Software Qual J (2011) 19:5–34
DOI 10.1007/s11219-010-9104-9

ICPC 2009

Resumption strategies for interrupted programming tasks

Chris Parnin · Spencer Rugaber

Published online: 10 August 2010
© Springer Science+Business Media, LLC 2010

Abstract Interrupted and blocked tasks are a daily reality for professional programmers. Unfortunately, the strategies programmers use to recover lost knowledge and rebuild context when resuming work have not yet been well studied. In this paper, we describe an exploratory analysis performed on 10,000 recorded sessions of 86 programmers and a survey of 414 programmers to understand the various strategies and coping mechanisms developers use to manage interrupted programming tasks. Based on the analysis, we propose a framework for understanding these strategies and suggest how task resumption might be better supported in future development tools. The results suggest that task resumption is a frequent and persistent problem for developers. For example, we find that only 10% of the sessions have programming activity resume in less than 1 min after an interruption, only 7% of the programming sessions involve no navigation to other locations prior to editing. We also found that programmers use multiple coping mechanisms to recover task context when resuming work.

Keywords Interruption · Resumption strategies · Task context · Task knowledge

1 Introduction

Professional developers must frequently resume an unfinished programming task that has been interrupted. The interruption may be due to an unexpected request from a co-worker, a scheduled meeting, or a task blockage resulting from an unanswered question to a colleague on vacation or from an unavailable server needed for development. Regardless of the source, the effects are often the same: when resuming interrupted work, developers

C. Parnin (✉) · S. Rugaber
Georgia Institute of Technology, Atlanta, GA, USA
e-mail: chris.parnin@gatech.edu
URL: http://cc.gatech.edu/~vector

S. Rugaber
e-mail: spencer@cc.gatech.edu
URL: http://cc.gatech.edu/~spencer

experience increased time to perform the task, increased errors, increased loss of knowledge, and increased forgetting to perform critical tasks (Parnin and DeLine 2010).

In order to resume an incomplete programming task, developers must recall their previous working state and refresh their knowledge of the software and task. Details of working state include plans, intentions, and goals; details of knowledge include relevant artifacts, component mechanisms, and domain representations. Unfortunately, the nature of the knowledge and program artifacts does not make this process any easier. First, much knowledge used for the programming task is of a tacit and ephemeral nature-it is not explicitly written down anywhere, and it involves many temporary details that have not yet deeply taken root in the mind. Therefore, an interruption can be detrimental to retaining this type of knowledge. Second, a software program, unlike a novel or television series, does not have a simple linear structure-a recap or rescanning the last few paragraphs of text is not enough to allow the developer to resume work. Instead, the relevant information exists across various artifacts and locations and must be actively sought out and restored in the mind.

Developers are certainly not alone in facing problems with interruptions. Several psychological studies of simple tasks and observational studies of knowledge workers have characterized the effects of interruptions on people. In psychology studies of interruptions, researchers have measured the effects on performance (Cutrell et al. 2001) and mental fatigue (Offner 1911) as well as understanding the contribution of interruption frequency (Monk 2004) and interruption length, task similarity, and task complexity (Monk et al. 2008; Gillie and Broadbent 1998) on performance. Despite efforts for managing interruptions, in situ studies suggest interruptions remain problematic for knowledge workers. Czerwinski's study (Czerwinski et al. 2004) showed that tasks resumed after an interruption were more difficult to perform and took twice as long. O'Conaill's study (O'Conaill and Frohlich 1995) found 40% of interrupted tasks are not resumed at all. Further research by Mark et al. (2005) observed 57% of tasks were interrupted and as a result work on a task often was fragmented into many small work sessions.

Studies examining software companies have replicated results from earlier workplace studies. van Solingen et al. (1998) characterizes interruptions at several industrial software companies and observed that an hour a day was spent managing interruptions, and developers typically required 15 min to recover from an interruption. Ko et al. (2007) used a *fly-on-the-wall* (Lethbridge et al. 2005) 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 (Latoza et al. 2006).

Although many studies have described the effects and circumstances of interruptions, few describe the strategies programmers use to cope with interruptions. Research on program comprehension has typically focused on 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 describes our study of the interruption problem and makes the following contributions:

1. A contextual inquiry and analysis of programming sessions to understand the various activities that take place before resuming work.
2. A characterization of several strategies for resuming programming tasks.

In the next section, we present some background material and relevant concepts for understanding interruptions. We then describe a schema for task knowledge and set of accompanying resumption and suspension strategies that can be used to understand how programmers manage interrupted programming tasks. This is followed by a discussion of datasets we use for evaluating the plausibility of our described strategies. Next, we describe the results from our analysis of the datasets. The strategies are then compared and discussed in the context of current tool support. We conclude with future directions for this research.

2 Background

In this section, we describe concepts related to interruptions, the timeline of recovering from an interruption, the nature of task knowledge and human memory, and the role that environmental cues and basic strategies play in managing interruptions.

2.1 Interruptions and multitasking

The nature of an interruption is an important determinate on the extent of the interruption's impact. Disruptive interruptions often comes at an inopportune moment and gives insufficient time for a programmer to preserve mental working state. These inopportune moments tend to align with programmers using large amounts of intermediate knowledge that does not yet have any tangible externalization representation nor have a firm 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 influences the type of suspension strategy used and the amount of knowledge at risk for loss. In this paper, we do not make any assumptions about the nature of an interruption, but instead focus on observing the activities of resuming a programming task after a break in programming activity.

Often, the term *multitasking* is confounded with the term *interruption*. To better situate these terms, Salvucci et al. (2009) describe a continuum of multitasking behaviors that can be used to understand the impact of interrupting a task. The continuum spans from concurrent multitasking (driving in car and talking) to sequential multitasking (writing a paper and reading email). In concurrent multitasking, task switching costs are high if both tasks require access to different problem states (Borst and Taatgen 2007). In sequential multitasking, task switching costs are especially high if working state has not been adequately preserved and suspended, which is typical with interruptions.

2.2 Resumption timeline

A key measure of the effect of an interruption on resumption is called the *resumption lag* (Adamczyk and Bailey 2004; Altmann and Trafton 2004, 2007). 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 made, such as mouse click. Similarly, *interruption lag* (Altmann and Trafton 2004, 2007) describes the time between when the user stop working on a task and when they begin addressing the interruption (e.g., writing a note before picking up the telephone).

Figure 1 shows the time line of a programmer editing code, stopping, and then resuming work. Each period of programming activity describes one *session* of work. The depiction of

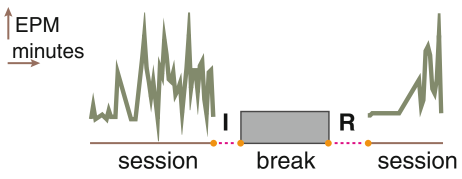

Fig. 1 Interruption lag (*I*) and Resumption lag (*R*). During the brief period of interruption lag, the programmer may actively encode their mental working state to prevent an increased latency when resuming a task (resumption lag)

the session is adopted from the visualization used in the SpyWare tool (Robbes and Lanza 2007). The graph displays the edits per minute (EPM) during a development session. The vertical axis corresponds to the number of edits, and a point on the horizontal axis corresponds to 1 min.

During or before the interruption lag, there are several strategies for negotiating the end of the interruption lag, often referred to as the *breakpoint*. McFarlane has evaluated and compared the different strategies for setting a breakpoint: immediately addressed interruption, scheduled (defined time limit), negotiated (user decides), mediated (third party decides if busy user looks interruptible) (McFarlane 2002).

Miyata and Norman (1986) suggest that people seek breakpoints at moments where the current task requires low mental workload. Indeed, studies of interrupted tasks such as document editing or route planning have demonstrated that breakpoints at moments of higher mental workload cause longer resumption lag than breakpoints at lower mental workload (Adamczyk and Bailey 2004; Iqbal and Bailey 2005). As observed in their studies, a less disruptive interruption arose when the breakpoint could be delayed by a few seconds.

Forgarty et al. (2005) performed an experiment to predict the interruptibility of programmers and the time needed to reach a breakpoint. In the experiment, programmers were interrupted every 3 min with a multiplication problem. The programmer had the choice to delay addressing the interruption until a good breakpoint was reached. For most interruptions, the programmers would address the interruption within 10 s; however, when more deeply engaged, they would defer for a mean of 43 s.

2.3 Task knowledge

In the course of a programming task, developers accumulate transient knowledge needed to perform the task. For example, during corrective maintenance of software, a developer may generate several hypotheses about how the program behaves (Mayrhauser and Vans 1997). *Task knowledge*, such as an explanatory hypothesis and the related program elements, is a temporary representation that assists in maintaining and manipulating relevant knowledge about a program and problem state. Loss of this knowledge can be costly, requiring the programmer to redo time-consuming and cognitively difficult work.

Kersten and Murphy (2006) have proposed a model that captures some aspects of task knowledge. *Task context* refers to the relevant artifacts of a program; that is, to a subset of program elements. Task context is a useful concept for building tools-Mylyn,[1] a plugin available with Eclipse, allows users to filter code documents based on a selected task and recent activity. The benefit of this representation is its ability to reduce the interference of

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

non-relevant code on recall, especially in the case of returning to a task that had be shelved for a long period of time. However, Mylyn's representation mainly assists with recovering the necessary artifacts for the task, but does not aid the user in planning the next steps or in evaluating the progress made.

Missing from task context are cognitive elements associated with the task such as goals or mental representations. Miyata and Norman (1986) describe performing a task in three phases: planning, execution, and evaluation. These phases are commonly hierarchical (Adamczyk and Bailey 2004). Interruptions during different phases have different effects (Zijlstra et al. 1999) that require different kinds of knowledge to support resumption. Researchers have tried to postulate what types of cognitive elements exist for programmers. A theme that emerges from different empirical studies is that programmers formulate two types of goals: those that relate to doing work and those that reflect on the status of their work. Anderson et al. (1984) describe *inherent* and *planned goals*. Allwood (1984) describes *progressive* and *evaluative goals*; O'Brien and Buckley (2005) refer to *progressive* and *evaluative* efforts in their work on information forging as a comprehension strategy. Pennington and Grabowski (1990) refers to these as *composition* and *evaluation goals*.

2.4 Memory

A programmer must be able to manage mental representations of both her program and problem state. In supporting this work, memory is an essential mechanism for maintaining and manipulating these mental representations. The human mind contains a variety of functional memory types that a programmer may take advantage of. For example, long-term memories help programmers remember facts about a program, whereas spatial memory (Ratwani and Trafton 2008) help programmers orient their attention among many artifacts and windows in a programming environment.

One of the earliest contributions to memory began in 1956 with Miller's work on limitations on information processing. Regardless of what item a participant was being asked to memorize, Miller observed that the capacity appeared to be five to nine items (Miller 1994). Recent research has suggested the actual limit is closer to four items (Cowan 2001). In 1968, Atkinson and Shiffrin presented an influential model of memory called the modal model of memory. In the modal model, information is first stored in sensory memory. Attentional processes select items from sensory memory and hold them in short-term storage. With rehearsal, the items can be moved into long-term storage. The model characterizes the process of obtaining long-term memory as a serial and intentional process with many opportunities to lose memory along the way via decay or interference from newly formed memories. Attempting to refine the modal model's account of short-term memory, in 1974, Baddeley and Hitch introduced the model of working memory. The concept of working memory was introduced to help explain how items could be manipulated and processed in separate modalities (e.g. visual versus verbal).

Attempting to understand how experts could exceed the limits of short-term and working memory, Chase and Simon proposed that experts such as chess players learn how to effectively *chunk* information after extensive practice and study (Chase and Simon 1973). Alternatively, Chase, Ericsson, and Staszewski (Chase and Ericsson 1982, Ericsson and Staszewski 1989) later proposed the skilled memory theory from observations of mnemonists and experts in a variety of domains. The skilled memory theory identifies two key strategies experts use to achieve their remarkable memory and problem-solving ability: (a) information is encoded with numerous and elaborated cues related to prior knowledge similar to Tulving's encoding specificity principle (Tulving and Thomson 1973); (b) experts develop a retrieval structure for indexing information in long-term memory (for example, experts might associate locations within a room with material to memorize by mentally visiting locations within the room, the expert could retrieve associated items from those locations).

Recently, the skilled memory theory has been extended into the long-term working memory theory, which claims many of the problems with previous theories can be explained if working memory actually involves immediate storage and activation of long-term memories (Ericsson and Kintsch 1995). This view is consistent with neurological views of memory. Morris and Frey (1997) postulate that the hippocampus provides the ability for an "automatic recording of attended experience". They argue that many important events cannot be anticipated nor may they occur again, and therefore traces and features of experiences must be recorded in real time as they happen and provide evidence from animal studies. Further, Morris (2006) makes the argument based on numerous neuroanatomical studies that the hippocampus does not store sensory stimuli directly, but rather associates indices into other cortical regions.

Researchers distinguish between sensory, short-term, working memory, and long-term memory. For long-term memory, Squire (2004) proposed a taxonomy that divides types of long-term memory hierarchically starting with a distinction between non-declarative (implicit) and declarative (explicit) memories. Non-declarative memory includes priming and muscle memory whereas declarative memory includes knowledge of facts and events. Tulving (1972) describes semantic memory as knowledge of facts and episodic memory as a recollection of past events. Other types of memory have been identified including *familiarity*, the 'feeling of knowing" that an object in a particular context has been encountered before without necessarily recalling the situation the object has been encountered (Gilboa et al. 2006); *recency*, recalling how long ago a recent event occurred (Milner et al. 1991); and *prospective memory*, remembering to remember to perform an action in the future under a specific context (e.g., setting up a mental reminder to buy milk on the way home from work) (Winograd 1988).

Interruptions place additional cognitive demands, introduce conflicting and interfering goals, and decay and delay processing of human memory; however, understanding how interruption affects memory is still an open research question. Abstracting from the underlying memory mechanics, Altmann and Trafton (2002) have proposed a general cognitive framework for modeling the effects of interruptions on tasks and associated problem state. *Memory for goals* is an activation-based approach for memory that builds on the ACT-R cognitive framework (Anderson et al. 2004). In this theory, goals and the associated problem state in working memory compete for attention based on their level of activation such that, at any given time, the most active goal directs behavior. As time passes, the activation levels of memory decay over time. When interrupted, the current primary task goal is suspended and the activation level of this goal decays. Upon resumption, the time required to begin work on the primary task reflects the process of retrieving the suspended goal. The higher the activation level of the suspended goal, the more easily that goal can be retrieved. There are several constraints that determine the activation level of the suspended goal. First, the history of the goal (i.e., frequency and recency of goal retrieval) impacts goal activation. Second, the availability of environmental context and cues may prime activation of the goal.

2.5 Environmental cues

The memory for goals framework predicts that environmental cues would reduce the time to retrieve a suspended goal. Recent research has demonstrated strong connection between the availability of environmental cues and reduced resumption costs (Altmann and Trafton 2002; Hodgetts and Jones 2006; Trafton et al. 2005). Conversely, if the cue is removed or tampered with, this benefit is removed. This holds even for implicit cues, such as the location of a mouse cursor. For example, in one experiment when the mouse cursor location was moved from where it was when it was last clicked to be in the corner of screen, the resulting resumption lag was higher than when the implicit cue left undisturbed.

Observations of developers suggest that they frequently rely on cues for maintaining context during programming. For example, Ko et al. (2005) observed programmers using open document tabs and scrollbars as aids for maintaining context during their programming task. However, environmental cues often do not provide sufficient context to trigger necessary memories: In studies of developer navigation histories, a common finding is that developers frequently visit many locations in rapid succession in a phenomenon known as *navigation jitter* (Singer et al. 2005). Navigation jitter takes the form of developers flipping through open tabs and file lists when trying to recall a location (Singer et al. 2005; Parnin and Görg 2006). Environmental cues such as open tabs may be insufficient because *what* a developer remembers may be spatial and textual cues within the code document (what they work directly with) and not the semantic or structural location of the code element when automatically encoding working state (Trafton et al. 2003).

An initial experiment by Safer and Murphy (2007) supports the insufficiency of even semantic-based cues. In the experiment, the authors investigated two cue designs for recalling information about recent programming tasks. The participant performed four programming tasks, with each task lasting about 15 min. After all the tasks were completed, the user was asked to identify the specific time at which each task was started and finished. The participant was aided by one of two cues for identifying the task boundaries. One cue stressed an *episodic* presentation of the activity history: a timeline of screenshots displayed a frame-by frame playback of the entire programming session. The other cue stressed a *semantic* presentation of activity: a timeline of tree views containing the programming artifacts (files and methods) that were visited or edited for an event. Using the episodic-based cue, users were able to identify task boundaries quicker and with less errors than with the semantic-based cue. However, there were several unresolved issues. The semantic-based cue was not as efficacious as the episodic cue-it did not include a way to navigate and view the source code, which is a standard interface for a tree view in a programming context, whereas the episodic cue inherently conveys the source code. This performance difference, nevertheless, suggests that file and method names alone may be insufficient for recalling certain details about recently performed work.

2.6 Interruption management strategies

Researchers studying interruptions have observed basic strategies people use for managing interruptions. Besides the strategies concerning timing of the task's breakpoint (McFarlane 2002) and priming based on environmental cues (Altmann and Trafton 2004), people also use prospective measures of rehearsal or various forms of externalization (note-taking).

Trafton et al. (2003) have observed people using prospective rehearsal of their tasks. Taking prospective measures at the time of interruption has both negative and positive effects (Trafton et al. 2003). 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 affected the ability to recall contextual details about the task. The evidence suggests controlled memory processes may override or interfere with automatic encoding.

Researchers have investigated what artifacts information workers used when managing tasks. Bellotti et al. (2004) studied the various media employed by nine different managers and found that, in addition to using paper media such as sticky notes, 50% of the tasks could be linked to emails. Brush et al. (2007) observed eight workers who needed to frequently file status reports of their weekly activity. Five out of the eight workers attempted to continually track information in a centralized location: e.g., notepad or ongoing email draft-but all participants also complemented their tracking strategy with memory triggers to retrieve details needed for composing the report. There is also evidence that developers use different sources of information for maintaining context. Software developers often need to remember *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 (Czerwinski et al. 2004). Other researchers have observed how programmers use task annotations (Storey et al. 2008) to record prospective task and proposed systems for capturing prospective tasks (Dekel 2008; Storey et al. 2008).

3 Resumption strategies

In this section, we describe various strategies for suspending and resuming an interrupted programming task. The general course of handling an interruption is described in three phases: *suspension*, *resolution*, and *resumption*. During the suspension phase, the programmer uses one of the basic strategies of rehearsal, serialization, and cue priming to save working state. Next, the programmer resolves the problem causing the interruption. Finally, the programmer returns to the programming task and reacquires the task's goals, plans, context, and representations with the assistance of the saved state. In the later sections, we compare these strategies to practice.

3.1 Working state

In this paper, we consider working state to be comprised of these four elements: artifacts (task context of programming elements), plans (actions to perform), goals (objectives to fulfill), representations (mental representations of task context and specialized domain knowledge needed in task (e.g., parsing, html flow layout).

3.2 Suspension strategies

During the interruption lag, a programmer has the chance to preserve working state for faster resumption. Depending on the interruption type (unanticipated or planned), the programmer may have more available time for this process. Although preservation of work environment (e.g., saving files, backups, background processes) is an important aspect, we do not discuss this further in the paper. We discuss suspension strategies in the context of an individual; however, these strategies can be relevant in other contexts such as a task hand-off to another team member.

A programmer may suspend a task through *internalization* and *externalization* of working state. With internalization, the programmer uses memory tactics to remember the working state, whereas with externalization, the programmer uses physical or electronic media to preserve the working state. Often these processes complement each other. For example, if a programmer externalizes working state through note-taking, then this process will also bolster internalization of the working state. On the other hand, the programmer may use an external cue when internalizing working state.

There are three types of suspension strategies programmers can use: *rehearsal*, *serialization*, and *cue priming*. With rehearsal, programmers retain working state by increasing the activation level of working memory. This strategy is mainly used for retaining high-level information over short to medium interruption durations. For short-term interruptions, the boost in activation level of the memory should sufficiently allow the working state to be recalled. However, this mechanism only enables a small amount of information to be rehearsed. During the interruption lag, the programmer must evaluate the progress of the task and determine what piece of task knowledge would be most important to retain. Often, the programmer will focus on the remaining actions that need to be performed and will rely on the availability of environmental cues to activate more details (Parnin and DeLine 2010).

Serialization, the externalization of working state is a powerful way of retaining task knowledge; however, it can be costly. The conflict that arises with serialization is due to the difficulty in recording knowledge (e.g., even jotting down a couple of class and method names can be a slow process). In general, there is a trade off programmers can make in serializing task knowledge between *visibility* and *context*. Programmers can reduce the effort representing task knowledge by situating the transient representation in the context where is it needed (e.g., placing the comment "Fix this" within the source code as a reminder). On the other hand, that same comment placed on a sticky note "Fix this" is more likely to be visible but offers little context in recalling detail. The risk the programmer runs in placing a situated note is failure to encounter and recall the note when intended, whereas the cost of writing an unsituated note is that the further away from the source, the more context has to be specified for it to be understandable. A note that reproduces context outside of the original source is a *reconstructive* note. Often this takes the form of a sketch or diagram (Cherubini et al. 2007). As a result of this serialization cost, programmers will often focus on recording task objectives and limited amounts of task context or plans of actions.

Cue priming is a strategy that can be frequently and quickly applied. Cue priming involves preparing or finding environmental cues to trigger recall upon resuming the task. A developer may leave open the last window she was editing, highlighting several lines of code, or leave open an error message that occurred while testing the program. This suspension strategy can be very useful because it only takes a few seconds to set up and often resumption can be very quick upon returning to work. However, this strategy is largely opportunistic and potentially fragile-the developer might not have such a convenient mnemonic available if she is in the middle of program exploration or effecting a large change. Further, transient cues may be lost or disturbed by new interactions with the environment or in the event of the IDE shutting down. Another problem with cue priming as a suspension strategy is that programming environments are not designed to easily leave behind cues from multiple locations within the program. As with a situated note, it may be easy to miss potentially useful cues. To compensate, a programmer can actively leave behind *roadblock cues*, such as compile errors, which will force the programmer to encounter the cues when resuming work.

Finally, there are some prospective measures programmers can take that are not specifically related to suspending working state during the interruption lag. For example, a programmer can clean up their task context by closing out windows that will not be needed upon returning to the task. This pruning process will assist the programmer in reducing interference with dangling pieces of context. Second, a programmer can use *task* *scaffolding* akin to externalization during performance of the task by first marking up the code with high-level comments describing intended changes. The comments can serve as a representation of the task's change plan and assist the programmer in resuming the task.

3.3 Resumption

During the resolution phase of an interruption, more time might have passed than the programmer expected, especially in the case of blocked tasks or changes in task priorities (Ko et al. 2007; van Solingen et al. 1998). Sometimes an interruption can consume the only available time within a day (the rest of the day is filled with meetings), causing the programmer to push back work to another day.

Rather than being exact duals of suspension strategies, resumption strategies often are more mixed and demand-driven due to the incomplete nature of suspension and unexpected degradation of memory from unexpected delay or failure. What makes the resumption strategies distinct is *what* needs to be restored—this is often reflected by the task's breakpoint (e.g., task is still in the planning stage or deep within the task hierarchy).

3.3.1 Global restoration

When returning to a programming task, a programmer might have only a vague idea of his last activity. In this scenario, a programmer might use global restoration strategies that elicit clues for recovering working state: *return to last stopping place* and *navigate to remember*. The most obvious and direct strategy a programmer might try is to return to the last scene where work took place. If the task is localized, then sufficient information should be available that the programmer can read the surrounding code, regain task knowledge, and then resume work.

If the programming environment is still available in its previous state, than numerous memory cues can be used. Memory cues are strong devices because they can trigger a variety of memory types and hasten the time of recovery through spreading activation of other memories. Cues are particularly important for prospective (reminders to perform an action) and episodic (temporal, spatial and visual memories of recent events) memories. Therefore, even seemingly random navigation has several tactical advantages. First, it increases the chance that programmers will encounter other memory cues and more memory cues will lead to more targeted navigation. Second, the process of navigation will prime the programmer and help him or her to maintain spatial awareness for the coming task.

3.3.2 Goal restoration

Sometimes if a task is not well specified or is resumed near its completion where there might be several clean-up work remaining, then the programmer might need to reacquire intermediate goals that were formulated previously. The strategies for restoring goal state can be top-down or bottom-up. *Review Task Assignment* is a top-down strategy where the programmer uses the original task description to refresh the goals and objectives of the task. With this strategy, the programmer uses the top-level goal to remember details about their last working activities and evaluate which steps need to be performed next. The programmer may use this reacquired goal as a filter for restoring other parts of the task context. *Execute Program* is a bottom-up strategy where the programmer uses the output or behavior of the program to evaluate the success of the task and determine any outstanding issues. Running the program has the advantage of priming the domain representations and quickly triggering goals that are not explicitly represented. However, the nature of some programs or the non-compiling state of a program may make this strategy not always applicable. A final strategy is *Restore from Task Breakdown*. This strategy relies on the programmer to maintain a written list or hierarchical breakdowns of the task into other smaller tasks. This strategy can be very successful in restoring goals, but its applicability depends on the level of detail a programmer will track explicitly.

3.3.3 Plan restoration

Whereas goals provide objectives for evaluating the completeness of the task, plans describe what sequences of actions to apply to a program. If the exact plan of action is not clearly recalled, then the programmer will use several strategies for recalling what actions to perform. A common strategy for restoring plan knowledge is *Review Source Code Change History*. There are several advantages to comparing differences in source code versions. Differences reveal a detailed account of progress in complementing a task. From the differences, the programmer has a baseline for evaluating the progress of the task. Second, the programmer can potentially identify deficiencies in the solution, such as he forgot to apply a global update to a certain location. Lastly, the differences can trigger memories of the associated plan for the task. *Search for Prospective notes* is a complementary strategy when programmers suspend prospective tasks. Searching for serialized task annotations, prospective notes such as TODO comments, or intentional compile errors provide reminders to perform certain actions. The collection of these reminders can help reformulate the plan for the task. For example, if the program has several compile errors, as the programmer addresses the errors, he recalls that the errors were caused by copying and pasting the code from another location, without renaming relevant variables. This further recalls why this change was made and helps restore the plan knowledge.

3.3.4 Context restoration

Often restoring task context can follow acquiring goal and plan knowledge. Even though a programmer can see if a certain code element was visited via an open document tab, without recalling the association with a particular goal or plan the programmer has not fully integrated this information with her task knowledge. However, if a programmer recalls a goal or plan, then this new knowledge provides enough clues to perform a search to identify or directly recall the relevant program elements. Besides using goals or plans to direct navigation, task context can be restored as a byproduct of efforts used to restore goals and plans. For example, a source revision difference will reveal the locations that belong in the task context. Finally, task context itself can bootstrap recall by triggering associated memories of goals and plans which in turn recall other relevant components of task context.

3.3.5 Representation restoration

Restoring temporary representations used in the task can be difficult if not actively serialized in a reconstructive note or if the representation is not familiar or common. Besides note-taking and learning representations, there are currently no available strategies that programmers can use to improve this process.

3.4 Summary

When a programmer suspends and later resumes a programming task, a variety of factors will determine which strategies the programmer uses. Some suspension strategies are more suited to quick and local suspension; others are better at capturing detailed task knowledge. Depending on the nature of the suspended task and the preparation taken to preserve task knowledge, the programmer can select the most appropriate combination of resumption strategies to restore missing task knowledge. We present a summary of the suspension strategies (see Table 1) and resumption strategies (see Table 2) the developer may use. In the next sections, we seek to measure, analyze, and understand how these strategies are used in practice and gain insight into where these strategies fall short and need support.

4 Datasets and measures

In this section, we describe the data sets and related measures 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.

Table 1 Suspension strategies vary in transcription speed, completeness, and durability of suspended task knowledge

Suspension strategy	Description	Resource
Rehearsal	Internalize thoughts and representations through subvocalization	
Serialization	Write down internal thoughts and representations	Situated, unsituated, reconstructive notes
Cue priming	Seek or enhance cues for aiding resumption.	Roadblock, env. cues
Context pruning	Discard irrelevant items	
Task scaffolding	Create placeholders in source code to outline task	Comments, method stubs

Table 2 Different resumption strategies restore different task knowledge

Resumption strategy	Description
Return to last stopping place	Review recent context to restore goals or plans
Navigate to remember	Restore context and goals through navigation
Review task assignment	Restore high-level goals
Execute program	Evaluate state of program to obtain goals
Restore from task breakdown	Restore from a low-level goal
Review source code change history	Evaluate past actions to remember next actions
Search for prospective notes	Seek reminders for next actions

4.1 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 et al. (2006). The researchers used the Mylyn Monitor tool to capture and analyze fine-grained usage data from volunteer programmers using the Eclipse development environment.[2]

The *Visual Studio data set* was collected in 2005 by Parnin and Görg (2006) at an industrial site. The previously published data included ten developers for 30 work days (1.5 months). Since the original publication, the data we report in this paper also includes two more developers and a few more months of data.

The *UDC dataset* is publicly available from the Eclipse Usage Data Collector (2009) 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 navigations or 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 min 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 (Zou and Godfrey 2006; Robbes and Lanza 2007).

Events collection is performed solely by 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 (Renaud and Gray 2004), then we could better explain these breaks. Nevertheless, we believe many breaks from programming activity are due to diversions (Bannon et al. 1983) that often derail programming efforts, as observed by Ko et al. (2007).

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, transition events were more finely observable because navigation events within a code editor were recorded, which was not true in the Eclipse data set.

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 seen in Table 3.

4.2 Survey

We conducted a survey of 371 programmers at Microsoft and 43 programmers from a variety of companies including small startups as well as defense, financial, and game companies. We initially conducted the external study and encouraged by the feedback, distributed the survey with slight modifications to employees at Microsoft. Overall, we find

[2] http://eclipse.org

Table 3 Summary of data in *UDC*, *Visual Studio*, and *Eclipse* data sets

Dataset	Stats				
	Users	Sessions	Filtered*	Edits+	Events
UDC	10,000+				
Visual Studio	12	1972	1561	1213	573,998
Eclipse	74	7927	5931	3962	3,937,526
Total	86	9,899	7492	5175	4,511,524

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

general agreement with the external survey and internal survey, but only report the internal results for this paper.

As we were primarily interested in professional software development, we selected our population from full-time software developers—excluding interns or people in roles such as tester or project manager. We also excluded developers that had been solicited for other surveys by our research group to avoid oversampling. Respondents were compensated by a chance to win a $200 gift certificate. From our participant pool, we randomly sampled 2,000 developers and invited them to participate through email.

Participants answered fifteen fixed- and open-response questions about interruptions. The participants were asked to describe in detail the cost of interruptions, the steps they took to prepare for an interruption, the resumption strategies they used, factors that made resumption difficult, and their thoughts on future tools. We drew the selection of future tools from current research proposals for improvements to programming environments. We wanted to determine if programmers were primarily interested in tools to help them manage task state or in tools that augment cues within the programming environment.

In the following section, we will compare results of our analysis to the results obtained through our survey.

4.3 Measures

4.3.1 Edit lag

Although previous work has demonstrated that strong correlations between interruptions and increased resumption lag exist, the measured resumption lag has been on 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 (van Solingen et al. 1998). 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 (see Fig. 2).

In this study, we focus most of our analysis on edit lag to gain insight into what activities developers perform before they have regained 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. For this reason, we only examine sessions with coding activity.

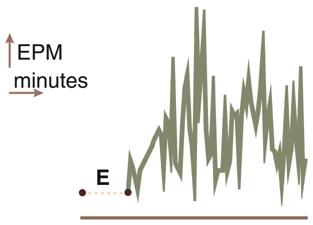

Fig. 2 Edit lag (*E*). When starting a new programming session, a latency can be typically observed before coding activity (edits to code) begins. We believe activities during this time period are used to prepare for the coding activity. Typically, once coding activity has been initiated, it strongly persists for the rest of the session (see *graph line*)

5 Analysis

When a programmer is interrupted from a programming task, what activities are needed 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 first measure the distribution of edit lag. If we observe that most values of edit lag are less than 1 min, 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 Fig. 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 min. 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.

In Table 4, we show a breakdown of the frequency and duration of sessions in a typical day.

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

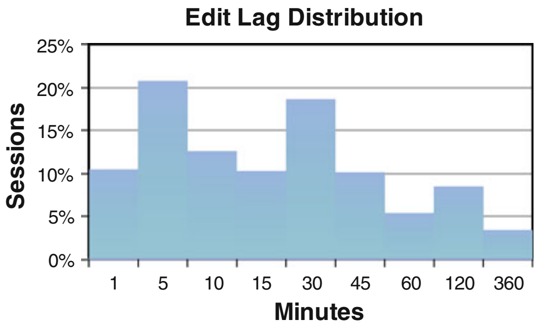

Fig. 3 Developers often do not make their first edit of the session until at least several minutes have past

Table 4 The frequency of sessions of various durations is shown

	Sessions in a typical day					
Sessions	1–3	1–2	1–2	1	0–1	Rarely
Duration	15 m	30 m	1 h	2 h	4 h	8 h

In a typical day, developers program in several short sessions with an additional one or two longer sessions. That is, in a normal day a developer might have three 15 min sessions, two 30 min sessions, a 1 h session, and another two hour session

5.1 Return to last method edited

One of the simplest methods for resuming a programming task is to return to the site where work was last performed. But how sufficient and 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.

Our analysis finds that in 91 of 1,213 sessions (7.5%) the programmers were able to make changes without navigating to other methods. Second, in 209 out of 1,213 sessions (17%), the programmers do eventually return from visiting 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 5. 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 1 min 35% of the time. We believe in these situations, the programmer can successfully use the method itself to remember sufficient details for completing the task. In other cases, the programmer may be spending time re-understanding the code or re-evaluating his or her 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.

Table 5 Developers are able to resume coding within 1 min for 35% of sessions and within 30 min for most sessions when the work involves completing the last edited method

	Resumption cost			
Sessions (%)	35	22	23	12
Edit lag (m)	1<	1–5	5–15	15–30

5.2 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 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 code elements. For the *Eclipse* data set, we measure the number of selection events prior to the edit.

The results can be seen in Table 6. 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 7, 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 5).

5.3 Execute the program

Debugging is an expensive but comprehensive way of obtaining knowledge about the program behavior. We suspect developers may be using debugging or program execution to measure progress or recover forgotten subgoals.

To measure the frequency of use of this strategy, we examined commands used in the *Eclipse* data set related to launching a debugging session or execution of the program. We do not know if these commands are being used intentionally as a resumption strategy; it may be possible that the programmer is continuing debugging the program from the last session.

In Table 8, we display the use of both the debug and run commands. Programmers execute the program prior to making edits through the run command 2% of the time and through the debug command 13% of the time. The higher rate of the debug command could

Table 6 Developers typically visit several locations of code before beginning a change

	Visual Studio	Eclipse
Locations	2–12 (7)	
Navigation distance	4–40 (27)	
Selection events		15–150 (135)

The range of elements covering 75% of values is shown with the mean in parenthesis

Table 7 Developers take a little longer to start coding when the work involves navigating to other locations first

	Resumption cost				
Sessions (%)	16	25	22	18	8
Edit lag (m)	1<	1–5	5–15	15–30	30–45

In contrast with Table 5, fewer sessions are quickly resumed (35% vs. 16%)

Table 8 Frequency of programmers debugging or running the program before making their first edit

	Pre-edit lag	Post-edit lag	% Sessions
Debug	521	1,666	13
Run	84	693	2
Total	605	2,359	15

Table 9 Participants reported their frequency of program execution for resumption

Frequency	5%	25%	50%	75%	95%	Weighted average (%)
Participants	126	89	68	48	22	33

suggest many of these sessions were continuations of previous debugging sessions. Another explanation could be that running the program through the debug command would provide advantages such as breaking on exception or retrospectively setting a breakpoint. Also of interest is the frequency of program execution: 59% of sessions involve some form of program execution. Even if program execution is not a resumption strategy, it is certainly a common strategy for evaluating the progress and correctness of a task.

From our survey results (see Table 9), participants indicated that they use program execution for a variety of reasons when resuming a task. When participants were asked if they run a program and then examine its output or UI, they responded positively 33% of the time. However, a large number of users indicated they never use this strategy or are not aware they are consciously doing it.

Participants described two ways they used program execution. One participant noted how he used program execution to assist in restoring his representation of the program:

> Sometimes I need to put in breakpoints and execute the application to recall the algorithm.

Another common strategy was to use program execution to check for problems when resuming work:

> Run unit testing lets me know what has passed and what has not.

5.4 Search for prospective notes

When a programmer leaves behind roadblock cues or prospective notes in the code, he can use these reminders to resume work. In this section, we are measuring how often programmers may leave behind these types of reminders.

To measure the use of roadblock cue reminders, we examine the *Eclipse* data set to measure usage of the Problem View, a view that lists current warnings or compile errors associated with the program. 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 simply that they could not be fixed during the previous session. We could not directly measure the use of prospective notes in the code through our interaction histories data sets, but have results from our survey.

The results show that the Problem View was used in 9% of sessions (see Table 10). From our survey results, many participants described the use of intentional compile errors placed in their open responses. When asked how frequently they would use compile or build errors (when available), participants estimated using this strategy in 43% of resumptions.

Table 10 Viewing problems left over from the last session is common

	Pre-edit lag	Post-edit lag	% Sessions
Problem view	301	265	9

Several participants described how they used compile errors. When first resuming, often participants would build the program to check for compile errors, almost compulsively. One used compile errors as a way of making prospective notes more visible:

> Add notations to the current code file of what needs to be done, but don't add the comment tag '//' so these comments will force a compile error.

Another noted how roadblock cues could be more prominent than environmental cues:

> If I literally have to drop something, I'll make sure that it has a compiler error and put comments to help me remember what I was doing and what I planned to do next. The compiler error prevents me from forgetting to finish. I almost never rely on Visual studio being open or in the same state as when I left—I've been burned by that strategy often enough.

5.5 Review task descriptions and notes

Programmers actively manage and perform many tasks when developing software. Some tasks may be simple, 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 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 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.

To measure the use of task tracking for management of subtasks, we examined the *UDC* dataset and *Eclipse* dataset. The *UDC dataset* includes commands recorded from users using Mylyn to track software tasks. We count the number of users that use Mylyn 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 11. Developers viewed task information in 9% of sessions. However, the associated edit lag was also very high. For 75% of the sessions that viewed task information, the edit lag was greater than 30 min. This suggests the start of these sessions might have been spent exploring and investigating the source code in order to formulate a change plan.

Table 11 Viewing the task list or bug list during the beginning of a session is common

	Pre-edit lag	Post-edit lag	% Sessions
Tasklist	274	246	9

Table 12 Tasking software is popular for reviewing assigned tasks but not for recording low-level tasks

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

In Table 12, the results display how many users recorded subtasks in Mylyn and compares this to other task operations available in Mylyn. A more detailed breakdown of the four operations in Mylyn is the following:

- list the tasks in an external task repository,
- open a task from the task repository,
- create a new task that was not be stored in the task repository, and
- create 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 for managing assigned tasks.

From our survey results, 77% of participants responded that they actively track the status of tasks with note-taking. In their open responses, the participants described where and what items they tracked using note-taking. Taking personal notes about tasks was often necessary because the originally assigned task was often vague or required many unspecified subtasks to accomplish. The participants consistently reported using a mix of media to record notes, including the bug tracking system (TFS or Product Studio), the note-taking product Microsoft OneNote, the source code itself, as well as traditional paper media. The choice of media often reflects whether the task is long-term and/or shared. One participant describes how he used different media for the tasks.

> Product Studio at a large scale, then TodoList…for longer-term checklists, the physical whiteboard for shorter term checklists and sometimes NotePad for very temporary stuff.

Another participant described what items he tracked about a task:

> I note down the big chunks of the task that needs to be done and also a list of corner cases that needs to be handled. While doing [so] I keep striking out the subtasks that have been completed. Main purpose of this exercise is to ascertain that the task is complete.

5.6 Review source code change 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

Table 13 Viewing history during the edit lag is as common as during the rest of the programming session

	Pre-edit lag	Post-edit lag	% Sessions
History	142	183	4
Commit	193	390	11

Table 14 Participants reported their frequency of using source differences as a resumption strategy

Frequency	5%	25%	50%	75%	95%	Weighted average (%)
Participants	79	104	68	57	39	39

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 occurrence 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 seen in Table 13. From this measurement, we observe only 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. Finally, edit lags were higher than 15 min for 59% of sessions when using revision history commands.

From our survey results (see Table 14), participants indicated they view a source difference 39% of the time. One motivation for checking differences is to resolve potential conflicts that may have occurred since the interruption. This may explain one reason why the edit lag can be high:

> A part of the source codes may have changed since the interruption, so I need to sync/resolve the difference.

Another reason for using a code difference was to obtain an overview of the changes:

> use windiff if needed to quickly understand where i was with my code changes (if i am in the middle of implementing some feature).

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.

6 Discussion

Several analyses were described in Sect. 5. In Table 15, a summary is given of the frequency of various activities developers performed when starting a programming session. Reviewing results from the survey and interaction history logs give us different insights; however, it is interesting to note that both sources agree in the order of frequency among the resumption strategies. The survey results for strategies frequency may give the impression of small differences between strategy usage. This is mainly an artifact of reporting the weighted mean of very different responses from individuals (e.g., 200 people using strategy A with 50% frequency would yield the same mean as 100 people using

Table 15 Developers perform various activities at the beginning of a session

Strategies	Usage (%)	Survey (%)
Note-taking (suspension)		77
Continue last edit	7.5	
Nav then continue last edit	17	
Navigate to new location	83	
Navigate to remember		58
View problem list	9	43
View task or bug list	9	42
View revision history	4	39
Execute/debug program	2/15	33

strategy B with 25% frequency and another 100 people using strategy B with 75% frequency); however, examining the underlying distributions show they are significantly different ($p < 0.05$) by a Kolmogorov-Smirnov test for every distribution except between View Problem List and View Task List.

From our survey results, we found participants using many coping mechanisms to recover from interruptions. Although these tools were not specifically built for interruption management, participants have devised several strategies in attempts to recover their task knowledge. One participant describes using several of the strategies we have investigated:

> Review any notes from when task was interrupted, discuss status with other teammates, view code differences, search code for previous TODO's, run build and test scripts and analyze output.

Finally, participants often described how they would select strategies based on how much task knowledge needs to be recovered. One participant described this process as the following:

> I tend to look at where I stopped and then, if need be, I will look over what I have done so far (using something like a differencing tool). If that isn't enough, then it's normally a task on which I would have at least jotted myself down some notes so I will read over those and then resume.

How do the presented results affect future research and tools? 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.

6.1 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 inless 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 in 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 these situations, 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 (Zeigarnik 1927; McKinney 1935). It is unknown how much contextual detail programmers lose after task completion. There is some evidence that gives direction. In Bailey's study (Adamczyk and Bailey 2004), 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 himself. Future research should continue to evaluate alternative navigation interfaces such as CodeThumbnails (DeLine et al. 2006); Relo (Sinha et al. 2006); and SHriMP (Storey et al. 2002) and their affect on resumption costs.

6.2 Other sources of task knowledge

We also reviewed several 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. Second, a formal representation of task structure including all relevant subtasks could be an important help in reducing the cost of task switching and interruptions management. Many research tools rely on the assumption that developers will participate in detailing tasking information. However, few users actively use available facilities for recording subtasks or small tasks. Instead, they use simple handwritten notes for recording task breakdowns. The issues that developers have with these tools must be discovered and addressed before such approaches becomes popular. Other interfaces for recording information may alleviate this problem: Developers may be receptive to simpler input mechanism that allows hand-written and 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 tasks. This would be consistent with other research: Status awareness was one of the top information needs found in Ko's study (Ko et al. 2007) of developers. Developers may also be using facilities such as a code differencing tool to remind themselves of what changes were made during their last session. Surprisingly, this process is still manual and this could explain the low use in comparison to other strategies. Participants noted using program execution for checking correctness and refreshing mental representations; however the expense in setting up the debugging environment may explain the infrequency of this strategy. 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.

6.3 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 to collecting task 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 sources 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.

To gain more insight into future research efforts, we surveyed developers about what features they would like in future tool development. The first set of features they described were several approaches for enhancing retrospective measures by enhancing environmental cues and providing different views of the source code:

- *Automatic Tags*: A tag cloud of links to recent source code symbols and names inside method bodies.
- *Change Summary*: Short summary of my changes.
- *Code Thumbnails*: Thumbnails of recent places I navigated or edited.
- *Instant Diff.*: Highlighted code showing how I changed a method body as well as providing a global view.
- *Activity Explorer*: Historical list of actions such as search, navigating, refactoring. Expanding item gives thumbnail of IDE action or code location.
- *Snapshots/Instant Replay*: Timeline of screen-shot thumbnails or an instant replay of my past work.

Continuing from the last state of the programming task can happen in a variety of ways: The developer might resume coding in the previous location, might need to transition to the next step of the task, or might 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 light-weight 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.

Table 16 Average ratings of possible resumption features, on a 5-point Likert scale

Ratings for features	1	2	3	4	5	Avg.
Automatic Tags	83	68	75	84	39	2.8
Change Summary	28	46	95	127	60	3.4
Code Thumbnails	41	63	84	112	53	3.2
Instant Diff.	20	24	63	125	122	3.9
Activity Explorer	45	75	89	104	41	3.1
Snapshots/Instant Replay	67	76	75	74	60	3.0
Task Sketches	69	77	83	83	41	2.9
Runtime Information	54	75	97	84	44	2.8
Prospective Cues	67	85	118	55	23	2.7

- *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.

The developers' ratings in Table 16 are notably consistent in what help they want when resuming tasks. In general, they favor tool support for retrospective measures versus prospective measures. This may be because they are already happy with the note-taking approaches they use or still think these features may be too heavy-weight. One participant noted:

> Task sketches—most task list "helpers" are more harm than help. I rated it a 2 for that reason. If you somehow made it awesome, then perhaps I'd use it more.

The top four choices show that they want to see a summary of the *content* that they edited or inspected before the interruption, whether it is shown integrated in the code (Instant Diff.), in a separate window (Change Summary), on thumbnail versions of the code (Code Thumbnails), or in a history view (Activity Explorer). On participant noted that what is really missing from current tools was the ability to recover fine-grained changes rather than a flat summary that a code difference presents:

> I use sd/sdv/windiff/windiffredirector/Beyond compare/related tools frequently already. The main drawback is that I don't have a good time guess on each of those changes… However, the ultimate way to improve the existing tools would be to add the time component.

Finally, one participant was enthusiastic to have any resumption support:

> All of the above would be absolutely amazing, PLEASE MAKE THIS. If I had to rank them, I'd say the top of my list would be Runtime Information (notably previous Debug values), Task Sketch, Instant Diff., Code Thumbnails. Prospective Cues might be ok, depending entirely on what conditions are defined.

6.4 Limitations of analysis

We have presented an analysis of developers' activity before coding and discussed several observations and hypotheses explaining their behavior. The nature of the datasets we have analyzed gives us a good overview of behavior but understandably several unknown factors might be at play.

One main threat to the validity of 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 need 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 to task repository activity.

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

Differences both across and within datasets threaten the generality of these results. For example, the *Visual Studio dataset* has finer granularity of navigation events and more detail on code entities; whereas the *Eclipse dataset* has more detail on IDE events. These differences across the datasets limit our ability to analyze our questions throughly (i.e., *Visual Studio dataset* was applicable for Sects. 5.1, 5.2). Within a dataset, differences between users such as experience and differences between tasks such as investigation or refactoring all influence factors surrounding resumption beyond our control. As such, any attempt to draw general assertions on the impact of interruption from these results (i.e., resumption time) should be avoided. Instead, we recommend researchers use these results as a baseline and guideline for future study.

Finally, the interaction history data does not capture a complete representation of all possible activities that were performed. We cannot account for productive time away from the keyboard that may have helped resume a task. Also, developers may be using other sources of reminder cues such as notes on their desk or comments in source code. Although we have corroborated this evidence with results from our survey study, other types of studies would need to be performed to measure the frequency and effectiveness of these techniques.

6.5 Future work

We are investigating several directions for expanding this research. We will perform 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 has a long productive session. Another trend we observed that 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.

7 Conclusions

In this paper, we theorize on suspension and resumption strategies that programmers use to resume work. 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. We conduct a survey to gain insights into why these activities are performed and how might future tools better support this process.

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. 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 an IDE organizes content beyond tabbed editing and hierarchical lists.

References

Adamczyk, P. D., & Bailey, B. P. (2004). 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* (pp. 271–278). New York, NY: ACM.

Allwood, C. (1984). Error detection processes in statistical problem solving. *Cognitive Science, 8*(4), 413–437.

Altmann, E. M., & Trafton, J. G. (2002). Memory for goals: An activation-based model. *Cognitive Science, 26*, 39–83.

Altmann, E. M., & Trafton, J. G. (2004). Task interruption: Resumption lag and the role of cues. In *Proceedings of the 26th annual conference of the cognitive science society*.

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

Anderson, J. R., Bothell, D., Byrne, M. D., Douglass, S., Lebiere, C., & Qin, Y. (2004). An integrated theory of the mind. *Psychological Review, 111*, 1036–1060.

Anderson, J. R., Farrell, R., & Sauers, R. (1984). Learning to program in lisp. *Cognitive Science, 8*(2), 87–129.

Atkinson, R. C., & Shiffrin. (1968). *The psychology of learning and motivation* (Vol. 2), chapter Human memory: A proposed system and its control processes (pp. 89–195). Baddeley: Academic Press.

Baddeley, A., & Hitch, G. (1974). *The psychology of learning and motivation: Advances in research and theory*, chapter working memory (pp. 47–89). New York: Academic Press.

Bannon, L., Cypher, A., Greenspan, S., & Monty, M. L. (1983). Evaluation and analysis of users' activity organization. In *CHI '83: Proceedings of the SIGCHI conference on human factors in computing systems* (pp. 54–57). New York, NY: ACM.

Bellotti, V., Dalal, B., Good, N., Flynn, P., Bobrow, D. G., & Ducheneaut, N. (2004). What a to-do: Studies of task management towards the design of a personal task list manager. In *CHI '04: Proceedings of the SIGCHI conference on human factors in computing systems* (pp. 735–742). New York, NY: ACM.

Borst, J. P., & Taatgen, N. A. (2007). The costs of multitasking in threaded cognition. In *ICCM '07: Proceedings of the 8th international conference on cognitive modeling* (pp. 133–138). Oxford: Psychology Press.

Brush, A. B., Meyers, B. R., Tan, D. S., & Czerwinski M. (2007). Understanding memory triggers for task tracking. In *CHI '07: Proceedings of the SIGCHI conference on human factors in computing systems* (pp. 947–950). New York, NY: ACM.

Chase, W., & Simon, H. (1973). Perception in chess. *Cognitive Psychology, 4*, 55–81.

Chase, W. G., & Ericsson, K. A. (1982). *The psychology of learning and motivation*, Vol. 16, chapter Skill and working memory (pp. 1–58). New York: Academic Press.

Cherubini, M., Venolia, G., DeLine, R., & Ko, A. J. (2007). Let's go to the whiteboard: How and why software developers use drawings. In *CHI '07: Proceedings of the SIGCHI conference on human factors in computing systems* (pp. 557–566). New York, NY: ACM.

Cowan, N. (2001). The magical number 4 in short-term memory: A reconsideration of mental storage capacity. *The Behavioral and brain sciences, 24(1)*, 87−185.

Cutrell, E., Czerwinski, M., & Horvitz, E. (2001). Notification, disruption and memory: Effects of messaging interruptions on memory and performance.

Czerwinski, M., Horvitz, E., & Wilhite, S. (2004). A diary study of task switching and interruptions. In *CHI '04: Proceedings of the SIGCHI conference on human factors in computing systems* (pp 175–182). New York, NY: ACM Press.

Dekel, U. (2008). Designing a prosthetic memory to support software developers. In *ICSE companion '08: Companion of the 30th international conference on software engineering* (pp. 1011–1014). New York, NY: ACM.

DeLine, R., Czerwinski, M., Meyers, B., Venolia, G., Drucker, S., & Robertson, G. (2006). Code thumbnails: Using spatial memory to navigate source code. In *VLHCC '06: Proceedings of the visual languages and human-centric computing* (pp. 11–18). Washington, DC: IEEE Computer Society.

Ericsson, K. A., & Kintsch, W. (1995). Long-term working memory. *Psychological Review, 102*, 211–245.

Ericsson, K. A., & Staszewski, J. J. (1989). *Complex information processing: The impact of Herbert A. Simon*, chapter Skilled memory and expertise: Mechanisms of exceptional performance (pp. 235–267). Hillsdale, NJ: Lawrence Erlbaum.

Fogarty, J., Ko, A. J., Aung, H. H., Golden, E., Tang, K. P., & Hudson, S. E. (2005). Examining task engagement in sensor-based statistical models of human interruptibility. In *CHI '05: Proceedings of the SIGCHI conference on human factors in computing systems* (pp. 331–340). New York, NY: ACM.

Gilboa, A., et al. (2006). Hippocampal contributions to recollection in retrograde and anterograde amnesia. *Hippocampus, 16*(11), 966–980.

Gillie, T., & Broadbent, D. (1998). What makes interruptions disruptive? a study of length, similiarity, and complexity. *Psychological Research, 50*, 243–250.

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

Iqbal, S. T., & Bailey, B. P. (2005). Investigating the effectiveness of mental workload as a predictor of opportune moments for interruption. In *CHI '05: CHI '05 extended abstracts on human factors in computing systems* (pp. 1489–1492). New York, NY: ACM.

Kersten, M., & Murphy, G. C. (2006). 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* (pp. 1–11). New York, NY: ACM.

Ko, A. J., Aung, H., & Myers, B. A. (2005). Eliciting design requirements for maintenance-oriented ides: A detailed study of corrective and perfective maintenance tasks. In *ICSE '05: Proceedings of the 27th international conference on software engineering* (pp. 126–135). New York, NY: ACM.

Ko, A. J., DeLine, R., & Venolia, G. (2007). Information needs in collocated software development teams. In *ICSE '07: Proceedings of the 29th international conference on software engineering* (pp. 344–353). Washington, DC: IEEE Computer Society.

Latoza, T. D., Venolia, G., & Deline R. (2006). Maintaining mental models: A study of developer work habits. In *ICSE '06: Proceeding of the 28th international conference on software engineering* (pp. 492–501). New York, NY: ACM Press.

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

Mark, G., Gonzalez, V. M., & Harris J. (2005). No task left behind? Examining the nature of fragmented work. In *CHI '05: Proceedings of the SIGCHI conference on human factors in computing systems* (pp. 321–330). New York, NY: ACM Press.

Mayrhauser, A. V., & Vans, A. M. (1997). Hypothesis-driven understanding processes during corrective maintenance of large scale software. In *ICSM '97: Proceedings of the international conference on software maintenance* (pp. 12–20). Washington, DC: IEEE Computer Society.

McFarlane, D. (2002). Comparison of four primary methods for coordinating the interruption of people in human-computer interaction. *Human-Computer Interaction, 17*(1), 63–139.

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

Miller, G. A. (1994). The magical number seven, plus or minus two: Some limits on our capacity for processing information. 1956. *Psychological Review, 101*(2), 343–352.

Milner, B., Corsi, P., & Leonard, G. (1991). Frontal-lobe contribution to recency judgements. *Neuropsychologia, 29*(6), 601–618.

Miyata, Y., & Norman, D. A. (1986). Psychological issues in support of multiple activities. In D. A. Norman & S. W. Draper, (Eds.), *User centered system design: New perspectives on human-computer interaction* (pp. 265–284). Hillsdale, NJ: Erlbaum.

Monk, C., Trafton, J., & Boehm-Davis, D. A. (2008). The effect of interruption duration and demand on resuming suspended goals. *Journal of Experimental Psychology: Applied, 14*, 299–313.

Monk, C. A. (2004). The efffect of frequent versus infrequent interruptions on primary task resumption. In *Proceedings of the human factors and ergonomics society 48th annual meeting*.

Morris, R. G., & Frey, U. (1997). Hippocampal synaptic plasticity: Role in spatial learning or the automatic recording of attended experience? *Philosophical Transactions of the Royal Society of London. Series B, Biological Sciences, 352*(1360), 1489–1503.

Morris, R. G. M. (2006). Elements of a neurobiological theory of hippocampal function: The role of synaptic plasticity, synaptic tagging and schemas. *European Journal of Neuroscience, 23*(11), 2829–2846.

Murphy, G. C., Kersten, M., & Findlater, L. (2006). How are Java software developers using the Eclipse IDE? (vol. 23, pp. 76–83). Los Alamitos, CA: IEEE Comp. Soc. Press.

O'Brien, M. P., & Buckley, J. (2005). Modelling the information-seeking behaviour of programmers—an empirical approach. In *IWPC '05: Proceedings of the 13th international workshop on program comprehension* (pp. 125–134). Washington, DC: IEEE Computer Society.

O'Conaill, B., & Frohlich, D. (1995). Timespace in the workplace: Dealing with interruptions. In *CHI '95: Conference companion on human factors in computing systems* (pp. 262–263). New York, NY: ACM Press.

Offner, M. (1911). *Mental Fatigue*. Warwick & York.

Parnin, C., & DeLine, R. (2010). Evaluating cues for resuming interrupted programming tasks. In *CHI '10: Proceedings of the 28th international conference on human factors in computing systems* (pp. 93–102). New York, NY: ACM.

Parnin, C., Görg, C. (2006). Building usage contexts during program comprehension. In *ICPC '06: Proceedings of the 14th IEEE international conference on program comprehension* (pp. 13–22).

Pennington, N., & Grabowski, B. (1990). The tasks of programming. In *Psychology of programming*, Computer and People Series, chapter 1.4 (pp. 45–62). London: Academic Press Ltd.

Ratwani, R. M., & Trafton J. G. (2008). Spatial memory guides task resumption. *Visual Cognition, 16*, 1001–1010.

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

Robbes, R., & Lanza, M. (2007). Characterizing and understanding development sessions. In *ICPC '07: Proceedings of the 15th IEEE international conference on program comprehension* (pp. 155–166). Washington, DC: IEEE Computer Society.

Safer, I., & Murphy, G. C. (2007). 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* (pp. 229–243). New York, NY: ACM.

Salvucci, D. D., Taatgen, N. A., & Borst, J. P. (2009). Toward a unified theory of the multitasking continuum: From concurrent performance to task switching, interruption, and resumption. In *CHI '09: Proceedings of the 27th international conference on human factors in computing systems* (pp. 1819–1828), New York, NY: ACM.

Singer, J., Elves, R., & Storey, M.-A. (2005). Navtracks: Supporting navigation in software maintenance. In *ICSM '05: Proceedings of the 21st IEEE international conference on software maintenance* (pp. 325–334). Washington, DC: IEEE Computer Society.

Sinha, V., Karger, D., & Miller, R. (2006). Relo: Helping users manage context during interactive exploratory visualization of large codebases. In *VLHCC '06: Proceedings of the visual languages and human-centric computing* (pp. 187–194). Washington, DC: IEEE Computer Society.

Squire, L. R. (2004). Memory systems of the brain: A brief history and current perspective. *Neurobiology of learning and memory, 82*(3), 171–177.

Storey, M.-A., Best, C., Michaud, J., Rayside, D., Litoiu, M., & Musen, M. (2002). Shrimp views: An interactive environment for information visualization and navigation. In *CHI '02: CHI '02 extended abstracts on human factors in computing systems* (pp. 520–521), New York, NY: ACM.

Storey, M.-A., Ryall, J., Bull, R. I., Myers, D., & Singer, J. (2008). 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* (pp. 251–260), New York, NY: ACM.

The Eclipse Foundation. (2009). Usage Data Collector Results. January 5th, 2009. Website, http://www.eclipse.org/org/usagedata/reports/data/commands.cs.

Trafton, J. G., Altmann, E. M., & Brock, D. P. (2005). 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*.

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

Tulving. E. (1972). *Organization of memory*, chapter Episodic and semantic memory (pp. 381–403). New York: Academic Press.

Tulving, E., & Thomson, D. M. (1973). Encoding specificity and retrieval processes in episodic memory. *Psychological Review, 80*, 352–373.

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

Winograd, E. (1988). *Practical aspects of memory: Current research and issues*, Vol. 2, chapter Some observations on prospective remembering (pp. 348–353). Chichester: Wiley.

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

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

Zou, L., & Godfrey, M. W. (2006). An industrial case study of program artifacts viewed during maintenance tasks. In *WCRE '06: Proceedings of the 13th working conference on reverse engineering* (pp. 71–82). Washington, DC: IEEE Computer Society.

Author Biographies

Chris Parnin is a PhD student in the Department of Computer Science at Georgia Tech. His research interests include human-computer interaction and software tools.

Spencer Rugaber is a faculty member of the College of Computing at the Georgia Institute of Technology. His research interests has concentrated on reverse engineering, software evolution, and user interfacess. In addition to his academic posts he also worked as an engineer at Bell Labs and program director for NSF. He holds a PhD in Computer Science from the Yale University.