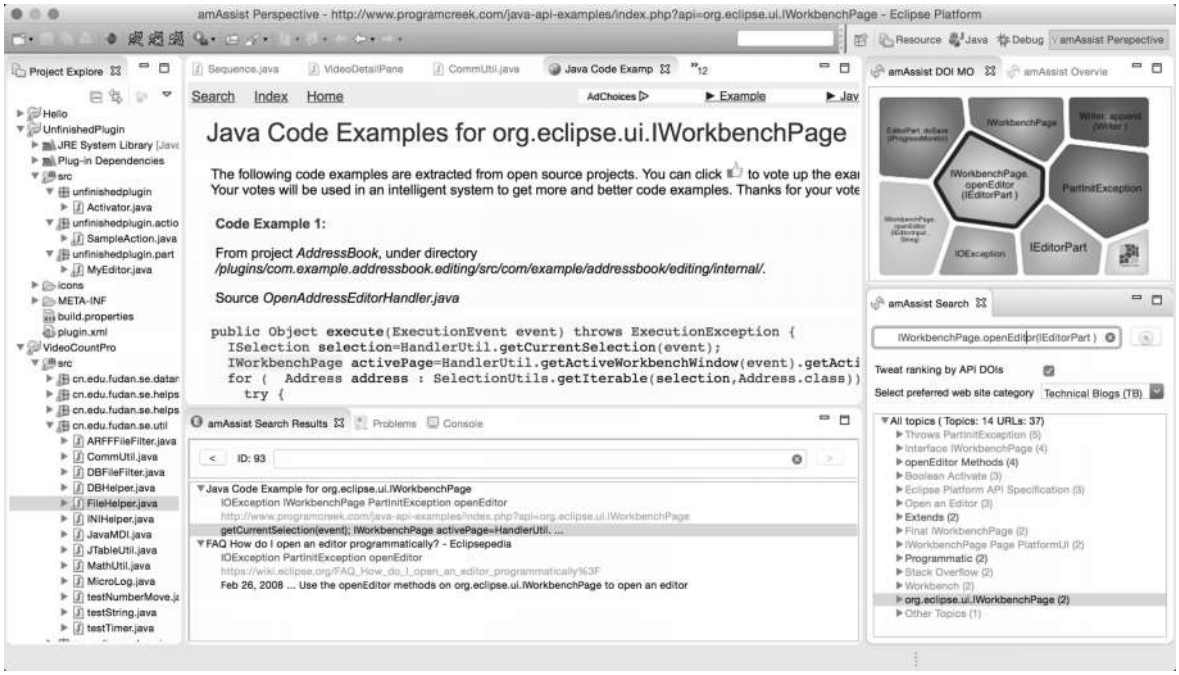


Fig. 1. The *amAssist* Tool

A. Experimental Design

Experiment Task: In this study we designed a software maintenance task. The participants were given the source code of a simple Eclipse editor plugin. They were asked to complete the two subtasks. First, the participants needs to fix a bug in the existing implementation. When the user opens a new editor, the existing implementation throws an $IllegalArgumentException$. Fixing this bug requires the knowledge of $EditorPart.openEditor()$ and $IEditorInput$ APIs. After fixing the bug, the participants needs to extending the editor plugin with file open/save/close and word count features. Implementing these new features require the knowledge of $EditorPart$'s file open/save/close APIs and status bar extension points.

Participants: We used between-subject design in our user study. We recruited 10 volunteer graduate students from the School of Computer Science, Fudan university. Based on our pre-experiment survey, all the participants described themselves as "above average" Java expertise or "Java experts". All the participants used Eclipse IDE regularly in their research work. We use P1,P2,...,P10 to represent them. These Participants were matched in pairs based on their programming experience and then randomly allocated to experimental group or control group. The participants of the experimental group G_1 used the *amAssist* tool to perform the software maintenance task. The participants of the control group G_2 used the Eclipse IDE and web browser to perform the same task. As an explanation for the pairs of participants, we use (P1,P6), (P2,P7), (P3,P8), (P4,P9), (P5,P10) to represent those pairs, the P1 to P5 from group G1 and the P6 to P10 from group G2.

Procedure: The participants were ask to work on the task in a 2-hour session. They were required to run a screen-capture software throughout the session. The screen-recorded task videos allow us to analyze the participants' behavior after the experiment. We also instrumented the *amAssist* tool to collect tool usage statistics.

After the experiment, we interviewed the two groups separately to collect their general feedbacks on the task and the tool usage. During our data analysis, we conducted individual interviews with participants as needed, for example to confirm the intention of their certain actions.

B. Results: Improvement on Search Performance (Q1)

We evaluate the search performance improvement of *amAssist* by comparing the following metrics of the experimental group (G_1) and the control group (G_2): task completion time, time spent on bug fixing subtask, the number of queries, the number of keywords, and the number of opened web pages. Table V shows the statistics of these performance metrics in the experiment group (G_1) and the control group (G_2).

We can see that there is no obvious difference in the overall task completion time. This is mainly due to the task design. All the participants in both the experiment group and control group completed the bug fixing subtask. However, none of them completed the feature extension subtask. That is why all the participants worked till the end of the experiment session. For the bug-fixing subtask that all the participants completed, the experimental group had shorter task completion time than the control group did.

We can see that the experimental group on average issued less search queries but they used more keywords, compared with the control group. Furthermore, the experimental group opened less web pages during the task. This can be attributed to the visualization of the developers' working context. The

visualization makes it explicit to the developers what they have been working on. It increases the chance that the developers can formulate more context-aware search queries using the API entities in the working context. More context-aware search queries can better reflect the developers' information needs, and thus can cause the more relevant web pages ranked higher in the search results. This may lead to the less effort in selecting relevant web pages, and consequently the less number of opened web pages.

We conducted Wilcoxon's matched-pairs signed-ranked tests to evaluate the significance of the differences in these search performance metrics of the experimental group and the control group. Our statistical tests show that the differences in these search performance metrics are not significant. This may be attributed to the limited number of participants and the limited task completed time. Our initial results show that the *amAssist* tool seems promising in improving the developers' online search by deepening the integration of the developers' working context in their online search. However, more systematic study is required to confirm the findings.

TABLE V. STATISTICS OF PERFORMANCE METRICS IN THE TWO GROUPS

Participant	Task Time	Bug Fix Time	#Query	#Key Words	#Web Pages
Experimental Group (G_1)					
p1	73m36s	60m20s	3	7	4
p2	72m47s	58m13s	3	8	6
p3	52m01s	08m00s	11	30	10
p4	68m25s	46m49s	9	24	15
p5	60m47s	35m25s	8	28	9
Min.	52m01s	08m00s	3.00	7.00	4.00
Max.	73m36s	60m20s	11.00	30.00	15.00
Average	65m31s	41m45s	6.80	19.40	8.80
Std.Dev.	08m08s	19m05s	3.25	9.91	3.76
Control Group (G_2)					
p6	67m08s	67m08s	9	9	10
p7	54m23s	54m23s	8	11	19
p8	69m40s	29m30s	11	27	14
p9	68m36s	53m29s	10	14	12
p10	59m45s	45m10s	14	17	12
Min.	54m23s	29m30s	8.00	9.00	10.00
Max.	69m40s	67m08s	14.00	27.00	19.00
Average	63m54s	49m50s	10.40	15.60	13.40
Std.Dev.	05m54s	12m34s	2.06	6.31	3.07

C. Results: Use of amAssist (Q2)

Figure 2 shows the ratio of the API entities in the developers' search queries divided by all the API entities in the foam tree at the time the developer issued the queries. On average, the four developers (P2, P3, P4, P5) used about 37%-50% of the API entities in the foam tree to formulate their search queries. It seems that the working context in the foam tree can effectively summarize the developers' information needs. The API entities in the foam tree seems useful for formulating context-aware search queries. The developer (P1) used on average about 18% of the API entities in the foam tree, which was less than the other four developers. This developer is an experienced Eclipse plugin developer. He seems to prefer to formulate the search queries with his own words.

D. Results: Behavior Change (Q3)

Figure 3 presents the time of the developers' first query and the time to opening the first web page. We can see

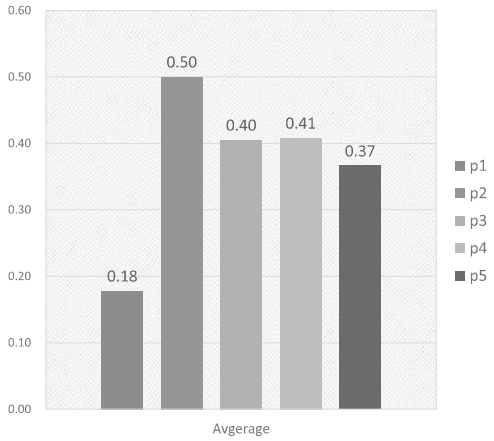


Fig. 2. The Ratio API Entities in query versus in foam tree

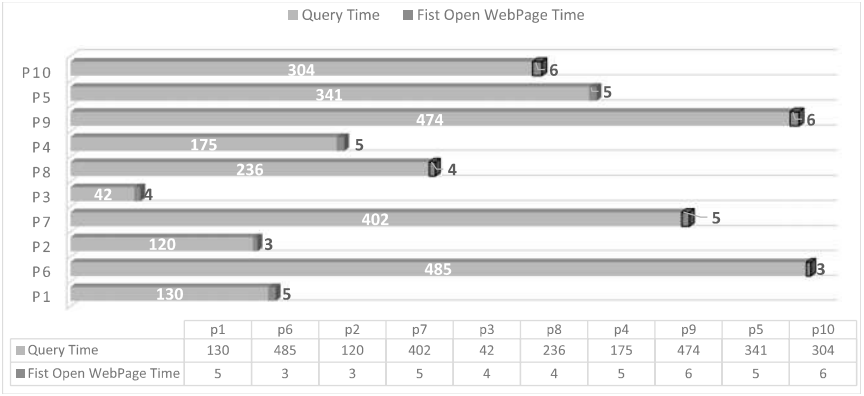
that four of the five experimental group developers (P1, P2, P3, P4) had much shorter time to first query than their counterpart developers in the control group (P6, P7, P8, P9). The experimental group developer P5 had slightly longer time to first query than his counterpart developer in the control group P10. All the developers in both experimental and control group opened the first web page shortly (in 4 - 6 seconds) after they obtained the search results. As a whole, the experimental group developer seems to be able to start their online search faster and find the potentially relevant web pages faster.

Figure 4 presents the time at which the developers integrated the online resources in their tasks for the first time. We identified such integration time by looking for actions such as copy-paste from the web browser to the IDE or the manual entering of web page content into the code editor. Three of the five experimental group developers (P1, P4, P5) had short time to start integrating the online resource than their counterpart developers (P2, P9, P10) in the control group. The two experimental group developers (P2, P3) had longer time to start integrating the online resource than their counterpart developers (P7, P8) in the control group. The results of first-time integration seems to be mixed. It seems that unlike search behavior, the integration behavior seems to be more affected by the programming habits of different developers.

V. RELATED WORK

Researchers have sought to monitor the users' interaction with software tools and documents to infer their interests or working context. Read and Write Wear [12] models the document places where users frequently read and write using wear metaphor. Similar concepts were used to visualize code changes [17]. The Jaba tool [18] elides code regions by means of Fisheye view which computes the DOI of code regions based on their distance to the selected region. The Mylar tool [19] monitors the Eclipse workbench selection and viewer services to infer the relevance of an element to a particular task. As we know, The Mylar project is now called Mylyn[4] , one of the top project in Eclipse. Now, It has powerful functions. It can recode programmer's behaviour and working context then outputting them with XML format file in silence way. Mylyn

[4]http://www.eclipse.org/mylyn/



	p1	p6	p2	p7	p3	p8	p4	p9	p5	p10
Query Time	130	485	120	402	42	236	175	474	341	304
Fist Open WebPage Time	5	3	3	5	4	4	5	6	5	6

Fig. 3. Time of First Query and Time of Opening the First Webpage

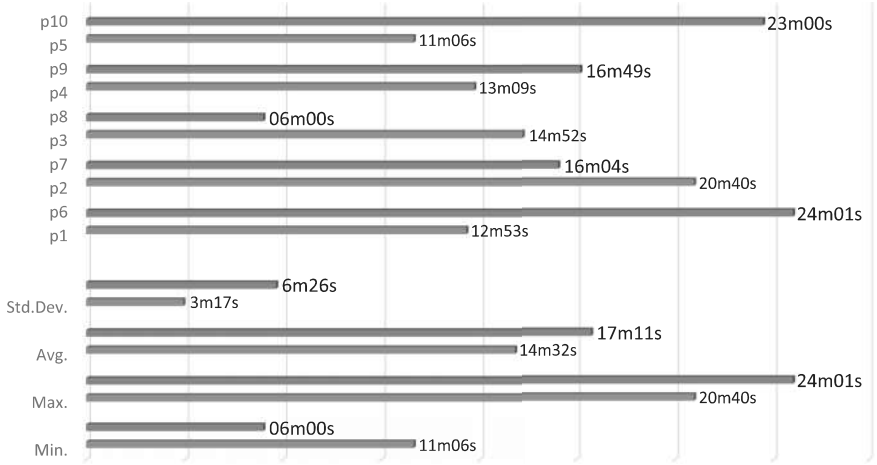


Fig. 4. Time of First Integration of Online Resources

can give the advices on the UI, It can help programmer focus on working contexts and code hits which are related current task. McKeogh et al. [20] implemented a plug-in, which use the monitor providing by SWT, to monitoring the programmer behaviour like exploring code or programming in the Eclipse Editor.

Robillard and Murphy [21] infers concern code from the source code visible to a developer over time. The *amAssist* DOI model was inspired by these related work. It tracks more comprehensive programming events in the developer's interaction with the IDE and program entities.

Researchers have used context in code search. CodeBroker [22] locates reusable components in a component repository based on task description and method signature. Strathcona [23] asks the developer to highlight a partially complete code fragment and retrieves code examples from existing framework applications based on structural context of the highlighted code fragment. Suade [24] recommends additional program elements based on their estimated structural relevance to a previously-identified set of program elements. MFIE [25] allows developers to interactively group, sort, and filter feature location based on automatically mined multiple syntactic and semantics facets. The tools consider context as a set of program entities at the current snapshot of the code. Our *amAssist* tool considers context as a time-series stream of programming events.

Researchers have used crowdsourced knowledge to assist software development. GraPacc [26] uses a snapshot of currently edited code to search and rank the relevant API usage patterns based on a database of API usage patterns mined from open source projects. HelpMeOut [5], [27] suggests program edits to fix compilation errors based on program edits from other programmers that fix the errors. These tools use only current code context and they do not search online resources.

Brandt et al. [28] investigated how programmers opportunistically interleave web foraging, learning, and writing code. The Seahawk tool [4] integrates Stack Overflow with the Eclipse IDE. It allows developers to query, view, and link Stack Overflow Q&As into their code. Blueprint [2] supports example-centric programming by code examples extracted from online forums. Dora [5] allows developers to query online discussions to locate relevant solutions to programming problems within the IDE. These tools still use only current code context. Furthermore, the code context was simply used to augment the search query. In contrast, our *amAssist* tool deeply integrate the working context in the entire search process from query formulation, custom search, to search results refinement and representation.

VI. Conclusions and Future Work

The paper presented the design of an in-IDE ambient search agent and the *amAssist* tool. The *amAssist* tool has two distinct characteristics. First, the *amAssist* tool unobtrusively monitors the developer's dynamic working context as a stream of time-series programming events observed from the developer's interaction with the IDE and the program. It uses interactive visualization to make the contextual information explicit to the developer what he has been working on over time. Second, the *amAssist* tool deeply integrate the developer's working context in the entire search process from query formulation, custom search, to search results refinement and representation.

Our initial evaluation suggests that the design of ambient search agent and its implementation in the *amAssist* tool is promising in better support the developer searching and using online programming resources while working in the IDE. By the deep integration of the developer's dynamic working context in the online search process, the developers seems to be able to formulate context-aware search queries with more specific API keywords and to locate and use relevant online programming resources faster to satisfy their information needs during software development.

In the future we will exploit Google Custom Search Engine features to better support custom search with context. We will also design and implement more intuitive ways to annotate the web page with relevant working context APIs. We will also conduct more comprehensive user study to evaluate the design of the ambient search agent and the benefits and limitations of deeper integration of the developers' working context in their online search.

Acknowledgements

This work is supported by National Natural Science Foundation of China under Grant No.61370079, National High Technology Development 863 Program of China under Grant No.2013AA01A605 and is partially supported by NTU SUG M4081029.020 and MOE AcRF Tier1 M4011165.020.

References

[1] J. Brandt, P. J. Guo, J. Lewenstein, and S. R. Klemmer, "Opportunistic programming: How rapid ideation and prototyping occur in practice," in *Proceedings of the 4th international workshop on End-user software engineering*. ACM, 2008, pp. 1–5.

[2] J. Brandt, M. Dontcheva, M. Weskamp, and S. R. Klemmer, "Example-centric programming: integrating web search into the development environment," in *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. ACM, 2010, pp. 513–522.

[3] B. Hartmann, S. Doorley, and S. R. Klemmer, "Hacking, mashing, gluing: Understanding opportunistic design," *Pervasive Computing, IEEE*, vol. 7, no. 3, pp. 46–54, 2008.

[4] A. Bacchelli, L. Ponzanelli, and M. Lanza, "Harnessing stack overflow for the ide," in *Recommendation Systems for Software Engineering (RSSE), 2012 Third International Workshop on*. IEEE, 2012, pp. 26–30.

[5] O. Kononenko, D. Dietrich, R. Sharma, and R. Holmes, "Automatically locating relevant programming help online," in *Visual Languages and Human-Centric Computing (VL/HCC), 2012 IEEE Symposium on*. IEEE, 2012, pp. 127–134.

[6] N. Sawadsky and G. C. Murphy, "Fishtail: from task context to source code examples," in *Proceedings of the 1st Workshop on Developing Tools as Plug-ins*. ACM, 2011, pp. 48–51.

[7] H. Li, Z. Xing, X. Peng, and W. Zhao, "What help do developers seek, when and how?" in *Reverse Engineering (WCRE), 2013 20th Working Conference on*. IEEE, 2013, pp. 142–151.

[8] L. Ponzanelli, G. Bavota, M. Di Penta, R. Oliveto, and M. Lanza, "Mining stackoverflow to turn the ide into a self-confident programming prompter," in *Proceedings of the 11th Working Conference on Mining Software Repositories*. ACM, 2014, pp. 102–111.

[9] J. Kohlhammer, T. May, and M. Hoffmann, "Visual analytics for the strategic decision making process," in *GeoSpatial Visual Analytics*. Springer, 2009, pp. 299–310.

[10] J. Matejka, T. Grossman, and G. Fitzmaurice, "Ambient help," in *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. ACM, 2011, pp. 2751–2760.

[11] G. W. Furnas, *Generalized fisheye views*. ACM, 1986, vol. 17, no. 4.

[12] W. C. Hill, J. D. Hollan, D. Wroblewski, and T. McCandless, "Edit wear and read wear," in *Proceedings of the SIGCHI conference on Human factors in computing systems*. ACM, 1992, pp. 3–9.

[13] B. Y. Kuo, T. Hentrich, B. M. Good, and M. D. Wilkinson, "Tag clouds for summarizing web search results," in *Proceedings of the 16th international conference on World Wide Web*. ACM, 2007, pp. 1203–1204.

[14] J. Stefanowski and D. Weiss, "Carrot2 and language properties in web search results clustering," in *Advances in Web Intelligence*. Springer, 2003, pp. 240–249.

[15] M. Hearst, *Search user interfaces*. Cambridge University Press, 2009.

[16] S. Osiński, J. Stefanowski, and D. Weiss, "Lingo: Search results clustering algorithm based on singular value decomposition," in *Intelligent information processing and web mining*. Springer, 2004, pp. 359–368.

[17] S. G. Eick, J. L. Steffen, and E. E. Sumner Jr, "Seesoft-a tool for visualizing line oriented software statistics," *Software Engineering, IEEE Transactions on*, vol. 18, no. 11, pp. 957–968, 1992.

[18] A. Cockburn and M. Smith, "Hidden messages: evaluating the efficiency of code elision in program navigation," *Interacting with Computers*, vol. 15, no. 3, pp. 387–407, 2003.

[19] M. Kersten and G. C. Murphy, "Mylar: a degree-of-interest model for ides," in *Proceedings of the 4th international conference on Aspect-oriented software development*. ACM, 2005, pp. 159–168.

[20] J. McKeogh and C. Exton, "Eclipse plug-in to monitor the programmer behaviour," in *Proceedings of the 2004 OOPSLA workshop on eclipse technology eXchange*. ACM, 2004, pp. 93–97.

[21] M. P. Robillard and G. C. Murphy, "Automatically inferring concern code from program investigation activities," in *Automated Software Engineering, 2003. Proceedings. 18th IEEE International Conference on*. IEEE, 2003, pp. 225–234.

[22] Y. Ye and G. Fischer, "Reuse-conducive development environments," *Automated Software Engineering*, vol. 12, no. 2, pp. 199–235, 2005.

[23] R. Holmes, R. J. Walker, and G. C. Murphy, "Strathcona example recommendation tool," *ACM SIGSOFT Software Engineering Notes*, vol. 30, no. 5, pp. 237–240, 2005.

[24] F. W. Warr and M. P. Robillard, "Suade: Topology-based searches for software investigation," in *Proceedings of the 29th international conference on Software Engineering*. IEEE Computer Society, 2007, pp. 780–783.

[25] J. Wang, X. Peng, Z. Xing, and W. Zhao, "Improving feature location practice with multi-faceted interactive exploration," in *Proceedings of the 2013 International Conference on Software Engineering*. IEEE Press, 2013, pp. 762–771.

[26] A. T. Nguyen, H. A. Nguyen, T. T. Nguyen, and T. N. Nguyen, "Grapacc: a graph-based pattern-oriented, context-sensitive code completion tool," in *Proceedings of the 2012 International Conference on Software Engineering*. IEEE Press, 2012, pp. 1407–1410.

[27] B. Hartmann, D. MacDougall, J. Brandt, and S. R. Klemmer, "What would other programmers do: suggesting solutions to error messages," in *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. ACM, 2010, pp. 1019–1028.

[28] J. Brandt, P. J. Guo, J. Lewenstein, M. Dontcheva, and S. R. Klemmer, "Two studies of opportunistic programming: interleaving web foraging, learning, and writing code," in *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. ACM, 2009, pp. 1589–1598.