

# Evaluating Cues for Resuming Interrupted Programming Tasks

(author names removed for blind review)

## ABSTRACT

Developers, like all modern knowledge workers, are frequently interrupted and blocked in their tasks. In this paper we present a contextual inquiry into developers' current strategies for resuming interrupted tasks and investigate the effect of automated cues on improving task resumption. We surveyed 371 programmers on the nature of their tasks, interruptions, task suspension and resumption strategies and found that they rely heavily on note-taking across several types of media. We then ran a controlled lab study to compare the effects of two different automated cues to note taking when resuming interrupted programming tasks. The two cues differed in (1) whether activities were summarized in aggregate or presented chronologically and (2) whether activities were presented as program symbols or as code snippets. Both cues performed equally well: developers using either cue completed their tasks with twice the success rate as those using note-taking alone. Despite the equal performance of the cues, developers strongly preferred the cue that presents activities chronologically as code snippets.

## Author Keywords

interruptions, task resumption, cues

## ACM Classification Keywords

H.5.2 Information Interfaces and Presentation: Miscellaneous

## INTRODUCTION

Every day, professional developers engage in a process of recovering knowledge about software. When resuming an incomplete programming task, the developer must recall their previous working state and knowledge about the software. Details of working state might include recalling plans, intentions, and relevant source code documents. Details of knowledge might include algorithms, component mechanisms, and domain representations.

In psychology studies of interruptions, researchers have characterized the effects on performance [4] and mental fatigue [19] as well as understanding the contribution of interruption frequency [17] and interruption length, task similarity, and complexity [10] on performance. Despite efforts for managing interruptions, *in situ* studies suggest interruptions remain problematic. Czerwinski's study [5] showed that tasks resumed after an interruption were more difficult to perform and took twice as long. O'Conaill's study [18] found 40% of interrupted tasks are not resumed at all. Further research by Mark *et al.* [16] observed that 57% of tasks were interrupted – as a result, work on a task often was fragmented into many small work sessions.

Studies examining software companies have also replicated

similar results from previous workplace studies. Solingen [32] characterizes interruptions at several industrial software companies and observed that an hour a day was spent managing interruptions, and developers typically required 15 minutes to recover from an interruption. Ko *et al.* [14] used a *fly-on-the-wall* [15] 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. Parnin and Rugaber [23] analyzed interaction logs from 10,000 programming sessions from 85 programmers and found that in a typical day, developers program in many short sessions (15-30 minutes) with an additional one or two longer sessions (1-2 hours). They also spend a significant amount of time at the start of a session navigating to various locations before resuming coding activity, which suggests that the last stopping point was insufficient for rebuilding their working context.

To resume a task after an interruption, a programmer needs to recover both the task's working context (the applications, windows and documents needed for the task) and her mental context (knowledge about the program, goals, plans). There has been a considerable line of research, summarized below, to discover and organize the working contexts that comprise a knowledge worker's tasks. The hope is that if a tool is able to persist the working context, when a task is resumed the user will have an easier time recovering the corresponding mental state. Researchers have proposed that maintaining a representation of task context would help reduce mental workload during development. Because the risk of interruptions includes the loss of details and increased time to recover working state, some have hypothesized that maintaining a representation of context in the IDE would reduce resumption costs [22, 13, 8, 7]. Of these, Mylyn [13] has emerged to become an integrated feature of the Eclipse development environment and is the most common standard for representing task context. Although there is some evidence these task context tools reduce navigation cost [22, 13, 27], there is little evidence describing the effect of using task context when resuming interrupted programming tasks.

In this paper, we investigate how programming environments can be enhanced to assist developers when resuming interrupted programming tasks. First, we surveyed 371 developers to learn about their current habits for coping with interruption and their preferences for potential treatments to help them recover from interruption. We then ran a controlled study looking at the effect of three different interruption recovery conditions on task completion and errors. Our base condition, note taking, represents the most prevalent existing practice for preserving mental state, based on our preliminary study. The other two conditions represent different automated cues, which the subjects could combine with note

taking. The first automated cue replicates Mylyn’s design; the second integrates the most highly rated features from our preliminary study.

## BACKGROUND

### Studies of Task Management

Researchers have investigated how information workers manage general tasks for insights into improving task management. Bellotti [2] studied the various media employed by 9 different managers and found in addition to using paper media such as sticky notes – 50% of the tasks could be linked to emails. Brush [3] observed 8 workers who needed to frequently file status reports of their weekly activity. 5 out of the 8 workers attempted to continually track information in a centralized location: *e.g.* notepad, ongoing email draft – but all participants also complemented their tracking strategy with memory triggers to retrieve details needed for composing the report. In a diary and field study of web tasks, tasks that required multi-sessions were common: 33% of web sessions involved at least one long-running (multi-session) task [12]. Users reported the need for better support in managing their tasks between sessions, restoring viewed pages, managing information associated with a task, and cleaning up content after completion.

Unfortunately, knowledge workers, such as programmers, face a different set of problems than previously studied. For example, many of the tasks previously studied by Bellotti involved managing a high volume of prospective tasks that could often be individually resolved with short-term activities or delegation. Many of the tasks were also typically associated with requests from clients or office mates; a factor likely contributing to the success of the email-centered approach. However, software developers are instead assigned a low volume of highly complex work items from a smaller network of people. Email-centered approaches would not be as likely successful for programming tasks. Like web tasks, programming tasks often involve long periods of investigation, but with several key differences: the documents are with more structure and are usually more complex, the documents are actively edited by the programmer and other teammates, and where searching on the web potentially is asking every expert on a subject, much programming knowledge is tacit and only available from other busy co-workers.

### Studies of Developers’ Tasks Management

Parnin and Rugaber [23] analyzed 10,000 programming sessions from 85 programmers and found that when developers resume a programming task, they spend a significant amount of time at the start of a session navigating to various locations before resuming coding activity. This information seeking activity suggests that relying on only the last stopping point was insufficient toward rebuilding working context. Other common activities at the start of a programming session included viewing task lists (*e.g.* *TODO* comments), viewing compile errors, and occasionally comparing the history of changes. Storey *et. al* have also observed programmers using task annotations (via *TODO* comments) [28] for recording prospective programming tasks. These activities suggested that programmers used different strategies

for regaining task context when resuming a programming session.

### Tool Support for Task Management

There have been many proposed tool designs for maintaining context during task-switching or recovering from interruptions. One proposed line of work has focused on how users can better organize their working context. These approaches focus on reducing the clutter associated with the mixed landscape of the computing desktop. Group Bar [24] allows users to organize windows and documents located in the task bar of Windows OS into similar groups based on their tasks. Other tools such as CAAD [25], attempt to do this automatically by clustering related work items based on interaction coupling. Rather than organize the documents and windows into definitive task categories, another approach allows users to associate personal tags with documents and activities [20] or shared work items [31]. For programmers, Mylyn<sup>1</sup>, a tool available with Eclipse, allows users to filter code documents based on a selected task and recent interaction history.

Another direction of research has investigated various ways of augmenting temporal, spatial, and contextual cues in the environment. Scalable Fabric [26] using a zoomable organization of windows and documents to leverage spatial memory for facilitating task switching. A similar effort has been made for viewing source code through Code Thumbnails [6]. Window Scape [29] introduced temporal snapshots of windows where users can return to recent contexts. In a study of searching open documents and windows [21], the researchers found improved performance for switching between interrupted tasks by introducing temporal and object similarity cues to the preview windows of the Alt-Tab interface of Windows OS.

Successful organization of work items through tools can reduce the overhead and friction between switching tasks; however, we believe it does not fully address the problems of recovering lost working state. Other than Mylyn, previous resumption aids have not taken advantage of the structure of the task content. That is, previous treatments focus on the gross-level structure of the tasks – windows and documents – not the fine-level structure of the task – methods or lines of code. Finally, many designs have been proposed, but few evaluate their performance on recovering from interruptions, especially knowledge-intensive tasks like programming. In this paper, we are concerned with evaluating the role and performance of resumption aids on interrupted programming tasks.

### Studies Evaluating Environmental Cues

Recent research has identified a strong connection between internalizing environmental cues and reducing resumption costs [30, 11]. In these experiments, the availability of cues during task resumption reduces the time to restart the task. Conversely, if the cue is removed or tampered with, then this benefit is removed. This even holds for implicit cues such as the location of a mouse cursor. For example, in one

---

<sup>1</sup>[www.eclipse.org/mylyn](http://www.eclipse.org/mylyn)

experiment when the mouse cursor location was moved from hovering over the last button clicked to be in the corner of the screen, the resulting resumption lag was higher than when the implicit cue was present.

Although previous work has demonstrated strong correlations between interruptions and increased resumption lag, the measured resumption lag has been in the order of seconds. In the domain of software development, the effects of interruption are believed to have a larger impact on loss of context and the recovery period to be on the order of minutes [32, 23].

### Interruptions

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

Forgarty *et. al* performed an experiment [9] to predict the interruptibility of programmers and the time needed to reach a breakpoint. In the experiment, programmers were interrupted every 3 minutes 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 seconds; however, when more deeply engaged, they would defer for a mean of 43 seconds.

For this paper, we divide interruptions into three categories: interruptions with no warnings (a phone call or office visitor); interruptions with warning (a scheduled meeting or a visitor who IMs ahead of time); and self-interruptions (task avoidance or task blocks on information needs). In our controlled study, we focus our on interruptions with warning which are common in software development [14].

### INVESTIGATIVE STUDY

While several previous studies have documented the fragmented nature of software developers’ work, there is no existing survey of the strategies developers use to resume interrupted tasks. We wanted to conduct a survey that would help us prioritize the conditions we would later test in the lab. Because the number of interesting conditions is too large to test exhaustively, we wanted to determine one experimental condition to represent current practice (our base line) and another to represent developers’ preferred treatments.

### Method and Procedure

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 industries. We initially conducted the external study and encouraged by the feedback, distributed the survey with slight modifications to employees at Microsoft. Overall, we find general agreement with the external survey and internal survey, but only report the internal results for this paper.

As we were primarily interested with professional software development, we selected our population from full-time software developers – excluding interns or roles such as testers or project managers. We also excluded developers that had been recently surveyed by our research group to avoid over-sampling. 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 feedback on future tools. We drew the selection of future tools from current research proposals for improvements to programming environments and wanted to determine if programmers were primarily interested if tools help them manage task state or tools augment cues within the programming environment.

### Findings

The results are as follow: We asked our participants to give us an estimate how long it takes to recover from a typical and worse-case interruption as seen in Table 1. We present our findings of typical work practices of our participants in Table 2. Finally, we obtained ratings of proposed tools as shown in Table 3.

We discuss these findings in detail in the following section.

	frequency		avg. length	
	typical	worst	typical	worst
seconds	5%	1%	12 s	15s
minutes	85%	56%	9.5 m	22 m
hours	8%	30%	2 h	2 h
days	1%	11%	.5 d	1.4 d

**Table 1. Interruptions of programmer typically takes several minutes to recover but ill-timed interruptions can significantly delay the resumption of a task by hours or days.**

### Discussion

The participants responses about the cost of typical interruptions were consistent with previous studies on programmers (see Table 1). In addition, participants reported that an ill-timed interruption could delay the resumption of the task for several hours or until the next day because they would not have sufficient block of time to rebuild context and make progress. Factors that made things worse included both commonly known factors: task complexity, interruption length, task concreteness, task completeness; and factors specific to

Question	Frequency
<b>Work assignments</b>	
Task is well-defined and includes some details of code implementation.	11%
Purpose of task is clear, but does not include plan on how to code it.	58%
Task broad in scope and needs to be refined into many sub-tasks.	55%
<b>Task tracking</b>	
Do you take notes to track your progress in programming tasks, for example, notes on paper, sticky notes, OneNote pages, documents in Notepad, etc.?	77%
physical notes (post-it, notebook, whiteboard)	46.6%
electronic notes (notepad, OneNote, Excel, Word)	39.9%
task database (bugs, scrum tasks, work items)	16.2%
email (flags, outlook tasks, notes to self, draft email)	15.1%
<b>Suspension Strategies</b>	
Write a physical note.	36%
Make a mental note.	58%
Leave a reminder cue such as text selection or active window.	50%
<b>Resumption Strategies</b>	
Return to last method modified, and navigate to related code to jog your memory.	58%
Use compile or build errors. (When available)	43%
Use a list of issues or a task description	42%
View source code difference.	39%
Run program and examine its output or UI.	33%

**Table 2. Frequency of work practices.**

software development: low understanding of code, number and size of artifacts, and difficulty setting up and reproducing state.

The complexity of the task and the amount of focus required to work on the task efficiently is proportional to how long it takes to resume the task.

The participants' free responses about tracking tasks indicate that note taking is by far the most prevalent practice for managing knowledge about a task. 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.

Instant Diff (Highlighted code showing how I changed a method body as well as a global view.)	3.9
Change Summary (Short summary of my changes.)	3.4
Code Thumbnails (Thumbnails of recent places I navigated or edited.)	3.2
Activity Explorer (Historical list of actions such as search, navigating, refactoring. Expanding item gives thumbnail of tool window or code location)	3.1
Runtime Information (Values or visualizations of variables or expressions from previous execution or debugging sessions)	3.0
Snapshots/Instant Replay (Timeline of screen-shot thumbnails or an instant replay of my past work.)	3.0
Task Sketches (Light-weight annotations of a task breakdown: steps, objectives, and plans.)	2.9
Automatic Tags (A tag cloud of links to recent source code symbols and names inside method bodies.)	2.8
Prospective Cues (Contextual reminders that are displayed when a condition is true)	2.7

**Table 3. Average ratings of possible resumption features, on a 5-point Likert scale.**

Several participants described their personal working style:

Steno Pad and Post Its - Lots and lots of Post Its.

Paper if personal tasks, or notes in Project Studio if public/shared, etc.

I track my progress in OneNote, but small issues that I notice and don't want to forget frequently end up as post-it notes or emails to myself. I'd like to be more consistent about where these things go.

Participants used various strategies for suspending a task in response to an interruption. For self-interruptions, the participants typically try to reach a breakpoint, save and prune their working context, jot down high-level notes about outstanding issues, and leave markers in the code. When participants anticipate a short interruption or are pressed for time, they typically use a more light-weight strategy by leaving the necessary context visible when they return to work with a quick note in the code.

Here are how some participants suspend their working state:

Write a short note of the next one or two steps to do. Save my work and leave one or more windows open at the tasks to be done.

Close out unnecessary applications and browser tabs. Make sure everything is saved. Lock my computer.

I use comments and TODO's in my code, coupled with a precise reminder of where I am in my notebook.

Several participants expressed frustration about the ineffectiveness of note taking:

I take notes on random scraps of paper. Sometimes I refer to them again, but often they migrate to odd corners of my office where they are never looked at again, except right before I throw them away the next time we

change offices. ... When I don't throw the notes away I invariably leave them at home and then I don't have them at the office the next day.

When returning to a programming task, notes, if available, and task description were commonly re-read to restore a high-level sense of the task. A common strategy was to navigate the source code to restore working state: participants mentioned how they attempted to work their way backward in time and task and formulate the next step before returning to the task. However, many times simply navigating was insufficient to restoring the working context; participants would often have to use tools to find the recent changes to code.

Review any notes left and then re-read the recent changes to the project. Not only does this get me back in the frame of mind to work on the project, I also find a good number of logic bugs.

I try to remind myself what it was that I was working on and continue where I left off. If I'm don't remember, I'll run the app and look at the code diff in order to see what state it was in before I left it.

The developers' ratings in Table 3 are notably consistent in what help they want when resuming tasks. 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). Based on these ratings, we designed one of our experimental conditions to show the content of a developer's activities.

### CONTROLLED STUDY

Based on the results of our preliminary survey, we ran a controlled study to test the effects of three different experimental conditions on developers' abilities to recover from task interruptions. Because of a prevalence of note taking for task management, we allowed subjects in all three conditions to take notes during task interruption and to review their notes during task resumption. Our base condition consists of note taking alone, representing current practice. In the other two conditions, the subjects supplemented their note taking by reviewing two different environmental cues that summarize their activities before interruption. The first of these cues is based on Mylyn, the most widely used research tool for developer task management; the second is based on developers' rating of task resumption features from our preliminary study. Our controlled study uses a within-subject protocol to allow subjects to compare the three conditions, as well as to mitigate differences due to subjects' varying programming and problem solving skills.

The two experimental cues feed from the same interaction log of developer activities and are shown in separate windows in the Microsoft Visual Studio 2008 development environment. The two cues, shown in Figure 1, differ only in

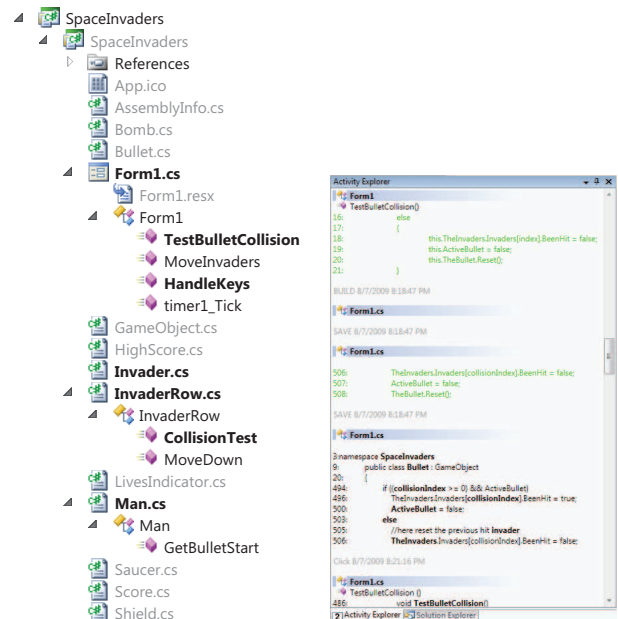


Figure 1. The experimental cues: DOI treeview (left) and content timeline (right).

how the developer's activities are presented. The first, called the *degree-of-interest (DOI) treeview*, consists of a treeview of names of the program's parts, namely, its projects, files within projects, types within files, members within types. In the style of Mylyn, these are filtered by a degree-of-interest model over recently visited or edited source code. The DOI model uses three levels: unvisited parts of the code are in gray, occasionally visited parts are in normal font, and often visited parts are in black boldface. The second cue, called the *content timeline*, shows a chronologically sorted list of the developer's activities (code selections, code edits, saves, and builds). A code selection is presented as the text that was selected (in black), a code insertion is shown with the new code in green, and a code change is shown as a pair of before and after text. In both cues, clicking on a program part causes the editor to navigate to the corresponding part.

The two cues differ in two dimensions. First, the DOI treeview presents program parts by name (project names, file names, type names, member names); the content timeline presents program parts by content (the program text). Second, the DOI treeview organizes the developer's activities by the code's structure; the content timeline organizes the activities by time. These dimensions are independent, so two other cues are possible, namely, content changes shown in a treeview and program names shown in a timeline. Testing two other conditions, however, would have made the experimental sessions too long and fatiguing. Hence we tested opposite corners of this two-by-two design quadrant in the hopes of maximizing the measured effects.

### Participants

Fifteen professional programmers (one female), average age of 39 years (range 31 to 56), participated in this study for receipt of 2 gratuitous copies of Microsoft software. The pro-

grammers were screened and then selected based on a series of profile questions. The profile questions were designed to recruit developers experienced with the development tools and environment used in the study (Visual Studio, C# and .NET, and GUI Software). In addition, we screened developers for experience with debugging and modifying the code of other developers and working in teams of at least 3 people.

## Methods and Procedure

Each subject was run individually in our user study laboratory. An experimenter was present for the entire session. The experimenter instructed the subjects to the goals of the study. Subjects were told that we were interested in how they would multi-task between several programming tasks and how they utilized different tools given to them. In addition, we stressed that we were not interested a perfect solution, but in completing as much of task as possible within a total of 20 minutes.

For our tasks, we used source code from three simple games (Tetris<sup>2</sup>, Pacman<sup>3</sup>, and Space Invaders<sup>4</sup>). The games are C# implementations of classic arcade games. Our motivation for using these games was based on the familiarity games offer and the nontrivial complexity of game source code. Further, we were assured from our previous user studies we conducted with Tetris and pilot studies that tasks involving these games were challenging yet feasible.

Each subject started with a warm-up period in which the experimenter demonstrated the DOI treeview cue and content timeline cue. Then, each subject ran a small application that we provided to coordinate the subject's activities over the three programming tasks, shown in Figure 2. For each task, the application automatically interrupted the subject several minutes after beginning the task, as described below. At the point of interruption, the subject would review the cue for that task (if any) and take notes, for up to one minute. The application would then start the subject on the next task. After all three tasks were interrupted, the application then asked the subject to resume each task, in turn. Upon resuming a task, the subject was given the opportunity to review that task's cue (if any) and his/her notes. The subject had up to 20 minutes to complete the task, with an additional 5 minutes, if the subject requested it. In short, the order of activities was this:

1. start Tetris;
2. get interrupted and review with condition 1;
3. start Pacman;
4. get interrupted and review with condition 2;
5. start Space Invaders;
6. get interrupted and review with condition 3;
7. finish Tetris, resuming with condition 1;
8. finish Packman, resuming with condition 2;
9. finish Space Invaders, resuming with condition 3;

<sup>2</sup> <http://www.codeproject.com/csharp/csgatetris.asp>

<sup>3</sup> <https://www.planet-source-code.com/vb/scripts/ShowCode.asp?txtCodeId=5669&lngWId=10#zip>

<sup>4</sup> <http://www.c-sharpcorner.com/UploadFile/mgold/SpaceInvaders06292005005618AM/SpaceInvaders.aspx>

Each subject was given the three conditions (no cue, DOI treeview cue, content timeline cue) exactly once. We counterbalanced the conditions across subjects to account for ordering effects. Each session ended with a questionnaire to rate the two cue and gather general feedback.

To provide use a consist and automated mechanism for interrupting the subjects, we used the following criteria: 15 seconds after a non-comment edit, after more than three non-comment edits, or after a twelve-minute timeout. We based these criteria on the factors Fogarty previously found to be associated with inopportune moments for interruption [9] and the nature of our tasks. The solutions for the programming tasks typically required changes to multiple parts of the source code. By setting the interruption point to be several moments after making a change, upon returning, the developer must ensure the change is completed correctly and then recall the next steps and relevant parts of code needed for the next change.

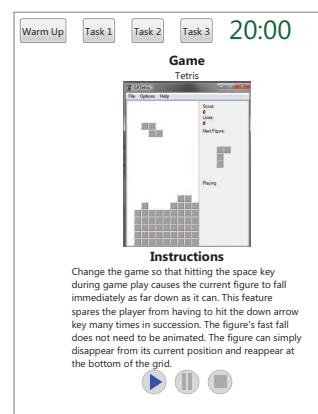


Figure 2. Application used for controlling interruptions and task switches in the experiment.

## Tasks

Each programming task was to add a small feature to each game:

1. Tetris: Change the game so that hitting the space key during game play causes the current figure to fall immediately as far down as it can. This feature spares the player from having to hit the down arrow key many times in succession. The figure's fast fall does not need to be animated. The figure can simply disappear from its current position and reappear at the bottom of the grid.
2. Pacman: In the existing game, if a power pellet is collected, the ghosts turn blue, slow down, and can be eaten for points. Change the game so that if a power pellet is collected, the ghosts instead freeze in place until the timer runs out.
3. Space Invaders: In the existing implementation, when the player fires a shot, it either hits an enemy or reaches the top of the screen. Change the game so that whenever the player fires a shot that misses an enemy, a new enemy is created. The new enemy is placed in the lowest row of an

existing enemy, filling in an empty position if possible. If all enemies positions are filled, no new enemy is created.

We used separate games, with separate code bases, for the tasks to keep the subject from polluting one task with the changes from another. For example, if the subject changed the Tetris code to the point where it would no longer compile, this did not prevent the subject from making progress on the next two tasks. The subject would have to fix the broken compilation after resuming the Tetris task. Using different code bases for all three tasks also ensured that there was no learning effect across tasks. Finally, the use of three unfamiliar code bases served to overload the subject’s memory, maximizing the detrimental effect of interruption.

### Dependent Measures

During the study, an error in the experiment tools prevented one subject from completing the tasks. In total, data from fourteen subjects was collected.

#### Task completion

Subjects completed tasks more frequently when one of the cues was available when resuming a task.

	Task Completion Rate			
	Task 1	Task 2	Task 3	Total
DOI treeview	3	2	2	7
content timeline	1	4	2	7
notes-only	1	2	1	4
Total	5	8	5	

**Table 4.** When developers had any form of activity history available, they were about twice as likely to complete a task than otherwise.

Breaking the data down by subject, 5 people did not complete any tasks, 2 completed all tasks, and 7 completed some tasks. Of the 7 people who completed some tasks, 5 of the incomplete tasks were in the notes-only condition, 2 were in the DOI treeview condition, and 2 were in the content timeline condition. (The low task completion rates in this study are consistent with a previous study using the Tetris task on the same code base [7, 6].)

#### Error Rates

To better understand why some subjects failed to complete the tasks in the allocated time, we categorize the type of error that lead to the failure:

1. Problem solving: The subject had all the pieces, he or she were stuck on the problem part.
2. Domain knowledge: The did not understand WinForms, event handler syntax, etc.
3. Relocation: The subject had difficulty relocating relevant parts of the code.
4. Programming: The subject had misunderstandings about the program or the changes he or she made.

Note that we kept the 20-minute deadline ”soft” to avoid failures due to arbitrary cut-offs. The subjects who did not succeed in a task were not close, even after 25 - 30 minutes. The

data in Table 5 categorizes the failures of the seven subjects completing at least one task.

	Problem Solving	Relocation	Errors	Total
DOI treeview	1	0	1	2
content timeline	2	0	0	2
notes-only	1	2	2	5
Total	4	2	3	

**Table 5.** When developers only had notes, they were more likely to make errors or to take longer to relocate necessary task context.

As one example of a programming error, one subject was interrupted in the middle of adding the event response for the space key in the Tetris task. When the subject later resumed the task, he completed the event handler, but forgot to assign the space key event to the handler in another part of the code. The subject moved on to the next part of the task, making the tetris block fall down, and quickly implemented the task specification much faster than other subjects. However, when the subject tested the solution, the missing association between the space key event and the handler prevented the new feature from working. The subject spent the remaining 10 minutes fiddling with the logic of the code, without realizing that the subject had simply forgotten to associate the handler with the space key event. This was the only task the subject failed.

As an example of a relocation error, a subject was interrupted in the Space Invaders task in the notes-only condition. The subject located the code for resetting the bullet. Having found this code, the subject needed to find out how to create a new invader, but the subject was interrupted during the search. The subject took notes for the information seeking task, but did not jot down the code location for resetting the bullet. When resuming the task, the subject eventually found the necessary information on how to create an invader after 4.5 minutes. The subject now needed to return to the previous location (code for resetting the bullet); however, the subject took another 5 minutes to relocate this code. The subject was frustrated that they knew what code they wanted, but could not find its location:

I remember finding that code, I just can’t remember where it is!

Unfortunately, by the time the subject had reached the code, the subject had used up most of their time and could not complete the task within the allocated time.

Experts could ward against these problems by proactively using markers, but might fail to do so when the necessary code is several subtasks away or when they did not anticipate being sidetracked on a subtask.

#### Lag Measures

To understand the cost of resuming tasks in the three conditions, we used two measures of lag:

1. Resumption lag: the time between clicking the ”play” button to restart a task and the first event initiated in the IDE.

2. Edit lag: the time between clicking the "play" button and the first code edit.

The resumption lags were quite similar in all conditions. The mean resumption lags (N=14) for each condition were: 20 sec, for the DOI treeview; 21 sec, for the content timeline; and 23 sec, for notes-only condition. These differences are not statistically significant by a one-tailed t-test. For edit lag, we took out the 5 subjects that did not complete any tasks because for at least one task, these subjects did even not make an edit after resuming their task. The edit lags (N=9) for each condition, in minutes and seconds, were: 2:30, for the DOI treeview; 3:26, for the content timeline; and 4:28, for notes-only. These differences are also not statistically significant by a one-tailed t-test.

### Note Taking

All subjects took some form of notes. We categorize note-taking into two categories: (1) situated: the notes are placed within the source code and (2) unsituated: the notes are written on paper or electronically within a notepad.exe window. Most of the subjects' notes were unsituated (75%).

The content of the notes would generally refer to either actions and objectives ("investigate where bullet goes off-screen") or specific code symbols (such as method names, or line number). During task suspension, subjects wrote down several types of information. Several subjects would re-summarize the task description into a simple one-line sentence ("Modify tetris so piece falls to bottom") or summarize what they had previously worked on. Most subjects tended to write notes about the most immediate subtask and would neglect to record other locations needed later for the task, that is they would tend to repeat the last location they were working on, but not where they might need to go next. Not surprisingly, 65% of the lines of unsituated notes would refer to code symbols, while only 14% of the lines of situated notes referred to code symbols. This suggests that when subjects write unsituated notes on paper, they need to spend several lines of text recreating the context in order to adequately record their note. The types of code symbols subjects wrote varied in the amount of code referred to: files (15 occurrences); methods (20); variables (9); code expressions (9); or line numbers (4).

### Subjective Ratings

After the tasks were over, we asked the subjects to rate, on a 5-point Likert scale, four aspects of the three conditions (the DOI treeview, the content timeline, and note taking): (1) how quickly it helped them resume their tasks (speed); (2) how well it reminds them of details from the interrupted task; (3) how useful they feel it would be after a week (durability); and (4) how well they liked it overall. Their average ratings are shown in Figure 3. Their ratings are notably consistent: in all four areas, they rated the content timeline higher than note taking and note taking higher than the DOI treeview. These differences are statistically significant by one-tailed t-test ( $p < 0.001$ ).

### User Feedback

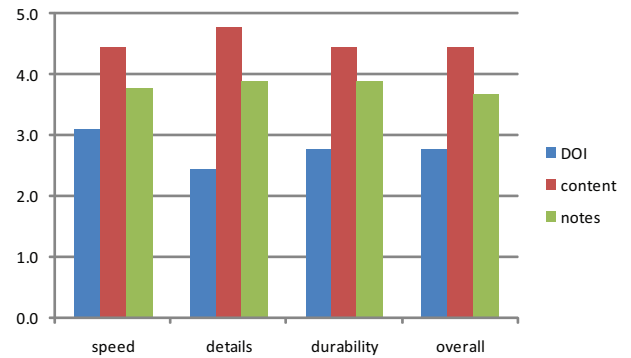


Figure 3. Subjects consistently preferred the content timeline over notes and notes over the DOI treeview.

When asked what they liked or disliked about the presentation of the cues, the subjects enjoyed the bolding of items in the treeview and the presentation of the code difference in the content timeline. Several subjects felt the DOI treeview was not granular enough and the method names was not as effective for trigger their memory about their activities. One subject said during the task, "[DOI treeview] does not show me where I just was. I would not remember this location if I had to come back in ten minutes". Another subject suggested the DOI treeview could be improved by distinguishing between items that were navigated to versus items that were edited.

Subjects felt the cues would offer interesting opportunities to complement their future note-taking habits. In our survey of developers, the developers said they placed temporary placeholders in the code that would be removed as soon as they returned to work. Most subjects enjoyed the synergy between the content timeline and TODO notes because it would be an automated place to view scattered notes in one place: "I would use comments in the code and the activities window - paper or notepad notes are not hooked to the code." They also mentioned wanting to directly enter or associate persisted TODO notes with items in the content timeline because many TODO personal comments are deleted right before checking in code. Subjects also talked about how they would change some of what they wrote down: "I wouldn't have to write down the location, just what needs to be done."

Subjects also had many general suggestions and ideas for interacting with the cues. Better support for filtering and different options for clustering the content (for example grouping changes by method or by file); quick gestures to select, save, and tag snippets of code that could be published to a personal repository and in the content timeline; and pin down or remove content from the content timeline.

Finally, subjects enjoyed using the content timeline during the feedback portion of the study to explain their thought processes and approach to the recently completed task. They were excited about the possibilities for using this to facilitate communication with other team members about their activities.

## Discussion

Subjects used the cues in different ways. For the DOI tree-view, subjects treated them like tabs in a multi-document interface. That is, when they could not remember which method contained the code they had in mind, they would use the names as information scent and click from one to another until they found the desired code. Many subjects with the content timeline would use it in two stages. They looked at the most recent entries for context, completing any work within their open document. Then, they would return to the activity timeline to rewind further back to review their previous actions before their last stopping point. From this review, subjects would often see the next place they wanted to return to.

Even though subjects had comparable performance with both the DOI treeview and content timeline, subjects rated the DOI timeline much lower. One explanation is the code was new to them, and therefore they were less familiar with the symbol names with which the DOI treeview refers to the code; they preferred the content timeline because it presents the code directly as they saw it in the editor. If the subjects had been previously familiar with these code bases, their subjective ratings might be different.

Although subjects did use the cues when resuming tasks, they did not use them much when preparing to suspend tasks. Instead, subjects focused on reaching a good stopping point or leaving a quick note. Even with cues in place to suggesting that programmers try minimizing the context they will actively suspend to the most immediate breakpoint.

## Threats to Validity

We did not have the opportunity to evaluate subjects younger than 31 years old. One possibility is that our experiment is biased toward older developers who may have different preferences for memory cues. Prospective memory performance is known to decrease with age if not counter-balanced by effective strategies for remembering. Regardless, our population represents a large and important portion of the active developer workforce.

We believe that the minute to prepare for interruption that we gave each subject makes our study unrepresentative of interruptions without warning but adequately representative of interruptions with warnings and self-interruptions, both of which frequently occur among professional developers [14]. We included this minute to prepare, despite the threat, because previous psychology studies show that having a period for studying an environmental cue for during interruption makes the cue for valuable during resumption [1]. Our focus in this study was to measure differences in the cues.

Finally, an additional threat to external validity is the nature of programming tasks we selected. Programmers encountering new code may not be able to as effectively utilize memory cues such as the DOI treeview because they are not as familiar with the specific names within the program and they have less opportunity to explore. However, our tasks are representative of many maintenance and bug fix tasks where

developers need to understand someone else's code enough to make a corrective change.

## CONCLUSIONS

We have presented an investigative study that provides new insight into how programmers currently recover from interruptions and a controlled study that evaluates how cues could assist resumption of programming tasks. The results provide a strong motivation for the need for tool support for helping developers resume tasks. Today, note taking is the most common way for developers to cope with task suspension and resumption, yet subjects in the note-taking condition had half the success rate as those using the automatic task resumption cues.

The two cues performed equally well, but subjects preferred the content timeline over the DOI treeview. This has important implications for knowledge workers beyond developers. The DOI treeview works well for source code, where the documents contain a hierarchy of named parts, but may not work as well for unstructured documents. The content timeline, on the other hand, does not take advantage of the code structure and would work equally well for unstructured documents. For example, office workers could use a content timeline to see changes to a spreadsheet by showing formula changes and cell changes in the timeline. Hence, using the content timeline for task resumption may be as effect for general knowledge workers as it is for developers.

Presenting information in an episodic fashion can also be useful for communicating work to others or for providing context when handing off a task from one person to another. For example, Mylyn already allows developers to store DOI trees with work items in bug database, as a form of guidance for the next developer who takes on the work item. To provide a similar capability for the content timeline creates privacy concerns, however, since the timeline contains the history of work activities. A developer, for example, may not want a coworker to see his "mistakes" when viewing his timeline. Yet, there is a fine line between showing "mistake" and providing the rationale behind the code (e.g. showing alternate rejected designs). A persistent form of the content timeline would have to be designed with these privacy concerns in mind.

## REFERENCES

1. E. M. Altmann and J. G. Trafton. Task interruption: Resumption lag and the role of cues. In *Proceedings of the 26th annual conference of the Cognitive Science Society*, 2004.
2. V. Bellotti, B. Dalal, N. Good, P. Flynn, D. G. Bobrow, and N. Ducheneaut. 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*, pages 735–742, New York, NY, USA, 2004. ACM.
3. A. B. Brush, B. R. Meyers, D. S. Tan, and M. Czerwinski. Understanding memory triggers for task tracking. In *CHI '07: Proceedings of the SIGCHI conference on Human factors in computing systems*, pages 947–950, New York, NY, USA, 2007. ACM.
4. E. Cutrell, M. Czerwinski, and E. Horvitz. Notification, disruption and memory: Effects of messaging interruptions on memory and performance. 2001.

5. M. Czerwinski, E. Horvitz, and S. Wilhite. A diary study of task switching and interruptions. In *CHI '04: Proceedings of the SIGCHI conference on Human factors in computing systems*, pages 175–182, New York, NY, USA, 2004. ACM Press.
6. R. DeLine, M. Czerwinski, B. Meyers, G. Venolia, S. Drucker, and G. Robertson. Code thumbnails: Using spatial memory to navigate source code. In *VLHCC '06: Proceedings of the Visual Languages and Human-Centric Computing*, pages 11–18, Washington, DC, USA, 2006. IEEE Computer Society.
7. R. DeLine, M. Czerwinski, and G. Robertson. Easing program comprehension by sharing navigation data. In *VLHCC '05: Proceedings of the 2005 IEEE Symposium on Visual Languages and Human-Centric Computing*, pages 241–248, Washington, DC, USA, 2005. IEEE Computer Society.
8. R. DeLine, A. Khella, M. Czerwinski, and G. Robertson. Towards understanding programs through wear-based filtering. In *SoftVis '05: Proceedings of the 2005 ACM symposium on Software visualization*, pages 183–192, New York, NY, USA, 2005. ACM.
9. J. Fogarty, A. J. Ko, H. H. Aung, E. Golden, K. P. Tang, and S. E. Hudson. 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*, pages 331–340, New York, NY, USA, 2005. ACM.
10. T. Gillie and D. Broadbent. What makes interruptions disruptive? a study of length, similarity, and complexity. *Psychological Research*, 50:243–250, 1998.
11. H. M. Hodgetts and D. M. Jones. Contextual cues aid recovery from interruption: The role of associative activation. *Journal of Experimental Psychology: Learning, Memory, and Cognition*, 32(5):1120–1132, 2006.
12. B. M. Kay and C. Watters. Exploring multi-session web tasks. In *CHI '08: Proceeding of the twenty-sixth annual SIGCHI conference on Human factors in computing systems*, pages 1187–1196, New York, NY, USA, 2008. ACM.
13. M. Kersten and G. C. Murphy. Using task context to improve programmer productivity. In *SIGSOFT '06/FSE-14: Proceedings of the 14th ACM SIGSOFT international symposium on Foundations of software engineering*, pages 1–11, New York, NY, USA, 2006. ACM.
14. A. J. Ko, R. DeLine, and G. Venolia. Information needs in collocated software development teams. In *ICSE '07: Proceedings of the 29th international conference on Software Engineering*, pages 344–353, Washington, DC, USA, 2007. IEEE Computer Society.
15. T. C. Lethbridge, S. E. Sim, and J. Singer. Studying software engineers: Data collection techniques for software field studies. *Empirical Software Engineering*, 10(3):311–341, July 2005.
16. G. Mark, V. M. Gonzalez, and J. Harris. No task left behind? Examining the nature of fragmented work. In *CHI '05: Proceedings of the SIGCHI conference on Human factors in computing systems*, pages 321–330, New York, NY, USA, 2005. ACM Press.
17. C. A. Monk. The effect of frequent versus infrequent interruptions on primary task resumption. In *Proceedings of the Human Factors and Ergonomics Society 48th Annual Meeting*, 2004.
18. B. O'Conaill and D. Fröhlich. Timespace in the workplace: dealing with interruptions. In *CHI '95: Conference companion on Human factors in computing systems*, pages 262–263, New York, NY, USA, 1995. ACM Press.
19. M. Offner. *Mental Fatigue*. Warwick & York, 1911.
20. G. Oleksik, M. L. Wilson, C. Tashman, E. Mendes Rodrigues, G. Kazai, G. Smyth, N. Milic-Frayling, and R. Jones. Lightweight tagging expands information and activity management practices. In *CHI '09: Proceedings of the 27th international conference on Human factors in computing systems*, pages 279–288, New York, NY, USA, 2009. ACM.
21. N. Oliver, M. Czerwinski, G. Smith, and K. Roomp. Relaltab: assisting users in switching windows. In *IUI '08: Proceedings of the 13th international conference on Intelligent user interfaces*, pages 385–388, New York, NY, USA, 2008. ACM.
22. C. Parnin and C. Görg. Building usage contexts during program comprehension. In *ICPC '06: Proceedings of the 14th IEEE International Conference on Program Comprehension*, pages 13–22, 2006.
23. C. Parnin and S. Rugaber. Resumption strategies for interrupted programming tasks. In *ICPC '09: Proceedings of the 17th IEEE International Conference on Program Comprehension*, Washington, DC, USA, 2009. IEEE Computer Society.
24. G. S. Patrick, P. Baudisch, G. Robertson, M. Czerwinski, B. Meyers, D. Robbins, and D. Andrews. Groupbar: The taskbar evolved. In *Proceedings of OZCHI 2003*, pages 34–43, 2003.
25. T. Rattenbury and J. Canny. Caad: an automatic task support system. In *CHI '07: Proceedings of the SIGCHI conference on Human factors in computing systems*, pages 687–696, New York, NY, USA, 2007. ACM.
26. G. Robertson, E. Horvitz, M. Czerwinski, P. Baudisch, D. R. Hutchings, B. Meyers, D. Robbins, and G. Smith. Scalable fabric: flexible task management. In *AVI '04: Proceedings of the working conference on Advanced visual interfaces*, pages 85–89, New York, NY, USA, 2004. ACM.
27. J. Singer, R. Elves, and M.-A. Storey. Navtracks: Supporting navigation in software maintenance. In *ICSM '05: Proceedings of the 21st IEEE International Conference on Software Maintenance*, pages 325–334, Washington, DC, USA, 2005. IEEE Computer Society.
28. M.-A. Storey, J. Ryall, R. I. Bull, D. Myers, and J. Singer. Todo or to bug: exploring how task annotations play a role in the work practices of software developers. In *ICSE '08: Proceedings of the 30th international conference on Software engineering*, pages 251–260, New York, NY, USA, 2008. ACM.
29. C. Tashman. Windowscape: a task oriented window manager. In *UIST '06: Proceedings of the 19th annual ACM symposium on User interface software and technology*, pages 77–80, New York, NY, USA, 2006. ACM.
30. J. G. Trafton, E. M. Altmann, and D. P. Brock. Huh, what was i doing? how people use environmental cues after an interruption. In *Proceedings of the Human Factors and Ergonomics Society 49th Annual Meeting*, 2005.
31. C. Treude and M.-A. Storey. How tagging helps bridge the gap between social and technical aspects in software development. In *ICSE '09: Proceedings of the 2009 IEEE 31st International Conference on Software Engineering*, pages 12–22, Washington, DC, USA, 2009. IEEE Computer Society.
32. R. van Solingen, E. Berghout, and F. van Latum. Interrupts: Just a minute never is. *IEEE Software*, 15(5):97–103, 1998.