Resumption Strategies for Interrupted Programming Tasks

Chris Parnin
College of Computing
Georgia Institute of Technology
Atlanta, GA U.S.A.
chris.parnin@gatech.edu

Spencer Rugaber
College of Computing
Georgia Institute of Technology
Atlanta, GA U.S.A.
spencer@cc.gatech.edu

Abstract

Interruptions are a daily reality for professional programmers. Unfortunately, the strategies programmers use to recover lost knowledge and resume work have not yet been well studied. In this paper, we perform exploratory analysis on 10,000 recorded programming sessions of 85 programmers to understand the variety of strategies used by programmers for resuming programming tasks. In our study, we find that only 10% of the programming sessions have coding activity start in less than a minute, only 7% of the programming sessions involve no navigation to other locations prior to editing, and find evidence of programmers seeking other sources of task context during task resumption. Based on the analysis, we suggest how task resumption might be better supported in future development tools.

1. Introduction

Everyday, professional developers engage in a process of recovering knowledge about software. When resuming an incomplete programming task, the developer must remember their previous working state and recover knowledge about the software. Details of working state might include recalling plans, intentions, and goals. Details of knowledge might include plan progress, component mechanisms, and domain representations.

Several researchers have characterized the effects of interruptions on performing tasks [6, 24, 12, 21]. Despite efforts for managing interruptions, *in situ* studies suggest interruptions remain problematic. Czerwinski's study [7] showed that tasks resumed after an interruption were more difficult to perform and took twice as long. O'Conaill's study [23] found 40% of interrupted tasks are not resumed at all. Further research by Mark *et al.* [18] observed 57% of tasks were interrupted – as a result work on a task often was fragmented into many small work sessions.

Studies examining software companies have also replicated similar results from studies of non-programming tasks in the workplace. Solingen [36] characterizes interruptions at several industrial software companies and observed that an hour a day was spent managing interruptions, and developers typically required 15 minutes to recover from an interruption. Ko [15] used a *fly-on-the-wall* [17] approach to observe software developers at Microsoft and found that they were commonly blocked from completing tasks because of failure to acquire essential information from busy co-workers. Another study at Microsoft indicated that 62% of developers surveyed believed recovering from interruptions was a serious problem [16].

Although many studies have been performed on interruptions, few describe what strategies programmers use for resuming a programming task. Research on program comprehension has typically focused on the perspective of exploring a new program or making a new change; however, with an interrupted task, programmers are instead re-comprehending a program and related artifacts in order to resume work.

This paper makes the following contributions:

1. An analysis of the beginning of programming sessions to understand the various activities that take place before resuming work.

2. A characterization of several possible strategies for resuming programming tasks.

The benefit of this paper is that other researchers and toolsmiths can apply these results when investigating and designing how to enhance tool support for resuming programming tasks.

In the next section of the paper, we describe related work for programming tool support. This is followed by a discussion of our datasets and relevant concepts. We then describe a set of resumption strategies and the experiments used to support the feasibility and plausibility of the strategies. The strategies are than compared and discussed in the context of

current tool support. Finally, we conclude with future work for this research.

2. Background

Studies have examined the idea of maintaining a representation of task context [14] to help reduce mental workload during development. Because the risk of interruptions includes the loss of details and increased time to recover working state, some have hypothesized that using a representation of context in the IDE would reduce resumption costs [25, 14, 11, 10]. Although there is some evidence supporting the reduction of navigation cost [25, 14, 29], there is little evidence describing what effects arise from using task context when resuming programming tasks. Further, the scope of task context has been primarily limited to the files and methods related to the task. However, an increasing amount of research raises the question of what other knowledge should be included in a context.

Miyata and Norman [20] describe task execution as having three phases: planning, execution, and evaluation. Interruptions during different phases have different effects [38] that require different sources of knowledge to support resumption. The current approach for representing a context typically relies on filtering the hierarchical list of files and methods in the IDE based on recent interaction or items explicitly specified to belong to the current task. The benefit of this representation would be in its ability to reduce interference on recall of non-relevant code, especially in the case of returning to a task that had be shelved for a long period of time. However, in the case of short term interruptions, this technique's representation mainly assists with continuing execution of the last task, but neither assists the user in planning the next steps or in evaluating the progress made. Because task execution is commonly hierarchical [1], this representation does not include information from other pending tasks. Finally, the representation of context would provide little benefit when compared with a user manually controlling the file manager by collapsing nodes; in either case, only the last editor window is in focus, and the user must still explore other code elements to recall detail.

There is some evidence that developers use different sources of information for maintaining context. Software developers often need to record *prospective tasks* that they are unable to complete at the moment but need to complete in the future. Failure to remember prospective tasks is a common occurrence with interrupted tasks [7]. Researchers have observed how programmers use task annotations [32] to record prospective task and proposed systems for capturing prospective tasks [8, 32].

Other researchers have explored the recall from *episodic memory* (contextual details including time and activity related to a past event). Safer [28] performed an investigation comparing the use of alternative user interfaces for recalling information about recent tasks. In the study, the user's task was to identify when a recent task was completed or deferred for another task. One interface displayed screenshots capturing different stages of performing the task to prompt episodic recall of the event. The other used a Mylyn context [14] indicating the files and methods visited or edited. An overall subjective preference was found for the screenshots, although the results overall remain inconclusive.

Finally, recent research has identified a strong connection between environmental cues and reducing resumption costs [2, 13, 34]. In these experiments, the availability of cues during task resumption reduces the time to restart the task. Conversely, if the cue is removed or tampered with, then this benefit is removed. This even holds for implicit cues such as the location of a mouse cursor. For example, in one experiment when the mouse cursor location was moved from the last button clicked to be in the corner of screen, the resulting resumption lag was higher than when the implicit cue was present.

Because most development environments limit the view to one active element but may hold many active elements in their working state, developers are limited to the details of one active element when automatically encoding [35] working state (making mental notes).

The types of knowledge and representations needed to restore working context remains an open research problem. Most proposed solutions have been limited to prescriptive approaches that have not been evaluated to determine their impact in assisting developers with recovering from interruptions. In our work, we are investigating if developers would like to see other sources of information for restoring working state.

3. Concepts and Datasets

In this section, we describe the time line of recovering from an interruption and related concepts. Secondly, we describe the data sets we use for analysis. The content of the data sets include the *interaction history* captured from developers programming in their natural settings. Interaction history is the record of low-level events (including navigation and edit events) from a programmer using an IDE.

3.1. Resumption Timeline

A key measure of the effect of an interruption is *resumption lag* [1, 3, 4]. Resumption lag measures the time it takes for a person to recollect their thoughts when returning to a task. In experimental studies, resumption lag is typically measured as the time between a subject being told to resume a task and the first physical response (such as mouse click).

Similarly, interruption lag [3, 4] describes the time between when the user stop working on a task and when they begin addressing the interruption (*e.g.* writing down a note before picking up the telephone). Taking prospective measures at the time of interruption has both negative and positive effects [35] on task resumption. Prospective measures are primarily successful when users have environmental cues about working state (*e.g.* an error message associated with a bug being fixed) that are present during both the interruption lag and the resumption lag. When comparing written notes versus mental notes, written notes were found to be effective in assisting recall, but negatively effected the ability to recall contextual details about the task when compared to mental notes. Another positive factor occurs when users have completed a subtask because there is less intermediate state that has to be recalled [1].

In Figure 1, we show the time line of a programmer editing code, stopping work, and then resuming work. Each period of programming activity describes one *session* of work. The depiction of the session is adopted from the visualization used in the SpyWare tool [27]. Basically, the graph displays the edits per minute (EPM) in a development session. The vertical axis corresponds to the number of edits, and a point on the horizontal axis corresponds to one minute.

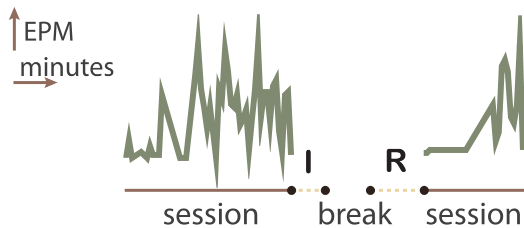

Figure 1. Interruption lag (I) and Resumption lag (R). During the brief period of interruption lag, the programmer may actively encode their mental working state to prevent an increased latency when resuming a task (resumption lag).

3.2. Edit Lag

Although previous work has demonstrated strong correlations between interruptions and increased resumption lag, the measured resumption lag has been in the order of seconds. In the domain of software development, the effects of interruption are believed to have a larger impact on loss of context and the recovery period to be on the order of minutes [36]. Therefore, we propose a specialized measure of resumption lag, called *edit lag*, which is the time between returning to a programming task and making the first edit.

In this study, we focus most of our analysis on edit lag to gain insight into what activities developers perform before regaining enough context to resume editing for that session. Undoubtedly, developers perform numerous activities other than coding during software development; however, studying the moment coding begins in a session serves as a logical starting point for asking why developers performed a series of activities before making an edit and how much of that is related to resumption costs.

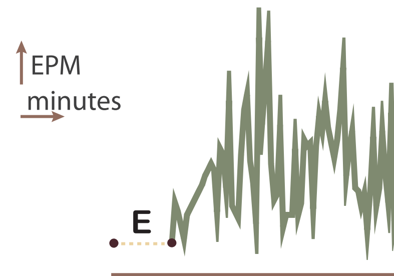

Figure 2. Edit lag (E). When starting a new programming session, a latency can be observed in the time it takes to begin coding and the time it takes to reach the peak of coding activity during the session.

3.3. Interruptions

The nature of interruption is an important determinate on the extent of an interruption's impact. The worst type of interruption often comes at an inopportune moment and gives insufficient time to a programmer for preserving mental working state. These inopportune moments tend to align with programmers using large amounts of intermediate knowledge that does not yet have any physical representation or have a firm mental foothold in the programmer's mind. However, not all interruptions are involuntary. Other reasons for suspending a programming task include fatigue, desire for reflection, road blocks, and reaching a stopping point. These types of interruptions are called *self-interruptions*. The nature and timing of an interruption will influence the type of suspension strategy used and the amount of lost knowledge at risk. In this paper, we do not know the nature of an interruption, but instead focus on observing the activities of resuming a programming task after a break in programming activity.

3.4. Interaction History

We draw upon a variety of data sets in our analysis.

The first data set we will call the *Eclipse* data set; it was originally collected in the latter half of 2005 by Murphy

and colleagues [22]. The researchers used the Mylyn Monitor tool to capture and analyze fine-grained usage data from volunteer programmers using the Eclipse development environment [1].

The second set we will call the *Visual Studio* data set; it was also collected in 2005 by Parnin and Görg [25]. The data was collected from 12 developers over several months at an industrial site.

The third set of data we will call the *UDC* dataset; it is publicly-available from the Eclipse Usage Data Collector [33], and includes data requested from every user of the Eclipse Ganymede release. Activity is recorded from over 10,000 Java developers between April and December 2008. The data counts how many programmers have used each Eclipse command, such as refactoring commands, and how many times each command was executed.

To obtain the developer's sessions, the events were segmented when there was a break in activity of 15 minutes or more. This segmentation is well supported by the nature of the data. The interval between events follows a Poisson distribution: for 98% of the 4.5 million events, the time between those events is less than a minute. This means tight clusters can be formed with any threshold above a minute. Similar thresholds have been supported in other studies [39, 27].

The events collected are solely from the IDE. Although, what appears to us is a break in programming activity, the developer may be engaging in a related task such as checking in source code or searching for online code examples. If we included window focus events [26], then we could better explain these breaks. Nevertheless, we believe many breaks from programming activity are due to diversions [5] that often derail programming efforts as observed by Ko [15].

There are some instrumentation differences between the data sets. In the Eclipse data set, an edit event was registered whenever a programmer typed (*i.e.* roughly an event per word); however, in the Visual Studio data set, an edit event was collected for each line edited. Another differences was that in the Visual Studio data set, edit lag was more finely observable because navigation events within a code editor were recorded which was not true in the Eclipse data set. This would mainly effect edit lags that did not require navigation to another file.

In our analysis, we accounted for these differences by running the experiments separately to check for any discrepancies between the data sets. A summary of our data sets can be see in Table 1.

Dataset	**Stats**				
	Users	Sessions	Filtered*	Edits+	Events
Visual Studio	12	1972	1561	1213	573,998
Eclipse	73	7927	5931	3962	3,937,526
Total	85	9,899	7492	5175	4,511,524

Table 1. Summary of data in *Visual Studio* and *Eclipse* data sets. The column Filtered* indicates the remaining number of sessions after removing sessions with a duration less than one minute and the column Edits+ indicates the number of filtered sessions with at least one edit event.

4. Resumption Strategies

When a programmer is interrupted from a programming task, what activities do they need to perform in order to resume work? In this section, we shed some light on this question – but a more pressing question should be answered first: Is there even any meaningful impact from interruption? Are programmers immediately able to return to effective work without a resumption delay?

To understand the possible impact of interruptions on developers, we measure the distribution of edit lag. If we observe most values of edit lag are less than one minute, then we would likely conclude the impact of interruptions are negligible or not immediately observable in this window of time. Second, we characterize the number and size of sessions in the day to see how often a programmer must resume work in a day.

In Figure 3, we display the distribution of edit lag among all sessions having editing activity. In 10% of the sessions, edit lag was less than a minute. For the rest of the sessions, several minutes pass until the first edit event occurs. In about 30% of sessions, the edit lag is over 30 minutes. In these sessions, we believe the developers may be engaged in debugging activities which require a longer investment of attention before a first edit can be made.

Finally, in Table 2, we show a breakdown of the frequency and duration of sessions in a typical day.

Sessions in a typical day						
sessions	1-3	1-2	1-2	1	0-1	rarely
duration	15m	30m	1h	2h	4h	8h

Table 2. In a typical day, developers program in several short sessions with an additional one or two longer sessions.

Overall, this evidence indicates that significant interruptions do occur and that significant time is required for effective resumption.

[1] http://eclipse.org

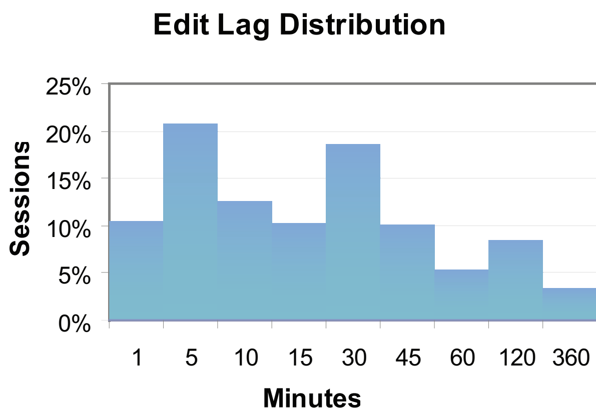

Figure 3. Developers often do not make their first edit of the session until at least several minutes have past.

4.1. Return to Last Method Edited

One of the most simple tactics for resuming a programming task is to return to the site where work was last performed. But how sufficient and how frequently is this tactic applicable for resuming work?

To measure how often programmers successfully perform this tactic, we make several measurements. First, we measure how often the first change is made without navigating to other methods or classes. Second, we measure how often the first change is eventually made in the last edited method even with intervening navigations to other methods or classes. Finally, we measure the edit lag for both cases. Only the *Visual Studio* data set had sufficient information about navigation for this analysis.

	Resumption cost			
Sessions	35%	22%	23%	12%
Edit Lag	1m	1-5m	5-15m	15-30m

Table 3. Developers are able to resume coding within 1 minute for 35% of sessions and within 30 minutes for most sessions when the work involves completing the last edited method.

The results show that in 91 of 1213 sessions (7.5%) the programmers were able to make changes without navigating to other methods. Second, in 209 out of 1213 sessions (17%) the programmers do eventually return from navigating to other locations and make changes to the last method edited. Otherwise, in the majority of sessions, the programmers navigated to other methods or classes to resume work.

The results of the edit lag can be seen in Table 3. When the programmers do continue work in the same method, they can often resume work quickly. When resuming coding in the last method, the programmer does this within one minute 35% of the time. We believe in these situations, the programmer can successfully use the method to remember sufficient details for completing the task. In other cases, the programmer may be spending time re-understanding the code or re-evaluating their implementation plan.

One explanation for the results is that programmers may prefer to stop working at a natural task boundary and therefore not need to complete work in that method upon resuming the task. The last completed task may serve as a convenient cue for remembering which next logical task to perform. Further investigation is needed to understand why programmers may spend more than a few minutes resuming work even when the work is in the same location. An experiment comparing the complexity of the source code with edit lag would confirm the hypothesis that developers need time to re-comprehend complex code when completing a task involving that code.

4.2. Navigate to Remember

In the previous subsection, we observed that in 118 of the 209 (56%) sessions (where the programmer made changes to the last method edited) the programmer *still* navigated to other locations first. Navigating to other locations is a natural tactic to use if the programmer needs to recall details from other parts of the code before making a change. But how many places do programmers need to visit, and what is the cost of performing all these navigations before a change can be finally be started?

Because of differences in the two data sets, we separate the analyses and use two different metrics. With the *Visual Studio* data set, to measure the number of places visited, we record the set of all methods or classes visited before making the change. We then measure the navigational cost as determined by the distance between a code element. For the *Eclipse* data set, we measure the number of selection events prior to the edit.

	Visual Studio	Eclipse
Locations	2-12 (7)	
Navigation Distance	4-40 (27)	
Selection Events		15-150 (135)

Table 4. Developers typically visit several locations of code before beginning a change. The range of elements covering 75% of values is shown with the mean in parenthesis.

Resumption cost					
Sessions	16%	25%	22%	18%	8%
Edit Lag	1m	1-5m	5-15m	15-30m	30-45m

Table 5. Developers take a little longer to start coding when the work involves navigating to other locations first.

The results can be seen in Table 4. In general, the developers navigated within the code to several locations before being able to begin coding. The developers also took considerably longer to start coding when navigation was involved. As shown in Table 5, sessions with navigations to other locations (in the Visual Studio data set) have higher edit lag when compared to results in the previous section (Table 3).

4.3. Task Tracking

Programmers actively manage and perform many tasks when developing software. Some tasks may be specific such as fixing a compile error, or as complex as implementing a complete system module. When resuming an incomplete programming task, developers may need to recall details of the task and other related subtasks. In this subsection, we are measuring the activity related to viewing details about tasks stored in a task repository. In addition, we also examine how developers use task tracking software to understand how willingly developers are for tracking subtasks or small tasks with task software.

To measure the use of task information, we examined commands used in the *Eclipse* data set related to accessing a bug repository such as Bugzilla, the Mylyn Task List, and the Eclipse Task List. We also separately measure the use of possible implicit task reminders such as the warnings or compile errors listed in the Problem View. Compile errors may serve as a good source of reminders, but we do not know if these are being intentionally left unresolved as a cue or could not be fixed during the previous session.

To measure the use of task tracking for management of subtasks, we examined the *UDC* data set and *Eclipse* data set. We count the number of users that use commands related to creating new subtasks and compare that to the number of users using other common task management commands. This gives an estimation of the developer population likely to use task software for small tasks.

The results for viewing task details are shown in Table 6. Developers viewed task information in 9% of sessions, and the Problem View was used in 9% of sessions. However, for both cases the associated edit lag was also very high. For 75% of the sessions that viewed either the Problem View or task information, the edit lag was greater than 30 minutes.

	Pre-Edit Lag	Post-Edit Lag	% Sessions
tasklist	274	246	9%
problem view	301	265	9%

Table 6. Viewing the task list or problem view during a session is common.

Commands	Users	
	10/2008	11/2008
View Task List	10,311	11,206
Open Task	861	953
New Local Task	101	101
New Sub Task	11	22

Table 7. Tasking software is popular for reviewing assigned tasks but not for recording low-level tasks.

This suggests these sessions might have been spent planning or refining tasks or resolving time-consuming configuration or compile issues.

In Table 7, the results of the task subtask recording usage is shown. A more detailed breakdown of the four commands in Mylyn is the following: users

- listed the tasks in an external task repository,
- opened a task from the task repository,
- created a new task that was not be stored in the task repository and,
- created a subtask of an existing task.

The table also shows the number of users that have used the command in the two most recent months of activity. Mylyn is a popular tool for viewing assigned tasks, but unfortunately the number of users using Mylyn for managing personal tasks (or willing to use the tool to enter new tasks or sub tasks) is several magnitudes less than managing assigned tasks.

4.4. Review Source Code History

Source code repositories hold a vast amount of information about the history of a project. Some questions might have to be answered by accessing the revision history before resuming a task.

To measure the frequency of using this strategy, we count the number of sessions that have cvs or subversion history commands during the edit lag. To obtain the activity related to history, we separate the actions concerned with retrieving information from the managerial repository commands. We were primarily concerned with comparing revisions, viewing the history of a revision, viewing the commit log, and viewing a revision annotation. We also measure the occur-

	Pre-Edit Lag	Post-Edit Lag	% Sessions
history	142	183	4%
commit	193	390	11%

Table 8. Viewing history during the edit lag is as common as during the rest of the programming session.

rence of commands in the rest of the session to gain insight into the significance of the command occurring during the edit lag. Finally, we are only able to measure the *Eclipse* data set for revision history usage.

The results can be see in Table 8. From this measurement, we only observe occasional use of revision history during the edit lag (only 4% of sessions). However, relative to commit commands, the history commands are used fairly frequently, and are as likely to be used during the edit lag as during normal coding activity.

The measures we have collected may be conservative. Not all users always use the Eclipse plugins for accessing revision history. Future studies should expand the instrumentation collection to directly measure access to source control.

5. Discussion

Several analyses were described in Section 4. In Table 9, a summary is given of the frequency of various activities developers performed when starting a programming session. How do the presented results affect future research and tools?

Strategies	Usage
Continue Last Edit	7.5%
Nav Then Continue Last Edit	17%
Navigate to New Location	83%
View Revision History	4%
View Problem List	9%
View Task or Bug List	9%

Table 9. Developers perform various activities at the beginning of a session.

In the following, we review our findings and provide hypotheses that might explain the observed behavior and implications for future tool research. Finally, researchers should be able to experimentally apply these latency measures when studying their proposed tools.

5.1. Edit and Navigation Behavior

Considerable doubt was placed on the sufficiency of using the last edit as an environmental cue. When the cue was applicable, it was very effective: 35% of these sessions were resumed in less than a minute, which is much higher than the 10% of sessions overall that could be resumed in less than a minute. However, this situation was only applicable 17% of the sessions and still required navigation to other locations. In other cases, developers had considerable edit lag even when continuing edits at the same location. It is likely that in this situation, the code or task was complex. This suggests there are opportunities for researchers to investigate how additional code representations or visualizations might eliminate this latency.

If programmers use the last site of completed work as a launching point, then there are some interesting implications. Several researchers have reported differences in recall ability in completed tasks versus incomplete tasks [37, 19]. It is unknown how much contextual detail programmers lose after task completion. There is some evidence that gives direction. In Bailey's study [1], the task-evoked pupillary response was recorded over time (the pupil size dilates during moments of increased memory access and mental load) while users performed tasks. In the study, immediately after subtask completion, pupillary response would return to baseline levels. This suggests that memory access ceases after task completion and may effect the retention of task-related information that was in short-term memory.

Finally, developers are spending a considerable amount of time navigating to several locations in code before beginning coding. Ideally, a developer should be able to resume working without having to spend several minutes of every session re-orienting themselves. Future research should continue to evaluate alternative navigation interfaces such as CodeThumbnails [9], or Relo [30], or SHriMP [31] affect on resumption costs.

5.2. Other Sources of Task Context

We also reviewed two possible sources of information that may be relevant in resuming a programming task. First, when developers refer to task or compile errors at the beginning of a session, the time to begin coding was much longer than in any other sessions. The extra latency needs to be further investigated. Secondly, a recording formal representation of task structure including all relevant subtasks could be an important method of reducing cost of task switching and managing interruptions. Many research tools rely on the assumption of developers participate in detailing tasking information. However, very few users actively use available facilities for recording subtasks or small tasks. The issues that developers have with these tools must be dis-

covered and addressed before such an approach becomes feasible. Other interfaces for recording information may alleviate this problem: Developers may be receptive to more simple input that allows whiteboard-like task sketching.

Developers actively refer to revision history information but are slower to resume coding when they do. Research needs to elaborate on the motivations and requirements for viewing revision history when resuming a task. One possible explanation is that developers need to review the status of other teammates activities to see if it impacts their task. This would be consistent with other research: Status awareness was one of the top information needs found in Ko's study [15] of developers. Developers may also be using facilities such as a code difference to remind themselves of what changes were made during their last activity. Surprisingly, this process is still manual and may be an explanation for the low observance of use. Future enhancement to IDEs may include the ability to automatically summarize and highlight code differences between programming sessions. This may provide additional cues for programmers to trigger reminders on any incomplete tasks or unresolved bugs.

5.3. Toward Supporting Task Resumption

Having seen the variety of activities developers perform at the beginning of a programming session, we attempt to identify how better tool support can be constructed for suspending and resuming programming tasks. Developers are devoting a significant amount of time toward building knowledge and planning how to perform a programming task during the start of a session. When resuming a programming task, developers have a spectrum of resumption strategies available. The selection of these strategies will largely depend on the manner in which the developer was interrupted and the current state of the programming task. However, the variety of suspended task states and types of knowledge programmers need to recall will pose many challenges to developing tool support.

With involuntary interruptions, developers have limited time to properly suspend the programming task and as a result will most likely select strategies that will prompt recall by using implicit cues. Unfortunately, the source of implicit cues are limited in current IDE systems. Because there are no explicit representations of plans, goals, or intermediate knowledge used during a task, developers rely on ad-hoc strategies to trigger and activate those memories. A likely reason why these strategies fail is caused by insufficient cues contained at the last site of work. There are several measures that future tools can make to improving this scenario:

- **Automatic Tags.** A tag cloud of links to recent source code symbols and names inside method bodies.
- **Instant Diff** Highlighted code showing how a developer changed a method body as well as a global view.
- **Snapshots and Instant Replay.** Time line of screenshot thumbnails or an instant replay of past work.
- **Change Summary.** Short summary of changes.

Continuing from the last state of the programming task can happen in a variety of ways: the developer may resume coding in the previous location, may need to transition to the next step of the task, or may need to evaluate the progress of the task. The complex nature of programming requires developers to concentrate on a fixed set of artifacts in the context of a task. Developers linearize tasks so that upon completing a subtask, they are aware of what the next subtask will be. In effect, developers maintain a rolling window of focus. Unfortunately, an interruption can severely derail this process due to the memory strategies used in this process. The diverging needs of different task states suggest that there are opportunities for devising notations for expressing the steps, objectives, and stages of a programming task. An IDE can formally support task stages by including perspectives for different task stages or introduce lightweight mechanisms for annotating programming artifacts. A list of some measures that can support task state include:

- **Task Sketches.** Light-weight annotations of a task breakdown: steps, objectives, and plans.
- **Runtime Information.** Values or visualizations of variables or expressions from previous execution or debugging session(s).
- **Prospective Cues.** Contextual reminders that are displayed when a condition is true.

5.4. Threats to Validity

One main threat to validity for this analysis is that the period of edit lag does not necessarily distinguish the time related to resuming a task from that of thinking about a new task. In a controlled study, a researcher would be able to control for whether a programmer was resuming an incomplete task or starting a new task. In this study, we can use the structure of activity to infer properties about task resumption, but ultimately the effects may be confounded. In the worst case, the experimental values found in this study serve as an upper bound. Future studies needs a stronger method of separating this effect by accounting for which task(s) comprise a session. Possible methods include relating sessions to source code check-ins and task repository activity.

Similarly, for the activities detected; such as task tracking usage, there is no direct causal relationship between starting a session and observance of an activity. Again, the experimental results found establish an upper bound on the activity corresponding with resuming a programming task

and set the basis for future confirmatory studies.

Finally, the data does not capture a complete representation of all possible activities that were performed. Developers may be using other sources of reminder cues such as notes on their desk or comments in source code. Other types of studies would be needed to be performed to measure the frequency and effectiveness of these techniques.

5.5. Future Work

We are investigating several directions for expanding this research. To continue our work, we are conducting a formal survey of developers to further understand the kinds of resumption strategies that are commonly used. Second, we will perform an ethnographic evaluation of developers resuming programming tasks to capture-information seeking requirements and common barriers faced. Finally, we will perform several experiments to better answer some of the questions raised in this study. What information are developers seeking (or what did they forget) when resuming a programming task? What are effective cues for resuming tasks: For example, would highlighting code differences of changes made during the last session better prompt developers' memories?

The analysis of session data can be expanded in several ways. An interesting aspect of the session data to investigate is the time after the last edit in the session (interruption lag). The interruption lag may include activities related to preparing the workspace to facilitate resumption. Another possible analysis includes examining the relationships between groups of sessions. For example, in reviewing visualizations of developer sessions, we observed a pattern of "dabbling sessions" early in the day where developers would navigate for a few minutes but make no changes. Presumingly, after "warming up" for an hour or two, the developer have a long productive session. Another trend we observed that was very productive days containing many long sessions were followed by several days of low productivity (a few small sessions). There may be the effects of mental fatigue and workload regulating the nature of sessions.

6. Conclusions

We have presented an analysis of three sets of data that provides new insight into how programmers resume a programming task — particularly characterizing what activities are performed at the beginning of a resumed session.

Some interesting results have emerged from these data. In only a small percentage of sessions did developers resume coding in less than a minute. Developers consistently spend a significant portion of their time doing non-editing activities before making their first edit in a session. During this time period, developers are performing a variety of activities that relate to rebuilding their task context.

However, there is still much work to be done. The psychology and environmental factors related to interruptions remains a complex topic. Fortunately, understanding this topic has broad impact on the everyday activities of developers. Future research should investigate *why* developers must visit several locations of code before coding and consider ways to rethink how IDE organizes content beyond tabbed editing and hierarchical lists.

References

[1] P. D. Adamczyk and B. P. Bailey. If not now, when?: the effects of interruption at different moments within task execution. In *CHI '04: Proceedings of the SIGCHI conference on Human factors in computing systems*, pages 271–278, New York, NY, USA, 2004. ACM.

[2] E. M. Altmann and J. G. Trafton. Memory for goals: an activation-based model. *Cognitive Science*, 26:39–83, 2002.

[3] E. M. Altmann and J. G. Trafton. Task interruption: Resumption lag and the role of cues. In *Proceedings of the 26th annual conference of the Cognitive Science Society*, 2004.

[4] E. M. Altmann and J. G. Trafton. Timecourse of recovery from task interruption: Data and a model. *Psychonomic Bulletin & Review*, 14:1079–1084, 2007.

[5] L. Bannon, A. Cypher, S. Greenspan, and M. L. Monty. Evaluation and analysis of users' activity organization. In *CHI '83: Proceedings of the SIGCHI conference on Human Factors in Computing Systems*, pages 54–57, New York, NY, USA, 1983. ACM.

[6] E. Cutrell, M. Czerwinski, and E. Horvitz. Notification, disruption and memory: Effects of messaging interruptions on memory and performance. 2001.

[7] M. Czerwinski, E. Horvitz, and S. Wilhite. A diary study of task switching and interruptions. In *CHI '04: Proceedings of the SIGCHI conference on Human factors in computing systems*, pages 175–182, New York, NY, USA, 2004. ACM Press.

[8] U. Dekel. Designing a prosthetic memory to support software developers. In *ICSE Companion '08: Companion of the 30th international conference on Software engineering*, pages 1011–1014, New York, NY, USA, 2008. ACM.

[9] R. DeLine, M. Czerwinski, B. Meyers, G. Venolia, S. Drucker, and G. Robertson. Code thumbnails: Using spatial memory to navigate source code. In *VLHCC '06: Proceedings of the Visual Languages and Human-Centric Computing*, pages 11–18, Washington, DC, USA, 2006. IEEE Computer Society.

[10] R. DeLine, M. Czerwinski, and G. Robertson. Easing program comprehension by sharing navigation data. In *VLHCC '05: Proceedings of the 2005 IEEE Symposium on Visual Languages and Human-Centric Computing*, pages 241–248, Washington, DC, USA, 2005. IEEE Computer Society.

[11] R. DeLine, A. Khella, M. Czerwinski, and G. Robertson. Towards understanding programs through wear-based filtering. In *SoftVis '05: Proceedings of the 2005 ACM sympo-*

sium on Software visualization, pages 183–192, New York, NY, USA, 2005. ACM.

[12] T. Gillie and D. Broadbent. What makes interruptions disruptive? a study of length, similiarity, and complexity. *Psychological Research*, 50:243–250, 1998.

[13] H. M. Hodgetts and D. M. Jones. Contextual cues aid recovery from interrruption: The role of associative activitation. *Journal of Experimental Psychology: Learning, Memory, and Cognition*, 32(5):1120–1132, 2006.

[14] M. Kersten and G. C. Murphy. Using task context to improve programmer productivity. In *SIGSOFT '06/FSE-14: Proceedings of the 14th ACM SIGSOFT international symposium on Foundations of software engineering*, pages 1–11, New York, NY, USA, 2006. ACM.

[15] A. J. Ko, R. DeLine, and G. Venolia. Information needs in collocated software development teams. In *ICSE '07: Proceedings of the 29th international conference on Software Engineering*, pages 344–353, Washington, DC, USA, 2007. IEEE Computer Society.

[16] T. D. Latoza, G. Venolia, and R. Deline. Maintaining mental models: a study of developer work habits. In *ICSE '06: Proceeding of the 28th international conference on Software engineering*, pages 492–501, New York, NY, USA, 2006. ACM Press.

[17] T. C. Lethbridge, S. E. Sim, and J. Singer. Studying software engineers: Data collection techniques for software field studies. *Empirical Software Engineering*, 10(3):311–341, July 2005.

[18] G. Mark, V. M. Gonzalez, and J. Harris. No task left behind? Examining the nature of fragmented work. In *CHI '05: Proceedings of the SIGCHI conference on Human factors in computing systems*, pages 321–330, New York, NY, USA, 2005. ACM Press.

[19] F. McKinney. Studies in the retention of interrupted learning activities. *Journal of Comparative Psychology*, 19 (2):265–296, 1935.

[20] Y. Miyata and D. A. Norman. Psychological issues in support of multiple activities. In D. A. Norman and S. W. Draper, editors, *User Centered System Design: New Perspectives on Human-Computer Interaction*, pages 265–284. Erlbaum, Hillsdale, NJ, 1986.

[21] C. A. Monk. The efffect of frequent versus infrequent interruptions on primary task resumption. In *Proceedings of the Human Factors and Ergonomics Society 48th Annual Meeting*, 2004.

[22] G. C. Murphy, M. Kersten, and L. Findlater. How are Java software developers using the Eclipse IDE? volume 23, pages 76–83, Los Alamitos, CA, USA, 2006. IEEE Comp. Soc. Press.

[23] B. O'Conaill and D. Frohlich. Timespace in the workplace: dealing with interruptions. In *CHI '95: Conference companion on Human factors in computing systems*, pages 262–263, New York, NY, USA, 1995. ACM Press.

[24] M. Offner. *Mental Fatigue*. Warwick & York, 1911.

[25] C. Parnin and C. Görg. Building usage contexts during program comprehension. In *ICPC '06: Proceedings of the 14th IEEE International Conference on Program Comprehension*, pages 13–22, 2006.

[26] K. Renaud and P. Gray. Making sense of low-level usage data to understand user activities. In *In Proceedings of SAICSIT '04*, pages 115–124, , Republic of South Africa, 2004. South African Institute for Computer Scientists and Information Technologists.

[27] R. Robbes and M. Lanza. Characterizing and understanding development sessions. In *ICPC '07: Proceedings of the 15th IEEE International Conference on Program Comprehension*, pages 155–166, Washington, DC, USA, 2007. IEEE Computer Society.

[28] I. Safer and G. C. Murphy. Comparing episodic and semantic interfaces for task boundary identification. In *CASCON '07: Proceedings of the 2007 conference of the center for advanced studies on Collaborative research*, pages 229–243, New York, NY, USA, 2007. ACM.

[29] J. Singer, R. Elves, and M.-A. Storey. Navtracks: Supporting navigation in software maintenance. In *ICSM '05: Proceedings of the 21st IEEE International Conference on Software Maintenance*, pages 325–334, Washington, DC, USA, 2005. IEEE Computer Society.

[30] V. Sinha, D. Karger, and R. Miller. Relo: Helping users manage context during interactive exploratory visualization of large codebases. In *VLHCC '06: Proceedings of the Visual Languages and Human-Centric Computing*, pages 187–194, Washington, DC, USA, 2006. IEEE Computer Society.

[31] M.-A. Storey, C. Best, J. Michaud, D. Rayside, M. Litoiu, and M. Musen. Shrimp views: an interactive environment for information visualization and navigation. In *CHI '02: CHI '02 extended abstracts on Human factors in computing systems*, pages 520–521, New York, NY, USA, 2002. ACM.

[32] M.-A. Storey, J. Ryall, R. I. Bull, D. Myers, and J. Singer. Todo or to bug: exploring how task annotations play a role in the work practices of software developers. In *ICSE '08: Proceedings of the 30th international conference on Software engineering*, pages 251–260, New York, NY, USA, 2008. ACM.

[33] The Eclipse Foundation. Usage Data Collector Results. January 5th, 2009. Website, http://www.eclipse.org/org/usagedata/reports/data/commands.csv.

[34] J. G. Trafton, E. M. Altmann, and D. P. Brock. Huh, what was i doing? how people use environmental cues after an interruption. In *Proceedings of the Human Factors and Ergonomics Society 49th Annual Meeting*, 2005.

[35] J. G. Trafton, E. M. Altmann, D. P. Brock, and F. E. Mintz. Preparing to resume an interrupted task: effects of prospective goal encoding and retrospective rehearsal. *International Journal of Human-Computer Studies*, 58:583–603, 2003.

[36] R. van Solingen, E. Berghout, and F. van Latum. Interrupts: Just a minute never is. *IEEE Software*, 15(5):97–103, 1998.

[37] B. Zeigarnik. Das behalten erledigter und unerledigter handlungen. *Psychologische Forschung*, 9 (1):1–85, 1927.

[38] F. R. H. Zijlstra, R. A. Roe, A. B. Leonova, and I. Krediet. Temporal factors in mental work: Effects of interrupted activities. *Journal of Occupational and Organizational Psychology*, 72:163–185, 1999.

[39] L. Zou and M. W. Godfrey. An industrial case study of program artifacts viewed during maintenance tasks. In *WCRE '06: Proceedings of the 13th Working Conference on Reverse Engineering*, pages 71–82, Washington, DC, USA, 2006. IEEE Computer Society.