SUPPORTING INTERRUPTED PROGRAMMING TASKS WITH MEMORY-BASED AIDS

A Thesis
Presented to
The Academic Faculty

by

Christopher Joseph Parnin

In Partial Fulfillment
of the Requirements for the Degree
Doctor of Philosophy in the
School of Computer Science

Georgia Institute of Technology
December 2014

SUPPORTING INTERRUPTED PROGRAMMING TASKS WITH MEMORY-BASED AIDS

Approved by:

Dr. Alessandro Orso, Committee Chair
School of Computer Science
Georgia Institute of Technology

Dr. Spencer Rugaber, Advisor
School of Computer Science
Georgia Institute of Technology

Dr. Mark Guzdial
School of Interactive Computing
Georgia Institute of Technology

Dr. Rosa Arriaga
School of Interactive Computing
Georgia Institute of Technology

Dr. Eileen Kraemer
School of Computing
Clemson University

Date Approved: November 14th, 2014

To Amy, Lana, and Miles,

the foundation of my life.

PREFACE

This dissertation is original, independent work by the author, C. Parnin.

ACKNOWLEDGEMENTS

I would like to thank the many people in my life who have supported me during my dissertation journey.

Contents

List of Tables

List of Figures

SUMMARY

Despite its vast capacity and associative powers, the human brain does not deal well with interruptions. Particularly in situations where information density is high, such as during a programming task, recovering from an interruption requires extensive time and effort. Although researchers recognize this problem, no programming tool takes into account the brain's structure and limitations in its design. In this dissertation, I describe my research collecting evidence about the impact of interruptions on programmers, understanding how programmers manage them in practice, and designing tools that can support interrupted programmers. I present a conceptual framework for understanding human memory organization and its strengths and weaknesses, particularly with respect to dealing with work interruptions. The framework explains empirical results obtained from experiments in which programmers were interrupted. For researchers, the intent is to use the framework to design development tools capable of compensating for human memory limitations. For developers, the insights and strategies from the framework should allow reflection on our own programming habits and work practices and how they may be tailored to better fit our human brain. The framework is evaluated by conducting two experiments that find that 1) developers can recall nearly twice as many past programming events using a tool designed with the framework over traditional tools, with comparable recall effort and 2) developers can remember to perform nearly twice as many prospective actions using a tool designed with the framework over traditional tools, with limited impact to cognitive load.

Chapter I

INTRODUCTION

Bob is a self-described multi-tasker. He works on multiple projects and teams, new initiatives, and has multiple responsibilities. One of his recurring tasks is to perform code merges with a team of 80 developers located in different time zones. Every two weeks, Bob must integrate and reconcile the conflicts between changes made by his local team and the external team, a process typically consuming 2-3 days.

During this process, Bob must coordinate with other developers using phone and email to help resolve the conflicts when integrating code. Unfortunately, this is further complicated by that fact that the integrated code cannot be tested until after the merge is done, because the code is often not compilable until then. Meanwhile, developers from other projects have bug fixes that cannot wait three days for Bob's attention, so Bob is required to frequently switch context into another codebase and perform complex debugging activities there. Keep in mind, not only is Bob affected by the code merge, but often his whole team remains idle until the updated and merged code has stabilized.

In summary, Bob must frequently switch between large contexts, for each of which he must maintain status on multiple tasks, people and code items. Any mistake can result in long delays for other projects and team members.

1.1 Programmer, Interrupted

Interruption to creative work extinguishes a creator's flame and stretches effort beyond original expectations. Consider one example: the commission of the tomb of Pope Julius II [48]. Initiated in 1505 by Michelangelo, work on the elaborate sculpture did not complete until 40 years later in 1545. Initially mired in financial problems, work stopped just after materials were procured and an initial design produced. Next, politics interfered when rival artists convinced the Pope to delay construction of his tomb until after his death. Accordingly, Michelangelo spent the next 35 years working intermittently on the tomb continually interrupted with other tasks and projects to the point where he never obtained fully uninterrupted focus on the project. Meanwhile, he completed many comparably sized works of art in the span of months or years, including painting the ceiling of the Sistine Chapel (a task other painters thought impossible) in under four years. Finally, nearing old age Michelangelo finishes the tomb, now a diminished shadow of the original design, leaving the artist deeply unsatisfied.

Many a software developer can find sympathy for Michelangelo's plight but nod with agreement that they have a more difficult fight. Software developers, unlike sculptors, cannot easily inspect the entirety of their unfinished work. There are no chisel marks and gouges in stone, nor a physical space that can be toured to remind themselves of their status. Rather, developers must refresh their previous knowledge and experiences about the incomplete programming task largely from memory.

Although a software program is mostly text, it is quite unlike resuming the reading of a novel. With a novel, text flows in one direction and often a reader can resume reading by simply rescanning the last few completed paragraphs. With a software program, ideas connect in all directions—the relevant information exists across various artifacts and locations and must be actively sought for. To read the text, programmers must engage in a process of program understanding, a suite of cognitive processes [28, 156, 109], which is considered to be one of the most time-consuming and expensive aspects of software

development [41].

Thus, an interruption to a programming task can be detrimental to productivity [203, 103, 151], as the programmer may need to act out these difficult cognitive processes again, in order to even understand the program enough to restart the programming task. But the abstractions and representations built during program understanding are not the only mental state at stake. Developers must also maintain their memories of the problem-solving state (data, possible solutions, calculations) and for the task itself (plans, intentions, artifacts, issues). Additionally, because a program is always under construction, programmers must continually refresh their mental blueprints of it (design, components, and run-time behavior) due to updates from other team members or external partners. These fragments of mental state specify a set of information needs that are difficult and costly to reacquire.

The impact of interruptions on software development is staggering. Solingen [203] 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 each interruption. Long-term interruptions are also common: Ko et al. [103] observed 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. In a study of interruptions, Parnin and Rugaber [151] analyzed interaction logs of 10,000 programming sessions from 86 programmers and found that in a typical day, developers rarely program in long continuous sessions. Instead, a developer's day is fragmented into many short sessions (15-30 minutes) with an additional one or two longer sessions (1-2 hours). Further, at the start of each of the longer sessions, a programmer often spends a significant amount of time (15-30 minutes) rebuilding working context before resuming coding. Interruption is a wide-spread and costly problem faced by programmers.

Looking behind the curtain, the root problem with interruption is the set of memory

failures that arise from sudden shifts in attention [88, 6] over long periods of time [1, 103]. In general, human memory is limited—for every one thing remembered, hundreds more are forgotten—the human mind cannot simply store every moment or fact. When a programmer recovers from an interruption, she must rely on incomplete fragments of memory to help restore working state. The programmer may have several information needs about the interrupted programming task and ask questions such as "What was I doing?", "Where was the code I was working with?", or "How does this code work, again?". To answer these questions, the programmer can use ad-hoc exploration of source code in hope of jogging her memory or refer to notes that were quickly scrawled down at the time of interruption. But these approaches are neither completely restorative nor take advantage of previously invested effort. Without improved support, programmers experience great difficulty in recovering the information they need at the necessary level of detail and completeness [148].

A potential solution is to use an external memory store that can provide access to the experiences of the interrupted programming task. Although the mind cannot store every moment or fact encountered, a computer can. Every click, error message, search result, exception, and edit can be stored in a history of events, files, or screenshots. Reviewing a complete replay of the interrupted task could, in principal, enable a programmer to recall and restore vital information. Unfortunately, watching a replay of a two hour programming task at normal speed would take another two hours. Without an effective way to index or summarize the history of the interrupted task, the raw history itself would remain overwhelming and near useless. The primary research challenge is to identify how memory failures about programming tasks can inform how a history of programming efforts can be encoded, consolidated, and recalled in a useful manner.

1.2 Research Overview

To inform how a stored history of programming efforts could assist programmers with interruption recovery, I first reviewed the literature of human memory limitations [147] from both psychological and cognitive neuroscience perspectives to understand the effects on memory after an interruption (see Appendix A). I then contextualized these memory failures in terms of programming tasks, by investigating what information developers needed to recover after task interruptions. I did this through a series of empirical, qualitative, and experimental studies ([151, 152, 148]) (see Chapter 4). From these insights, I devised a set of tool requirements for analyzing, organizing, and presenting summarized programming activities.

I then developed a suite of interruption recovery aids, called memlets, that can be used to address different types of memory deficiencies that may have occurred after an interruption of a programming task. Formally, a *memlet* is a conceptual model and accompanying visualization that supports a particular cognitive operation needed by a programmer and is available in the programming environment. Each memlet can be constructed preemptively prior to an interruption (to suspend information) or built automatically (to recover information) after an interruption. In studies of programmers, I found that the ability to both preemptively and reactively deal with interruptions was an important strategy used by the programmers.

In the next subsections, I provide more details on my conceptual framework, describe the derived memlets, provide an illustrated scenario of a developer using memlets to recover from an interruption, provide details on my implementation of memlets, and describe my evaluation of two memlets.

1.2.1 Conceptual Framework

Based on a series of empirical studies and reviews of cognitive neuroscience literature, I derive a set of information needs for recovering from interrupted programming activities.

My conceptual framework structures the information needs in terms of five memory types particularly relevant to programmers: attentive, prospective, associative, episodic, and conceptual.

Attentive memory holds conscious memories that can be freely attended to. Attentive memory is easily disrupted by short-term interruptions. *Prospective memory* holds reminders to perform future actions in specific circumstances (e.g., to buy milk on the way home from work). Prospective memory failures often involve forgetting to perform an action in a timely manner. *Associative memory* holds a set of non-conscious links between manifestations of co-occurring stimuli. Associative memory failures often occur when recalling information from computer interfaces with insufficient information cues. *Episodic memory* holds the recollection of past events. Episodic memory failures often involve forgetting contextual details about events and omission of events. *Conceptual memory* holds representations in a continuum of perceptions and abstractions. Conceptual memory is often disrupted by long-term interruptions, which in turn requires that conceptual memory be restored and primed before becoming useful.

My conceptual framework can be used to explain the information seeking and preserving behavior of interrupted programmers who are attempting to recover from a memory failure. The conceptual framework is then used to guide the development of several memlets.

1.2.2 Memlets

Memlets help developers recover from a memory failure experienced after an interruption. Memlets target a specific programming activity and memory type.

Based on my conceptual framework, the following memlets were derived: **TOUCH POINTS**, **SMART REMINDERS**, **CODE NARRATIVES**, **ASSOCIATIVE LINKS**, and **SKETCHLETS**. **TOUCH POINTS** support attentive memory by allowing a programmer to focus on programming elements in the environment at a much longer duration and number than she

would be able to do if relying solely on her own memory. **TOUCH POINTS** are designed for programming tasks that involve refactoring, which may involve having to attend to many locations in code to apply a change. **SMART REMINDERS** support prospective memory by allowing a programmer to designate reminders that trigger based on an external conditions. **SMART REMINDERS** are designed for situations such as blocked programming tasks, which may delay when a programmer can resume a task. **CODE NARRATIVES** support episodic memory by allowing a programmer to review a narrative of the events and experiences related to their programming task. **CODE NARRATIVES** are designed for programming tasks that involve recall of past events, such as code reviews. **ASSOCIATIVE LINKS** support associative memory by providing spatial, temporal, and activity-related links between various programming elements. **ASSOCIATIVE LINKS** are designed for programming tasks that involve code navigation, in which developers frequently experience disorientation. **SKETCHLETS** support conceptual memory by allowing the representation and naming of new concepts derived from existing programming elements. **SKETCHLETS** are designed for programming tasks that involve learning new concepts or restoring concepts that need to be need to be refreshed.

1.2.3 Example Workflow

To understand how a developer could use the approach to support resuming an interrupted task, I provide an illustrated scenario of a developer using my approach to resume an interrupted programming task, shown in Figure 1.

> *Ana has just received renewed interest on a project that was previously canceled. The problem is that it has been over a month since she last worked on the project. With only limited time to complete the project, Ana finds it difficult to jump back into the project and start working again. Details of the code have slipped her mind and strategies, such as browsing through the code, are failing to get her in the right state of mind. Luckily, with* GANJI*, an implementation*

of my approach, Ana can use **CODE NARRATIVES** *to be reminded of key events that occurred during her last programming session. One event, receiving a troublesome run-time exception, reminds Ana of a roadblock that she could not overcome but had since commented out. From the detailed code history, Ana can see what code was throwing the exception, and a detailed stack trace.*

While Ana continues her work, she notices another event describing the code provenance *(the origin of the code). She now remembers that she had copied that code from a blog post online. The memlet displays the original blog, allowing Ana to revisit the blog post. Reading through the blog comments, Ana notices that one reader has posted a solution to the same exception she was experiencing.*

Now, finally able to move forward, Ana wants to see what she was doing before she got stuck with the exception. Using **TOUCH POINTS**, *Ana sees the status of the code she visited and edited before. From this status, she recalls several incomplete changes and returns to the code locations to finish the work. Having the information necessary to resume her work, Ana, in a flurry of coding, is able to complete the project and seal the deal with her client.*

In summary, GANJI allowed Ana to recall important events, fill in important details that she had forgotten, and restore important artifacts and resources necessary for completing her project.

1.2.4 Implementation

To implement the memlets, an infrastructure was developed to collect a comprehensive history of programming effort, called the *code history*, which requires that the software development work performed in the programming environment is collected and stored within a database of events and a repository of artifact snapshots. Further, various history analysis algorithms were developed to reason over the code history and construct the

data model for each memlet. Finally, several visualizations and services were developed in support of the memlets.

1.2.5 Evaluation

CODE NARRATIVES and **SMART REMINDERS** were chosen for evaluation. In deciding which claims to focus the evaluation on, the frequency of a particular task and its impact of interruption were considered. Interruptions to different types of programming tasks tax different types of memory. My conceptual framework suggests that interruption to attentive and associative memories are short-term and more isolatable whereas prospective and episodic failures are long-term and more difficult to isolate.

Prospective and episodic failures are likely to have long-term impact and disruption. In the case of prospective memory failures during blocked tasks, the delay in forgetting to perform an action can be on the order of days and impact other teammates downstream. For example, *prompting failures* occur when visual cues for reminders are not perceived and *monitoring failures* occur when applicable reminders are not tracked and acted upon. In the case of episodic memory failures, for example when recalling code in preparation of a task hand-off, the wide variety of information types and the temporal ordering of steps can be difficult to recall unaided. For example, *source failures* occur when contextual information related to a event is forgotten and *recollection failures* occur when events are omitted or the sequence of events is recalled out of order.

My evaluation investigates the following claims:

1. **SMART REMINDERS** reduce monitoring failure by introducing mechanisms for conditional triggers and reduce prompting failure by introducing mechanisms for enhancing awareness of reminders.

2. **CODE NARRATIVES** reduce source failures by maintaining and presenting contextual details of programming events and reduce recollection failure by maintaining and presenting the ordering and fidelity of programming events.

1.3 Dissertation Outline

In subsequent chapters of the dissertation, I describe how a comprehensive history of programming effort, called the *code history*, can help developers recover from interrupted programming tasks. The developers can index into this history through visualizations called *memlets* that aid the developers in recalling and restoring knowledge. Further, the memlets are designed so that developers can recover from particular kinds of memory failures commonly associated with programming tasks. The organization of the chapters are as follows:

In the background chapter, I first give a general review of the literature on interruptions from workplace and psychology studies that highlight the various effects of interruption. This is then followed by a general overview of terminology and research in programming environments and software development practices. I then summarize the findings from my review of human memory, which explain why some types of memory are affected differently by interruption and thus need different types of support.

In the remaining chapters, I describe my research approach, exploratory research, conceptual framework, derived memlets, infrastructure and implementation needed to record and summarize development work, and the evaluation performed. In the end, I provide appendices with my review on human memory, and research on developer resumption behaviors.

Finally, throughout this work, I will use examples and stories drawn from real-world experiences with programmers to give flesh and context to the problems that arise from interruption.

1.4 Thesis statement

My research argues the following thesis statement: *Memlets, conceptual models and visualizations that support limitations in programmers' memories, reduce the negative effects of interrupted programming tasks by overcoming memory failures experienced.*.

My thesis is supported by the following findings:

1. Developers can recall nearly twice as many events using **CODE NARRATIVES** over traditional tools, with comparable recall effort.

2. Developers can perform nearly twice as many prospective actions using **SMART REMINDERS** over a TODO note, with limited impact to cognitive load.

1.5 Research contributions

My research contributes the following to the software engineering community:

1. A review of human memory research in the context of programming.

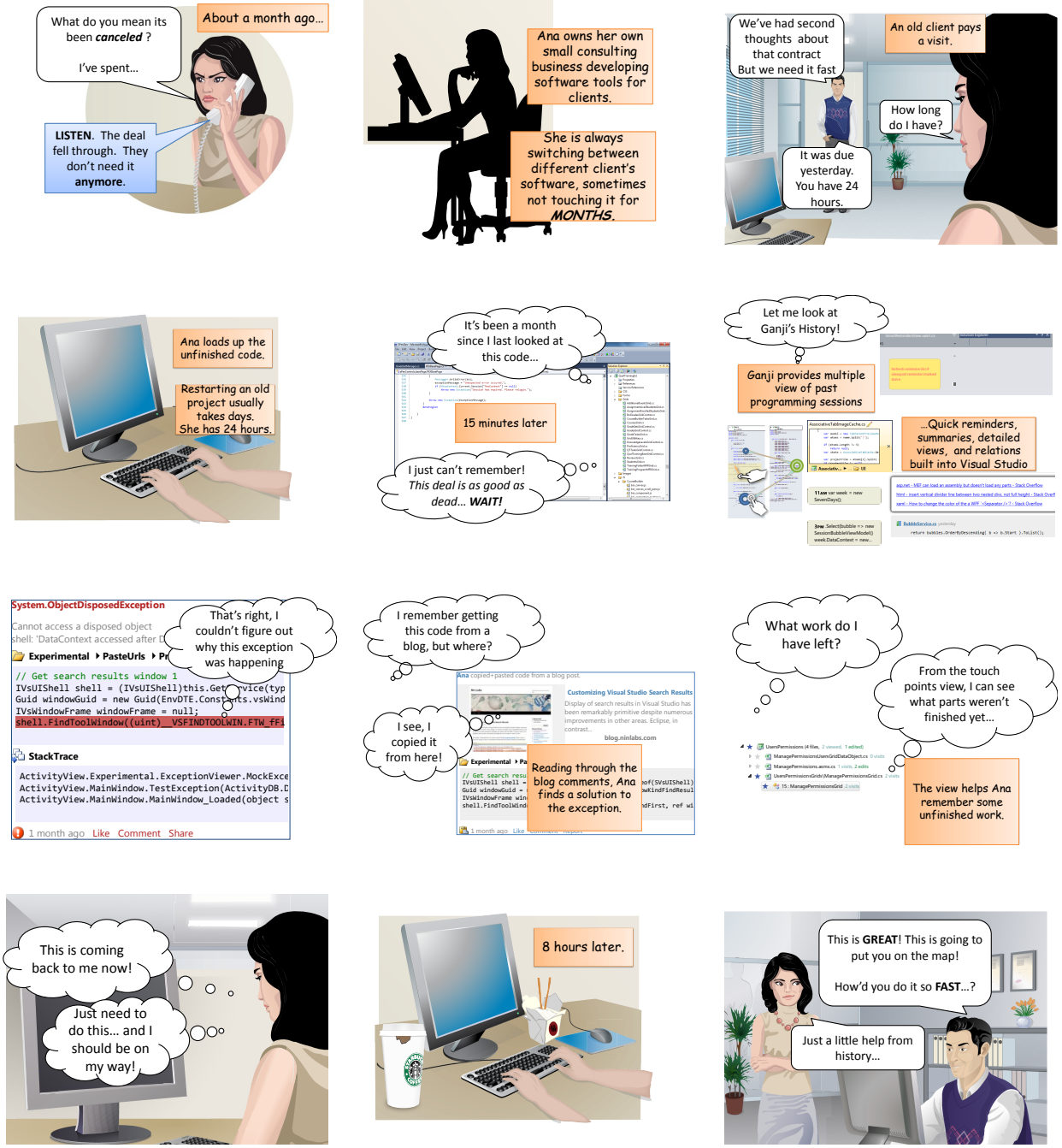
2. A systematic study of how programmers currently manage interruptions.

3. A conceptual model for supporting recovery of programming information.

4. An infrastructure for obtaining code history and characterizing code activity.

5. A series of empirical evaluations that explore the benefits and disadvantages of code history toward interruption management.

Figure 1: An illustration describing how Ana uses GANJI.



Chapter II

BACKGROUND: INTERRUPTIONS, MEMORY, AND PROGRAMMING ACTIVITIES AND PROGRAMMING ENVIRONMENTS

Ana *runs a home business requiring infrequent but important software development efforts. Ana is one of the estimated 13 million end-user programmers [173] (compared with 3 million software developers) who customize his or her own software, but are not professionally trained and rarely have time to devote full attention to it. Ana must often make changes at unexpected times in response to client demands or to develop new features. On average, Ana makes changes to her product every two weeks. Unfortunately, the long delays between programming sessions means that Ana often wastes a large part of her day just trying to refamiliarize herself with her code and remember the effects of recent changes.*

Ana represents the many programmers who resume programming tasks after periods of inattention. Exacerbating these interruptions is the burden of having low conceptual knowledge of newly learned tools and features of APIs. In sum, task resumption represents a large overhead in development effort for infrequent programming tasks.

2.1 Interruptions

2.1.1 Concepts

What does it mean to be *interrupted*? I adopt the definition from Merriam-Webster [121]:

> ***interrupt:*** *(verb) to break the uniformity or continuity of work.*

When work is interrupted, what exactly is broken and what are the resulting fragments called? Some definitions: Work is often performed as a hierarchy of tasks. By *task*, I mean a unit of work assigned to a developer, which may be refined into further subtasks. An *interruption* alters the course of work on a task so that it must be performed in many discrete chunks, called *sessions*, separated by periods of time called *breaks*. The length of a break can be a few minutes or a few weeks. I call the moment the developer returns to work the *resumption point*. Of course, work and interruptions are not limited to a single task; during a break in a task, a developer may be working on other tasks that may also become interrupted.

When the performance of tasks becomes interleaved, we often use the term *multitasking*. Still, the term *multitasking* can be ambiguous and confounded with other terms such as *task-switching* and *interruption*. To clarify the matter, I use the continuum of multitasking behaviors as defined by Salvucci and colleagues [171]: multitasking spans from *concurrent multitasking* (driving in car while talking on a phone) to *sequential multitasking* (alternating between writing a paper and cooking). In this dissertation, I usually mean multitasking in the sense of sequential multitasking—developers are not typically driving while programming. And, to be clear, it follows from the definition of interruption that any switch between tasks are also considered to be an interruption.

Are there ways to classify interruptions? Miyata and Norman [128] describe interruptions as either internal or external. *Internal* interruptions are self-initiated breaks of work for reasons including fatigue, desire for reflection, road blocks, or reaching a stopping point. In contrast, *external* interruptions are generated from external sources

such as computer notifications, inquiring colleagues, emergency bug fixes, or scheduled events. With external interruptions, there are several strategies for negotiating the moment of interruption, referred to as the *breakpoint*. McFarlane has evaluated and compared different strategies for selecting the moment of time that a breakpoint occurs: immediate response, scheduled response with a defined time limit, negotiated response (interruptee decides), and mediated response (third party decides if busy person looks interruptible) [117]. Negotiated breakpoints are generally the best strategy for stopping work because the interruptee can choose an opportune moment.

Psychologists have devised a vocabulary for describing the components of interruptions and resumptions in more detail. Imagine that Bob is working on a task when suddenly the telephone rings. The onset of an interruption (the telephone ringing) is called the *interruption point*. Before a person fully addresses an interruption, there is a critical window of opportunity where a person can attempt to mitigate the disruption. Bob uses this time interval, called the *interruption lag*, to quickly jot down a note before picking up the ringing telephone. When Bob picks up the telephone (the breakpoint), he is now interrupted from his primary task and becomes engaged with a secondary task. After the telephone call, Bob wants to resume his primary task, but needs a few moments to recollect his thoughts. The first observable action related to the primary task occurs at the *resumption point*. The time interval separating the end of the secondary task and resumption point is called the *resumption lag*. In psychology studies, the length of the resumption lag is often used to measure the mental effort required to disengage from the secondary task and shift attention back to the primary task [5].

In Figure 2, I depict these terms with a timeline illustrating two tasks affected by an external and internal interruptions.

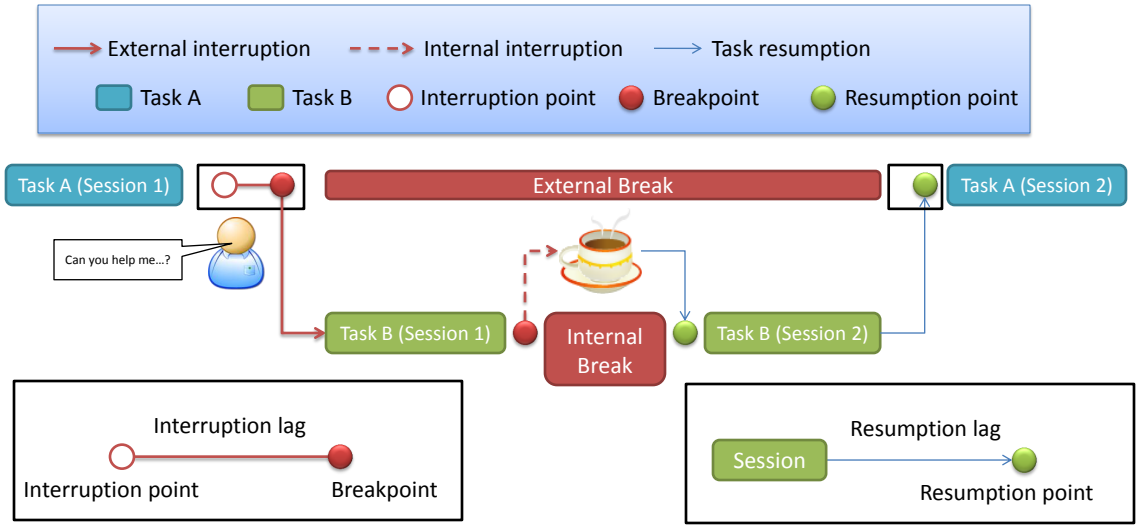


Figure 2: Bob is working on a task A. Another teammate interrupts this task with a question about another project (external interruption). This prompts Bob to begin working on task B. After some progress, Bob decides to take a coffee break (internal interruption). He returns and finishes up task B. Finally, Bob resumes task A.

2.1.2 Workplace Studies

How do interruptions affect tasks in the workplace? Despite efforts to manage interruptions, *in situ* studies suggest interruptions remain problematic. Czerwinski's study [46] showed that tasks resumed after an interruption were more difficult to perform and took twice as long to complete when compared to uninterrupted tasks. O'Conaill's study [140] found 40% of interrupted tasks are not resumed at all. Further research by Mark et al. [114] observed that 57% of tasks were interrupted. As a result, work on a task often was fragmented into many small work sessions. In a field study of 27 workers, Iqbal and Horvitz [90] found that workers had difficulty in recovering state from suspended tasks due to loss of context and low visibility of work artifacts, such as minimized or covered computer application windows.

Researchers have investigated what artifacts knowledge workers used when multitasking. Bellotti [20] studied the various media employed by 9 different managers and found that, in addition to using paper media, such as sticky notes, 50% of the tasks were

associated with emails. Brush [30] 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 single location: e.g., Notepad or ongoing email drafts. But all participants also complemented their tracking strategy with reminders to retrieve details needed for composing their reports. There is some 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 [46]. Other researchers have observed how programmers use task annotations [188] to record prospective task and proposed systems for capturing them [49, 188].

Unfortunately, knowledge workers, such as programmers, face a different set of problems than previously studied. For example, many of the tasks 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 likely be as successful for such tasks. Like knowledge-based tasks, programming tasks often involve long periods of investigation, but with several key differences: The programming documents are more structured and are usually more complex; the documents are actively edited by the programmer and other teammates; and much programming knowledge is tacit and only available from other experienced but busy co-workers.

2.1.3 Psychological Effects

What happens when a person is heavily involved in a mentally challenging task and is suddenly interrupted? What happens when a person has made great headway into a task and has reached a good stopping point but cannot resume it until a few days later? To answer such questions, we draw insights from psychological and cognitive studies.

Scientific inquiry of the psychological effects of interruptions did not begin until the early 20th century. In 1911, studies of school children and various configurations of class lesson schedules found that interruption caused by short lesson periods disrupted learning and introduced mental fatigue [32]. However, certain schedule configurations could invigorate students if breaks were appropriately spaced. Another study found that incomplete tasks could be recalled better than completed tasks [213, 120]. That is, when subjects were asked to recall tasks previously performed during the experiment, even after a few days, they remembered the tasks they did not complete. Participants, when left to their own devices, had a compulsive need to complete the tasks even when not instructed to do so. Only recently have we learned that *prospective* remembering (the act of remembering to perform an action in the future) has distinct brain circuitry dedicated to supporting it [165]. More modern research on interruptions has focused on other effects of interruption, such as task performance, especially in the context of mission critical, computing- or knowledge-intensive tasks. For example, McFarlane [117] characterizes the effects of interruptions and different alert mechanisms on tasks in environments such as an aircraft cockpit or operational planning for the military [117].

Much work on interruptions over the past century has been successful in observing broad outcomes but has had difficulty in explaining how interruption affects memory. One recurring issue dogging experimenters has been the difficulty in predicting which internal factors are responsible for the disruptive effect of interruptions. Consider what tools early psychologists had available: Armed with a mechanical stopwatch, a psychologist would peer over the shoulder of a participant and interrupt the participant based

on his or her perception of task engagement. Thus, it was uncommon to control for the timing and duration of an interruption. But even with the arrival of computerized task monitoring, inconsistent and conflicting findings are still common.

In 1989, Gillie and Broadbent made one of the first attempts to systematically measure the impacts of interruption length, task similarity, and task complexity on the performance of computer-related tasks [72]. Participants were presented with a list of shopping items and needed to navigate a virtual town to acquire the items. Over a course of four experiments, participants were interrupted with either a simple task of mental arithmetic, a similar task of memorizing words, or a complex task of mental arithmetic from an alphabetic representation of numbers: A short, simple, and dissimilar interruption (30 seconds of mental arithmetic) did not have a significant effect on the main task nor did extending the interruption length to nearly 3 minutes. However, free recall of another list of words (similar task) had a significant effect in increasing the main task completion time by a few seconds. Finally, a short, but complex interruption did not have significant effect on main task completion time but did have the secondary effects of increased navigation time.

In 2001, Cutrell and colleagues [44] studied the effect of instant messages on search tasks. In the experiment, participants scanned a list of book titles to locate a target. Some trials were interrupted with instant messages. Additionally, the search task was varied so that instead of finding an exact book title, participants were asked to find a book title matching a topic (requiring semantic processing). The study found that receiving notifications affected the search task, and even more so when the search task involved semantic processing. A possible explanation was that semantic processing of the title required more memory workload than perceptual matching and thus was more affected by the interruption. Examining interruption frequency, Monk [129] found frequent interruptions defeats concentration, but eventually had a net positive effect as subjects were forced to adopt more aggressive suspension strategies.

2.1.4 Task Structure Effects

By 2003, experiments began to vary the cognitive nature and structure of the task being interrupted. By studying tasks with more complex structures and more mental workload, researchers began to understand the critical relation between the moment of interruption and level of mental workload demanded by different parts of the tasks. Consider a simple task of listening to numbers being read aloud and then multiplying the previous number by the current number if the current number is evenly divisible. The structure of the task could be described as having two phases: the monitoring stage and the multiplication stage. The mental workload of the monitoring stage requires the person to hold the previous number and the last digit of the current number in mind. The mental workload of the multiplication stage requires the person to hold the previous number, current number, and intermediate state such as carried digits, current digit position, and the intermediate solution. Understandably, interruption to the multiplication stage of the task is more disruptive than those occurring during the monitoring stage. *Interruptibility* is a term describing the suitability of interrupting a task at a given moment.

To measure mental workload, psychologists use physiological indicators that are associated with states of increased mental workload. For example, the pupil of the eye expands with increased levels of mental workload (See Figure 3). If the pupil size is measured on a person performing the monitoring and multiplication task described above, then the pupil can be observed growing when transitioning to difficult parts of the task and shrinking when transitioning to easier parts of the task, reaching its maximum size during performance of the multiplication itself.

Experiments applying these physiological measures confirmed that interruptions were more disruptive when they occurred during more difficult or cognitively demanding parts of a task [88]. Conversely, a less disruptive interruption can be achieved when delaying the interruption until a moment of lower mental workload is reached. Unfortunately, physiological measures are problematic. For example, to measure pupil size, expensive

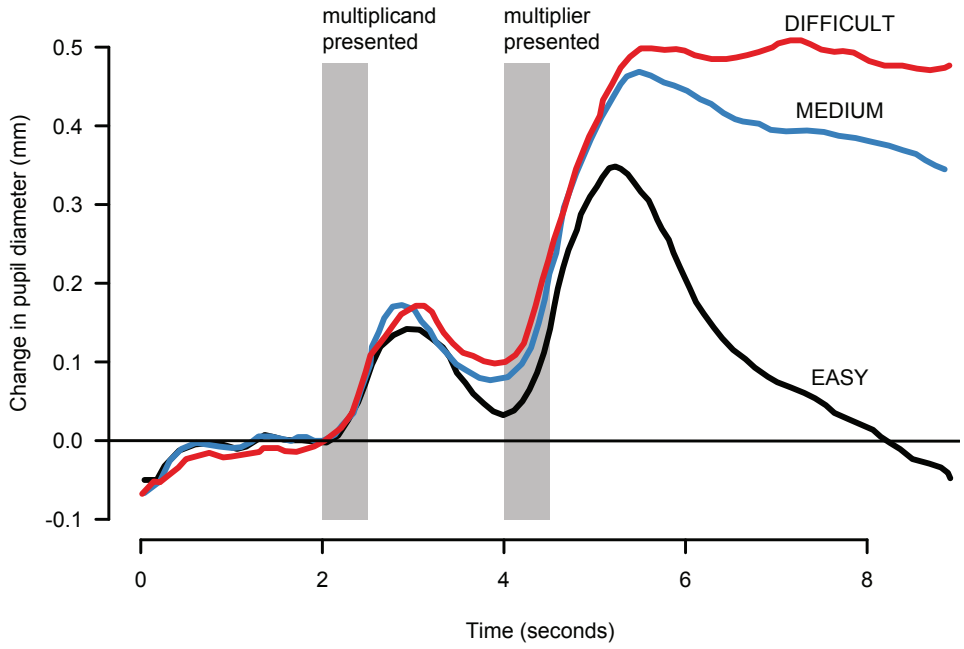


Figure 3: Pupil size over time during a easy, medium, and difficult multiplication task. Disruption and increased levels of anxiety accompany interruptions at high levels of mental workload. Figure from Klinger's experiment on vigilance [101].

eye-tracking hardware and rooms with controlled lighting need to be used.

In a series of experiments relying on self-assessment of mental workload, researchers have empirically determined how long it takes to transition from a point of high mental workload to a point of low mental workload. For example, finding a moment of low mental workload (a good stopping place) is a common strategy for reducing the disruptiveness of an internal interruption. In essence, a task filled with mostly long periods of high mental workload is more vulnerable to disruption and more difficult to voluntarily stop. Fogarty et al. performed an experiment [64] to predict the interruptibility of programmers and the time needed to reach a good stopping point. In the experiment, programmers were prompted every 3 minutes to complete 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 delay addressing the interruption for a mean of 43 seconds.

Instead of relying solely on self-assessment, researchers have found that the granularity of activity predicted different levels of mental workload. They describe three different levels of granularity of activity: *coarse, medium, and fine* [212]. Coarse levels of activity involve less mental workload and higher levels of goal processing whereas finer levels of activity are highly engaging and resource intensive. Iqbal and Bailey performed an experiment where observers reviewed videos of recorded workplace tasks and then identified transitions in granularity. For programming tasks, the average time to transition from a moment of high mental workload (fine activity) to low mental workload (coarse activity) was nearly 7 minutes. This experiment demonstrated that for workplace programming tasks, the strategy of quickly reaching a good stopping place is often not practical.

2.1.5 Environmental Effects

How can a person recover from an interruption successfully? Previously, we found that being interrupted at a moment of high mental workload is correlated with the extent of disruption. Researchers have found this effect can be reduced or completely eliminated if the environment (computer display, physical workspace) provides a rich set of reminders. That is, if the environment can provide *cues*, information that aids with the recall of memories associated with the task, then the person can more readily resume it.

In a series of experiments [4, 86, 193], the availability of cues during task resumption was correlated with a reduction of the time to restart the task. Interestingly, this effect holds even for implicit cues that the person may not have intentionally associated with the task, such as the location of a mouse cursor. For example, in one experiment when the mouse cursor location was moved from the last button clicked on the screen to the corner of screen, the time to resume the task became significantly longer. Enhancing the cue by highlighting the last element on the interface related to the task (such as last

button clicked) resulted in an even greater benefit. Cues that are associated with the goals of a task are especially helpful in rapidly recovering from an interruption because recalling the goal triggers retrieval of related memories [6].

Observations of developers suggest they frequently rely on cues for maintaining context during programming tasks. For example, Ko et al. [102] observed programmers using open-document tabs and scrollbars as aids for maintaining context. However such cues often do not provide sufficient context to trigger memories: In studies of developer navigation histories, a common finding is that developers frequently have to view many locations in rapid succession in a phenomenon known as *navigation jitter* [180, 149].

2.1.6 Summary

To date, interruption research has produced several insights but also consistently pointed to a root problem: the memory failures that result from interruption of a high mental workload. Although a key finding, there are limits to what can be concluded and applied from this research area to software development. First, most experiments studied short interruptions on simple tasks: Studied tasks lasted on the order of seconds to minutes with similar interruption durations. The observed difference in experimental conditions, although significant, have been on the order of seconds; a far cry from the many minutes observed with interrupted developers in workplace studies. A few studies have considered short interruptions on more complex work, but little or no research has examined extended interruption to complex work such as programming.

Second, interruption research has yielded little understanding of what memory processes underlie the resulting memory failures. Thus, we have difficulty explaining successful support mechanisms or identifying what strategies are successful in memory recovery. Certainly, as the length of the break increases, the initial effect of a sudden shift in attention caused by the original interruption matters less and less, and the question of how people maintain memory of their work matters more and more.

2.2 Memory

Human memory is both fragile and resilient. Why do we seem unable to remember the simplest of things, like a phone number, for more than a few moments, but are able to recite the gist of conversations or complicated movie plots in vivid detail for many years?

Previously, we provided a general review of psychology and cognitive neuroscience research on the brain and memory [147] (see Appendix A for more detail). In the section, we synthesize our findings in terms of five different types of human memory that are heavily used during programming. Our categories, derived from Fuster's [67] and Morris and Frey's [131] accounts of memory, are the following: prospective, attentive, associative, episodic, and conceptual.

2.2.1 Basic Concepts

What is *memory*? Memory is not a simple concept—philosophers and researchers today struggle to provide a satisfying answer. To some, memory takes the form of a conceptual storehouse. To others, it is a collection of processes or abilities that include recollecting facts and experiences. And for some a definition of memory must imply that changes in behavior can result from previous storage of information. Foster succinctly captures these characteristics of memory: "a complex, diverse, and heterogeneous entity...[that]...lies at the core of our whole mental life, behavior and sense of personal identity". In this dissertation, I use *memory* in the following senses: (1) a repository of information in the brain and (2) the information itself that is stored in brain.

The basic processes of memory include: encoding, storage, consolidation, and recall. *Encoding* is the process of extracting relevant information into a construct suitable for storage; moreover, multiple levels of encoding may occur. *Storage* is the persistence of information and its associated context. *Consolidation* is the process that combines, augments, migrates, or simplifies stored memory. Finally, *recall* is the retrieval or recollection of stored information.

What is forgetting? *Forgetting* is the process by which previously stored information becomes inaccessible. Classically, two factors are used to describe forgetting: decay and interference. *Decay* is the result of physical processes that weaken the storage of memory. *Interference* is an emergent condition preventing the relevant information from being retrieved [199]. Interference is caused by a failure to suppress other competing or irrelevant information. Some researchers also suggest that processes similar to consolidation may cause certain information to be replaced or diminished [17]. In modern perspectives of memory, decay is now seen as the failure of neurons to undergo or maintain long-term potentiation in the hippocampus [210].

2.2.2 Memory Types and Networks

Rather than using memory categories based on time, such as short-term or long-term memory, modern researchers have adopted a more distributed and specialized view of memory [197, 67], where memory is retained throughout the brain within the neurons that processed the original information [108]. Neurons are able to store memories through the process of *long-term potentiation*, a biological process which strengthens the physical connections between neurons by temporary modulation of synapse excitability or induction of new synapse growth [1].

In the next sections, I describe the prospective memory, attentive memory, associative memory, episodic memory, and conceptual memory. A summary of the memory types and memory failures is provided at the end.

2.2.2.1 Prospective Memory

Prospective memory holds reminders to perform future actions in specific circumstances (e.g., to buy milk on the way home from work) [209]. When forming a prospective memory, both an intended action and a retrieval cue are stored. Subsequently, perceptual processes monitor the environment for the cue, retrieve the memory, and bring cognitive

attention to the intended action.

Given the complexity of the process for storing into and recalling from prospective memory, naturally there are several points of failure [104]: When an intention is held in prospective memory, a monitoring process continually scans for the conditions for acting upon the intention. These monitoring processes compete with other cognitive resources, leaving prospective memory susceptible to *monitor failure*, failure to act on an applicable intention. When a condition is realized, prompting processes must also compete against active goals in order for the intention to receive conscious attention. Therefore, prospective memory is also susceptible to *engage failure*, a failure to acquire conscious attention.

2.2.2.2 Attentive Memory

Attentive memory holds conscious memories that can be freely attended to. Within it, goals, plans, and task-relevant items can be sustained for substantial periods of time. Attentive memory has two complementary operations: focusing and filtering.

Attentive memory is highly volatile and prone to frequent failures. When a programmer is actively engaged in a programming task, attentive memory allows a programmer to maintain focus on particular programming elements or goals that are relevant to a programming task. Although residuals of previously attended items can be found after switching attention [162], task switches often result in *concentration failure*, a failure to maintain focus on an item. Attentive memory can only provide reliable focus on a few consciously accessible items at a time. Constraints imposed by phase coherence and modality separation frequently induce *limit failure*, a failure to hold the required number of items. Moreover, interruption is very likely to disrupt a programmer's maintenance of attended items, such as a programming location being edited.

2.2.2.3 Associative Memory

Associative memory holds a set of non-conscious links between manifestations of co-occurring stimuli. Associative memories are essential for the "automatic recording of attended experience" [131]. The reason why the brain evolved the ability to record such activity is that many important events cannot be anticipated and do not recur, and therefore traces and features of experiences must be recorded in real-time.

Despite the raw power of associative memory, it has several weaknesses. When an associative memory is first formed, its expected lifetime is only a few hours. Formation of an association is determined by uncontrollable factors such as novelty or interest. For example, in brain imaging studies of subjects memorizing words, the experimenters could predict which words would be forgotten based on their activation strength in associative memory [54]; that is, forgotten words did not produce a strong enough response to engage associative memory during the memorization period. In such cases, the result is a *retention failure*. To combat this failure, people often form intentional associative memories though internal speech (activation of speech motor systems and speech comprehension [84]), which is nearly equivalent to hearing ourselves speak aloud and may subsequently excite auto-associative mechanisms [119].

Other times, an associative memory is formed, but with weak or missing associations. For example, it is common to associate with the visual features of an item, but fail to associate with other attributes such as its name, limiting our ability to recall it. This phenomenon is evident when someone says, "I'll recognize it when I see it". As a result, *association failure*, a failure to form complete or strong associations, frequently occurs.

2.2.2.4 Episodic Memory

Tulving, an influential memory researcher, describes *episodic memory* as the recollection of past events [196]. Whereas associative memory provides the facility for soaking up

raw experiences, episodic memory involves a much more complex network of memory processes. Episodic memories involve highly processed input from every sensory modality, as well as input relating to ongoing cognitive processes. Additionally, as the brain develops over time, the ability to learn and anticipate complex forms of episodic structures enables more concise representations of experiences to be retained [100].

In order to form episodic memories cognitive resources are required, and when those resources are otherwise engaged (on a hard programming task, for example), memory failure can occur. For example, episodic memory requires processes for maintaining information about recency and ordering about events [73]. As a result, when learning new experiences it is common to incur a *recollection failure*, a failure to recall a sequence of events in a complete and orderly fashion. However, research has shown that episodic cues can assist in improving episodic memory, even in memory-impaired patients [85].

Episodic memory is not as fully automatic as associative memory and can be easily disrupted [96]. For example, a person may remember the experience of hearing sentences being read aloud, but forget details such as whether the voice was male or female or the specific order of the sentences. Therefore, experiences requiring heavy cognitive load are susceptible to *source failure*, a failure to recall contextual details associated with an experience.

2.2.2.5 Conceptual Memory

Conceptual memory is best understood as a continuum between perceptions, actions and abstractions. Conceptual memory is consciously accessible. *Priming* is a process that allows responses related to a particular perception, action, or abstraction to become more probable.

There are several failures possible when remembering perceptions and actions. The effects of priming fade over time. In general, the state of unprimed memory results in *activation failure*, an inefficient state of conceptual memory. Interruption may reduce

the effect of priming of concepts needed for a programming task, requiring that the programmer refresh his/her memory.

There are also weaknesses possible when remembering abstractions. Because the formation of abstractions in conceptual memory rely on systemic consolidation, it is common to experience *formation failure,* a failure to form an abstraction in a timely manner. Interruption may reduce the ability for a programmer to hold together newer ideas that do not yet have conceptual memory support.

2.2.3 Summary

I have presented the background on five memory types: prospective, attentive, associative, episodic, and conceptual (see Figure 42 in Appendix A). I briefly summarize the salient points, in Table 1. In the next subsections, I present a detailed summary on the programming tasks that tax a memory type, the resulting memory failures, and cognitive support that can help programmers prevent or recover from these memory failures.

Table 1: Functions, and memory failures associated with different memory types.

MEMORY	FUNCTION	FAILURES
prospective	Holds reminders to perform future actions in specific circumstances.	Monitoring and prompting failures
attentive	Holds conscious memories that can be freely attended to.	Limit and concentration failures
associative	Holds a set of non-conscious links between manifestations of co-occurring stimuli.	Retention and association failures
episodic	Holds the recollection of past events.	Source and recollection failures
conceptual	Holds representations in a continuum of perceptions and abstractions	Activation and formation failures

Prospective Memory

- Prospective remembering allows us to perform postponed actions.
- Background processes monitor the environment for reminder cues and automatically retrieve a goal.

- Prospective remembering can take a toll on ongoing cognitive processes and can fail to trigger at the appropriate time (prompting failure).
- Interruption is likely to interfere with the monitoring of prospective memory causing a programmer to fail to perform important agenda actions (monitoring failure).

Attentive Memory

- Attentive memory is largely driven by intentional processes that allow information such as goals to be coded as abstract visual targets, internal speech, or concepts and maintained for a several minutes or hours.
- Attentive memory is provided by neurons that create temporary bindings of plans of action and task-relevant knowledge and have affinity to specific goals and modalities.
- Phase coherence constrains the number of attended items (limit failure).
- Interruption is likely to disrupt a programmer's maintenance of attended items (concentration failure).

Associative Memory

- Connected to multiple processing centers of the brain, associative memory is basis for most initial declarative memories.
- No conscious effort is required to form associative memories; however, we can easily form intentional memories through internal speech (activation of speech motor systems and speech comprehension), which is nearly equivalent to hearing ourselves speak aloud.
- We cannot control whether information is deemed interesting enough to trigger recording in associative memory (formation failure).
- Interruption may disrupt the state of cues in the environment, making it difficult to recall information associatively (association failure).

Episodic Memory

- Episodic memory does more than binding perceptual events, but can also relate semantic, temporal, and contextual information into episodic structures.
- Episodic memory is important cognitive mechanism for learning a sequence of new experiences as well as recognizing and summarizing them.
- Cognitively demanding work can interfere with maintenance of this contextual information, resulting in the failure to form important temporal or contextual associations (source failure).
- Interruption may prevent source information from being maintained, creating incomplete memories of experiences (recollection failure).

Conceptual Memory

- Conceptual memory is essential for encoding, remembering and reasoning about everyday objects and situations.
- Conceptual memory spans a continuum from perceptions to abstractions.
- Perceptions are the memory of stimuli and their features.
- Spatial, visual, verbal perceptions can be primed in conceptual memory for several minutes before needing to be refreshed.
- Abstractions are formed over perceptions and actions.
- Conceptual memory supports the ability to abstract information after several exposures to objects, experiences, or actions.
- Unfortunately, many exposures are required before items can be remembered at a conceptual level, and interference can affect our ability to leverage executive concepts in controlling what information we recall that would be relevant for a goal (formation failure).
- Interruption may reduce the effect of priming of concepts needed for a programming task (activation failure).

In the next sections, I review background on programming activities and programming environments.

2.2.4 Programming Activities

Software developers perform of activities during software development. In this section, I present some key definitions of programming activities and its relation to memory.

Editing. The activity of modifying source code text. Editing requires use of attentive memory to keep track of the current region of text that is being modified, and episodic memory to reason about a series of edits that have been made.

Refactoring. The activity of restructuring source code without modifying its semantics. Editing requires use of attentive memory to keep track of other locations affected by the restructuring process.

Comprehension. The activity of understanding a program's source code. Comprehension requires use of conceptual memory to keep important abstractions in mind and potentially form new ones.

Searching. The activity of finding documents and source code lines based on based on keywords or patterns. Searching requires associative memory to reason about search terms for locating recently visited code.

Navigating. The activity of transitioning between documents in a IDE. Navigating often involves visual search and requires associative memory to identify cues that guide how the documents are navigated.

Structured Navigation. The activity that allows navigation based on the structure of the program. For example, within an document, commands are available to navigate

from a program element's usage to it's definition, via commands such as *Goto Definition*. Comprehension requires use of conceptual memory to reason about relations between the program elements.

Building. The activity of compiling a program's source code into executable machine code. It is possible that a program may not successfully build due to problems present in the source code. Programmers often exploit build errors as make-shift prospective memory aids.

Debugging. The activity of controlling the execution of a program and inspecting its runtime state for the purpose of locating program faults. Debugging requires use of attentive memory to track the state of execution and episodic memory to reason about the runtime state.

Programming environments often provide specific tools in support of programming activities, and is described in the next section.

2.3 Programming Environments

Software developers rely on a variety of tools for creating software. In this section, I present some key definitions and tools related to software development.

2.3.1 Definitions

A *programming environment* is the setting that provides the applications and documents used for a programming task, with the ultimate goal of creating a software program. Within a programming environment, programmers typically use a tool framework called an *integrated development environment* (IDE) for performing many programming activities. A *programming activity* is a coordinated sequence of actions performed in the programming environment. There are many kinds of programming activities typically centered around documents, such as editing, debugging, refactoring, documenting, or

navigating. Other applications may be used as well for programming activities such as source-control applications for synchronizing and committing documents, or web-browsers applications for finding and researching online programming resources.

The documents that compose a software program are referred to as *source-code documents* or *source code*. Source code is written in a *programming language*, a formal notation of human-readable instructions for computers. Programming languages come in various flavors that differ in structure and semantics. In this dissertation, we focus on *object-oriented languages*, a common flavor of a programming language that emphasizes encapsulating data and operations on that data into an *object* (a runtime entity with behavior). Object-oriented languages are structured hierarchically with programming elements. A *programming element* is a named entity that contains either a sequence of instructions or other programming elements. For example, C# [1], is structured with top-level programming elements called *namespaces*, which contain other programming elements called classes. A *class* is an unit of source code that describes a static representation of an object. A class contains programming elements called *fields* and *properties* for encapsulating data and programming elements called *methods* for encapsulating instructions.

2.3.2 Interface Elements of an Integrated Development Environment

A mature software program can contain hundreds of thousands of documents. An Integrated Development Environment (IDE) is designed with the goal of managing these many documents and providing support for programming activities such as navigation, editing, and debugging source code. An IDE supports programming activities by providing several *interface elements* (a user interface specialized for a programming activity).

[1] http://msdn.microsoft.com/library/z1zx9t92

2.3.2.1 Interface Elements

Two popular IDEs—Visual Studio [2] and Eclipse [3]—provide several interface elements for helping developers. These interface elements can be arranged and docked within the IDE. The area where interface elements can be arranged is a called the *viewport*.

Tree View. To organize the source-code documents of a program, an IDE can group documents into units such as folders, projects, or workspaces. These units are organized hierarchically and may be orthogonal to organization of the program's programming elements. The interface supports opening documents and collapsing and expanding units.

Class View. An alternative to the Tree View, the Class View displays the program's classes and its members in a hierarchical view, supporting structured navigation.

Document Tab Strip. To switch between recently open documents, an IDE provides a strip of tabs each labeled with the document name and decorated with an "*" if the document is unsaved.

Document Panes. Typically, only one document and one document tab strip is visible in an IDE. The viewport can be split, allowing two document tab strips, and thus two documents to be displayed concurrently.

Editor Window. A word processor for modifying a document, supporting syntax-highlighting for programming language keywords, and tab-completion for automatically filling out program elements. Editor windows are hosted within a document pane.

[2] http://www.microsoft.com/visualstudio/en-us
[3] http://www.eclipse.org/

Search Window. Documents can be searched for keywords or patterns. Search results display source code lines and their files and line numbers that match a search query. Double clicking on the search result navigates the IDE to the source location. In Eclipse, the Search Window organizes search results hierarchically by program elements. In Visual Studio, the Search Window provides a flat list of search results. See Figure 4.

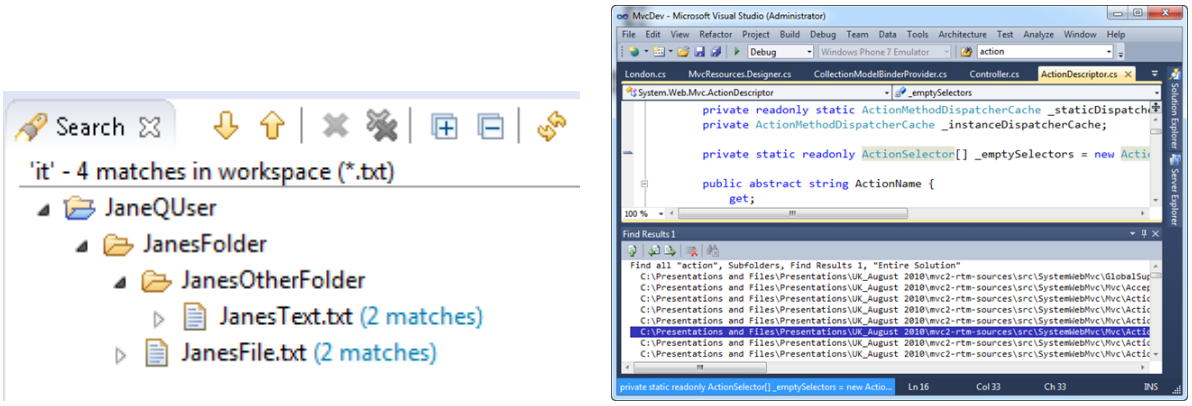


Figure 4: An Eclipse Search Window (left). A Visual Studio Search Window (right).

Bookmarks. A source code line can be flagged as a *bookmark*. Bookmarks are displayed in the margin of an editor window. A *Bookmarks Window* displays a list of bookmarks in the program. See Figure 5.

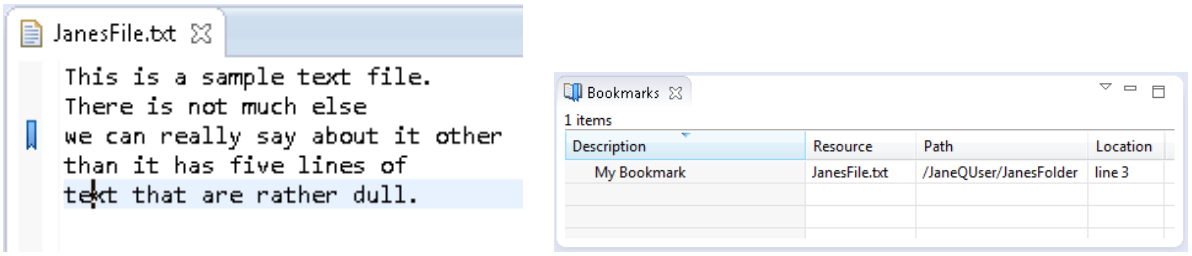


Figure 5: In Eclipse, a bookmark added in the margin (left). A view listing all bookmarks (right).

Task List. Tasks can be stored by the IDE. A common way to create an entry in the Task List is by creating a TODO comment. A source code line can be marked with a *TODO*

comment, by using a notation like `// TODO`. Eclipse provides an extension that allows a Task List to be connected to an external task tracking repository. See Figure 6.

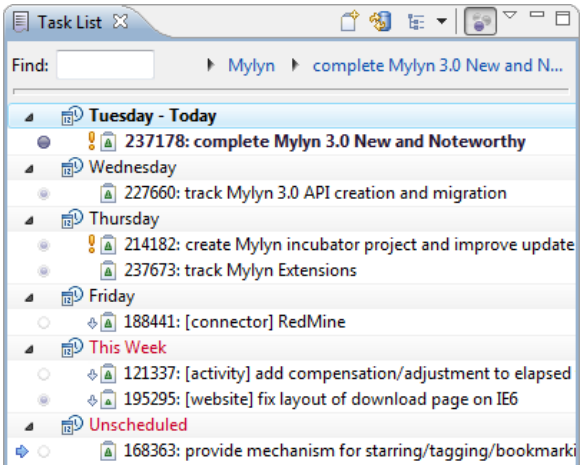


Figure 6: A Task List in Eclipse.

Error Window. When a program is compiled, errors that violate the programming language's syntax specification or semantics, called *compile errors*, are displayed in the *Error Window*. Double clicking on a compile error navigates the IDE to the associated source location.

Watch Window When a program is being debugged, inspected runtime values can be viewed in the window.

2.3.3 Source Code Repositories

A *source code repository* is a data store for tracking revisions made to source code. A *Source control tool* is tool for managing the source code repository. Often, source control tools can be integrated directly into the IDE; however, developers also use source control tools in a standalone manner.

Source code repositories provide many operations. A developer can *commit* a file, which will add the current version of the file to the repository along with additional information, such as authorship and programmer commentary. Most source code repositories

allow for multiple files to be committed at once in an enclosure called a *changeset*. Source code repositories also provide operations for showing the difference between versions in the repository.

Finally, source code repositories make the distinction between centralized repositories and distributed repositories. A *centralized repository* stores data on a single server. A *distributed repository* allows commits to be stored on a local copy of the repository, which can be later merged into other, distributed, repositories.

2.3.4 Task Repositories

Previously, I have defined, *task*, as a unit of work assigned to a developer, which may be refined into further subtasks. In software projects, it is common to assign primary responsibility for completing a task to a single developer. To manage assignment and status of tasks, a task repository is often used.

A *task repository* is a data store for tracking tasks assigned to developers. A *task tracking tool* is a tool for managing the task repository. In a programming environment, a developer have many tools available to managing task repositories. Unfortunately, the terminology used between different task repositories often vary. Task repositories such as VersionOne[4] or Pivotal[5] that support agile development often refer to a task as a "story" or "defect". Microsoft's TFS (Team foundation server)[6] or IBM Jazz[7] instead use the term "work item". Even more confusing, before the term *task repository* came into prominence, the concept of an "issue repository" was also used. To make a clear distinction, an *issue repository* generally differs from a *task repository* in that it stores bug reports and feature requests that are collected from users and other stakeholders.

Another difference among task repositories is how assigned work is structured—there may even be differences in how a team uses a task tracking tool to structure the tasks in

[4] `http://www.versionone.com/`
[5] `http://www.pivotaltracker.com/`
[6] `http://www.visualstudio.com/en-us/products/tfs-overview-vs.aspx`
[7] `www.ibm.com/software/rational/jazz/`

the task repository. For example, Pivotal, allows the specification of a “story” and a “story type” of “chore”, “bug”, “release”, or “feature”, as well as a list of “tasks” associated with the story. In contrast, VersionOne, makes a “defect” a first class unit of work. VersionOne also allows stories to belong to “epics”. Further, VersionOne “stories” and “defects” can be assigned tasks, with different and multiple story owners, whereas in Pivotal “stories” have a single owner and tasks belong to that story owner. As an example of a team-specific variation I personally experienced—even though VersionOne allows the specification of “tasks” for “defects”, by team convention, we only defined tasks for “stories”.

Chapter III

RESEARCH APPROACH

Charles *is a seasoned programmer, who long ago wrote the majority of the code for a core product. He is considered the product owner of that software code, which is very complex and still requires constant tweaking and maintenance. Charles has no formal requirements to work from, but instead receives a steady stream of emails or walk-in visits from other developers and project managers who request enhancements, report issues, or ask for advice in how to effectively use the product's framework. Recently, management has become concerned with an over-reliance on Charles and has suggested he take an apprentice that can learn the product and take on some of the maintenance tasks. Charles, used to solitude, finds the additional responsibilities overwhelming. He often complains that he can never concentrate long enough on a task to get into the right state of "flow" to get work done. Increasingly, Charles has resorted to staying late to code in isolation, but he still has difficulty in completing work.* Charles's story describes an unfortunately large number of developers who have a single focus on a complex project, but find the workplace full of interruption and distractions. Charles needs tools and strategies for maintaining concentration on complex tasks and that help him share his knowledge with others with minimum interruption.

Despite its vast capacity and associative powers, the human brain does not deal well with interruptions. Particularly in situations where information density is high, such as during a programming task, recovering from an interruption requires extensive time and effort. Although modern programming environments have begun to recognize this problem, none of these tools take into account the brain's structure and limitations. Existing frameworks of programming information needs neither reflect the wide variety of programming tasks that exist nor conform to the underlying needs of the human brain. For example, Storey et al.'s framework on information needs [186] is limited to considering only exploration tasks and does not incorporate memory failures and resulting information needs reflected in other everyday programming tasks.

What is needed is a conceptual framework for understanding human memory organization and its strengths and weaknesses, particularly with respect to dealing with work interruptions. The framework should explain empirical results obtained from experiments in which programmers were interrupted. The framework should aid with the design of development tools capable of compensating for human memory limitations and delineate programmer information needs such tools must satisfy and suggest several memory aids such tools could provide.

The overall thesis of my research is that programmer recovery from interruption can be improved by making use of tools specifically designed to address the limitations of human memory. To validate this thesis, I must demonstrate the effects of interruption on a programmer and causally relate them to limitations of human memory, i.e., explain programmer performance problems in terms of memory limitations. Further, I need to devise strategies for overcoming these memory limitations and contextualize them in terms of programmer information needs.

In this chapter, I outline the research approach (see Figure 7 for an overview), which begins with research questions about the problems discussed in Chapter 2. The first three research questions explore the problem. The fourth research question asks how to

conceptualize it. The last three research questions involve identify the solution approach, implementation and evaluation of the solution.

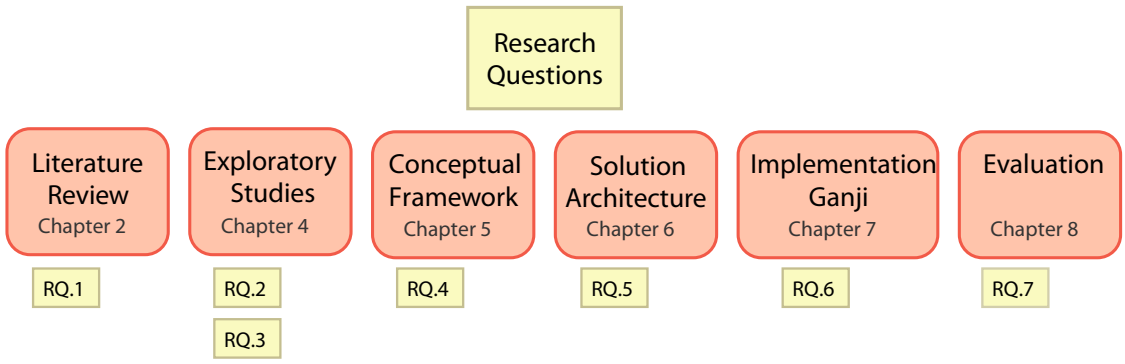


Figure 7: Illustrated steps of my research approach.

3.1 Research Questions

The following research questions guide my research efforts.

Research Question 1 - What memory deficiencies arise from interruption of programming tasks?

Research Question 2 - What strategies and sources of information do programmers use to recover from interruptions?

Research Question 3 - What memory aids can programmers use to recover from interruptions?

Research Question 4 - To what extent can memory deficiencies be linked to resumption strategy failures or lack of information?

Research Question 5 - To what extent can the innate properties of human memory be used to *derive* more effective resumption aids?

Research Question 6 - Which new algorithms and concepts are needed for building effective resumption aids?

Research Question 7 - What are the observable benefits and disadvantages with using memlets for interruption recovery in software development?

3.2 Research Approach Outline

The first three research questions are answered in my exploratory research work.

To answer Research Question 1, I conducted an extensive literature review on interruptions and memory [147], which has been discussed in Chapter 2 and can be seen in full in Appendix A.

To answer Research Question 2, I examined over 10,000 recorded programming sessions [151] and surveyed over 400 programmers to understand the strategies and sources of information that were useful for recovering from interruptions [152]. The results of the study is summarized in Chapter 4 and is presented in full in Appendix B.

To answer Research Question 3, I performed a controlled experiment [148] with 14 programmers and investigated how the programmers used different types of memory aids to recover from interruptions to programming tasks. From this laboratory study, I observed instances where memory deficiencies occurred because there was a failure to record important information. However, memory deficiencies were mitigated when memory aids were introduced, helping the programmer recover from the interruption faster. The results of the study is summarized in Chapter 4 and is presented in full in Appendix C.

The remaining four research questions (4-7) represent the conceptualization, realization, and evaluation of my research work. To answer Research Question 4, I constructed a conceptual framework that could explain the results obtained in answering the previous research questions. From it, I derived a set of programmer information needs based on memory failures that programmers experience [153]. To answer Research Question 5, I constructed a set of memory aids, called *memlets*, derived from the conceptual framework. To answer Research Question 6, I implemented these memory aids and devise the

necessary algorithms and data structures for realizing the implementation. To answer Research Question 7, I evaluated a set of claims about the memlets' abilities to aid recovery from interruptions.

Chapter IV

EXPLORATORY RESEARCH

I have performed exploratory work that has answered several research questions.

To answer Research Question 2, *What strategies and sources of information do programmers use to recover from interruptions?* I examined over 10,000 recorded programming sessions [151] and surveyed over 400 programmers to understand the strategies and sources of information that were useful for recovering from an interruption [152]. From the study, I was able to identify several important strategies and sources of information that programmers use to recover from interruption.

To answer Research Question 3, *What memory aids can programmers use to recover from interruptions?* I performed a controlled experiment with 14 programmers and investigated how the programmers used different types of memory aids to recover from an interruption to a programming task. From this laboratory study, I was able to observe several instances where memory deficiencies occurred when there was a failure to record important information. However, memory deficiencies were mitigated when memory aids were introduced, helping the programmers recover from the interruption faster.

The two studies differed in scope and design. The first study used a survey and analysis of interaction logs to extract information about the frequency and nature of strategies that developers use to recover from interruptions. The second study is a more focused but still formative study that evaluated how well different resumption aids might help developers in recovering from interruptions. More details about these two studies can be found in Appendix B and Appendix C, respectively.

Finally, I describe programmers' ratings for potential research tools, lessons learned, and limitations to the studies.

4.1 Empirical Study: How Developers Recover From Interruptions

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.

To understand the information-seeking behavior of developers after an interruption, I performed an empirical analysis of 10,000 recorded sessions of 86 programmers performing software development tasks [151] and a qualitative survey of 414 programmers [152]. From these studies I was able to learn much about the various strategies and information sources developers use to manage interrupted programming tasks.

Based on the analysis of programming sessions and survey results, I propose a framework for understanding these strategies and suggest how task resumption might be better supported in future development tools.

4.1.1 Analysis

Data was collected from developers using instrumented versions of Eclipse (74 developers) and Visual Studio (12 developers).

To obtain the developer's sessions, the events were segmented when there was a break in activity of 15 minutes or more. This segmentation is well supported by the nature of the data. The interval between events follows a Poisson distribution: for 98% of the 4.5 million events, the time between those events is less than a minute. This means tight clusters can be formed with any threshold above a minute. Similar thresholds have been supported in other studies [214, 166].

Events collection is performed solely by the IDE. It should be noted that what appears to us that as 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 I included window focus events [164], then I could better explain these breaks. Nevertheless, I believe many breaks from programming activity are due to diversions [14] that often

derail programming efforts, as observed by Ko [103].

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 leading to a recovery period on the order of minutes [203]. Therefore, I 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 to it.

In this study, I focus most of the 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 activity is related to resumption costs. For this reason, I only examined sessions with coding activity.

4.1.2 Survey

While several previous studies [114, 90] have documented the fragmented nature of software developers' work, there is no existing survey of the strategies developers use to resume interrupted tasks. The survey asked 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, and any factors that made resumption difficult.

Initially, the survey was administered to 43 programmers from a variety of companies including small startups as well as from the defense, financial, and game industries. Encouraged by the feedback, I distributed the survey with slight modifications to 371 employees at Microsoft.

When I distributed the survey at Microsoft, I selected the survey population from full-time software developers—excluding interns and people filling roles such as testers or project managers. I also excluded developers who had been recently surveyed by our research group to avoid oversampling. Respondents were compensated by a chance to win a $200 gift certificate. From the participant pool, I randomly sampled 2,000 developers and invited them to participate through email.

4.1.3 Strategies

Developers use a mixture of suspension and resumption strategies to manage their memories about programming tasks. *Suspension* strategies are preemptive strategies applied prior to a breakpoint. 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. These processes complement each other. For example, if a programmer externalizes working state through note-taking, then this process may also bolster internalization of the working state.

4.1.3.1 Suspension Strategies

Developers apply suspension strategies opportunistically based on factors such as availability of external cues, estimation of required retention period, or perceived importance and recoverability of information. Two that we discuss here are note-taking and cue-priming.

Note-taking. *Note-taking* is a strategy for recording thoughts and representations on a variety of paper or electronic media (such as sticky notes or email messages), within code, or via sketched diagrams. When recording information, a programmer must evaluate the status of a task and determine what information would be valuable to retain. Note-taking reduces the burden on prospective memory when writing down reminders and

on conceptual memory when drawing a sketch or diagram [39]. Often, programmers are spartan in their note-taking (e.g., even jotting down a couple of class and method names can be too time-consuming for a programmer). Programmers can reduce the effort in making a note by situating the note in the code (e.g., placing the comment "Fix this" within the source code as a reminder), but risk losing sight of the note among all the text. On the other hand, "Fix this" written on a sticky note is more likely to be visible but offers little context. In fact, Czerwinski observed many workers forget the purpose of a sticky note after about a week [46]. In general, there is a trade off among visibility, brevity, and durability of note content.

Cue-priming. *Cue-priming* is a strategy that can be frequently and quickly applied. Cue-priming involves preparing or finding a cue for triggering recall upon resuming a task. Commonly, this strategy uses readily available cues drawn from interface elements in the programming environment (*environmental cues*), to serve as reminders. For example, a developer may leave open the last window she was editing, highlighting several lines of code, or leave open an error message display that occurred while testing a program. Cue-priming allows a programmer to quickly set up prospective reminders, but the ad-hoc nature of environmental cues can be limited in expressiveness and potentially fragile as they may be lost or disturbed by new interactions with the environment or in the event of the IDE shutting down. To compensate, a programmer can leave behind *roadblock cues*, essentially intentionally made compile errors, which force the programmer to address the errors before resuming work. Roadblock cues are more resilient and manifold and therefore more suited for supporting attentive memory, as they allow attention to be maintained across several locations of the code.

4.1.3.2 Resumption Strategies

With *resumption strategies*, a developer uses previously internalized or externalized information to recall detailed knowledge about a programming task. Resumption strategies are applied at the resumption point and involve seeking lost information and the reconstruction of mental state. Two that we discuss here are cue-seeking and systematic review.

Cue-seeking. *Cue-seeking* is a light-weight strategy for seeking reminders, including activities such as returning to the stopping place or navigating through the source code for the purpose of recalling details about the program. Reengaging with the task, even without a targeted goal, can be effective for activating associative memories and triggering prospective reminders. Therefore, even seemingly random navigation has several tactical advantages. First, it increases the chance that programmers encounter other cues that in turn lead to more targeted navigation. Second, the process of navigation primes the programmer's conceptual memory and helps him or her maintain abstractions and spatial awareness for the coming task. However, cue-seeking is not a thorough strategy and it becomes less effective with the passing of time as associative memory degrades or another task disrupts the state of the programming environment.

Systematic review. When working on especially demanding cognitive tasks involving complex and dispersed changes in the source code, important details about a programming task may be forgotten. *Systematic review* is a strategy for inspecting previous progress on a programming task in order to re-acclimate the programmer and identify incomplete work. As a prerequisite, systematic review requires a transcript of how the task was performed; in practice, a programmer may have to either log the changes she makes or use a tool for obtaining a source code difference (a diff) between the current and older versions of the code. A systematic review without a transcription could work in tandem

with episodic memory, but at the cost of thoroughness. For completely recovering from an interruption, a systematic review is one of the most thorough and methodological strategies in a programmer's arsenal. But unlike cue-seeking, performing a systematic review can be time consuming, and without proper tool support and with an imperfect episodic memory—often an impractical one.

4.1.4 Information Sources

To identify the different information sources used by programmers for interruption recovery, I analyzed 10,000 recorded sessions from 86 programmers. In particular, I measured the programming activity that occurred at the start of each session and characterized the sources of information that developers sought in the program environment. I also asked developers about the difficulty they had with obtaining the information they needed from these sources.

Developers used the following information sources for interruption recovery, ranked by frequency of use, starting with the most frequent.

4.1.4.1 Documents

No other source of information is as widely used as the *source code* itself. After an interruption, developers are most likely to navigate to several code files before resuming coding. For example, if a developer was editing a method (e.g., `AuthenicateTwitterKey()`) and then was interrupted, in over half of the cases, the developer would navigate to several other documents not containing the method prior to resuming editing the `AuthenicateTwitterKey()` method. Only 7% of interruptions involved no navigation to other locations. Overall, this sort of behavior is consistent with developers applying the cue-seeking strategy.

Despite the popularity of cue-seeking, gathering information from the source code is a slow process. For instance, only 10% of interruptions had coding activity start in less than a minute; instead most sessions required 10-15 minutes of initial exploratory navigation. From our surveys, developers reported feeling disoriented and resorted to clicking and scrolling through open files

hoping to find something relevant.

4.1.4.2 Notes

Notes were another popular source of information, according to self-reported responses. The participants consistently reported using a mix of media to record notes, including a bug tracking system (TFS[1] or Product Studio[2]), the note-taking product Microsoft OneNote[3], the source code itself, as well as traditional paper media. Overall, note-taking was a prominent suspension strategy.

Several participants described how they used different media and their personal working style:

> *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.*

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

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

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.*

4.1.4.3 Compile Errors.

Compile errors were a common source of information for developers resuming from an interruption. Although we could not distinguish between compile errors that were left intentionally

[1] `http://www.visualstudio.com/en-us/products/tfs-overview-vs.aspx`
[2] `http://weblogs.asp.net/scottgu/251930`
[3] `http://office.microsoft.com/en-us/onenote/`

and ones that were present at the time of interruption, many developers reported the use of intentional compile errors [4]

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 reminders more visible:

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

Another noted how compile errors 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.*

Developers' use of compile errors as roadblock cues was consistent with the cue-priming suspension strategy. Although compile errors can be effective as reminders, they do have some tactical weaknesses. First, compile errors are drastic measure that prevent the code from compiling and being tested or debugged. Second, if an interruption requires a switch to another task in the same codebase, then the compile error's presence would interfere with resuming another task. Moreover, compile errors are not effective in situations where there is a long-term interruption or multitasking.

4.1.4.4 Run-time Information

Another source of information developers used was the result of simply executing or debugging the program. Developers most often used debugging rather than simple program execution.

[4] In discussions with a programmer, I once asked what he would do if I told him he could had to stop his programming task right then and there and not pick it up for a month. He said, he'd quickly insert a compile error where he was. Later, by an automated mental process, he would pick up the task by immediately trying to compile the project and discover those errors, even though he would have completely forgotten about leaving them there.

There are several possible explanations why developers use this source of information. In some cases, developers were interrupted when debugging and were simply continuing the debugging activity. But debugging information can be helpful to an interrupted developer; it provides the opportunity to recover previously set breakpoints and inspected runtime values. Also of interest is the frequency of debugging found in the analysis of programming sessions: 59% of programming sessions involve some form of debugging.

The use of run-time information is consistent with the resumption strategy of cue-seeking. The program flow provides a means of structured navigation. However, debugging can be cumbersome and infeasible if the task is in an unfinished state. Developers often complained of the IDE "forgetting" runtime information as soon as a debugging session is closed.

4.1.4.5 Task List.

Although not as frequently used as other information sources, the Task List was a useful source for developers wanting to refresh the goals and criteria specified in the task. However, available tools for specifying programming tasks did not resonate with developers: Less than 1% of developers created their own personal tasks or breakdown existing assigned tasks into subtasks using the Task List.

One reason was that the Task List was not the only place where developers recorded their own notes about programming tasks. Furthermore, developers felt that the assigned task, more often than not, was not an adequate representation of the work that needed to be performed. Developers felt the actual work had to be broken down much further, usually after exploration and experimentation, and as a result is not formally recorded. Further, developers need to make a distinction between public and private work. Other researchers have found similar conclusions: In a survey of 782 programmers and examination of work artifacts from 62 programmers [112], only half of the actual work performed by developers was found to be represented by assigned tasks.

4.1.4.6 Source Control History.

One information source developers often used as a last resort was to perform a source code difference between the current and older versions of the code they were working on in order to recover details about their past progress on a task.

Source control change history provides a means for applying the systematic review strategy, but there are still many problems with it. Many differencing tools view differences only on a per file basis. Further, developers complained about other problems they had with using existing diff tools including the following: The information provided was unordered, verbose, time-consuming, and cognitively demanding.

4.2 Lab Study: How Developers Recover From Interruptions With Visualizations

To answer Research Question 3,*What memory aids can programmers use to recover from interruptions?* I performed a controlled experiment with 14 programmers and investigated how the programmers used different types of memory aids to recover from an interruption to a programming task. Because my previous study examined recorded programming sessions and analyzed surveys studying note-taking, I used this study to also further my understanding of the problems associated with note-taking. Full details for the study can be found in Appendix C.

The study tested three different experimental conditions on developers' abilities to recover from task interruptions. Because of a prevalence of note taking as a suspension strategy, I allowed subjects in all three conditions to take notes prior to the breakpoint and to review their notes during task resumption. The base condition consisted of note taking alone, representing current practice. In the tool condition, the subjects supplemented their note taking using a resumption aid when resuming the task.

4.2.1 Resumption Aids

In the study, I wanted to understand in what ways a resumption aid can help a programmer recover from an interruption. Two resumption aids were created for the experiment (see Figure 8). The first resumption aid, called the *degree-of-interest (DOI) treeview*, was created as a replica of

Mylyn's treeview [98], which highlighted recently and frequently viewed locations in code. The second resumption aid, called the *content timeline*, showed fragments of recently edited and visited code, in order of occurrence.

The two resumption aids are populated from the same record of code history, but present them in different ways; as a result, they support different resumption strategies and memory processes.

The DOI treeview supports cue-seeking and attentive memory. It discards temporal information, displaying only the names of program parts—projects, files, classes, and methods—and highlighting the ones that were edited or viewed just before the interruption. Developers can use the highlighted names as cues for guiding their navigation in support of a cue-seeking resumption strategy. The highlighted names also supports the recall of source code locations held in attentive memory, which are otherwise likely to be forgotten during the interruption.

The content timeline supports systematic review and episodic memory. The content timeline uses temporal information to order the fragments of code. The ordered fragments of code allow the developer to review the previous changes and navigations in the order that they occurred. This ordering also supports the ability to recall events in episodic memory.

4.2.2 Method

Participants performed three programming tasks in two stages. Each task involved modifying a video game in order to add a new feature. In the first stage, participants were interrupted during the task and instructed to move on to the next task.

In the second stage, participants resume the interrupted tasks. To assist the resumption of the interrupted task, participants were assigned one of the resumption aids. More detail on the experimental methods, procedures, tasks, participants can be found in Appendix C.

4.2.2.1 Participants

Fifteen professional programmers (one female), average age of 39 years (range 31 to 56), participated in this study for receipt of two copies of Microsoft software. The programmers were screened and then selected based on a series of profile questions. The profile questions were

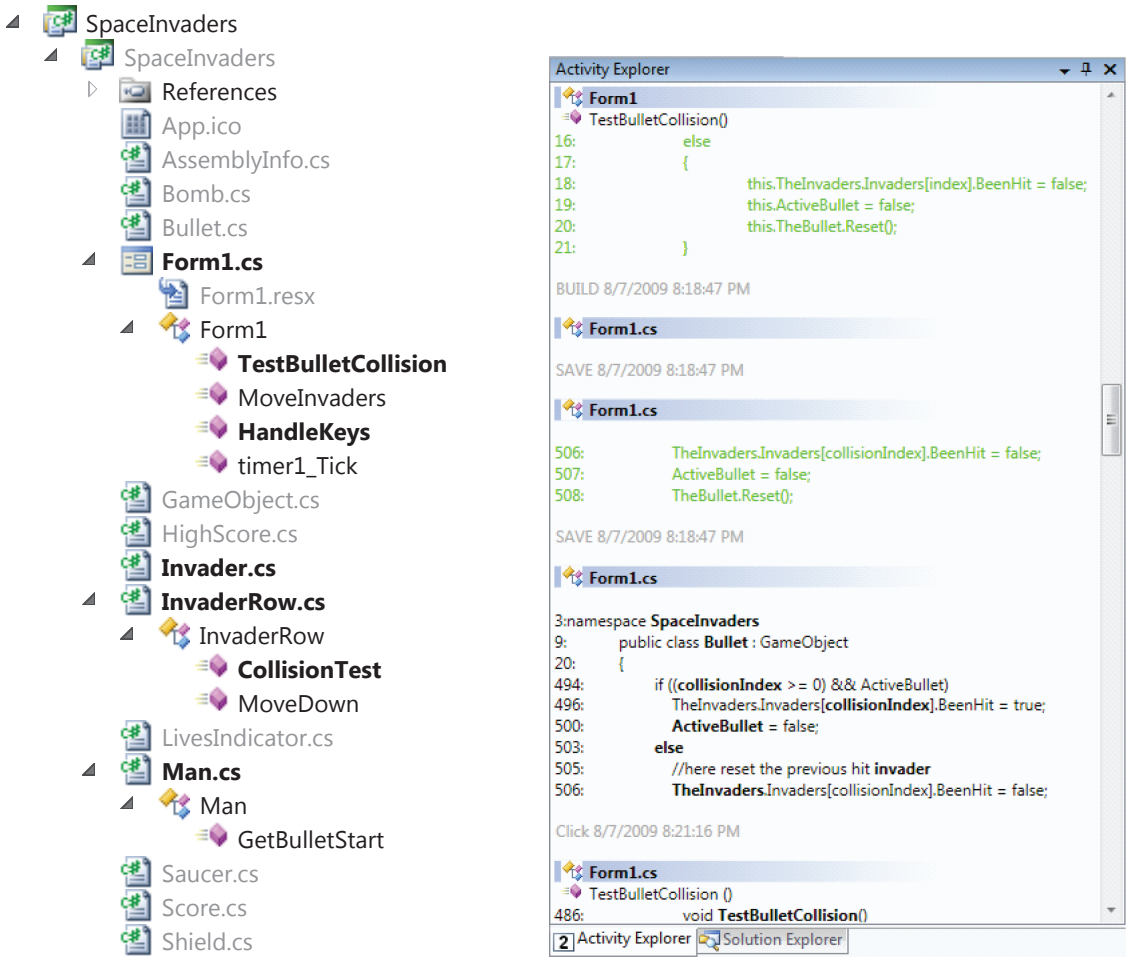


Figure 8: The experimental resumption aids: DOI treeview (left) and my content timeline (right).

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, I screened developers for experience with debugging and modifying the code of other developers and working in teams of at least 3 people.

4.2.2.2 Procedure

Each subject was run individually in a user-study laboratory. I was present for the entire session. I informed the subjects of the goals of the study. Subjects were told that I was interested in how they multi-tasked among programming tasks and how they used the different tools given to them. In addition, I stressed that I was not interested a perfect solution, but in completion of as much of

experimental task as possible.

For the tasks, I used source code from three simple games (Tetris [5], Pacman [6], and Space Invaders [7]). The games are C# implementations of classic arcade games. The motivation for using these games was based on the familiarity the games offered and the nontrivial complexity of their source code. Further, I had confidence from previous user studies 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 I provided to coordinate the subject's activities over the three programming tasks, shown in Figure 46 in Appendix C. 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 reviewed the cue for that task (if any) and took notes, for up to one minute. The application then started 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 (in total), 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 Pacman, resuming with condition 2;
9. finish Space Invaders, resuming with condition 3;

[5] http://www.codeproject.com/csharp/csgatetris.asp
[6] https://www.planet-source-code.com/vb/scripts/ShowCode.asp?txtCodeId=5669&lngWId=10#zip
[7] 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. I counterbalanced the conditions across subjects to account for ordering effects. Each session ended with a questionnaire to rate the two kinds of cues and gather general feedback.

To provide a consist and automated mechanism for interrupting the subjects, I used the following criteria: 15 seconds after a non-comment edit, after more than three non-comment edits, or after a twelve-minute timeout if no changes were made. I based these criteria on the factors Fogarty previously found to be associated with inopportune moments for interruption [64] and on the nature of the tasks. The solutions for the programming tasks typically required changes to multiple parts of the source code. By setting the interruption point several moments after making a change, upon returning, the developer must ensure the change is completed correctly and then recall the steps and relevant parts of code needed for the next change.

4.2.2.3 Tasks

Each programming task was to add a small feature to a 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 a 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 enemies, filling in an empty position if possible. If all enemies positions are filled, no new enemy is created.

I 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.

4.2.3 Results

In this section, I describe the basic results of the study, the failures that occurred after an interruption, the note-taking behavior of the programmers, and their behavior when using the resumption aids.

4.2.3.1 General results

Subjects using either resumption aid successfully completed nearly twice as many tasks in comparison with subjects relying solely on notes. Subjects with only notes were much slower to resume editing the code, nearly twice as slow on average as the other subjects, contributing to their failure to complete the task within the time limit. Further, subjects with only notes also suffered the consequences of memory failures more frequently, contributing to errors and forgetting important information from previous work on the task.

Resumption Speed. I list the time to resume editing, by resumption aid used, in minutes and seconds:

- Subjects that used the DOI treeview were the quickest, resuming in 2:30.
- Subjects that used the content timeline followed, resuming in 3:26.
- Subjects that only had notes available were the slowest, resuming in 4:28.

Although the trend of resumption speed is consistent with task performance, these differences are not statistically significant. Even though subjects using the content timeline were slower than

those using the DOI treeview, it did not impact their overall success in completing the task. This difference may be attributed to the resumption strategy employed: Subjects using the content timeline followed a systematic review strategy that is more thorough but slower, whereas subjects using the DOI treeview followed a cue-seeking strategy that is less thorough but quicker.

4.2.3.2 Failures

We looked at the time for a subject to resume editing after an interruption and characterized the failures that contributed to a subject not completing the task.

Failures Categories. To better understand why some subjects did not complete the tasks in the allocated time, I partition the failures that the programmers experienced when resuming the task after an interruption into the following categories:

1. Problem solving: The subject had all the pieces; he or she was stuck on the problem part.
2. Domain knowledge: The subject did not understand WinForms, event handler syntax, etc.
3. Location Amnesia: The subject had difficulty relocating relevant parts of the code.
4. Change Amnesia: The subject forgot to complete a step.
5. Program Amnesia: The subject forgot or misunderstood how the program worked.

The first two failures are not strongly related to interruption, but rather stem from subject's programming experience and problem solving ability. For example, some subjects simply lacked appropriate domain knowledge, which greatly hindered their progress. However, the last three failures were more strongly linked to memory failures after interruptions.

Subjects uniformly experienced problem solving and domain knowledge failures, but subjects with notes-only suffered the most from memory failures. Furthermore, for subjects using the content timeline, the slower but steady systematic review strategy paid off; in those cases, no relocation, change amnesia, or program amnesia failure occurred. And speed can kill: One subject with the generally faster DOI timeline, experienced a change amnesia failure that resulted in the subject failing to complete the task.

Change Amnesia Example. One subject was interrupted in the middle of connecting an input event to its event handler. When the subject later resumed the task using the DOI timeline, he thought he completed the connection, forgetting he had not yet registered the handler for the event. The subject moved on to the next part of the task and implemented the required functionality much faster than other subjects. However, when the subject tested the solution, the missing association 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 complete the change. This was the only task the subject failed.

Location Amnesia Example. As part of the programming task, a subject needed to find two locations in the code in order to add a feature. The subject had found the first location and was searching for the second location—but was interrupted. The subject took notes about her progress in the search, but no information about the first location. After resuming the task in the notes-only condition, the subject continued searching and soon found the second location. Unfortunately, overconfident she would remember the first location at the time of the interruption, she never made any note of it. The subject then spent the next 5 minutes trying to find the first location again, greatly frustrating her:

> *I remember finding that code, I just can't remember where it is!*

Unfortunately, by the time the subject located the code, she had used up most of her time and could not complete the task by the end of the trial.

4.2.4 Note-taking

In order to understand why note-taking was often insufficient for resuming from an interruption, we observed what type of information participants recorded in their notes.

All subjects took some form of notes. I partitioned note-taking into two categories: *situated*: The notes are placed within the source code, and *unsituated*: The notes are written on paper or electronically within a notepad.exe window. Most of the subjects' notes were unsituated and were evenly divided between paper and electronic forms.

During task suspension, subjects wrote down several types of information. The content of the notes generally referred to either prospective actions (e.g., "investigate where bullet goes off-screen") or specific program elements or locations (such as method names or line numbers). But a few developers also wrote down other kinds of information. Several subjects summarized the task description with a one-line sentence ("Modify tetris so piece falls to bottom") or briefly described what they had worked on previously.

Not surprisingly, many unsituated notes referred to program elements, whereas situated notes rarely needed to refer to program elements. This suggests that when subjects write unsituated notes on paper, they need to spend more time recreating the context of the note in order to adequately record information. The types of program elements subjects wrote favored elements higher in the hierarchy than lower—in descending order of frequency: files, methods, variables, code expressions, and line numbers.

Finally, subjects' notes focused on the most immediate subtask and neglected to record other locations needed later or visited before. As seen earlier, this often led to location amnesia failures.

4.2.5 User Behavior with Resumption Aids

As predicted, the subjects used the resumption aids in different ways. For the DOI treeview, subjects treated them like tabs in the IDE. That is, when they could not explicitly remember which method contained the code they had in mind, they followed the cue-seeking strategy to click from one to another until they found the desired code. Subjects with the content timeline followed a systematic review strategy. They looked at the most recent document entries for context, completing any work within the open document. Then, they would return to the content timeline to rewind further back to review their previous actions before their last interruption point. From this review, subjects would often see the next place they wanted to return to.

Although our study found no difference in task completion rates between the DOI treeview and content timeline, subjects rated the DOI treeview significantly lower on overall preference. One explanation is that the code was new to them, and they therefore were less familiar with the program element names displayed in the timeline, possibly lowering the effectiveness of a cue-seeking strategy.

The subjects strong preference for the content timeline can be explained by its better affinity with episodic memory. Because it presented fragments of code as it appeared in the editor and in the same order as their interactions, they could better recognize and recall information from episodic memory even without having any previous experience with the source code. Episodic memory is especially suited for storing and recalling new experiences, but tools need to have interfaces that properly exploit it.

Although subjects did use the resumption aids 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. This suggests notes remain an important mechanism for suspending interrupted tasks. As one participant explained how his note-taking habits would change with a resumption aid:

> *I wouldn't have to write down the location, just what needs to be done.*

4.3 Desired Tools

To gain more insight for future research efforts, I analyzed developers' responses to the previously administered survey about what features they would like in future tool development. I drew the selection of future tools from current research proposals for improvements to programming environments in order to determine if programmers were primarily interested in to tools help them manage task state or tools to augment cues within the programming environment.

The first set of features support resumption strategies and can be grouped by memory type supported.

- **Automatic Tags** (*attentive*). A tag cloud of links to source code symbols.
- **Code Thumbnails** (*associative*). Thumbnails of recent places I navigated or edited.
- **Instant Diff** (*associative*). Highlighted code showing how I changed a method body as well as providing a global view.
- **Activity Explorer** (*episodic*). Historical list of actions I undertook such as search, navigating, refactoring. Expanding item gives thumbnail of IDE action or code location.
- **Change Summary** (*episodic*). Short summary of my changes.

- **Snapshots/Instant Replay** (*episodic*). Timeline of screen-shot thumbnails or an instant replay of my past work.

We also asked about features that would support suspension strategies:

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

In Figure 9, we display comparative ratings of tool value from surveyed developers. Overall, developers desired features that supported resumption strategies over suspension strategies. This largely stemed from wary perceptions that the described tools for suspension strategies might be too heavy weight. Developers gave high ratings to tools that could support a systematic review such as **Instant Diff** or **Activity Explorer**, but not to ones that do so in a too detailed manner, such as **Instant Replay**. For tools supporting cue-seeking, such as **Automatic Tags** or **Code Thumbnails**, developers favored tools that depicted more realistic renditions (thumbnails) over abstract ones (tag clouds). This preference is consistent with underlying mechanisms of associative memory which works more effectively with concrete stimuli.

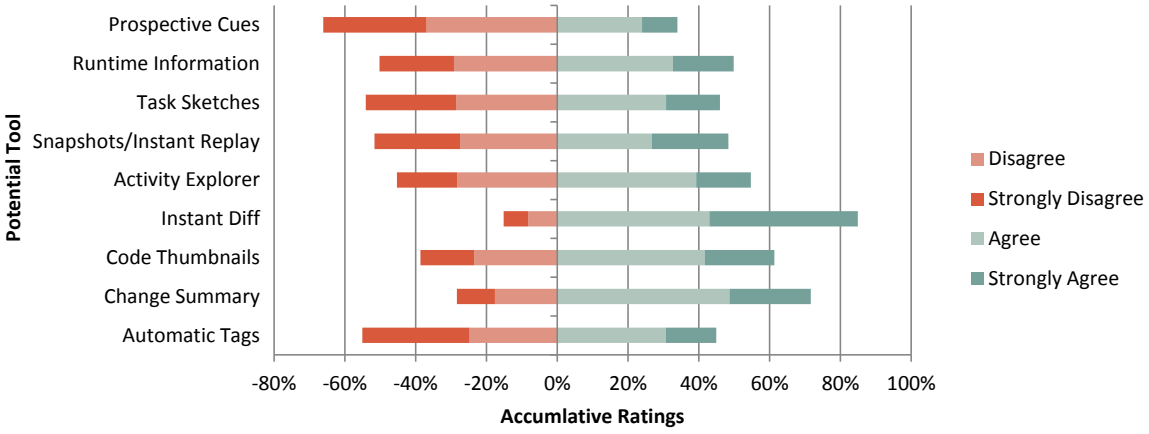


Figure 9: Survey ratings of possible resumption features, on a 5-point Likert scale, with neutral ratings removed.

4.4 Lessons Learned and Limitations

The exploratory studies allowed me to gain insight into the behaviors, practices, and sources of information developers used in interruption management and establish preliminary evidence that introducing interventions via visualization could aid developers in interruption recovery.

From my analysis of recorded history and surveys, I reached a deeper understanding of how interruption affected developers and the strategies and sources of information they used in recovery. I also found several deficiencies with available tools and source of information supporting those strategies. I also received feedback on possible improvements of those tools.

From my lab study, I demonstrated a decrease in task performance and an increase in failures when programmers were interrupted. I also demonstrated an increase in task performance and an decrease in failures when those programmers were given a resumption aid (DOI treeview or content timeline). I did not observe a performance difference between the two resumption aids, and, due to the exploratory nature of the study, I could not fully determine under which circumstances one would strongly outperform the other.

The design of the resumption aids influenced user behavior. Developers used the DOI treeview in a manner consistent with the cue-seeking strategy, exploring with a quick but less thorough investigation resulting in occasional failures. Developers used the content timeline in a manner consistent with the strategic review strategy resulting in slower but more thorough investigation, and no observed failures.

The design of the resumption aids had several limitations. First, the collected history was constrained in what events were captured and displayed: fixed to changes and visit actions, excluding the many other actions available in a programming environment. Second, for the content timeline, there was no way to consolidate or abstract any of the events in the history—not an issue for my study—but a big issue for scaling to real-world programming tasks that might have several hours or days of accumulated history. Third, for the DOI treeview, there was no way to distinguish which actions occurred at a program element or segment locations by different activities. Also, the developer did not have the ability to mark or annotate a code location.

Finally, due to the nature of the exploratory study, elements were designed without a strong

conceptual model in place. For example, there is no consideration of how to support interruption of prospective, associative, or conceptual memory. Further, there was little consideration given to the type of programming activity that would be supported by a particular resumption aid.

In the next chapter, I describe a conceptual model that considers specific types of memory failures and information needs of various programming activities from which I could design a more complete set of memory aids.

Chapter V

CONCEPTUAL FRAMEWORK

In this chapter, I present a conceptual framework for recovering from interrupted programming activities based on different types of memory failures. The conceptual framework provides a *conceptualization* of human memory rather than a *model* of human memory. That is, I provide a set of concepts that are useful for expressing common concerns related to memory failure rather than models that express the mechanics of human memory. I then derive information needs that are a result of these memory failures.

The information needs explain the information seeking and preserving behavior of interrupted programmers. The information needs are then used to guide the development of targeted memory aids. For each memory aid, I provide an information model that satisfies the information needs. The information models explains deficiencies in alternative devices previously proposed by researchers.

5.1 Programmer Information Needs

In this section, for each memory type, I first describe a programming activity and how it stresses that type. I then demonstrate how various memory failures affect developers in practice and the mechanisms they use to cope with those memory failures. Finally, I generalize these behaviors into information needs that each correspond to a particular memory failure. In Table 2, I summarize the information needs that result after memory failure.

5.1.1 Prospective Memory Support

5.1.1.1 Task: Resuming a Blocked Task

Developers often become blocked on a task: i.e., in a state where no progress can be made until an external constraint is resolved. For example, developers can become blocked when coordinating with other developers [103]. Other reasons include holding off on a task due to an unexpected

Table 2: Information needs and memory aids for different memory failures.

MEMORY	PROGRAMMING ACTIVITY	FAILURE	INFORMATION NEED	MEMORY AIDS
prospective	Resuming blocked tasks	Monitor failure Engage failure	Support monitoring applicability Provide multi. levels of engagement	*smart reminders*
attentive	Refactoring large code	Concentration failure Limit failure	Provide persisted and stateful focus Facilitate multiplicity	*touch points*
associative	Navigating unfamiliar code	Retention failure Association failure	Provide distinguishable features Support indexing by multi. modalities	*associative links*
episodic	Review and recall recent code changes	Source failure Recollection failure	Store context Support narrative	*code narratives*
conceptual	Program understanding	Activation failure Formation failure	Support priming Support abstraction	*sketchlets*

shift in scheduling priority or server/database down-time [152]. Regardless of how the developer becomes blocked, the consequence is the same: A blocked developer must later remember how to perform the task after a potentially lengthy interruption. Unfortunately, prospective memory's ability to prompt us at the appropriate time can be unreliable.

5.1.1.2 Developer Studies

Various studies have described how developers try to cope with prospective memory failures. For example, developers often leave TODO comments in the code they are working on [188]. To leave a TODO comment, a developer writes a comment beginning with the text "// TODO: Remember to fix ...", which can later be seen in a list of TODOs collected in a tool view such as the Task List. A drawback of this mechanism is that there is no impetus for viewing these reminders. Instead, to force a prospective prompt, developers may intentionally leave a compile error to ensure they remember to perform a task [152]. A problem with compile errors is that they inhibit the ability to switch to another task on the same codebase. Finally, developers also do what other office workers do [20]: leave sticky notes and send emails to themselves [152].

5.1.1.3 Information Needs

Monitor failures are a common reason why programmers fail to act on applicable prospective actions. Monitoring can be a cognitively demanding and distracting activity, especially in cases requiring the polling of an external condition, such as another team member's progress.

Information Need 1 - Programmers need facilities for monitoring and polling the status of external constraints inhibiting prospective actions.

Engage failures are a common reason why programmers fail to recognize the reminder for a prospective action. Passive reminders, such as sticky notes or comments in the code often fail to engage conscious attention.

Information Need 2 - Programmers need facilities for modulating their levels of engagement in prospective actions.

To address the shortcomings of existing coping mechanisms, I introduce the concept of a *smart reminder* (see section 5.2.1), which compensates for prospective memory failures by providing facilities for monitoring and polling external conditions and for modulating levels of engagement.

5.1.2 Attentive Memory Support

5.1.2.1 Task: Tracking Refactoring Changes

Some programming tasks require developers to make similar changes across a codebase. For example, if a developer needs to refactor code in order to move a component from one location to another or to update the code to use a new version of an API, then that developer needs to systematically and carefully edit all those locations affected by the desired change. Unfortunately, even a simple change can lead to many complications, requiring the developer to track the status of many locations in the code. Even worse, after an interruption to such as task, the tracked statuses in attentive memory quickly evaporate and the numerous visited and edited locations confound retrieval.

5.1.2.2 Developer Studies

Studies examining refactoring practices of programmers have found several deficiencies in tool support [136, 200]. One essential deficiency is the lack of ability to track the statuses of many locations in code. As a work-around, developers abandon refactoring tools and instead rely on compile errors that were introduced when refactoring. Interactive compile errors (which appear

or disappear automatically as a programmer makes changes) can represent task status well in an automated fashion: A correct change removes the compile error, whereas an incorrect change adds more compile errors. A programmer can be interrupted in this state and still have a means to continue the task. Unfortunately, using compile errors to track changes is not a general solution and can still lead to incorrect refactorings [71].

5.1.2.3 Information Needs

Concentration failures arise when programmers need to shift attention away from a programming task. Interruption to tasks can cause programmers to lose track of the status of previously attended locations of code.

Information Need 3 - Developers need support for a persisted and stateful focus on program locations.

Limit failures occur when programmers need to hold many items related to a programming task in attentive memory. Such tasks can often involve hundreds of program locations, a number well beyond the handful of items that attentive memory can support.

Information Need 4 - Developers need support for attending to numerous program locations.

In support of tasks that heavily tax attentive memory with many points of attention, I introduce the concept of *touch points* (see section 5.2.2), which supports maintaining status across many locations in code.

5.1.3 Associative Memory Support

5.1.3.1 Task: Navigating Unfamiliar Code

Some programming tasks require developers to explore and understand unfamiliar code. For example, if a developer newly joins a project or is assigned to fix a bug in an unfamiliar region of code, then he must become familiar with it. This includes learning new identifiers, locations, relationships, conventions, and behaviors. Such a task deeply taxes associative memory.

5.1.3.2 Developer Studies

Observations of developers suggest they frequently rely on associations with *environmental cues*—interface elements of the programming environment—for navigating and understanding new code. For example, Ko et al. [102] observed that programmers use cues, such as open-document tabs and scrollbars, for maintaining context during their programming tasks. However, these cues are often insufficient: The act of navigation often disturbs the state of environmental cues, and the paucity of interface elements, such as tabbed panes, which often only contain a file name, starves associability. In studies of developer navigation histories, a common finding is that developers frequently spend time flipping through open tabs because they fail to associate the tabs with desired code locations [180].

5.1.3.3 Information Needs

Retention failures occur when environmental stimuli do not offer sufficient features to trigger associative memory mechanisms. Unfortunately, for programmers, source code text is often visually repetitive, lacking highly distinguishing visual features. Further, interface elements, such as tabs, only provides one consistent associative feature: a file name (tab position is frequently unstable, making spatial positioning an unreliable associative feature).

Information Need 5 - Developers need diverse and distinguishable user interface features for building associations with code locations.

Association failures occur when incomplete or weak associations are formed. For example, when programmers are interrupted after exploring new code, it is common for developers to associate a block of a code with semantic information, such as functionality, but fail to form strong associations with details, such as name or location [148]. As a result, developers often spend significant time relocating code after an interruption [152].

The *modality* of sensory input can have a strong influence on associative memory. Modality is a channel for processing information, such as sound or color. Multiple modalities provide a good opportunity for forming associations on input.

For a code location, some example modalities include:

- *lexical*: alphabetic combinations, i.e., the name of the code location.
- *structural*: relation in program element hierarchy. For example, the containing project and package.
- *spatial*: physical position in programming interface. For example, indicating the coordinates of a location relative to the document.
- *operational*: user action taken at the source code location. For example, a search or edit action associated with the location.
- *syntactical*: grammatical structure and content of the source code location. For example, source code text and its associated structure (for statements, if statements, etc.).

Information Need 6 - Developers need support for indexing into associative memory via multiple modalities in order to recall code locations.

To address these needs, I introduce *associative links* (see section 5.2.3), which are memory aids that provide distinguishable features and indexing by multiple modalities. In particular, I show how a code tab can be made more associable using alternative modalities.

5.1.4 Episodic Memory Support

5.1.4.1 Task: Reviewing and recalling code changes

Software developers often need to recall recent coding experiences after an interruption. For example, remembering recent changes in order to write a commit message, vetting proposed changes with a teammate, performing a systematic review of recent changes, documenting a coding experience as a blog post, or preparing for a code review. Developers often forgot information when reviewing code changes. For example, developers commonly forget that they had refactored code when writing commit message [136].

5.1.4.2 Developer Studies

A common strategy developers use for recovering from episodic memory failures is to use source control history in order to perform a systematic review of changes [152]. However, developers

complain of the problems they have with using existing diff tools including that the information provided is unordered, verbose, time-consuming to peruse, and cognitively demanding.

Studies of programmers have found that presenting information about a past programming session in an episodic manner [170, 148, 81] improves recall of the session. Similarly, studies that examined recall of life experiences have shown that when a sequential display of events (pictures) from a past experience is given, that presentation can be more effective at encouraging recall than when other contextual details (names or locations) are given [170, 85]. Also, recall is boosted when the pictures are combined with more contextual elements (such as street locations on a map). Finally, psychology studies have shown that presenting information in a narrative form is a useful learning strategy for intermediate learners [211].

5.1.4.3 Information Needs

Source failures are common when learning new experiences. For example, a programmer may undergo a source failure if she knows that she copied code from an online example, but cannot recall the origin of the code.

Information Need 7 - Developers need support in retaining contextual details about their programming experiences.

Developers often need to recollect a past programming experience. After returning to an interrupted learning experience, a developer may need to reflect on her current status. Developers also need to tell stories in different ways, and they also occasionally need to relate their programming experiences to a colleague who wants to perform a similar task. In all three cases, it is difficult to provide a faithful account of how the programming task was done, resulting in *recollection failures.*

Information Need 8 - Developers need support in recollecting personal and social narratives of their learning experiences.

To address episodic memory failures, I introduce the concept of a *code narrative* (see section 5.2.4), which support developers in retaining and recollecting contextual details and narratives about their learning experiences.

5.1.5 Conceptual Memory Support

5.1.5.1 Task: Program Understanding

Developers are expected to maintain expertise in their craft throughout their careers, while tackling new paradigms of emerging technology and revisiting the architectures and concepts of previously developed programs that need maintenance. Furthermore, for experts trying to become skilled in a new domain, like the desktop developer becoming a web developer, there are many concepts that must be put aside and new ones learned. When understanding a program's source code, forming new concepts or activating previously learned concepts is an essential task.

5.1.5.2 Developer Studies

Studies suggest that sketching, diagrams, and note-taking are important ways for developers to capture and conceptualize development knowledge. Sketches are used throughout the lifetime of a project, expanding to include different facets and migrating to different media along the way [206]. Diagrams are used to form and retain early concepts [39]. Note-taking is a also commonly employed strategy to retain information about a programmed task. However, notes can often be incomplete and as a result lead to resumption failures [148].

5.1.5.3 Information Needs

Formation failures occur when a concept has not been consolidated into conceptual memory, which may require several months. Until that time, developers use intermediary devices such as notes or sketches to assist in viewing and reasoning about concepts. However, these devices are expressed with media that are neither long-lasting nor linked into the software system.

Information Need 9 - Developers need support in annotating and abstracting code as intermediary steps to forming concepts.

Activation failures occur when a concept has not been recently used, lessening a programmer's ability to use it. Developers that have been interrupted from a programming task after a long time away from it, need to refresh the concepts associated with the task.

Information Need 10 - Developers need support in reviewing relevant concepts in order to promote priming.

To address conceptual memory failures, I introduce the concept of a *sketchlet* (see section 5.2.5), which support developers in abstracting and refreshing concepts about source code.

5.2 Memory Aids

In this section I describe memory aids that address the information needs articulated in the previous section. Each memory aid is presented in terms of the information needs served and alternative devices that address the same needs.

Information needs are translated into an *information model*, which describe the concepts and their relationships required for representation. To present the information model of a memory aid, I use a simple subset of UML notion [1]. Figure 10 lists the notations used. A *class* represents a modeled concept. An *attribute* is a data element in the class. An *operation* is an action provided by the class. A *generalization* is a relation (denoted by triangle endcap) between two classes, where a child class inherits attributes and operations from a parent class. A *composition* is a relationship between two classes, describing how one class comprises another. Multiplicity of a participant is expressed near each endcap (denoted by a number, or * for unbounded). An *association* is a general relationship between two classes.

In the next chapter, details about the visual design and implementation of the memory aids are provided.

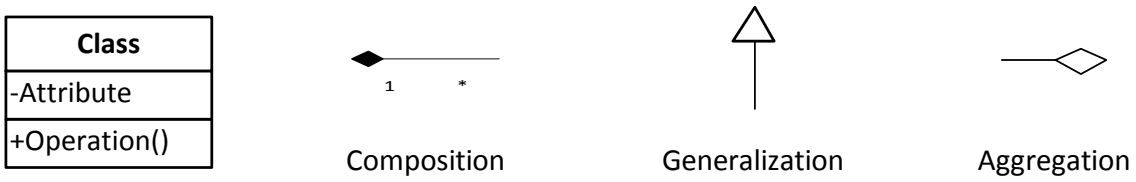


Figure 10: Elements of UML notation used in Information Models.

[1] `http://www.uml.org/`

5.2.1 Smart Reminders for Prospective Memory

5.2.1.1 Information Model

A *smart reminder* is a prospective memory aid that enables a programmer to condition the timing and modulate the level of engagement provided by a reminder. **SMART REMINDERS** are composed of: a reminder condition, a notification mechanism, and a reminder message. The *reminder condition* is an objective determining the applicability of a reminder. The *notification mechanism* is a device by which the reminder is conveyed to the user. The *reminder message* is a notice encoded in text.

To support **Information Need** 1, **SMART REMINDERS** can be created with a reminder condition that monitors its applicability. Studies of prospective memory show that using conditions, such as entrance to the physical space related to a task, can be an effective strategy [118]. To support such strategies, I have created *proximity conditions*, which condition the display of a reminder based on proximity to relevant code locations, such as a class or namespace path. To support monitoring of external conditions, I have devised several domain-specific conditions that check on task completion in a task tracker and checkins of source files into a code repository. Ultimately, a rich space of reminder conditions exists, tailorable to different programming environments, team compositions, personal preferences, and software development processes.

To support **Information Need** 2, **SMART REMINDERS** can be created with a notification mechanism that varies in strength. *Passive notifications* do not force attention, but remain passively visible until dismissed. For example, I have created **SMART REMINDERS** that are persistently visible in the lower righthand corner of the editor viewport (the viewport is always visible regardless of scroll position of the editor). In contrast, *obstructive notifications* force immediate attention of a programmer until dismissed. *Constrictive notifications*, do not directly force attention, unless a programer attempts to proceed with a certain activity. For example,**SMART REMINDERS** can be shown when a developer attempts to perform activities such as a checkin, program build or program execution. Finally, it is possible to specify notifications that transition between or blend these different levels.

The information model for **SMART REMINDERS** is presented in Figure 11.

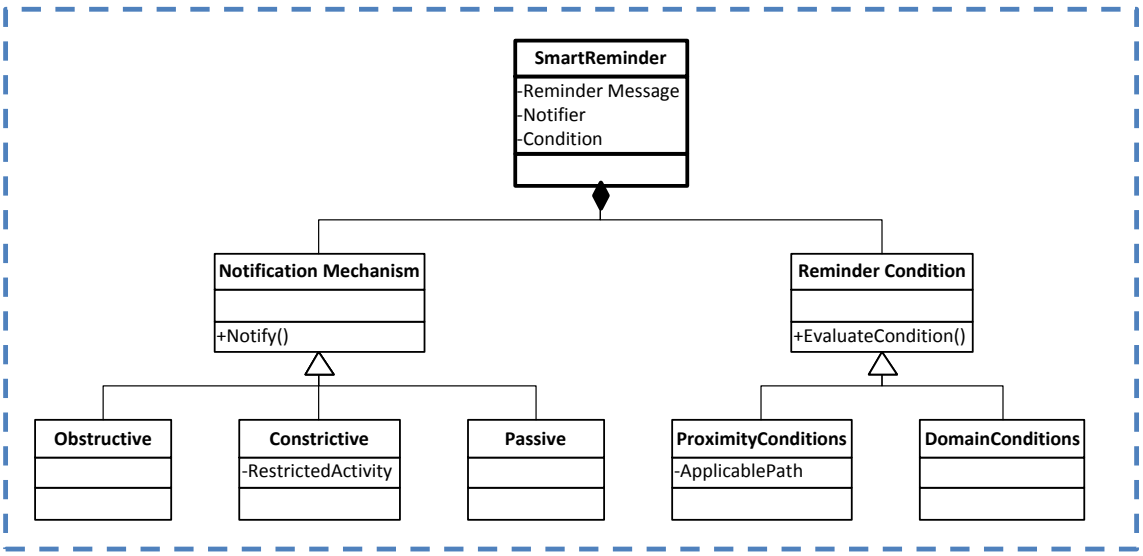


Figure 11: Information model for **SMART REMINDERS**.

5.2.1.2 Related Devices

- *Todo*: a reminder message embedded in code.
- *Roadblock*: an intentional compile error that must be addressed before compiling a program.

Todo comments represent the reminder message implicitly, but do not support engaging a user's attention or conditioning the display of the reminder. As such, Todo comments often get treated as documentation, with its known limitations, and not as prospective reminders. Roadblocks can be viewed as constrictive notifications but not as the other types of a smart reminder.

5.2.2 Touch Points for Attentive Memory

5.2.2.1 Information Model

Touch points are attentive memory aids that enables a programmer to maintain persisted and stateful attention to programming elements. A *programming element* is a named entity in a program's source code such as a sequence of statements, projects, or methods. A programming element can also refer to a statement with an internally specified name.

To support **Information Need** 3, a touch point tracks information about an element's state and can be further highlighted and annotated. To track, a touch point maintains internal state about edits, visits, and time since the last edit and visit. This internal state helps programmers track and filter **TOUCH POINTS** that have not been attended to, and review the ones that have. Finally, highlighted and annotated track points enable developers to preserve a long-term focus on problematic areas of code.

To support **Information Need** 4, **TOUCH POINTS** can be hierarchically organized and grouped. **TOUCH POINTS** can be expanded and collapsed based upon structure of the tracked programming elements. Groups of **TOUCH POINTS** can be created, merged, and split to reflect different investigations.

Finally, there are several ways to automatically create **TOUCH POINTS**. A group of **TOUCH POINTS** can be created interactively from the result of keyword or structured searches or based on recently recorded programming activity.

The information model for **TOUCH POINTS** is presented in Figure 12.

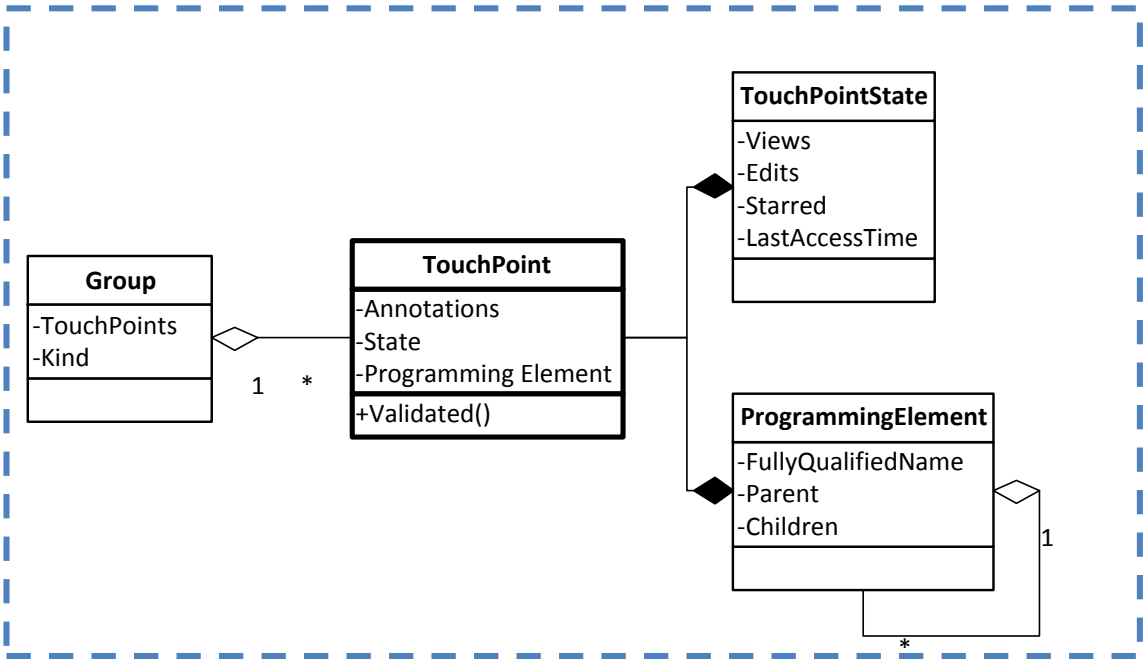


Figure 12: Information model for Touch Points.

5.2.2.2 Related Devices

- *bookmarks*: statements that have been flagged by the user.
- *task context*: a tree-like collection of programming elements, excluding statements, weighted by frequency of activity with a programming element [98].

Bookmarks are designed to indicate points of interest but do not scale well, whereas **TOUCH POINTS** are designed to handle ephemeral explosions of demand on attentive memory. Like task context, **TOUCH POINTS** can be manually specified or automatically generated from programming activity. **TOUCH POINTS** differ in that they are tree-like collections of programming elements, including statements indicating activity, annotations, and issues. That is, they support managing multiple locations at fine granularity and the tracking of progress.

5.2.3 Associative Links for Associative Memory

5.2.3.1 Information Model

An *associative link* is a memory aid that helps a programmer form and recall associations by providing distinctive features and multi-modal indexing. In addition to a code location, an associative link has a modal property. A *modal property* is information about the code location or an event undertaken by the programmer at the code location that emphasizes a specific aspect of interest.

For supporting navigation within unfamiliar code, **ASSOCIATIVE LINKS** can be used to improve the retention and recall of code locations from a tabbed pane containing code. In most program development environments, a tabbed pane provides only a lexical association to a code location. That is, the tabs only show a file name.

To support **Information Need** 5 (retention), three additional **ASSOCIATIVE LINKS** are added to tabbed panes: operational, syntactical, and structural. The operational associative link provides information about the last programming action that the programmer undertook at the code location, such as an edit or search. The syntactical associative link provides a thumbnail of the code in the current document viewport. The structural associative link provides a subset of the programming element hierarchy containing the code location. Overall, the presence of the

additional modalities may encourage the formation of associative memories, as there are more distinctive elements present during the act of navigation.

To support **Information Need** 6 (recall), modal queries can be used to recall code locations. For example, it is common for several tabbed panes to be opened after performing a search or stepping through a program when debugging. Using associative queries based on operational associations, a programmer can filter out tabbed panes that were used for debugging and show only the ones that were visited from a search. By scanning the list of thumbnails in the tab bar, a programmer can use syntactical associations to recall the code location. By examining the partial hierarchy of programming elements, the programmer can use structural associations, such as the namespace or project location, to recall the desired code location. Overall, **ASSOCIATIVE LINKS** allow multiple modes of indexing into code locations to improve access.

The information model for **ASSOCIATIVE LINKS** is presented in Figure 13.

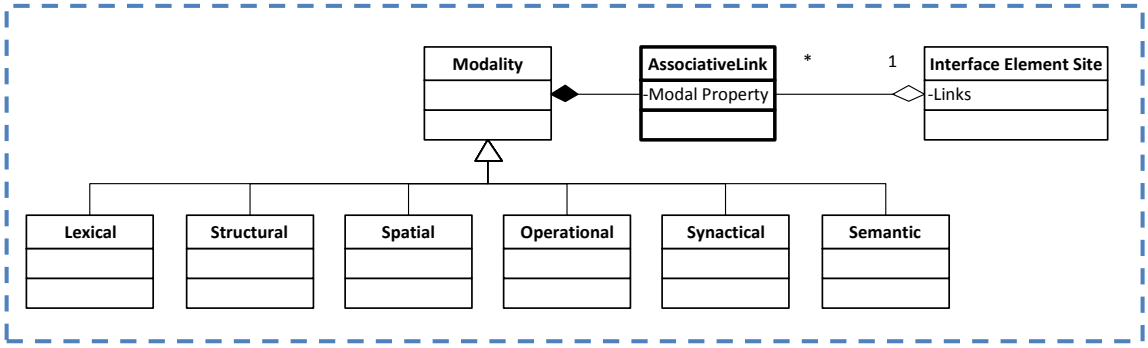


Figure 13: Information model for **ASSOCIATIVE LINKS**.

5.2.3.2 Related Devices

- *NavTracs*: a set of files associated by frequent co-visitation [180].
- *Code Canvas*: a fixed layout of source code content, associating each file with a spatial position on a zoomable plane [52].
- *Code Bubbles*: a dynamic layout of source code fragments, associating each fragment with a spatial position on a scrollable plane [25].

There are several related devices that exhibit characteristics similar to **ASSOCIATIVE LINKS**. NavTracs provide ways of indexing into code locations via operational modality, specificly navigation actions. By redesigning the entire programming interface, both Code Canvas and Code Bubbles provide ways of indexing into code locations via spatial modality. Nevertheless, none of these devices systematically consider which modal properties to support in the context of forming associative memories, nor provide multiple modalities for improving access.

5.2.4 Code Narratives for Episodic Memory

5.2.4.1 Information Model

A *code narrative* is an episodic memory aid that helps a developer recall contextual details and the history of programming activity. It is composed of a stream of programming events woven into a narrative structure. A *programming event* is an action performed in a programming environment (editing a method, performing a search, or debugging a program) and its resulting outcome (new method text, search results, or run-time exception). A *narrative structure* provides a schema for anticipating and organizing events. Narrative structures are socially constructed [100], meaning certain groups, such as developers, have their own learned narrative structures. Based on our study of how developers present their learning experiences on blogs, Treude and I found two common narrative structures used by developers: *overcoming obstacles* and *teaching tutorials* [154].

To support **Information Need** 7, a programming environment is heavily instrumented, such that, in addition to recording a stream of programming events, contextual details such as code snapshots, search terms and results, addresses of code samples, and stack traces are retained.

To support **Information Need** 8, the stream of events are organized into a series of episodes. An *episode* is an abstraction of a series of events, as defined by the narrative structure's schema. For obstacle narrative structures, the following details are populated: setting, conflict, investigation, and resolution. The *setting* is an overview of files encountered and programming tasks undertaken. The *conflict* is the encountered problem, such as a runtime exception, that prevented a task from being completed. The *investigation* is the series of programming events used to determine the problem. The *resolution* is the series of programming events that solved the conflict.

For tutorial narrative structures, the following details are populated: setting (a series of alternations between procedure and code snippet) and conclusion. The *procedure* is a textual description of how code is changed and where the change was made. The *code snippet* is a set of source code lines that was created as a result of the procedure. The *conclusion* is a textual description of limitations or future directions related to the procedure. I have prototyped algorithms that semi-automatically populate a series of programming events into a tutorial-style narrative and publish it as a blog post. These details are discussed in the next chapter.

Finally, a distinction is made between **REVIEW NARRATIVES** and **SHARED NARRATIVES**. When a narrative is shared, care must be taken so that it is understandable by others, who may lack context. Therefore, **SHARED NARRATIVES** tend to have a flat structure, as events must be related in strict order. In contrast, **REVIEW NARRATIVES**, can leverage existing episodic memories, enabling a programmer to move fluidly through his own experiences. In support of **REVIEW NARRATIVES**, I allow coding details to be organized hierarchically, by clustering and grouping programming events into programming activities, supporting improved indexing.

The information model is presented in Figure 14.

5.2.4.2 Related Devices

- *information quests*: are a collection of files visited, annotated with an information seeking goal and shared with others [76].
- *code replays*: are a stream of change events that can be replayed and shared with others [81].

Information quests provide a mechanism for sharing and visiting files during a programming experience, but not for relating a general narrative or contextual details about the experience. Code replays and **CODE NARRATIVES** both share a stream of change events. However, **CODE NARRATIVES** include additional programming events, such as navigation and search, and further organize those events into a narrative structure. Code replays provide an excellent mechanism for recollecting an coding experience as a "flash-bulb experience". However, for a programming task that can span several days, a code replay can overwhelm a programmer with an excessively long and unstructured sequence of code changes.

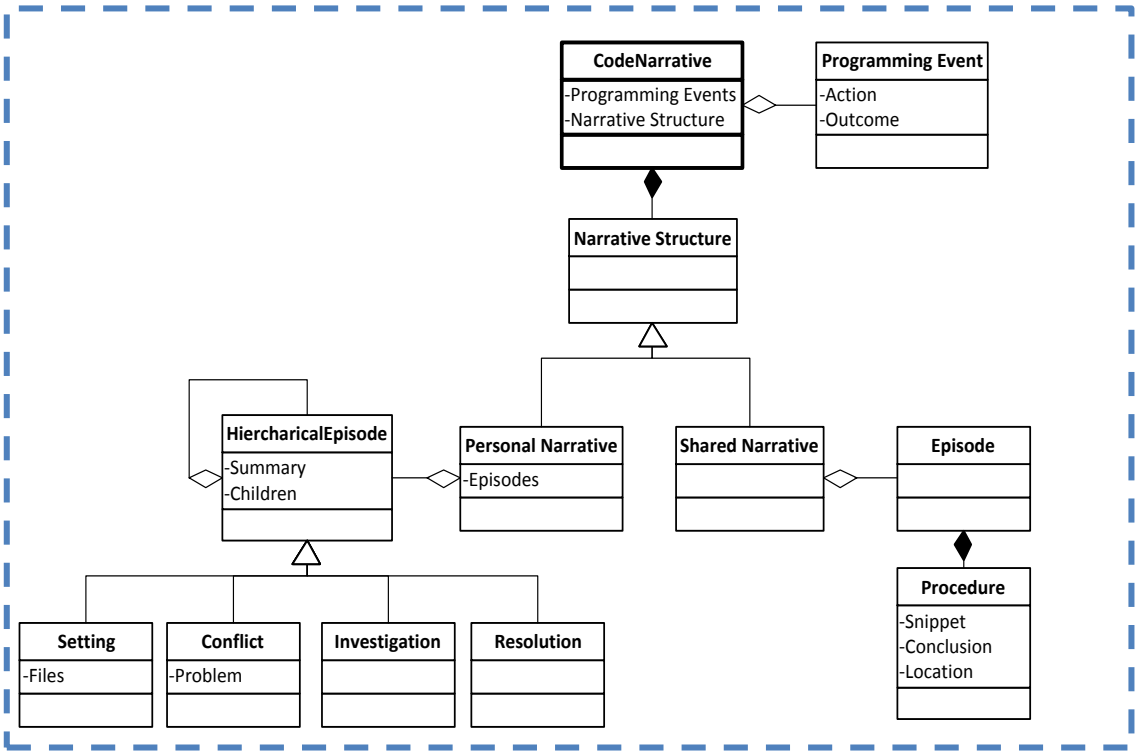


Figure 14: Information model for **CODE NARRATIVES**.

5.2.5 Sketchlets for Conceptual Memory

5.2.5.1 Information Model

A *sketchlet* is a conceptual memory aid that helps a programmer form and prime concepts by supporting abstraction and reviewing concepts that need to be refreshed. A sketchlet is composed of a programming element, an overlay, and a set of interactive spaces. An *overlay* is a set of annotations projected onto the programming element. An *interactive space* [157], is a container for an alternative representation of the programming element, such as a sketch or diagram.

To support **Information Need** 9, annotations and abstractions are provided. Annotations can be viewed in conjunction with a programming element. Interactive spaces can be used to provide other abstract representations of a program element. Interactive spaces can be used to view abstractions of other programming elements, enabling a programmer to represent abstractions among programming elements.

To support **Information Need** 10, programming elements are tracked for recency of interaction by a developer. The recency information can be used to calculate how much conceptual decay may have occurred. User interfaces can use this to provide visual cues indicating non-recency in order to encourage the reviewing of a relevant workspace or automatically display annotations.

The information model for **SKETCHLETS** is presented in Figure 15.

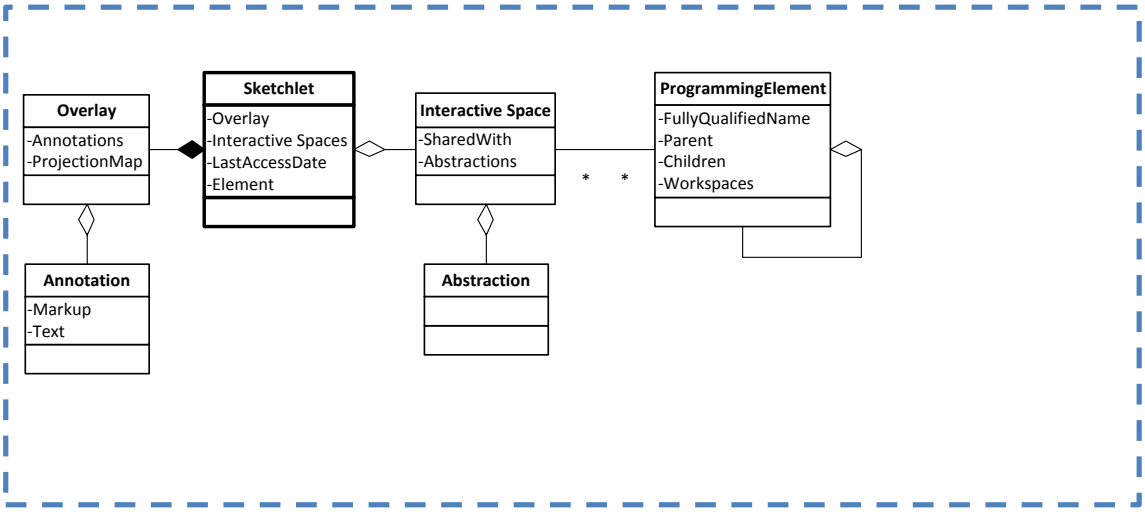


Figure 15: Information model for **SKETCHLETS** with example annotations, abstractions, and workspaces.

5.2.5.2 Related Devices

- *ConcernMapper*: a set of concerns organizing projections of programming elements [169].
- *TagSea*: a set of hierarchical tags organizing annotated source code lines [189].
- *Code folding*: a set of hierarchical directives to the IDE controlling the visibility of source code lines.

In ConcernMapper, a concern allows a single abstraction to be built over many programming elements; whereas **SKETCHLETS** enable abstractions to happen at a finer granularity. In TagSea, a tag is a subset of the possible annotations provided by **SKETCHLETS**. Code folding interleaves organization with source code; whereas **SKETCHLETS** provide overlays and alternative workspaces.

In contrast with **SKETCHLETS**, these devices do not integrate developer's sketching-like behavior, nor consider which organized knowledge may need to be primed. **SKETCHLETS** can support program understanding by offering separate contexts for different concerns, annotating code and interim understanding, and maintaining simultaneous perspectives of code and abstractions.

5.3 Summary

Based on a series of empirical studies and reviews of cognitive neuroscience literature, I derive a set of information needs for recovering from interrupted programming activities. My conceptual framework structures the information needs in terms of five memory types particularly relevant to programmers: attentive, prospective, associative, episodic, and conceptual. My conceptual framework can be used to explain the information seeking and preserving behavior of interrupted programmers who are attempting to recover from a memory failure and be used to derive a set of memory aids that can support them.

In Table 3, the memory aids and supported information needs are listed.

Table 3: Memory aids that satisfy information needs of a particular memory type.

MEMORY	MEMORY AID	SUPPORTED INFORMATION NEED
prospective	smart reminders	monitoring and polling the status of external constraints inhibiting prospective actions. modulating their levels of engagement in prospective actions.
attentive	touch points	persisting stateful focus on program locations. attending to numerous program locations.
associative	associative links	diverse and distinguishable user interface features for building associations with code locations. indexing into associative memory via multiple modalities in order to recall code locations.
episodic	code narratives	retaining contextual details about their programming experiences. recollecting personal and social narratives of their learning experiences.
conceptual	sketchlets	annotating and abstracting code as intermediary steps to forming concepts. reviewing relevant concepts in order to promote priming.

Chapter VI

MEMLETS

Previously in Chapter 5, I described how programmer information needs can be derived from studies of developer information-seeking behavior and related to different types of memory failures. These information needs has been categorized into a set of memory aids, each specified by a description of relevant information needs and specified by an information model. In this chapter, I describe a set of *memlets*, that provide visual representation and operations over the previously described information models for the purpose of assisting developers in recovering lost information after an interruption. The described memlets represent concrete design choices in the design space described by the conceptual model.

For each memlet, I describe the information that needs to be collected and logged by the programming environment, the algorithms that are needed to populate the memlet, the graphical views available provided by the memlet, and the operations enabled by the memlet. Together, the memlets and the comprehensive history of programming events support recovery of missing information after memory failures.

In crafting the design of the memlets, there were several considerations used as guidance. First, based on the distinct nature of the memory types and the programming activities considered, I avoided integrating the visual designs of memlets and instead kept them as distinct tools in the IDE. A benefit of this separation is that each memlet can be studied in isolation. Second, based on my studies of programmers, I found that the ability to both preemptively (in the form of suspension) and reactively (in the form of information seeking) deal with interruptions was a useful strategy for programmers. Therefore, in the design, I accounted for memlet being used to create information interactively (for suspension prior to an interruption) and automatically (for recovery after an interruption). Finally, for **CODE NARRATIVES** and **SMART REMINDERS**, I used design surveys and feedback from early adopters to iterate on the design of the memlets.

In the next section, for each memlet, I describe

1. the data collected from the programming environment,

2. the population of the memlet's information model, including the algorithms for doing this automatically from the collected data,

3. the visualization and the interactions and an example workflow for using the memlet.

Finally, I identify other design decisions and document the reasoning for the decisions I made.

6.1 Smart Reminders for Prospective Memory

Programmers experience long-term interruptions from tasks that are blocked by external constraints. In these situations, programmers are likely to experience prospective memory failures because of failure to monitor the external constraints or notice reminders.

SMART REMINDERS help a programmer recall prospective memories by supporting the monitoring of external constraints and timely prompting of reminders. A *smart reminder* is composed of a reminder message, a reminder condition, and a notification mechanism. The reminder condition controls the applicability of the reminder, whereas the notification mechanism drives its prominence. For example, a developer may only want to see a reminder if he is in the vicinity of specific code, or a developer may only permit certain reminders to be obtrusive.

After exploring several combinations of notification mechanisms and reminder conditions, the following two smart reminder types were designed:

Attachable reminders A reminder that can be attached to one or more editor viewports. The reminder is only applicable when the developer has a particular viewport in focus.

Due By reminders A reminder that appears as a compile error if it is not completed by a specified due date.

The two smart reminder types differ in how they support prospective memory. Attachable reminders prompt repeatedly developers about a reminder, without being obstructive. Due By reminders do not prompt developers until the reminder deadline has been reached. In that case, the reminder constricts the developer's ability to compile the program. In terms of memory support, attachable reminders support a weak forms of monitoring and prompting, because

they still rely on developers to notice the reminder and decide to act on it, whereas Due By reminders support a strong form of monitoring and prompting, because monitoring can be handled completely by the smart reminder and the constrictive nature of compile errors.

6.1.1 Monitored Information

A smart reminder requires that data is monitored and collected from the programming environment. Determining if a reminder is applicable requires significant monitoring.

Three categories of conditions measure the programming environment state: temporality, proximity, and activity. Temporal-based reminder conditions trigger before a scheduled calendar date and time or session. For example, "Show on next day". To support temporality, the current time and timestamp of the most recent programming activity in the IDE is monitored.

Proximity-based reminder conditions trigger on the current position of a programmer in a document. Two levels of proximity are supported: a specific file or any file. For example, a developer can configure a reminder to "Attach here", meaning it will only appear if the developer is in the proximity of the current file.

Activity-based reminder conditions are triggered based on the current activity of the programmer (internal) as well as the remote activities of other programmers (external). Internal activities include: compiling, debugging, editing, and committing to a source code repository.

Attachable reminders use a combination of temporality and proximity when determining applicability, whereas Due By reminders use a combination of temporality and activity when determining applicability.

6.1.2 Population

SMART REMINDERS are created interactively, using a "auto-complete" interaction style. Whenever a developer creates a TODO message, they are prompted with the ability to promote the TODO note as a smart reminder, using a menu of options. To create a Due By smart reminder, developers use the following format: `// TODO BY <a date> message`, where date is a day of the week, today, tomorrow, next day, next week, or a specific date such as 5/31/14 (date format is flexible). Day of week and date format support internationalization.

In Figure 16, the autocomplete menu for creating different kinds of smart reminder is shown.

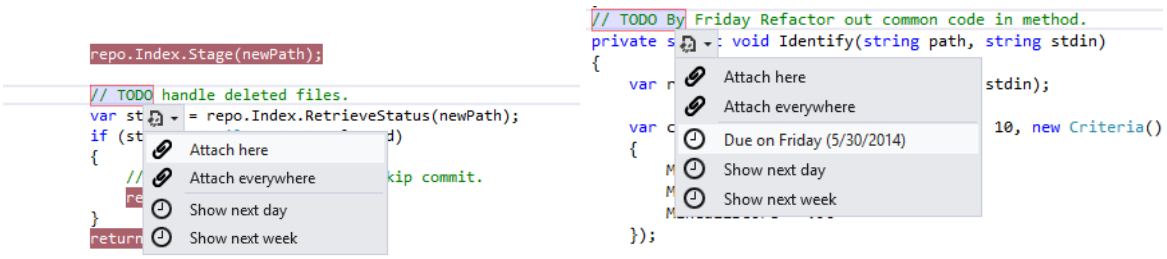


Figure 16: Attachable and Due By **SMART REMINDERS** can be created from TODO notes.

6.1.3 View

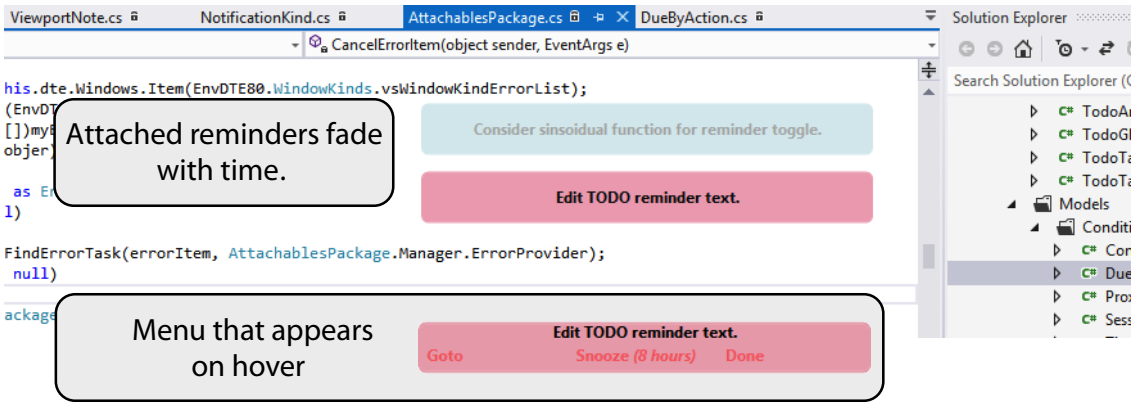


Figure 17: Attachable reminders displayed in the editor viewport

The display of **SMART REMINDERS** is driven by which notification mechanism is selected. Attachable reminders use a passive notification mechanism and Due By reminders use a constrictive notification mechanism.

6.1.3.1 Attachable Reminders

The design of passive notifications trades off noticeability and ambience. The presentation of attachable reminders catches attention without interfering with programming activities, essentially providing an ambient visualization [145]. The notification is displayed in the top right-hand corner of the code editor viewport, where the view is alway be visible, regardless of scroll position.

Justification of this design decision is found in Murphy-Hill's observation that the right-side of a code editor often does not contain any code [134]. Figure 17, shows the passive notification

mechanism. If there is more than one active reminder, they are stacked below each other. Notice, that if there is an exceptionally long line, the source code always has the highest z-order, meaning it would never be occluded by the reminders.

To prevent a reminder becoming too intrusive, an attachable reminder fades in opacity over time during a programming session.

6.1.3.2 Due By reminders

Constrictive notifications display when developers attempt to perform an associated activity. Due By reminders are shown when developers attempt to compile their programs. An example of an constrictive notifications is show in Figure 18, where a developer is prevented from building a program and must first interact with the reminder message.

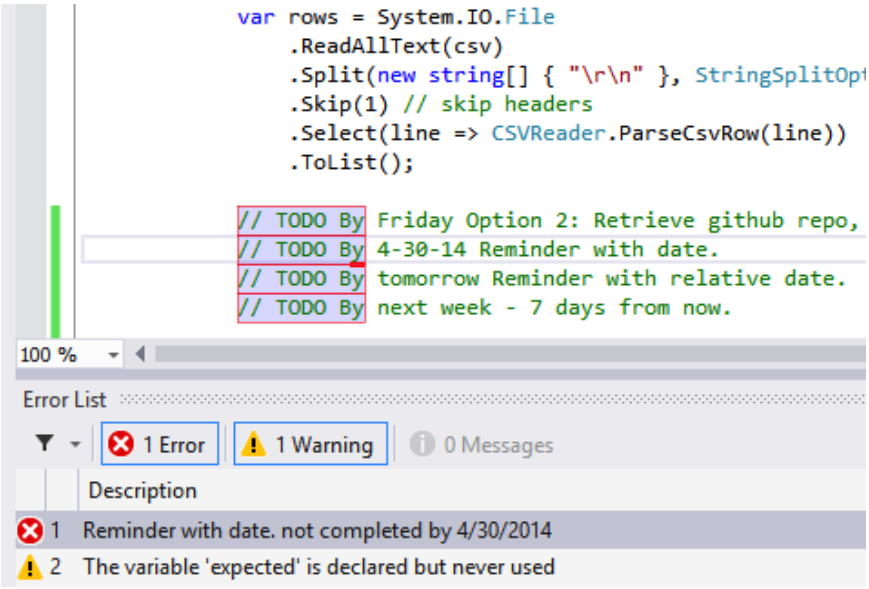


Figure 18: A Due By reminder shown as a compile error.

6.1.3.3 Operations

There are several operations available to assist the developer in interacting with **SMART REMINDERS**.

Autocomplete menu actions

Attach here: The attachable reminder is attached to the corner of the editor viewport and only displayed when you viewing the file.

Attach everywhere: The attachable reminder is attached to the corner of the editor viewport in all code windows.

Show next day: The attachable reminder will be displayed the next day in all code windows.

Show next week: The attachable reminder will be displayed the next week in all code windows.

Due on: The Due By reminder that if not addressed by a specified date will be displayed as a compile error.

Mark Complete: The reminder that has a due by date can be marked as complete from the menu.

Attachable hover actions Additional actions are available if you hover over an attachable reminder.

Goto: Goto the place the reminder was originally created.

Snooze: Hide a reminder for the next 8 hours.

Done: Complete and remove the reminder from the display.

6.2 Touch Points for Attentive Memory

Programmers often need to focus on many code locations. For example, during programming tasks like refactoring or applying a systematic change, a programmer may need to keep track of locations that exhibit complications or revisit locations to requiring a change. Maintaining focus on many locations taxes attentive memory by exceeding its limitations.

TOUCH POINTS enable a programmer to attend to many locations in source code during programming activities by persisting source code locations that need attention. A *touch point* is composed of a program element and its state, and may be aggregated by a set of group containers, which hierarchically organize the **TOUCH POINTS**.

6.2.1 Collected Information

The primary data collected from the IDE by my code history infrastructure are the set of source code lines visited, the program elements (methods, classes, files, directories) containing the lines, and the set of edits and navigations targeting the lines. Timestamps of the edits and navigations are also collected.

When programmers make systematic changes, they often use structured navigation, such as editing all references to a programming element or searching for all occurrences of a name. I collect the programming elements and queries and extract the resulting navigations from the list of results.

Alternatively, programmers may use unstructured navigation, such as switching code documents and scrolling to view code. To include all program locations involved in unstructured navigation would result in many spurious locations. Instead, I only track the changes to the position of the cursor, filtering out scrolling operations, keydown, and keyup movements. Note that the cursor does not change during scrolling. I also collect the word extent of the cursor, which provides more context about the cursor placement.

6.2.2 Population

The procedure for creating **TOUCH POINTS** is straight-forward. First, a touch point is created for each collected source code line and its associated program element. **TOUCH POINTS** are organized hierarchically in accordance to the program elements' structure. A touch point is classified as having been edited or navigated based on the collected history.

TOUCH POINTS provide the facility to organize a set of **TOUCH POINTS** into a top-level group node. Recall that attentive memory is more effective when it is paying attention to groups of similar items. Therefore, I sought several ways to automatically group **TOUCH POINTS**.

All of the **TOUCH POINTS** associated with an instance of structured navigation are organized under a single group node. The group node contains information about the structure navigation, such as the programming element or search term used to do the search. **TOUCH POINTS** associated with unstructured navigation and edits are also organized under a single group node. I had

considered using further heuristics to group these **TOUCH POINTS**. Previous research on programmer navigation suggests that word similarity and recency are strong predictors for navigation and edits of program locations, and thus are a promising heuristics to use for creating more groups. However, I decided to keep the initial design simple before introducing more complicated groups.

Another design issue: Should **TOUCH POINTS** that were edited and navigated be grouped separately or jointly? I decided to group them jointly, using the intuition that a programmer may have not finished changing a group of program locations, and that it would be more helpful if the changed and viewed locations were in the same group.

6.2.3 View

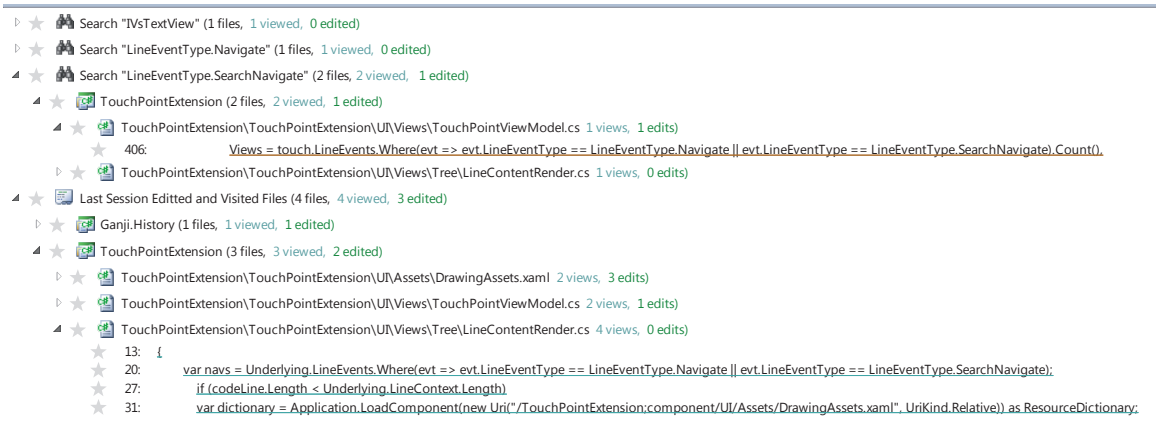


Figure 19: TOUCH POINTS created through auto-population.

Figure 19 shows a visualization for **TOUCH POINTS**. A collapsible treeview is used to represent the groups of **TOUCH POINTS**. Treeview icons distinguish the following kinds of nodes: group, project, file, and source code line. The top-level nodes are the group nodes. In the figure, the first three group nodes correspond to structured navigation activities. Notice, the third node with the label, 'Search "LineEventType.SearchNavigation"', includes source code lines that were navigated via search activity and then subsequently edited. Each parent node includes summary information about its children: the total number of files and the number of files that have been viewed or edited. I decided to exclude class and method nodes from the treeview, in order to avoid the extra overhead of navigating a too deeply nested tree for each touch point.

The leaf nodes show the source code lines of each touch point. The line node displays the line number and the file content at that line number. Double clicking on a touch point will navigate the editor's focus to the code location. I avoid using numeric or nominal values to indicate edit or visit state of a particular source code line to reduce clutter. Instead, lines that have only been navigated to are underlined in blue. Lines that have been edited are underlined in mustard. Finally, nodes can be starred by programmers to indicate mark important locations.

6.2.3.1 Operations

There are several operations available to assist the developer in interacting with **TOUCH POINTS**. There may be some scenarios where too many **TOUCH POINTS** have accumulated and, as a result, developers may need help searching through them. Nodes can be filtered based on starred, viewed, or edited state. Individual nodes can be hidden. Groups of nodes can be merged if similar code was accessed through different queries.

There may also be scenarios where a developer has not completed a systematic change, while applying changes locations listed in a search result query. Developers can execute the search query again and load non-visited locations and non-edited locations under the currently selected search group node.

Finally, there are several operations that support a programmer in specifying **TOUCH POINTS** interactively, i.e., before an interruption. Operations in the editor window allow a programmer to pin a source code line or to perform a structured query with the intention of viewing the resulting items in the touch point memlet. Search operations can be interactively created in a touch point memlet. Finally, annotations can be added to any node.

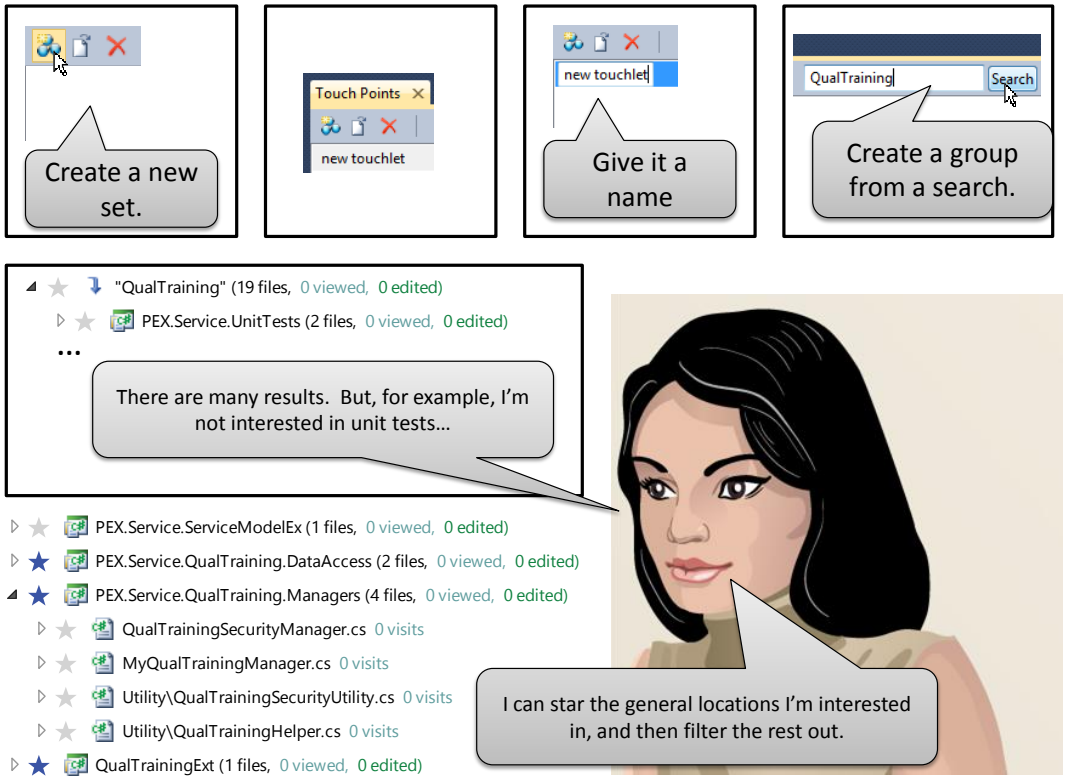
Table 4 summarizes the touch point's operations.

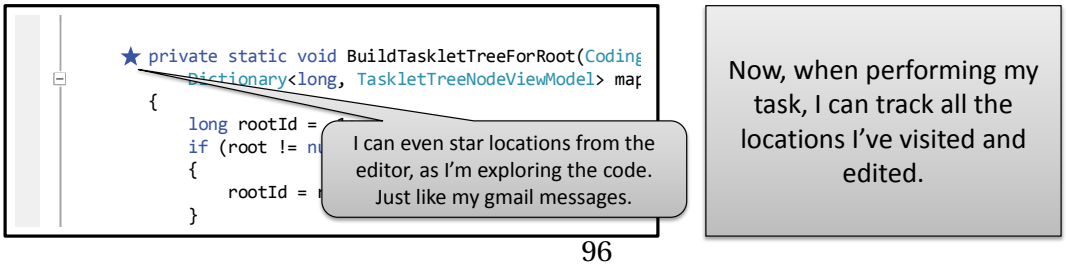
6.2.3.2 Workflow

In Figure 20, Ana describes how she uses **TOUCH POINTS** to track the status of a large systematic change task.

Table 4: Table of operations for **TOUCH POINTS**.

OPERATION	DESCRIPTION
Filter	Filtered **TOUCH POINTS** based on starred, viewed, or edited state.
Hide	Remove an individual touch point from view.
Merge	Join two groups of **TOUCH POINTS**.
Reload	Refresh the status of **TOUCH POINTS** visited via a search result.
Pin	Add a program element as an explicit touch point.
Annotate	Add a note to a touch point.





6.3 Associative Links for Associative Memory

Developers commonly experience disorientation when navigating to code that has not been recently visited. The disorientation stems from associative memory failures that arise when a developer must recall information about the places of code they are viewing or what to view next. Researchers believe the lack of rich and stable environmental cues in interface elements, such as document tabs, prevent associative memories from being recalled.

The presence of multiply modalities in a stimulus increases the ability to form an associative memory. In this sense, a *modality* refers to distinct types of perceptions that are processed by distinct regions of the brain, such as the auditory or visual pathways. Examples of different modalities include: lexical, spatial, operational, and structural. When multiple modalities are present in the same stimulus, more pathways are activated, thus increasing the chance of forming an associative memory. In contrast, a monotonous stimulus with a single modality is less likely to form an associative memory.

An *associative link* is an associative memory aid that helps a programmer to form and recall associative memories by situating information of multiple modalities with a program element. In particular, I am focusing on improving navigation by document tabs, which are often used for navigating between different program elements. The default configuration of document tabs in IDEs are especially spartan, with a single modality (lexical) showing the name of the document. Figure 21 displays a pane of document tabs in Visual Studio.

To make document tabs more associable, **ASSOCIATIVE LINKS** to syntactical, structural, and operational modalities of information are added. The following list reviews the definitions of the modality types:

- *lexical*: alphabetic combinations, i.e., the name of the code location.
- *structural*: relation in program element hierarchy. For example, indicating the containing project and package.
- *operational*: user action taken at the source code location. For example, a search or edit action associated with the location.

- *syntactical*: grammatical structure and contents of the source code location. Source code text and its associated structure (for statements, if statements, etc.).

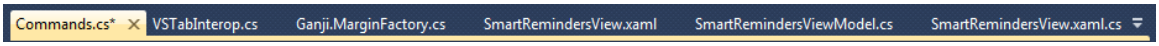


Figure 21: A pane of document tabs in Visual Studio, displaying document names.

6.3.1 Collected Information

To support **ASSOCIATIVE LINKS**, the primary information collected from the IDE is the set of open documents and their associated states that corresponds to operational, structural, and syntactical modalities.

To obtain operational state, I record the actions performed on a document, referred to as the *document actions*. Table 5, lists the document actions collected. Previous research [214] has found that documents are viewed more than they are edited in a typical session, suggesting that the documents that were edited and the different ways the document was viewed provide distinguishing features.

Table 5: Table of operations for **SMART REMINDERS**.

DOCUMENT ACTION	DESCRIPTION
Search	The document has been viewed as the result of a search operation.
Edit	The document has been edited and saved at least once.
Navigate	Movement within the document has occurred.
Debug	The document has been viewed as the result of a debug operation.

To obtain structural state, I record the logical placement of the document within its project's hierarchy. The hierarchy information indicates where to find the document in future interactions.

To obtain syntactical state, I record a snapshot of the current document's viewport. I decided against recording a snapshot of the entire document. Any visualization using the entire document would not likely contain any readable text or recognizable text patterns (such as blocks of if statements), and thus would not be syntactical in nature.

6.3.2 Population

ASSOCIATIVE LINKS are fully automated. However, there are a few additional steps that must be taken in populating and managing their lifecycles. Documents may have more than one document action, and some of those document actions may not be recent. When there is more than one document action, the primary documentation action is determined by the following order of precedence: edit, debug, search, navigate. Edit is always shown first because it is the only document-altering action. The remaining actions are ordered by frequency of occurrence (least frequent first) in order to maintain distinction. Further, document actions can expire. An exponential decay function is used to determines when a document action should be removed, with an additional guard that the document action at least persists to the next session. Finally, documents may be renamed or moved. Changes to documents are tracked and state is migrated to the new document.

6.3.3 View

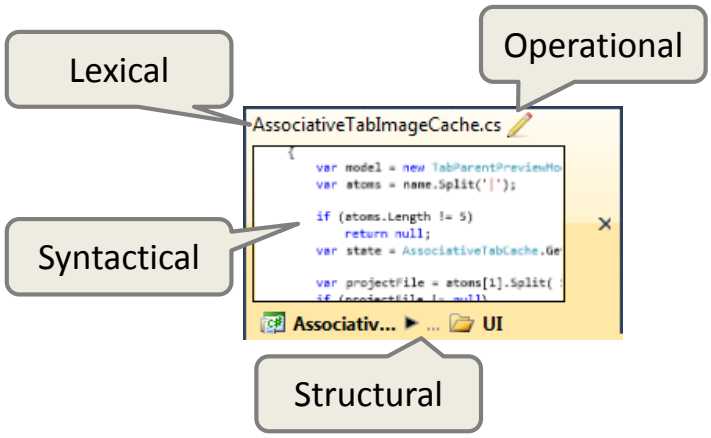


Figure 22: Different modalities provided by **ASSOCIATIVE LINKS**.

In Figure 22, a single associative tab is displayed and annotated to show which parts of the view correspond to which modality. The operational modality is indicated by an icon representing the primary document action. The structural modality is indicated by a partial breadcrumb trail. The structural modality is indicated by a thumbnail of a document's viewport.

An alternative design for presenting the structural modality was a treeview displaying the

document's hierarchy. However, there were several disadvantages to using a treeview. First, considerably more vertical space is required to display the entire depth of the hierarchy. Second, depths are variable between documents, resulting in wasted space and in poor alignment for other associative tabs. The partial breadcrumb trail resolves these problems by displaying the hierarchy in a single line, conserving vertical space. Instead of showing the full hierarchy, the partial breadcrumb trail displays the name of the highest ancestor and most direct parent of the document's hierarchy.

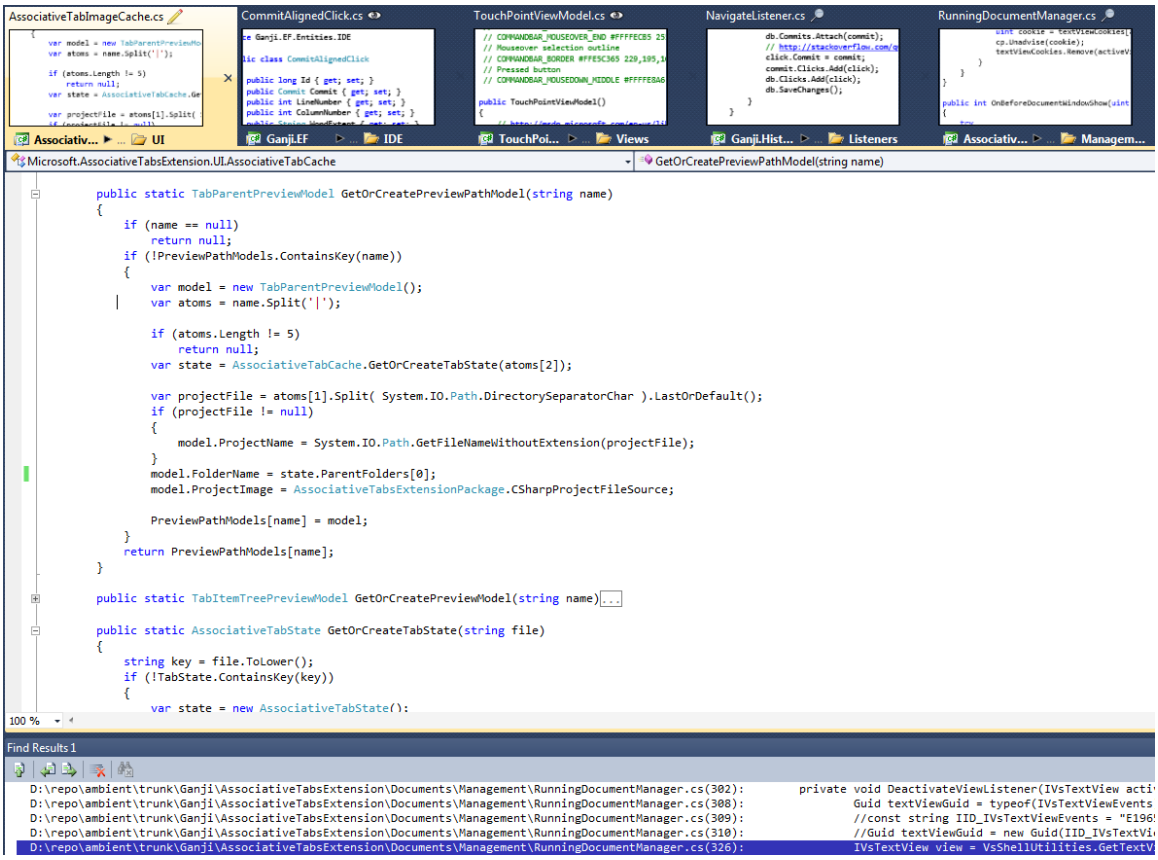


Figure 23: A screen shot of several **ASSOCIATIVE LINKS** in Visual Studio.

When rendering the thumbnail, a region of the document's viewport is selected and scaled at 66% and rendered in a 200x100 pixel image. The region is selected to avoid whitespace (in the left and right margins of the document). The scale used and thumbnail size is configurable.

Finally, there is a general issue concerning screen real-estate. Associative tabs consume

more vertical space than is available with the default document tabs in Visual Studio. For some workstations, this may not be of concern: Consider Figure 23, which is viewed from a 22 inch monitor. Ultimately, experiments and user configurations should determine the ideal number and placement of modalities used by an associative link.

6.3.3.1 Operations

Although **ASSOCIATIVE LINKS** work in an fully automatic fashion, there are a few operations that assist developers in working with **ASSOCIATIVE LINKS**, listed in Table 6.

Table 6: Table of operations for **ASSOCIATIVE LINKS**.

OPERATION	DESCRIPTION
Close All Same Document Action Group	Close all tabs with the same operational association.
Keep All Same Document Action Group	Keep all tabs with the same operational association.

6.4 Code Narratives for Episodic Memory

Software developers often need to recall recent coding experiences after an interruption. For example, remembering recent changes in order to write a commit message, vetting proposed changes with a teammate, performing a systematic review of recent changes, documenting a coding experience as a blog post, or preparing a code review.

Developers need to recollect this experience from their episodic memory. Developers commonly experience episodic memory failures that limit their ability to recall essential details or to recollect key events. For example, a developer may experience a recollection failure when they forget the changes they performed for a programming task, or a source failure when they forget details such as the URL of a blog post that was used for implementing a part of the task.

A common strategy developers use for recovering from episodic memory failures is to use source control history in order to perform a systematic review of changes [152]. However, developers complain of the problems they have with using existing diff tools including that the

information provided is unordered, verbose, time-consuming, and cognitively demanding. Finally, there is no preservation of the ordering of events, which makes recall more difficult.

A *code narrative* is an episodic memory aid that helps a developer recall contextual details and the history of programming activity. It is composed of a stream of programming events woven into a narrative structure. Two narrative structures are currently supported: A *review narrative* aimed at recapping a recent experience at a detailed level, including changes that were later deleted, and a *shared narrative* aimed at relating a experience to other developers in a high-level representation, focusing on presenting an step-by-step guide to the final changes achieved. **REVIEW NARRATIVES** are designed for recall tasks that involve personal reflection on a programming task, which is better suited for interruption recovery. **SHARED NARRATIVES** are designed for recall tasks that involve relating information about a programming task to another person, which is better suited for documentation or knowledge sharing tasks.

6.4.1 View

Based on design surveys with 100 developers and feedback from early adopters, I designed **CODE NARRATIVES** with a simplified process for creating and viewing **CODE NARRATIVES**. Instead of requiring a developer to specify a time frame upfront, a collapsible view shows all recent history. In early design prototypes, **CODE NARRATIVES**, regardless of its narrative structure, could be edited and annotated in a rich editor interface. Although developers in the survey believed that the rich editor could be useful, they wanted an option to have a more light-weight, text-based interface. Finally, developers felt their were certain types of information they would not be comfortable sharing in a public manner.

To streamline the design, **SHARED NARRATIVES** are generated using a plain-text mark up format, called markdown[1], that allows easy annotation of code snippets and that is supported by a wide variety of blogging platforms. **REVIEW NARRATIVES** are generated in a stand-alone file and displayed in the web browser, allowing rich interaction.

[1] `http://daringfireball.net/projects/markdown/`

6.4.2 Collected Information

To provide **CODE NARRATIVES**, several types of events must be collected from the programming environment. The different types of events reflect the many programming activities that a developer performs, specifically those that indicate interesting or significant events during a programming experience.

A *change event* occurs when a developer saves a change to a document. Changes to the source code documents are tracked by maintaining a repository of time-stamped snapshots of the documents. Source code differences are derived from the snapshots.

An *exception event* occurs when a developer experiences a run-time exception while debugging a program. Exceptions occur when an error in the executing program has occurred. Not all exceptions events are collected. By design, exceptions are commonly caught and handled by the program, and therefore collecting all these events would be superfluous. Instead, exception events are restricted to uncaught exceptions and exceptions that have exception handlers containing a debugging breakpoint. The reason for including the latter is that a programmer has explicitly indicated that she is interested in seeing if that particular exception has occurred by placing a breakpoint in the hander. Collected information from an exception event includes: the stack trace, the exception message, and the exception type.

A *web visit event* occurs when a developer has visited a webpage during a programming session. When developers do not understand how to do something, they often suspend their programming task and perform web searches for more information. Commonly, developers use resource such as question-and-answer sites such as `stackoverflow.com` or blog posts to find relevant code examples they can use. Additionally, web paste events are collected. A *web paste event* occurs when a developer copies code from a webpage and pastes it into the current project. Collected information from a web paste event includes: the url, website snapshot, and pasted code.

A *commit event* occurs when a developer checks code into a source control repository. A commit provides a natural milestone representing the progress of a programming experience. Collected information from a commit event includes the time of the commit, the commit message,

commit id, and the files affected by the commit.

A *search event* occurs when a developer performs a structured or unstructured search through the code base. Searches often occur during transition points or distinct periods of a programming task. A period of searching often occurs while a developer explores a program before editing code. Once a developer has started coding, searches also occur during transitions between heavy periods of editing, as the developer looks for the next regions of code to change. Collected information from a search event includes: the search term or programming element used for structured search and the visited locations.

In Table 7, a summary of the different types of events collected for **CODE NARRATIVES** are listed.

Table 7: Table of collected events for **CODE NARRATIVES**.

EVENT	COLLECTED INFORMATION
Change Event	File snapshot
Exception Event	Exception message, type, stack trace, and source code at the site of the raised exception
Web Visit Event	URL and title of visited page
Web Paste Event	URL, copied code, web page snapshot
Commit Event	Commit message and affected files
Search Event	Search term and visited locations

6.4.3 Population

The procedures for populating **CODE NARRATIVES** is determined by the respective narrative structures. To create **CODE NARRATIVES**, developers can bring up the view as shown in Figure 24.

6.4.3.1 Review Narratives

REVIEW NARRATIVES are designed to provide a detailed account of a recent programming experience. To populate **REVIEW NARRATIVES**, initially all collected events related to the programming experience are gathered. Events from external resources such as source code repositories or local browser history are also integrated into the collected history.

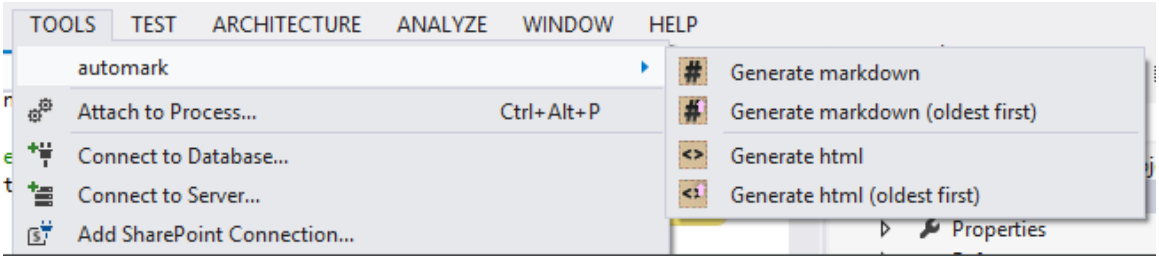


Figure 24: Menu for creating **CODE NARRATIVES**.

Because a programming experience can involve many thousands of events, simply showing all events in chronologically decreasing order can quickly become overwhelming. Instead, events are clustered into windows. A *window* is a container for events of the same type that occur in temporal proximity to each other. A sliding window algorithm [97] is used to group events into windows without exceeding a specified threshold—the threshold allows temporally close events to be grouped together.

6.4.3.2 Shared Narratives

SHARED NARRATIVES are designed to provide a high-level recap of a recent programming experience starting with the most recent event. However, not every detail is included such as the mistakes made along the way. Instead, **SHARED NARRATIVES** are intended for relating how to perform a task in a tutorial, step-by-step manner. For this reason, other types of programming events, such as visited web sites and exceptions are excluded, and instead primarily focus on source code change events. As a result of the population procedure for **SHARED NARRATIVES**, source code changes during the programming experience are organized into small temporarily ordered code snippets.

To populate **SHARED NARRATIVES**, the documents changed during the programming experience are collected. An analysis is performed that only include the changes that survived in the latest snapshot of the document. The analysis starts with each line in the latest snapshot and walks backwards in history to find change events related to that line. Based on several heuristics, these lines are then grouped into code snippets. The order of code snippets is determined by the median time of all associated change events. More detail on this analysis is described in the next

section.

6.4.4 Views

Figure 25: Interface for browsing and viewing **REVIEW NARRATIVES**.

CODE NARRATIVES can be displayed in one of two different modes: a review mode that displays the events of a programming task in a file with **REVIEW NARRATIVES**, and a sharing mode that displays the event of a programming task in a markdown file with **SHARED NARRATIVES**, which can be annotated and published to a blog.

6.4.4.1 *Review Narratives*

In Figure 25, the view for **REVIEW NARRATIVES** is shown. Events are divided by days and segmented by coding sessions. This enables developers to quickly browse through events and find the section of history relevant to them. To avoid clutter, events associated with a session are collapsed. A preview of the first event is shown. A developer can toggle a collapsed session by clicking on the first event.

Change events are rendered as a source code difference, with contextual lines that occur before and after the change added for reference. Web visit events are rendered with the title and URL of the visited web page.

6.4.4.2 *Shared Narratives*

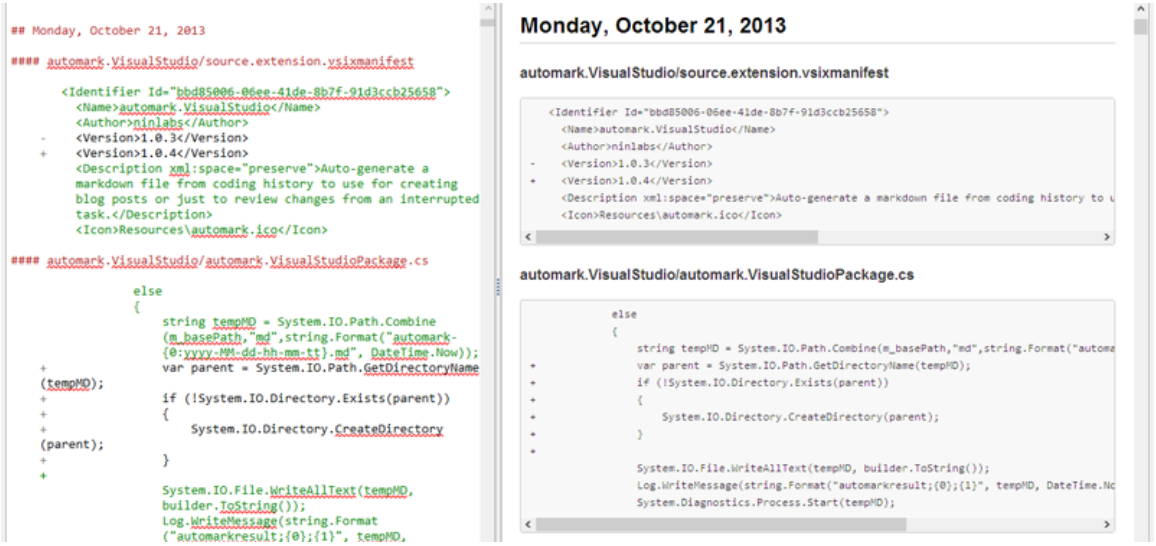


Figure 26: SHARED NARRATIVES being edited in a markdown editor.

In Figure 26, the view for **SHARED NARRATIVES** is shown. Change events are rendered as code snippets in markdown. File locations are included in the generated output. Figure 27, shows the snippet of a blog post that was generated from the markdown file. The full blog post can be seen online [2].

[2] `http://chrisparnin.github.io/articles/2013/10/creating-a-visual-studio-extension-shim-for-automark/`

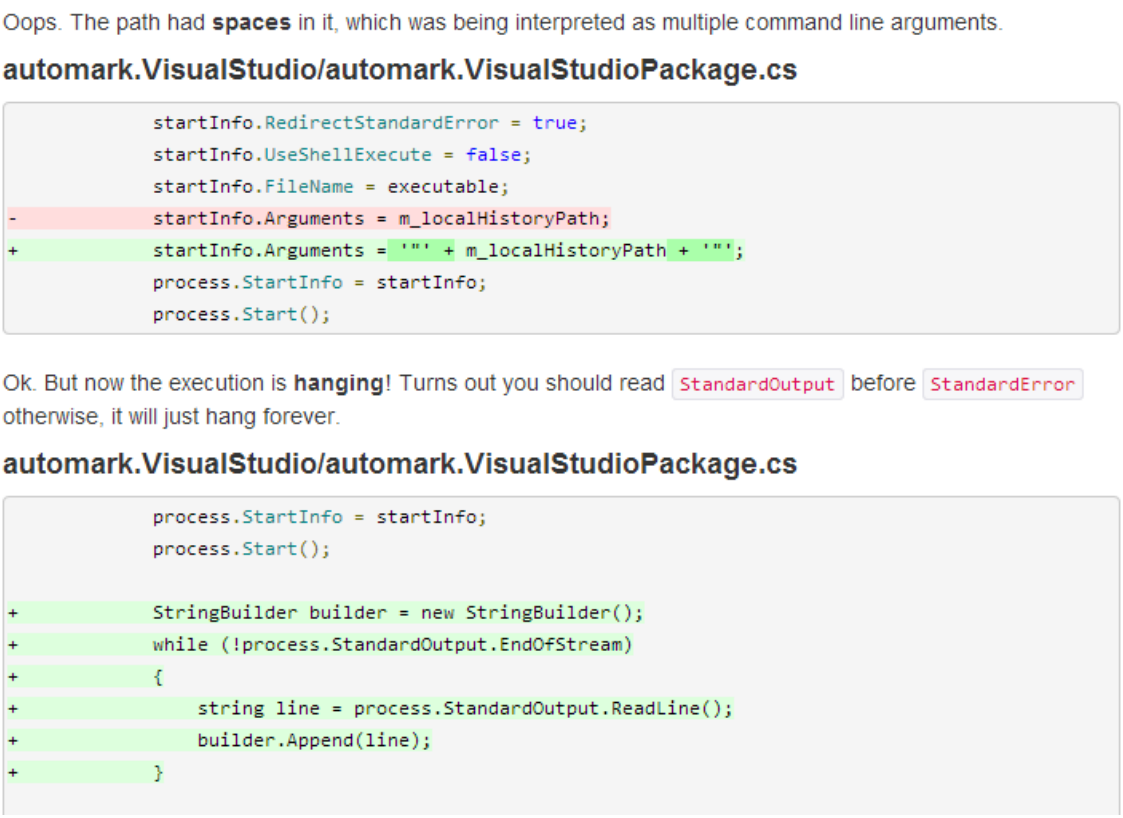


Figure 27: A blog post created from **CODE NARRATIVES**.

6.5 Sketchlets for Conceptual Memory

Software developers must learn and remember abstractions over information in source code. Because consolidating an abstraction in conceptual memory (such that it becomes second nature) can be a long process, developers often use intermediate representations to help them retain the abstraction. Additionally, if a developer needs to revisit a concept that has not been used in a while, that concept must be primed. Reviewing the intermediate representations is useful for priming concepts—unfortunately these representations are often kept as sketches on paper media and are easily displaced or discarded.

A *sketchlet* is a conceptual memory aid that helps a developer create and prime concepts. A sketchlet is a composed of overlays for annotating source code and interactive spaces [157] for expressing abstractions. **SKETCHLETS** are available when viewing a program element.

6.5.1 View

Görg, Rugaber and I developed a prototype tool called `CodePad`, which supports maintaining interactive spaces on tablet devices and allows annotations of source code. In the following text, we detail some of the initial directions we have explored that are relevant for **SKETCHLETS**. As the tool mostly involves interaction, there is not any special information collected from the programming environment to support **SKETCHLETS**.

CodePad is a tool that provides interactive spaces that are physically separate from the programming environment. Each interactive space has additional modalities available for interacting with and annotating content. Together, these interactive spaces create a mental playground that supports conceptual memory. Figure 28 shows a CodePad used with a pen-tablet monitor and tablet computer.

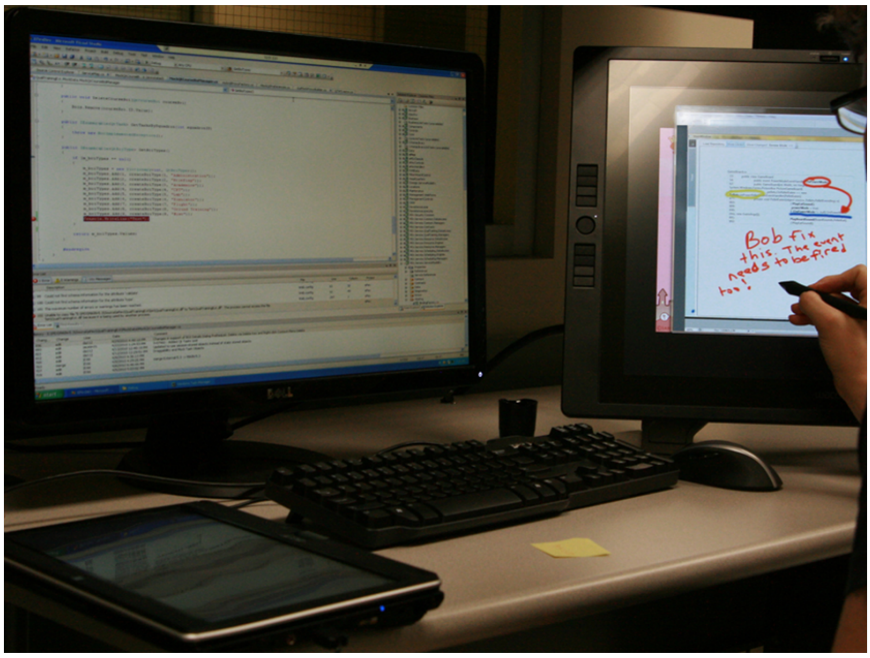


Figure 28: Pen and touch environment for annotating source code.

6.5.1.1 Sketchlet Views

Several types of **SKETCHLETS** (Figure 29) can be displayed and interacted with while it is on an interactive space:

file content sketchlet a zoomed out overview of an entire file.

code summary sketchlet a subset of the source code that was viewed.

code snippet sketchlet a manually created snippet of code.

data snapshot sketchlet a runtime snapshot of a source code variable and a treeview of its properties.

project snapshot sketchlet a snapshot of a project's structure.

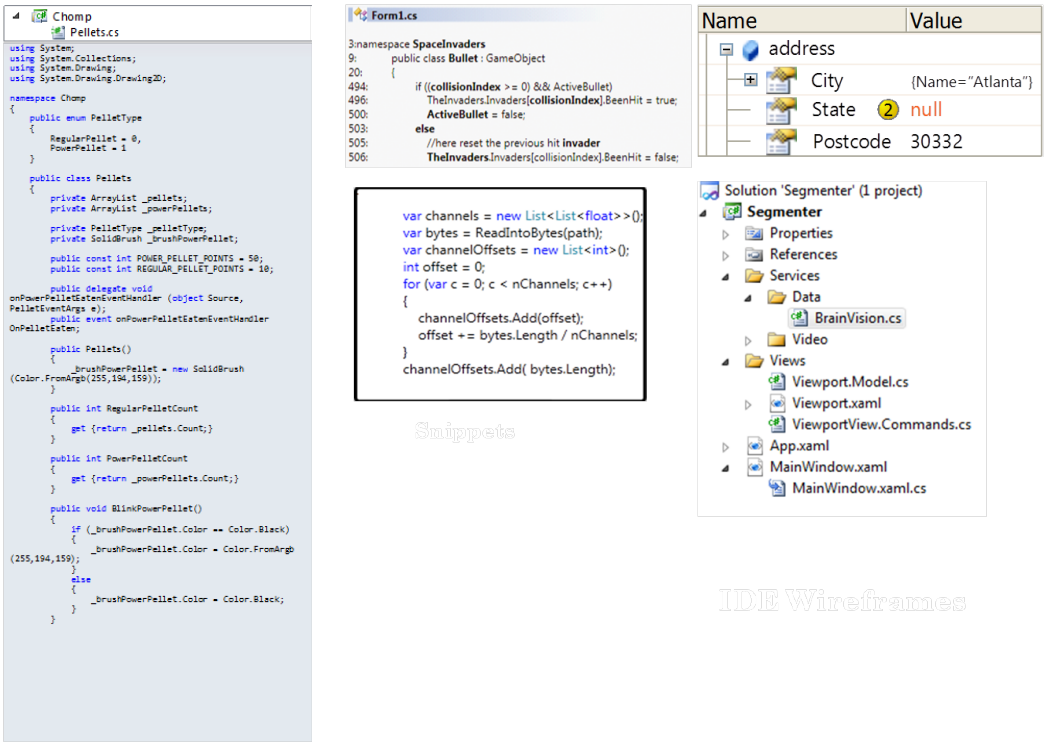


Figure 29: Different types of content supporting annotation with CodePad.

6.5.1.2 Annotations

The design supports several types of annotations for **SKETCHLETS**. These annotations are similar to those annotations that developers make on paper media [206].

Note add a note to a program element.

Link Form a link between two program elements.

Highlight Assign a highlight color to a region of text.

Mark Use a pen to mark a region of text.

6.5.1.3 Operations

There are also several sketchlet operations available in the views, Table 8 lists the different types of operations available for **SKETCHLETS**.

Table 8: Table of operations for **CODE NARRATIVES**.

OPERATION	DESCRIPTION
CreateAndSend	Create a sketchlet and send it to a CodePad device.
Focus	Navigate and focus a display to a sketchlet's associated program element in the pro
Sync	Refresh a sketchlet with any changes from programming environment.

Figure 30 demonstrates some of the gestures that can be used for annotating and interacting with **SKETCHLETS**.

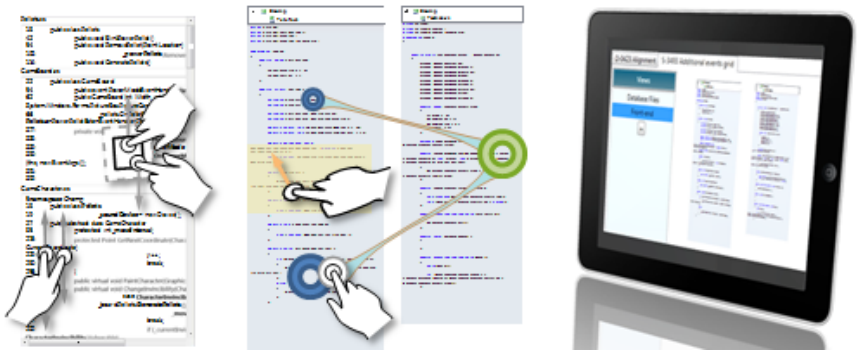


Figure 30: Several gestures for annotating and interacting with **SKETCHLETS** in CodePad.

6.6 Status

Several memlets have been developed to the point where they are actively being used by developers. In this section, I give the current status of the memlets' implementation.

6.6.1 Smart Reminders

SMART REMINDERS are available as an extension to Visual Studio. The extension has been deployed in the Visual Studio Gallery and is actively being used by developers. In future work, I would like to improve the support for more external repositories and constrictive reminders. For example, I would like to better support creating reminder conditions based on the external state of items held in source control and task repositories. The only fully supported constrictive reminder mechanism is for building programs. Other activities, such as debugging programs or committing source code can be supported in the future.

6.6.2 Touch Points

TOUCH POINTS are available as an extension to Visual Studio. The extension is mature enough for evaluating programming tasks in experiments. The Reload and Merge operations are not yet implemented. In future work, I would like to consider other visual designs that better incorporate the editor view for working with **TOUCH POINTS**.

6.6.3 Associative links

ASSOCIATIVE LINKS are available as an extension to Visual Studio. **ASSOCIATIVE LINKS** has its core functionality implemented, which is sufficient for running experiments. Because there is no official support for modifying tabs within Visual Studio, I currently employ a custom hack which will sometimes the cause the tabs to revert back to the original state. In future work, I would like to make the tabs more stable and explore other interface elements that can use **ASSOCIATIVE LINKS**.

6.6.4 Code narratives

CODE NARRATIVES are available as an extension to Visual Studio. **CODE NARRATIVES** are available in the Visual Studio Gallery, under the rebranded names autogit and automark and are

actively used by developers. In future work, I would like to support incorporate more advanced summarization and other types of events as development of those features stabilize.

6.6.5 Sketchlets

SKETCHLETS have been developed as a proof-of-concept and are not mature enough for use in experiments or development. In future work, I would like to improve annotation support and incorporate those annotations to be viewed in the programming environment. Furthermore, there is more design work needed for supporting abstractions of program elements.

6.7 Summary

In this chapter, I have presented a set of memlets for supporting the information needs of programmers after memory failures. Five types of memory are supported each with a corresponding memory aid: attentive with **TOUCH POINTS**, prospective with **SMART REMINDERS**, associative with **ASSOCIATIVE LINKS**, episodic with **CODE NARRATIVES**, and conceptual with **SKETCHLETS**. For each memory aid, I described how information from the programming environment can be automatically collected and populated into the memory aid's information model. Finally, I described how each memory aid can be viewed and interacted with as a memlet in the IDE.

In Table 9, I summarize the set of memlets described in this chapter.

Table 9: A description of how each memlet supports a particular memory type.

MEMORY	MEMLET	DESCRIPTION
prospective	smart reminders	designate reminders that trigger based on an external conditions.
attentive	touch points	increase duration and number of focus on program elements.
associative	associative links	provides spatial, temporal, and activity-related links between various program elements.
episodic	code narratives	reviews a narrative of the events and experiences related to their programming task.
conceptual	sketchlets	enables the representation and naming of new concepts derived from existing program elements.

In the next chapter, I describe my implementation (GANJI) of an environment including its high level architecture, services and algorithms, instrumentation, and data storage.

Chapter VII

IMPLEMENTATION

The name, GANJI*, derives from two Mandarin words (in simplified form):* 干 (gān) and 记 (jì). *While having no true historical origin,* ganji *can be roughly translated as* a dried note or preserved memory *in the same sense that dried food is preserved.*

In Chapter 6, I described a set of memlets for supporting programmers with interrupted programming tasks, including the visual design, the collected data, and the interactions needed to realize them. In this chapter, I describe my implementation for memlets and implementation architecture needed to support them, called GANJI. The implementation architecture includes the services needed for interfacing external resources, such as task repositories, and the infrastructure for logging development work and storing it within a database of events along with a repository of artifact snapshots.

There are several benefits served by the implementation architecture: First, the resulting implementation is something that professional developers can directly benefit from and use. Second, the implementation architecture supports other researchers with building new tools. Third, the implementation architecture serves as a test bed for validating my research hypotheses, described earlier.

In designing the implementation architecture, inspiration came from the concepts and principles prescribed by web service architectures [61] and service-oriented architectures [201]. In web service architectures, web applications are designed so that resources and services can be easily accessed via resource paths and service interfaces. The simplified interoperability allows for extension to other devices and platforms. *Representational state transfer (REST)* is a popular kind of web service architecture that exposes resources through resource paths [61]. A service or API is referred to as *RESTful* when it is designed with the principles described by REST. In service-oriented architectures, each service is designed so that it fulfills just one use case, such

as “retrieving most popular customer purchases associating with a given brand name”. In this way, many services can access the same resources while maintaining isolation between services. This isolation supports development for multiple programming environments, devices, and user interfaces. Both architecture styles make use of layers that separate responsibilities for persistence, data access, services, and user interfaces.

I will first describe the implementation architecture, its architectural layers, and key concepts. Then, for each architecture layer, I will describe the responsibilities and detailed mechanics provided by the layer.

7.1 Overview

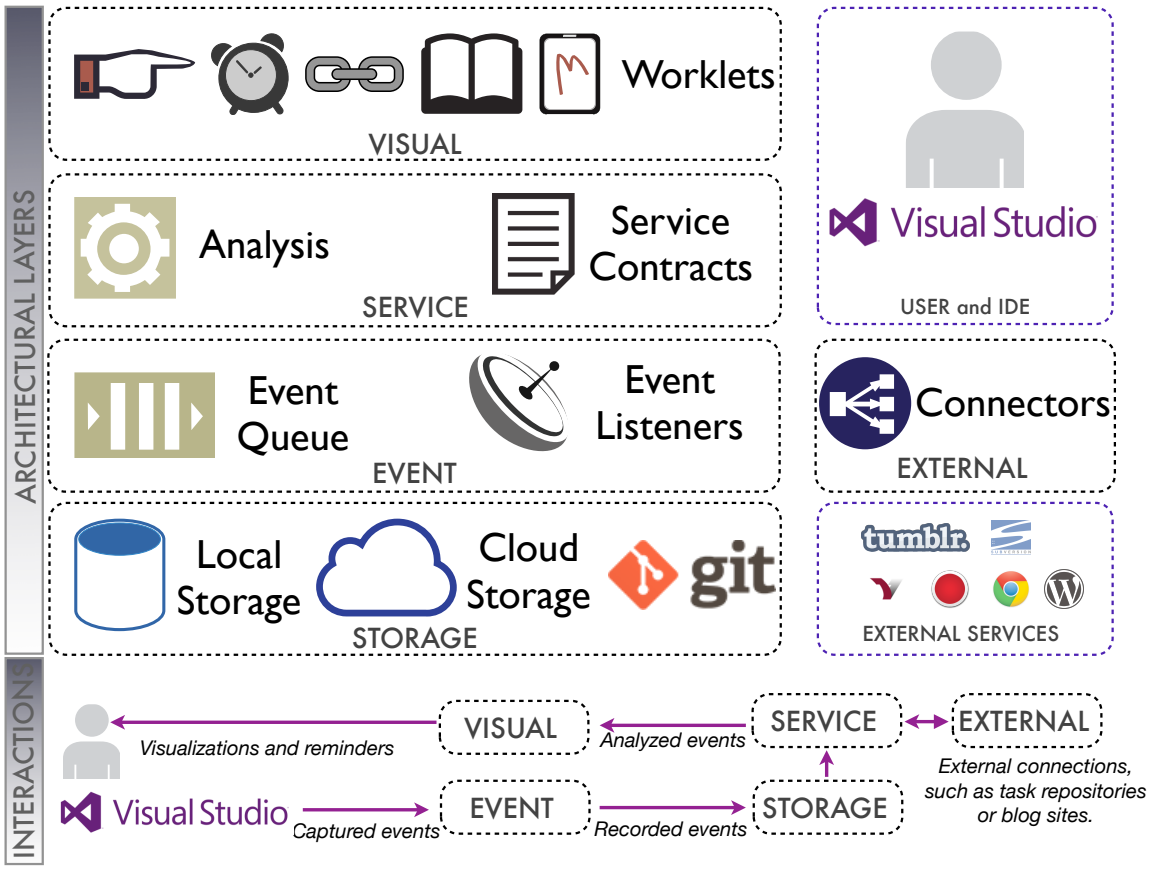


Figure 31: Architecture for GANJI.

GANJI is designed for use with Microsoft’s Visual Studio programming environment and written in the C# programming language. In Figure 31, the implementation architecture of GANJI

is shown.

GANJI spans five layers: a visual layer providing the user interactions and visualizations for developers, a service layer providing analysis services of the recorded history, an external layer providing connections to external services, a history layer providing instrumentation of programming events, and a storage layer providing persistence of the events and related artifacts.

7.1.1 Visual Studio Extensiblity

Visual Studio provides interfaces and the ability to add enhancements through a software development kit (SDK). The Visual Studio Extensibility SDK [1] provides three options for extensibility. The first option, allows the creation of an *Addin*, which is a plugin that accesses Component Object Model (COM) interfaces to Visual Studio services and objects. *COM* is a Microsoft Windows technology for interfacing for running programs. The second option, allows the creation of an *extension*, which is a plugin that enables deep integration with Visual Studio at the cost of considerable complexity. The third option, allows the import and export of components through the Managed Extensibility Framework (MEF) library [2], which is a C# library for providing extension points to C# objects at runtime.

GANJI uses a combination of the last two options. A Visual Studio extension is created for each memlet and each architectural layer. MEF components are used to add enhancements to existing interface elements in Visual Studio. For example, **SMART REMINDERS** expose an new reminder adornment for the Editor Window, which is imported and applied to the Editor Window by the MEF library.

7.1.2 Architectural Layers

The *visual layer* is responsible for handling user interactions with and rendering the visualizations of the various memlets that GANJI provides. Each memlet operates independently as a standalone extension in Visual Studio. Because I previously detailed the visualizations and interactions of the memlets in Chapter 6, I will only briefly describe the visual layer and focus on the other layers in this chapter.

[1] `http://www.microsoft.com/download/en/details.aspx?id=21835`
[2] `http://msdn.microsoft.com/en-us/library/dd885013.aspx`

The *service layer* is responsible for providing the services needed by the memlets. The services are tailored toward the specific analyses of the history required by each memlet, such as automatically populating a touch point memlet. A *contract* is an interface specification that describes the services and data that is transported between each and the visual layer. A *service contract* is a contract that specifies the parameters and return value of a service. A *data contract* is a contract that specifies the field names and types of a parameter or return value used in a service.

The *event layer* is responsible for collecting all the events and artifacts that are needed by the memlets. An *event listener* is a component that subscribes to a specific class of programming events published by Visual Studio, such as cursor movement events. The *event queue* is a component that collects snapshots of source code documents as well as associated events such as cursor movements together with associated locations in a particular snapshot.

The *external layer* is responsible for coordinating the external services needed by the memlets. A *connector*, is a component that enables interactions with an external service such as a task repositoruy, blog site, or source code repository. For example, **SMART REMINDERS** rely on connectors to task repositories for monitoring the status of a task that is in progress.

The *storage layer* is responsible for persisting all data for GANJI. An embedded database stores events for each project locally. An artifact repository stores document snapshots. Cloud storage allows certain information such as **SMART REMINDERS** and configuration information to be stored in a cloud-based server.

7.1.3 Architecture Usage Scenarios

To illustrate the interactions between the architectural layers, I provide a scenario that walks through using a memlet and the corresponding layers and components that are involved.

7.1.3.1 Auto-Populating Touch Points

> *Bob was in the middle of a complex refactoring campaign that touched many locations in the code. He used several keyword and structural searches to find places in the code that are affected by the refactoring operations. It has been several days since he has worked on the task, and he must remember which places in the code still need to be*

changed in order to accommodate the refactoring operations. Luckily, Bob had GANJI *tracking his searches and the places that were visited or changed. Now, Bob can use* **TOUCH POINTS** *to help him recall which places were changed and which ones still need work.*

Recording Searches, Navigations, and Edits As Bob is working on his programming task, the event layer is recording the programming events that occur in the programming environment. The event listeners subscribe to events published by Visual Studio and the event queue processes the events and sends them to be recorded in the storage layer.

For example, when Bob performs a keyword search, the `SearchNavigationListener` receives notification that a search is in progress, along with other details such as the keyword used and search scope. Additionally, the window that displays the search results is instrumented so that it can be determined if any navigations to program elements occur that resulted from a search result. These events are sent to the event queue, where they are processed and sent to be persisted in the storage layer. There, the events are stored in the embedded database.

Whenever Bob saves a change, the `SaveListener` receives notification before and after the save operation, allowing snapshots of the code to be captured. The save events are processed by the event queue and persisted by the storage layer. The snapshots are committed to the git repository in the storage layer.

When Bob moves the cursor as he is navigating in a document, the `ClickListener` receives notification of a click event. Because a code document is constantly changed, simply recording the line number of the file is insufficient. Instead, the event queue aligns the click events in the queue to reference a line number of a snapshot also in the queue.

Populating Touch Points from Event History Now that Bob is ready to return to his programming task, he would like to view **TOUCH POINTS**. He clicks the "Auto-Populate" link, which populates the places his has visited and edited, organized under past searches. To retrieve these places, an `ITouchPointManager` service analyzes events stored in the storage layer and returns the data to the visual layer.To support the `ITouchPointManager`, a special data structure called

a *snapshot graph* is used to analyze the past change events for each code document. A forward analysis over the snapshot graph is used to determine whether an event and a source code line it references still exists. For example, if a source code line is deleted, then the events associated with that line are excluded. The snapshot graph is retrieved from the service layer via the `ISnapshotGraphFactory` service contract. The `ISnapshotGraphFactory` retrieves the snapshots from the git repository in the storage layer and maps lines between a previous and next snapshot.

When constructing a set of **TOUCH POINTS**, the `ITouchPointManager` service queries search events and navigation events from the local database in the storage layer. The events are mapped to a snapshot in the snapshot graph via the snapshot reference stored with the event.

Using the touch point memlet, Bob notices several locations where he did not complete making a necessary change. He is able to navigate to those locations and complete the changes.

7.2 Visual Layer

The visual layer is responsible for hosting the memlets in Visual Studio. The design of the visual layer is based of the architectural pattern called Model View ViewModel (MVVM) [182], which is a specialized version of the Presentation Model design pattern [3]. MVVM is composed of user interface templates (View) that bind to a model of view state (ViewModel), which is derived from a data source (Model).

To implement the visualizations and interactions, Windows Presentation Framework (WPF) is used. WPF allows creation of user interface elements through creation of an user interface template language called the Extensible Application Markup Language (XAML).

To host the memlets in Visual Studio, a special component called a Tool Window can be used to contain a WPF-based user interface element. A Tool Window can be displayed, docked and tabbed like other standard Visual Studio windows. However, for memlets like **SMART REMINDERS**, which need to display items within the viewport of a text document, a Tool Window is insufficient. Instead, a MEF component called an *Adornment* allows content to be hosted directly within a text document. Adornments allow visualizations to be added to the background of a text document as

[3] `http://martinfowler.com/eaaDev/PresentationModel.html`

well as modifying the appearance of the text itself.

Although GANJI is implemented to be displayed in Visual Studio, the design of the visual layer can be generalized for other IDEs such as Eclipse, Xcode, or Cloud9. The architectural design of GANJI should be reusable for implementing the same views in other IDEs.

7.3 Service Layer

The service layer is responsible for providing analyze services on the event history for the visual layer. In this section, I describe the services provided by the service layer and some underlying data structures that are key for supporting those services.

7.3.1 Data Structures

There are several data structures that are used by the service layer.

7.3.1.1 Snapshot Graph

Many services require reasoning about extensive revisions to a source code file over time. Although there are techniques that support tracking changes across revisions (see [34, 33]), and implemented in source code repositories such as git, they often focus on comparing two sequential revisions and do not work bi-directionally.

Therefore, I build on the work of Canfora et al. [34], which uses heuristics for matching changed lines across two revisions, to create a more general data structure called a snapshot graph.

Definitions A *snapshot graph* is a bi-directional chain of commit snapshots that can be used to relate the different versions of the file over time. A *commit snapshot* is a structure that contains two timestamped versions of a file: the state before and the state after a change event. A *line mapping* is an index that relates source code lines between the versions of a file in a file snapshot. Figure 32 shows an example of a commit snapshot and its line mappings.

A *forward chain* is an edge from one commit snapshot to the next commit snapshot. A *backward chain* is an edge between a commit snapshot to the previous snapshot. A forward chain and backward chain always starts at a line in the commit snapshot's rightmost version and ends at

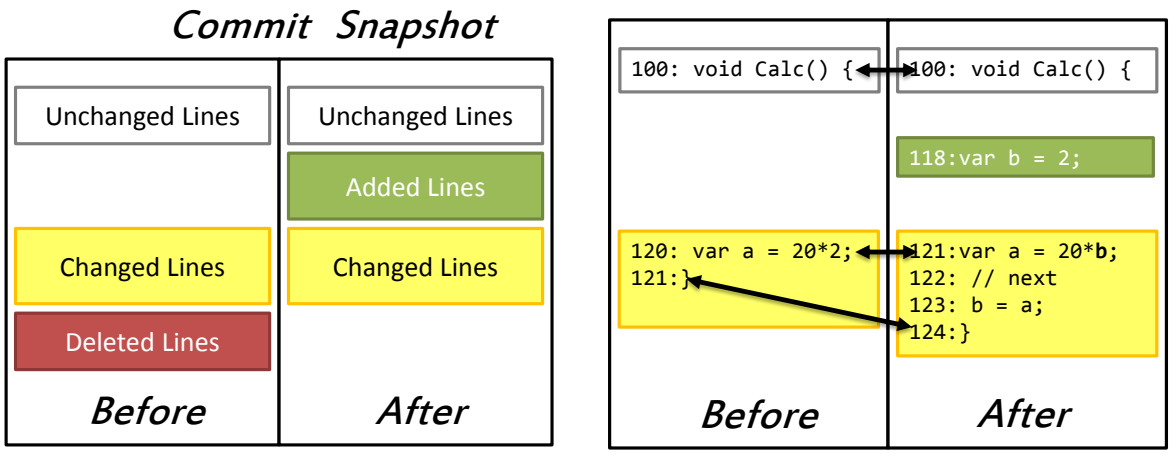


Figure 32: A commit snapshot (left) and its line mappings (right)

a line in the next or previous commit snapshot's rightmost version. A forward chain and backward chain also have an associated chain type.

A *chain type* is a category used to classify the status of a line after following a chain. A chain type can be one of:

- **Carry.** The line is not affected by any change in the previous or next commit snapshot.
- **Transform.** The line is modified by a change in the previous or next commit snapshot.
- **Kill.** The line is deleted by a change in the next commit snapshot.
- **Null.** The line is invalid in a previous or next commit snapshot.

Figure 33 shows an example of forward chains and their chain types linking several commit snapshots.

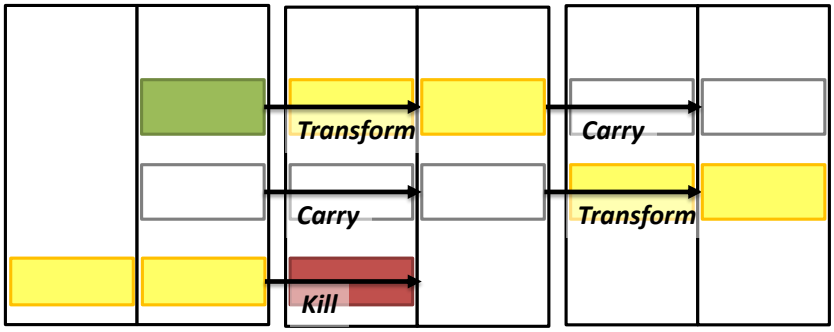


Figure 33: Forward chains between commit snapshots.

Construction To create a snapshot graph, a set of revisions of the file and differences are obtained from git. The differences from git categorize a changed line as either a deletion, addition, and modification.

N-1 commit snapshots are created from the N revisions of a file. Each commit snapshot is assigned a before and after revision. A line mapping between the before and after revisions is created by incrementing a counter for the before revision and a counter for the after revision based on the differences that occurred.

Table 10: Transition table for forward transitions

STATE	TRANSITION RULE
None → None	*Carry*
Modified → None	*Carry*
Add → None	*Carry*
None → Modified	*Transform*
Modified → Modified	*Transform*
None → Delete	*Kill*
Modified → Delete	*Kill*
Add → Modified	*Transform*
Delete → *	*Terminate*
$\perp$	*Error*

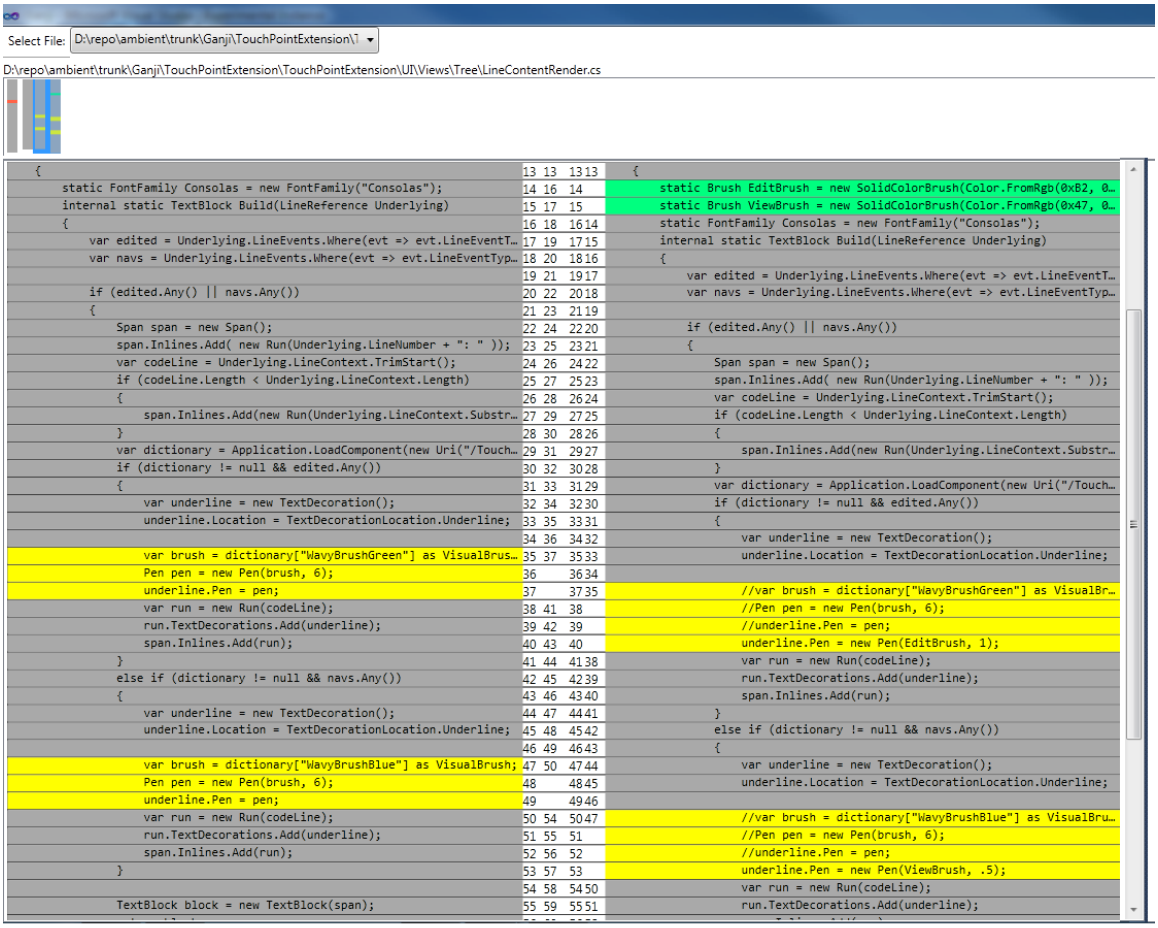


Figure 34: Snapshot Graph Visualizer

Forward chains are created for each line in the after revision and then linked to the future version of the line in the next commit snapshot. A transition table determines the chain type. Table 10 shows the transition table that is used to determine the chain type. The state for selecting a transition rule is composed of a difference type at the originating site and difference type at the destination site. Backward chains are created in a similar manner as forward chains, except that they are created in the opposite direction, starting with the latest revision.

Figure 34 shows a constructed snapshot graph displayed in a tool for visualizing snapshot graphs.

7.3.2 Internal Services

There are several internal services that are commonly used across other services in the service layer.

7.3.2.1 Parse Analysis via Roslyn

Several services need to parse source code in order to extract structured information. The Roslyn extension is used to parse C# code. [4] The following are services that support common parse requests:

- `IMethodCallParser` - Retrieve list of method calls from a block of code.
- `IProgramElementParser` - Retrieve a hierarchal collection of program elements in a file.

7.3.2.2 Origin Analysis

Origin analysis [74], is an analysis technique which can compare blocks of changed code and determine if they are similar by comparing sets of method calls contained by the blocks of code. Origin analysis is effective for determining when code has been moved or extracted. These origin analysis services are provided:

- `IOriginAnalysis` - Return the Jaccard similarity [91] between two sets of method calls contained in a block of code.
- `IExtractRefactoringDetector` - Classify a set of code changes as an extract method refactoring.

7.3.2.3 Sessions

Services often need to know when the last programming session that occurred, or retrieve a list of *n* recent sessions.

- `ISessionManager` - Retrieves a list of sessions based on query criteria.

[4] `http://www.microsoft.com/en-us/download/details.aspx?id=27746`

7.3.2.4 Snapshot Graph

Memlets such as **TOUCH POINTS** or **CODE NARRATIVES** need snapshot graphs in order to perform analysis over a source code file's history. The following service is used to construct and retrieve a populated snapshot graph:

- `ISnapshotGraphFactory` - Construct a snapshot graph for a file's revisions.

7.3.3 Memlet Services

These services are used by the visual layer for supporting interactions with and visualizations for the memlets.

7.3.3.1 Touch Points

TOUCH POINTS rely on internal services to construct snapshot graphs of changed and visited files. When constructing a set of **TOUCH POINTS**, the `ITouchPointManager` service will query search events and navigation events from the local database in the storage layer. The events are mapped to a snapshot in the snapshot graph via the snapshot reference stored with the event.

- `ITouchPointManager` - Return a set of **TOUCH POINTS**.

7.3.3.2 Smart Reminders

SMART REMINDERS rely on external services that must be polled as well as monitoring current state of the programming environment. The following services help bridge external services and monitor the programming environment state:

- `ISmartReminderManager` - Manages the configuration and storage of **SMART REMINDERS**.
- `ISmartReminderMonitor` - Manages external connectors and reminder conditions to remote services.

7.3.3.3 Associative Links

ASSOCIATIVE LINKS require monitoring the current state of the programming environment with a bit more requirements than **SMART REMINDERS**. The following service helps monitor the

programming environment state:

- `IAssociativeLinkManager` - Manage the state and persistence of an associative link.

7.3.3.4 Code Narratives

CODE NARRATIVES need to work with external services for publishing **CODE NARRATIVES** and retrieving additional event such as web browser history and to acquire snapshot graphs for analyzing history. The following services help create and publish **CODE NARRATIVES**:

- `INarrativeManager` - Returns a set of narrative items.
- `IEventBubbleManager` - Returns a set of event bubbles from event history.
- `IPublishNarrative` - Coordinates publishing a narrative via a blog connector.

7.3.3.5 Sketchlets

SKETCHLETS need to communicate with remote devices with a Code Pad environment. The following services support **SKETCHLETS**:

- `ICodePadReceiver` - Register and receive events from Code Pad.
- `ICodePadManager` - Send content from Visual Studio to Code Pad.

7.4 External Layer

The external layer is responsible for providing a bridge to external services and external events. A set of connectors mediate the external resources. By providing additional connectors, it is possible to extend GANJI to work with other systems. Before using a connector, a developer must provide configuration details, such authentication and urls to the external services.

7.4.1 Blog Connections

CODE NARRATIVES need connections to blog publishing services.

7.4.1.1 Wordpress

Wordpress [5] is one of the most popular platforms for publishing blog sites. The `WordPressConnector` uses XMLRPC [6], which is a specification for remote procedure calls, to create a new blog post. The connector also handles formatting the content of the blog post to account for syntax highlighting and other formatting issues.

7.4.1.2 Tumblr

Tumblr [7] is another blogging platform targeted at publishing and sharing simple posts. The `TumblrConnector` uses an RESTful API to create a new blog post. The connector also handles formatting the content of the blog post to account for syntax highlighting and other formatting issues.

7.4.2 Task Repository Connections

GANJI needs to be able to connect task repositories in order to obtain status about tasks. For example, **SMART REMINDERS** have reminder conditions that need to trigger on task completion. A challenge in providing connections to task repositories is mapping the different vocabulary and subtle differences in items contained in a task repository. For example, task tracking tools such as Version One or Pivotal that support agile development often refer to a task as a "story" or "defect". Microsoft's TFS (Team foundation server) or IBM Jazz instead use the term "work item". Other challenges of task repositories were also described in section 2.3.4.

7.4.2.1 Pivotal

Pivotal [8] is a task tracking tool for managing iterations of agile development. Pivotal tracks stories, tasks, and bugs. The `PivotalConnector` uses a RESTful API to retrieve the list of stories, tasks, and bugs assigned to an user, and to check on the status of stories, tasks, and bugs.

[5] `http://wordpress.org/`
[6] `http://xmlrpc.scripting.com/`
[7] `http://www.tumblr.com/`
[8] `http://www.pivotaltracker.com/`

7.4.2.2 Version One

Version One [9] is a task tracking tool for the entire development lifecycle of an agile-managed project. Version One tracks requirements, epics, stories, tasks, and defects. The `VersionOneConnector` uses a RESTful API to retrieve the list of epics, stories, tasks, and defects assigned to an user (more than one user can be assigned to an item), and to check on the status of epics, stories, tasks, and defects.

7.4.3 Browser History

CODE NARRATIVES need connections to browser histories to collect links to visited web sites traversed during software development.

7.4.3.1 Chrome

Chrome [10] is a popular web browser, especially among software developers. Chrome stores the history of visited urls in an embedded SqLite database [11]. The `ChromeHistoryConnector` uses the System.Data.SQLite library [12] to read Chrome's database and extract the time and url of a visit.

The `ChromeHistoryConnector` uses a whitelist to filter which results are read from the browser history. Figure 35 shows a visualization of a developer's web history that has been filtered to include only developer-relevant sites. The visualization show's the icons corresponding to the visited web site, leaving blanks for filtered out sites. The sites shown in the summary are filtered sites, Google searches, stackoverflow.com visits, Android documentation sites, Google user groups sites.

7.4.4 Source Code Repository Connections

SMART REMINDERS and **CODE NARRATIVES** need to monitor the status of source code commits and to retrieve commit messages.

[9] `http://www.pivotaltracker.com/`
[10] `https://www.google.com/intl/en/chrome/browser/`
[11] `http://www.sqlite.org/`
[12] `http://system.data.sqlite.org/index.html/doc/trunk/www/index.wiki`

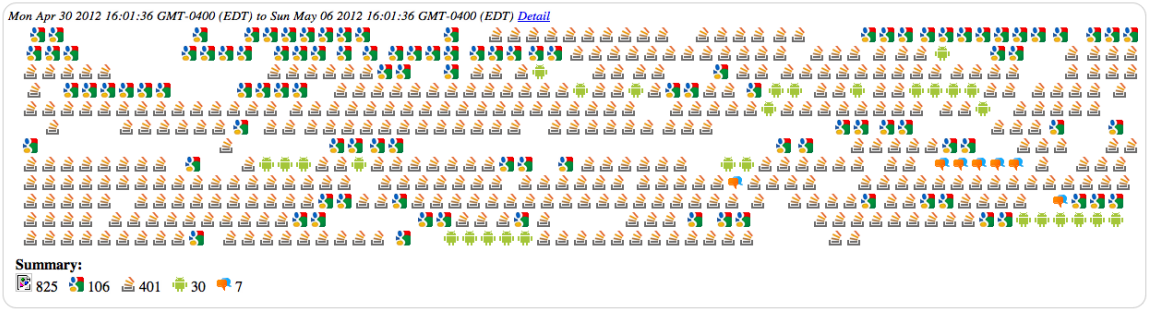


Figure 35: A visualization showing one week of a developer's web history.

7.4.4.1 Subversion

Subversion [13] is source code repository that allows a set of files to be committed to the repository in a single transaction. The repository is typically hosted on a remote server. The `SvnCommitConnector` uses the SharpSvn library [14] to retrieves a list of source code commits to extract the time of the commit, the commit message, committer, and the list of files changed.

Source control repositories often are accompanied by a client user interface for performing a commit. **SMART REMINDERS**, which may use constrictive notifications to prevent a commit from occurring, can be connected with the client user interface via a client-side hook. A *client-side hook* is a script that is executed prior to an action on a source code repository. TortoiseSVN [15] is a popular user interface for Subversion. The `TortoiseSVNConnector` uses WScript, a Microsoft Windows scripting language much like JavaScript, to implement the constrictive notification for commits.

7.5 Event Layer

The event layer is responsible for subscribing to programming events from Visual Studio and recording them to the storage layer. The two main types of components of the event layer are event listeners that subscribe to programming events and an event queue that processes the events and sends them in the storage layer.

[13] `http://subversion.tigris.org/`
[14] `http://sharpsvn.open.collab.net/`
[15] `http://tortoisesvn.net/`

7.5.1 Event Listeners

The event listeners are hosted within a custom extension loaded in running instances of Visual Studio 2010 or higher. The extension handles subscribing and shutting down the event listeners as projects open and close. Each event listener is responsible for logging instances of related event types. For example, a build listener will listen to when a developer requests that a solution is compiled and built. This may generate multiple types of events beyond the build command itself, such as build errors or even navigation events when the developer clicks on the build error.

The following are the event listeners and events (shown in italic) supported by GANJI:

- `SaveListener` - Logs saves to documents (*save events*) and collects snapshots.
- `BuildListener` - Logs *build events* and successful compilation or build errors (*build error events*).
- `ErrorNavigationListener` - Logs navigation through compilation errors (*error navigation events*).
- `CursorListener` - Logs text input cursor movement and the words under the cursor (*click events*).
- `RefactoringListener` - Logs refactoring commands such as rename or extract method (*refactoring events*).
- `CopyPasteListener` - Logs copy/paste/cut commands (*copy and paste events*). Extracts the source url when pasting from code collected from the web browser.
- `ExceptionListener` - Logs runtime exceptions encountered while debugging (*runtime exception events*).
- `SearchNavigationListener` - Logs searches and navigation through search results (*search navigation events*).
- `ProjectStructureListener` - Monitoring files that are removed, renamed or added (*document events*).

When an event listener receives an event and is ready to record its occurrence, the event listener will send the event to the event queue.

7.5.2 Event Queue

The *event queue* is a component responsible for the processing and storing of events. The event queue is needed for several reasons. First, if high-frequency events, such as clicks, were sent directly to the storage layer, this could introduce considerable delay in the user interface due to latency in the storage operations. Instead, the event queue allows events to be buffered and eventually flushed to the storage layer more efficiently. Second, the event queue allows events to be processed and associated. For instance, *event alignment* is a process that allows events to be associated with other events, as illustrated in Figure 36.

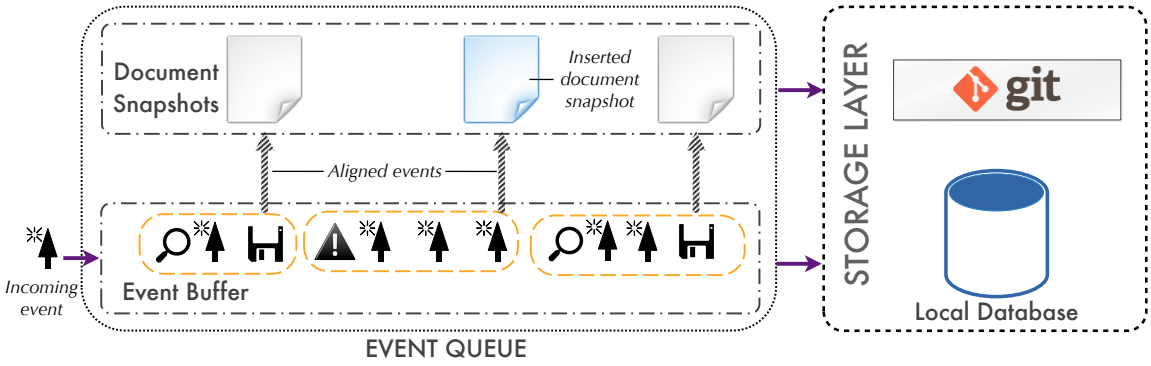


Figure 36: The event queue with several events buffered and aligned to document snapshots.

When events enter the event queue, they undergo the process of event alignment. Event alignment is performed on events that need to be associated with document snapshots. Because a code document may be frequently changed, simply recording the line number in the document is insufficient. During event alignment, events are associated with a document snapshot that validly matches document references in the event. If no valid document snapshot exists, a new snapshot is inserted and associated.

7.6 Storage Layer

The storage layer is responsible for persisting programming events, document snapshots and other data needed by memlets. The storage layer is designed to store data for each project into a corresponding database and artifact repository. The ability to store data in cloud-hosted databases

is also supported.

7.6.1 Event Database

The event database stores instances of programming events (see Figure 37) sent to the storage layer and other information needed by the memlets (see Figure 38). In order for GANJI to access the event database, it uses an Object-Relational-Mapping (ORM) [18] framework. ORM is a programming technique for transporting data from the tables of the database to objects in the program. Rather than manually create the database tables and the objects, usually one can be generated from the other. GANJI uses Entity Framework [16] to automatically generate the local database from data contract objects. The underlying database provider is SQL Compact Edition 4.0 [17].

7.6.2 Artifact Repository

The artifact repository stores document snapshots. To implement the artifact repository, a git repository is used. Git[18] is a distributed version control system. In this sense, *distributed* refers to the ability to maintain local instances as well as remote instances of the repository that stores multiple versions of a document. Because a git repository can be easily embedded, it is useful for storing a local history of a project's documents. An additional responsibility of the artifact repository is the retrieval of source code differences. Version control repositories such as git can efficiently store and retrieve differences between versions of a document.

7.6.3 Cloud Storage

Relying solely on local storage has several limitations. Cloud storage is a on-line storage service that provides pools of resources that are typically distributed, automatically replicated, and easily accessed. Because a programmer often develops using multiple computers or may need to share information with other teammates, GANJI includes the ability to use cloud storage. Furthermore, cloud storage makes it easier to develop new memlets that are accessible from a web browser.

[16] `http://msdn.microsoft.com/en-us/data/ef.aspx`
[17] `http://www.microsoft.com/en-us/download/details.aspx?id=17876`
[18] `http://git-scm.com/`

Currently, **SMART REMINDERS** and connector configuration information is also stored in cloud storage.

To support cloud storage, GANJI stores data hosted by MongoLab[19], which is a cloud-based service that hosts MongoDB databases. MongoDB[20] is an object-based database commonly available in cloud services. MongoDB stores objects expressed using a JSON (JavaScript Object Notation)[21] representation.

To store data in a JSON format, a data-contract-to-JSON convertor is used. The convertor enables any data that can be stored locally to also be stored in cloud storage without any additional overhead.

An on-line GANJI account is created for each developer that uses GANJI during setup. To access cloud storage, a developer simply supplies credentials for his or her GANJI account.

7.7 Summary

This chapter describe the implementation of memlets and its implementation architecture, called GANJI. The implementation architecture is inspired by the principles and concepts of the web-service and service-oriented architectural styles. The implementation architecture composes five layers: the visual layer, the service layer, the event layer, the external layer, and the storage layer. Each layer has a primary responsibility that ultimately support the ability to create and display memlets for a developer recovering from an interruption.

The next chapter describes how I evaluate the remaining research questions and hypotheses of my thesis.

[19] `http://mongolab.com/`
[20] `http://www.mongodb.org/`
[21] `http://www.json.org/`

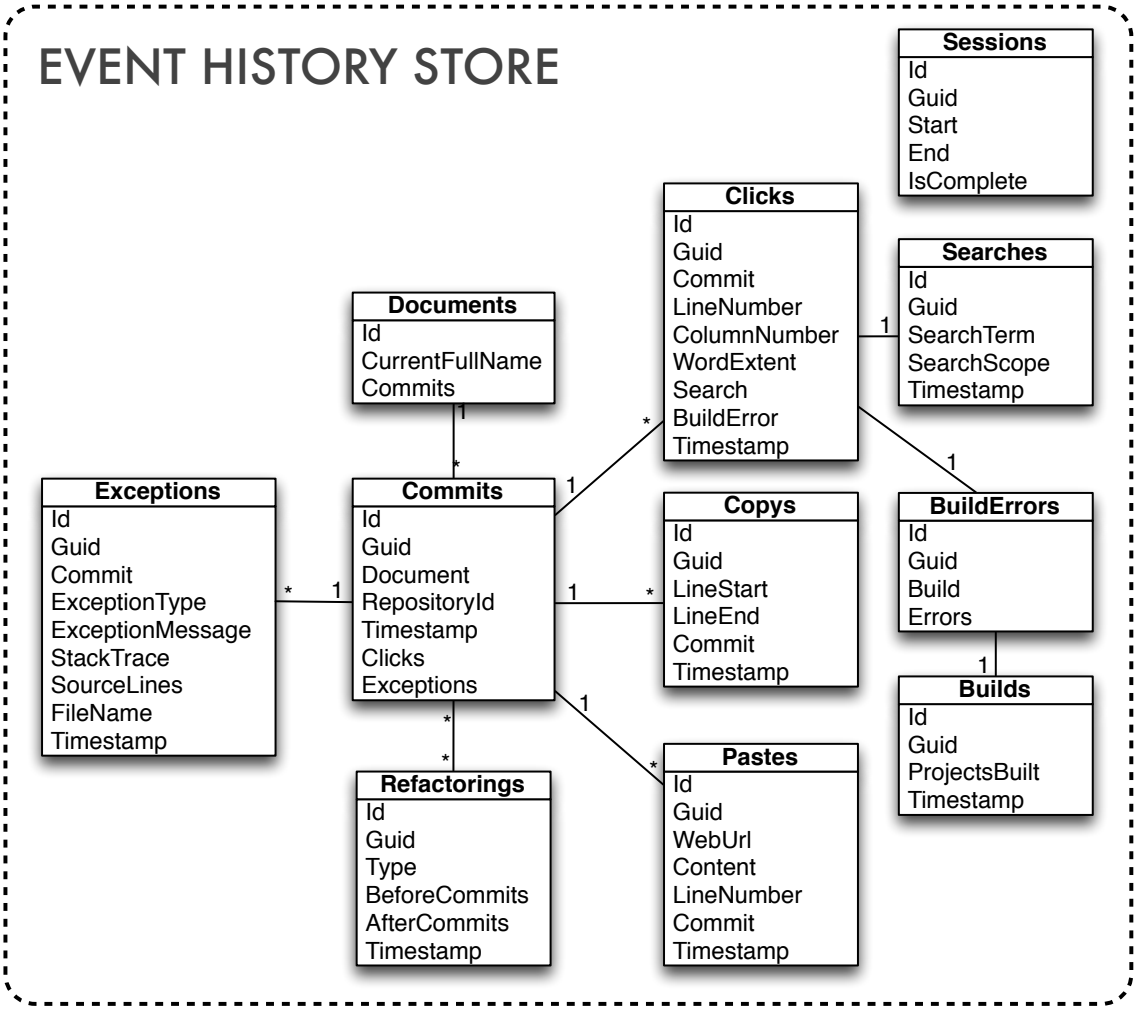


Figure 37: Data model for programming events used by GANJI.

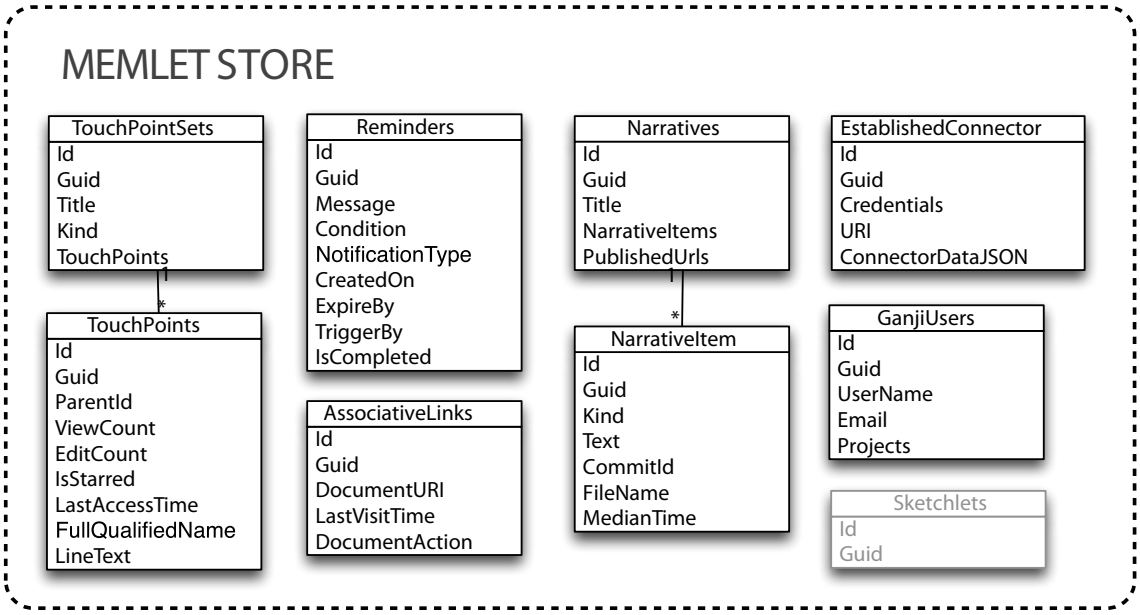


Figure 38: Data model for the persisted memlet state and other GANJI data stored locally for each project.

Chapter VIII

EVALUATION

In this chapter, I describe how I evaluate the research questions and claims related to my thesis. First, I review my thesis statement and research questions and enumerate the evaluated claims. Then I describe my approach to evaluation and the experimental results supporting the claims.

8.1 Evaluation of Thesis

The thesis to be evaluated is as follows:

Memlets, conceptual models and visualizations that support limitations in programmers' memories, reduce the negative effects of interrupted programming tasks by overcoming memory failures experienced..

To elaborate, the thesis statement states that the negative effects of interrupted programming tasks, such as increased errors and time-to-task completion, can be reduced if programmers are supported by appropriate memory aids in the form of memlets. The dissertation argues that the memlets must satisfy a programmer's cognitive needs (see Chapter 2) and explains that these information needs are dependent on the various constraints and memory failures that arise in different areas of memory during specific programming tasks (see Chapter 5). My conceptual framework structures these constraints and failures as information needs in terms of five memory types particularly relevant to programmers: attentive, prospective, associative, episodic, and conceptual. Based on these information needs and constraints, the design and implementation of memlets was undertaken (see Chapter 5 and 6). The implementation is realized in an extension built for Visual Studio (see Chapter 7).

8.1.1 Research Questions

The research on my thesis has been guided and progressed by the following research questions (see Figure 39):

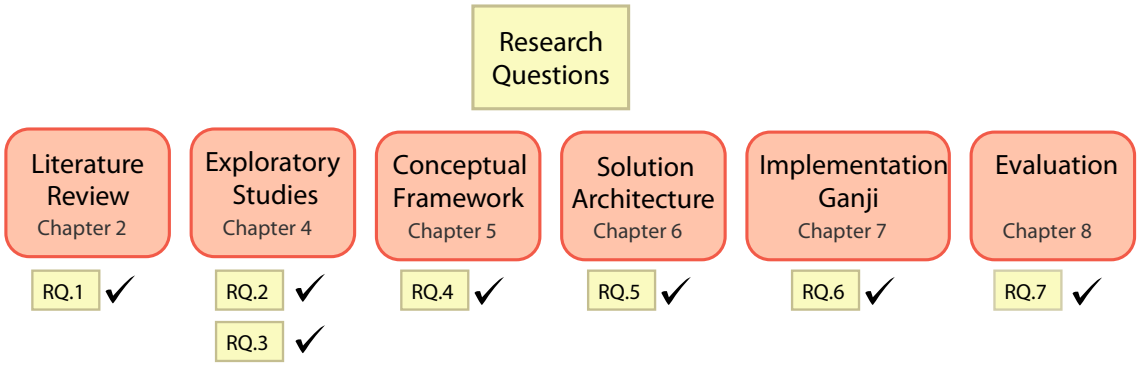


Figure 39: Illustrated steps of my research approach. A check indicates that a research question has been addressed.

Research Question 1 - What memory deficiencies arise from interruption of programming tasks?

Research Question 2 - What strategies and sources of information do programmers use to recover from interruptions?

Research Question 3 - What memory aids can programmers use to recover from interruptions?

Research Question 4 - To what extent can memory deficiencies be linked to resumption strategy failures or lack of information?

Research Question 5 - To what extent can the innate properties of human memory be used to *derive* more effective resumption aids?

Research Question 6 - Which new algorithms and concepts are needed for building effective resumption aids?

Research Question 7 - What are the observable benefits and disadvantages with using memlets for interruption recovery in software development?

The first three research questions have been answered in my exploratory phase of research. To answer Research Question 1, *What memory deficiencies arise from interruption of programming tasks?* I conducted an extensive literature review on interruptions and memory [147], which has been discussed in Chapter 2 and is presented in full in Appendix A.

To answer Research Question 2, *What strategies and sources of information do programmers use to recover from interruptions?* I examined over 10,000 recorded programming sessions [151] and surveyed over 400 programmers to understand the strategies and sources of information that were useful for recovering from an interruption [152]. From the study, I was able to identify several important strategies and sources of information that programmers use to recover from interruption.

To answer Research Question 3, *What memory aids can programmers use to recover from interruptions?* I performed a controlled experiment [148] with 14 programmers that investigated how the programmers used different types of memory aids to recover from an interruption to a programming task. From this laboratory study, I was able to observe several instances where memory deficiencies occurred when there was a failure to record important information. However, memory deficiencies were mitigated targeted when memory aids were introduced, helping the programmers recover from the interruptions faster.

The next three research questions have been addressed in my explanatory phase of research and implementation work. To answer Research Question 4, *To what extent can memory deficiencies be linked to resumption strategy failures or lack of information?* I constructed a conceptual framework that could explain the results obtained in answering previous research questions. From it, I derived a set of programmer information needs based on memory failures that programmers experience [153].

Based on the results of the exploratory investigations, two additional research questions were investigated. To answer Research Question 5, *To what extent can the innate properties of human memory be used to derive more effective resumption aids?* I constructed a set of memory aids derived from the conceptual framework. However, in answering Research Question 5, I made several claims about the ability of a memlet to reduce the negative effects of interruptions that remained unevaluated. The claims are discussed in the next section.

To answer Research Question 6, *Which new algorithms and concepts are needed for building effective resumption aids?* I implemented these memory aids and devised the necessary algorithms and data structures for realizing their design. The final research question and its accompanying research claims are evaluated in this chapter: Research Question 7, *What are the observable*

benefits and disadvantages with using memlets for interruption recovery in software development?

8.1.2 Claims

There are several unevaluated claims about a memlet's ability to reduce the negative effects of interruptions.

Claim 1 - **SMART REMINDERS** *reduce monitoring failure by introducing mechanisms for conditional triggers and reduce prompting failure by introducing mechanisms for enhancing awareness of reminders.* There has been no study to evaluate if **SMART REMINDERS** outperforms a programmer's innate prospective memory supported by ad hoc measures such as TODO messages.

Claim 2 - **TOUCH POINTS** *reduce the number of errors and omissions made by a programmer during an interrupted task by increasing the ability for a programmer to attend to multiple programming locations during the programming task.* There has been no study to evaluate if **TOUCH POINTS** outperforms a programmer's innate attentive memory supported by existing IDE interfaces.

Claim 3 - **ASSOCIATIVE LINKS** *reduce the navigational effort of a programmer recovering from an interruption by improving the recognition and recall of program elements visited during the task.* There has been no study to evaluate if **ASSOCIATIVE LINKS** outperform a programmer's innate associative memory supported by existing IDE interfaces.

Claim 4 - **CODE NARRATIVES** *reduce source failures by maintaining and presenting contextual details of programming events and reduce recollection failure by maintaining and presenting the ordering and fidelity of programming events.* There has been no study to evaluate if **CODE NARRATIVES** outperform a programmer's innate episodic memory supported by differencing tools.

Claim 5 - **SKETCHLETS** *reduce the effort of priming concepts necessary for understanding code.* There has been no study to evaluate if **SKETCHLETS** outperform traditional programming understanding efforts.

Experiments are needed to validate these claims. I describe how evaluating these claims support my thesis, my evaluation approach, experimental designs, experimental results in the next sections.

8.1.3 Evaluating Claims

I have focused my evaluation efforts on **Claim** 1 and **Claim** 4 by performing the following studies described in the next two sections. By evaluating these claims, I provide evidence that my conceptual framework can be used to derive resumption aids, memlets, that are more effective than existing software development tools or human memory unaided (Research Question 5). Memlets can be more effective because they support properties and processes of human memory that are otherwise not well supported during interruption of software development tasks. Further, the studies—as a result of performing studies in the context of real development tasks—revealed several benefits and barriers to using memlets in ecologically valid settings (Research Question 7).

8.2 A Field Study of Episodic Review with Code Narratives

To evaluate **Claim** 4: **CODE NARRATIVES** *reduce source failures by maintaining and presenting contextual details of programming events and reduce recollection failure by maintaining and presenting the ordering and fidelity of programming events*, I performed a field study involving participants using **CODE NARRATIVES** to undertake episodic review tasks.

In designing the study, an important objective was to be able to measure recall of code change events after interruption. Based on this objective, a laboratory study was undertaken for two reasons. First, because a participant works on experimental tasks with unfamiliar code, more time is spent navigating than editing as there is considerable time needed to first comprehend the code. As a result, there are not many change events generated or available to recall. Second, in the laboratory setting it is difficult to convince participants to be involved in long-term interruptions. In short, laboratory experiments are better suited for studies involving short-term interruption on comprehension tasks.

A second objective was to identify the practical concerns and obstacles with using the tool in the context of natural settings. For instance, when developing a research tool for adoption by industry, unfounded assumptions and unexpected constraints often render the research tool useless in industrial contexts and harming the external validity of resulting research findings.

Details of my method and obtained results are described in the next sections.

8.2.1 Method

I chose to perform a field study because it offered a viable way to observe programmers performing programming tasks with sufficient amounts of changes when measuring recall while satisfying my secondary objective. A *field study* is a research method for collecting contextual information about a problem, understanding the unique challenges posed by it, and collecting data about how a tool was used in practice by a developer. One constraint imposed by a field study is avoiding methods or measures that participants view as too intrusive or time-consuming during the performance of their daily work.

The study uses a mixed-method approach and is divided into three phases. The first phase involves a *contextual inquiry* [22], an investigation method for establishing basic constraints and for receiving initial feedback on tool design. This phase reduces the chance that poor tool design interferes with the experimental goals. The second phase involves *field observation* [107], a data collection method for recording events in the context of natural settings. During the field observation, developers perform a tool observation and episodic review tasks, which are described in detail below. The third phase involves an exit survey and collection of tool logs.

To measure the improvement of recall, a content analysis of events recalled with and without **CODE NARRATIVES** was performed when developers were resuming interrupted programming tasks. To ensure that differences in recall are not simply explained by increases in recall effort, I also measure the time used to perform the episodic review tasks. Further, by allowing participants to use existing source difference tools, they had access to the same information provided by **CODE NARRATIVES**, but without temporal support and with less fidelity. To identify benefits and barriers of using **CODE NARRATIVES**, I observed participants doing walk-through tasks with the tool and collected data from surveys and direct feedback from participants.

8.2.1.1 Participants

18 participants were enrolled in the study, split across three teams. The developers' experience with C# ranged from 2–5 years, and they were all familiar with Visual Studio. Three participants were female.

8.2.1.2 Tasks

Tool Observation Task To better understand how a participant uses the tool, the experimenter observed at least one usage of the tool by the participant. During the observation, the participant was asked about how he would use the tool to recover from an interruption. Further, he was asked to perform a simple content-sharing task, which involves the participant indicating which information viewable in his coding history he would feel comfortable sharing and which information he would feel uncomfortable sharing with his teammates.

Episodic Review Task During the episodic review task, participants were asked to recount the events of their previous programming sessions to me. Participants were asked to explain the events in detail sufficient enough for a newcomer to be able to take over the task. To ensure developers did not try glossing over details to a perceived non-expert, I described my credentials as a developer and my familiarity with C# and some of the frameworks they were using, but not with their program.

Participants performed the episodic review task under one of two conditions. In one condition, the participant could use free recall or use the source code difference view provided by Visual Studio to recall recent changes. In the other condition, the participant could use the output of **CODE NARRATIVES** to recall changes. Participants were randomly assigned to a condition prior to doing the task.

8.2.1.3 Tools and Instrumentation

Developers in the study were using Visual Studio 2012. After enrolling in the study, participants installed **CODE NARRATIVES**. Additionally, in order to log code history, developers installed GANJI.

8.2.1.4 Procedure

The first phase was conducted several months before the field observation. The results of the first phase were used to make final design decisions on **CODE NARRATIVES** and the remaining study design. The second phase took place at a major industrial company over a two week period.

Phase 1: Contextual Inquiry and Preparation A contextual inquiry is performed in order to identify constraints and issues that might impact the design and implementation of **CODE NARRATIVES**.

The information obtained during the contextual inquiry highlighted several necessary changes to the implementation of **CODE NARRATIVES**. One major concern was with stability of the tool. Essentially, there was a zero-crash policy instituted by management, if the tool crashed and disrupted the programming environment during development, then the tool would be immediately uninstalled. Another concern was that developers felt that although the proposed tool interface required to review and annotate code history could be useful, they worried it would be too heavy-weight for initially learning and using.

As a result of the inquiry, I redesigned the tool to be more streamlined and to be able to display its output in a web browser. The design decisions and resulting interface for **CODE NARRATIVES** are discussed in section 6.4.

To help prepare for the field observation, **CODE NARRATIVES** was distributed to several developers to help test it for stability. Four developers also prepared each prepared one written report using **CODE NARRATIVES** to describe a recent coding session and another written report written from free recall. The written reports were used to generate seed codes for further content analysis.

Phase 2: Field Observation The second phase spanned two weeks. Upon arrival at the field site, the first priority was to ensure all the components for performing the study were in working order. A volunteer participant helped ensure the tool worked in the local environment. Several issues were identified and addressed before rolling out the tool to other participants.

During the first week, a set of semi-structured interviews were performed. During an interview, a participant provided background on her organization, how her team handled storing organizational knowledge, and what her personal habits for managing interruptions and tracking tasks were. Participants were also asked about possible incentives and obstacles for making contributions to the corporate knowledge base.

To perform field observations, I use *shadowing* [107], a technique for observing actions performing by a participant without requiring direction interaction. In addition, I attended

weekly planning meetings and daily *scrum meetings*, meetings where every developer updates the team on progress, issues, and plans in a time box of a few minutes.

Participants performed the tool observation tasks during the first week and episodic review tasks during the second week. To select when to perform the episodic review task with a participant, I waited for a developer to resume working after a naturally occurring external interruption, such as a developer going to a meeting or teammate asking a question. As soon as the interruption finished, I prompted the developer to ask if he or she would do the episodic review task. I call this method an *interruption tag-a-along*. No developer refused, the task began as soon as the interruption ended. To minimize interference with daily work, the events from episodic review tasks were collected from their verbal recall because the developers perceived written recall as too disruptive. In addition, there was the existing precedent of a very similar task: Reviewing and vetting recent code changes with a teammate before a code commit.

Phase 3: Tool Logs and Exit Survey The third phase occurred after the field observation. An exit survey was administered that asked participants about their experience using **CODE NARRATIVES**. Participants were asked to rate several attributes, such as overhead and usefulness. Participants were asked to identify one positive and one negative aspect of using the tool. Finally, participants offered suggestions on how they thought the tool might be improved and how they would rate possible incentives that would encourage use of the tool. Participants were asked to send their tool usage logs and history to me for analysis.

Participants were also encouraged to continue using **CODE NARRATIVES** to help resume an interrupted programming task. Participants then exported their tool log and code history after a week of additional usage.

8.2.2 Results

During the field study, seven tool observations were performed and twenty-two episodic reviews were collected from eleven participants. After the field observation, seven participants returned exit surveys and three returned tool logs. Performance limitations in the tool limited the number of participants that could return tool logs after extended use.

The data collected from participants allowed me to understand the different type of recall

events that occurred when using **CODE NARRATIVES** (or not), determine if there was additional effort required of developers using **CODE NARRATIVES**, and observe if any reduced resumption time occurred during usage of **CODE NARRATIVES** in the field. More details are available in Appendix D.

8.2.2.1 Recall Events

A content analysis was performed on the collected episodic reviews. In the initial stage, *open coding* [21], a qualitative research method for labeling concepts in data, was performed to define a set of codes that corresponded to different types of recall events. The codes were then categorized to group related codes together. Low-frequency or redundant codes were discarded.

Four categories and twenty-three different type of recall events emerged from the content analysis. Details about the categories and recall event types follow.

Code-oriented recall events occur when developers are referring to making changes to features, mechanics, or structure of the code. *Domain-oriented* recall events occur when developers are referring to the behavior, rules, and purpose of the program. *Narrative-oriented* recall events occur when developers are referring to the context, progress, and status of a programming task. It is narrative-oriented in the sense that the events correspond to common elements observed in narrative structures. *Problem-oriented* recall events occur when developers are referring to the issues, approaches, and solutions associated with problem-solving.

The events types for each category are listed in Table 11.

Table 11: Resulting event types recalled by developers during episodic recall task.

EVENT TYPE	DESCRIPTION
Code-oriented	
Add/Remove Feature	A programming effort related to adding or removing a desired feature from the code.
Mechanics	A programming effort related to low-level programming language concerns (e.g. casting, generics).
Refinement	A small change made to new or recently changed code, such as optimizations, corrections, and guards.
Restructing	A refactoring effort, such as moving code, organization, extracting common code, clean-up.
Domain-oriented	
Logic	A business rule or specification related to a program change.
Rationale	A reason why a change was made.
Purpose/Role	A role of a program element.
Program Behavior	A user-visible changes to the program.
Program/Data Flow	A change to data and program flow.
Data	A desired result of a data query (e.g. get all people with account access).
Narrative-oriented	
Context of Work	A setting for the task. High-level purpose for task.
Transition	An event described in relation to another event or task.
Sterotype/Pattern	A common idiom to describe events.
Milestone	A statement of progress on task, (e.g. got a feature working).
Side Note	An external event that review made them remember.
Incomplete Work	A statement about work that is still not done yet.
Problem-oriented	
Problem	An issue faced by the programmer.
Approach	A strategy for solving a problem.
Resolution	An implemented approach that serves as a solution for the problem.
Mistake	An error they corrected (e.g. wrongly pasted code, bug).
Incorrect Code	A realization that they did not finish something.
Backtracking	A reversion on a decision to add code.
Experimenting	A set of different solutions and approaches that were attempted.

8.2.2.2 Recall Events

Recall events that occurred in the free recall condition were compared with recall events that occurred in the memlet condition. To determine differences in recall, both the total number of recall events, category of recall events, and type of recall events were compared between the two conditions.

A median of twelve events were recalled in the free recall condition versus a median of twenty events in the memlet condition. When recall events are aggregated into the four categories of recall events, narrative-oriented and problem-oriented events were recalled more often in the memlet condition. Code-oriented and domain-oriented events were evenly recalled in both conditions. A breakdown of the recall frequency can be observed in Table 12.

Table 12: Programmers A–K number of recall events as aggregated by category and separated by task condition.

	A	B	C	D	E	F	G	H	I	J	K
FREE RECALL											
Code-Oriented	7	3	4	3	3	3	2	10	5	5	2
Domain-Oriented	8	7	2	9	2	5	3	10	8	6	7
Narrative-Oriented	1	0	0	1	1	1	0	0	1	0	1
Problem-Oriented	1	1	2	2	4	2	0	1	0	4	2
MEMLET											
Code-Oriented	7	3	3	11	4	4	12	16	16	1	1
Domain-Oriented	11	8	5	18	8	4	7	8	8	6	0
Narrative-Oriented	4	1	3	4	3	4	9	11	11	2	3
Problem-Oriented	7	7	2	2	6	1	11	7	7	3	2

Further differences can be observed when examining the frequency of recall events in each category individually.

Code-Oriented Developers recalled a similar number of code-oriented recall events in both conditions. Developers in the free recall condition had more Add/Remove Feature recall events, whereas developers in the memlet condition had more Mechanics, Refinement, and Restructuring recall events.

Domain-Oriented Developers recalled a similar number of domain-oriented recall events in both conditions. Developers in the free recall condition had more Purpose/Role recall events, whereas developers in the memlet condition had more Program/Data Flow and Data recall events.

Narrative-Oriented Developers in the memlet condition recalled more narrative-oriented recall events. Developers in both conditions had Context of Work recall events, but developers in the free recall condition had little narrative-oriented recall events in general.

Problem-Oriented Developers in the memlet condition recalled more problem-oriented recall events. Developers in both conditions had Problem recall events, but developers in the memlet condition had more Approach, Mistake, Backtracking, and Experimenting recall events.

8.2.2.3 Recall Effort

The recall effort was measured and compared between the free recall and memlet conditions. To determine recall effort, the time spent performing the episodic review task was measured. To ensure that the number of recalled events was not simply a function of recall effort, a correlation analysis was performed.

Developers generally spent several minutes performing the episodic review task. Developers in the free recall condition spent a median four minutes performing the episodic review, and developers in the memlet condition spent a median five minutes performing the episodic review.

To test if there is any relation between the time spent on the recall task and the number of events recalled, a correlation analysis was performed. A Spearman rank correlation was performed between the number of events recalled in the free recall condition and time spent on task. A Spearman rank correlation was performed between the number of events recalled in the memlet condition and time spent on task. The resulting Spearman rank correlations were -.29 and 0.55, respectively.

The results do not indicate that a strong correlation exists (a value greater than .7) between time spent on the recall task and the number of events recalled. One explanation for the moderate correlation between time spent on the recall task and the number of events recalled in the memlet

condition is that there were two episodic review tasks that took ten and twelve minutes to perform. In both cases there was some considerable refactoring work that had been done that had touched many files. **CODE NARRATIVES** were not particularly effective at presenting these types of changes, making the developer spent more time scrolling through the tool's view of the code.

8.2.2.4 Tool Log Analysis

The three logs returned by participants revealed a total of ten instances of tool usage prior to resuming a programming session that had at least one edit. To determine the speed of resumption, the edit lag was measured by taking the difference between the time of the first edit and the time of the first navigation. Ten programming sessions that had at least one edit, but no tool usage, were randomly selected from the code history to perform this comparison.

The average edit lag of a programming session resumed with **CODE NARRATIVES** was 3 minutes and 42 seconds, and the average edit lag of a programming session resumed without was 6 minutes and 18 seconds. However, these differences are not statistically significant from a 2-tailed paired t-test ($p < 0.08$, effect size .87).

8.2.2.5 Benefits and Barriers

Observations from walk-through tasks, shadowing developers, and data from surveys were collected and analyzed to determine a set of perceived benefits and barriers to using **CODE NARRATIVES** in practice.

Benefits There were several benefits that were reported by developers using **CODE NARRATIVES**.

Episodic coding companion. Some developers found that if they kept open **CODE NARRATIVES**, that they would often refer back to it as they performed the programming task. One developer said when he forgot where one piece of recently edited code was, rather than remembering how to navigate to it within their IDE, he would instead find it very easily in the display of **CODE NARRATIVES**. Additionally, viewing content in **CODE NARRATIVES** allowed the developer to avoid potentially disrupting his own IDE windows and lose focus when searching for lost code.

Reducing overhead One developer described how having **CODE NARRATIVES** could reduce overhead typically associated with he manually performed an episodic recall task:

> *Locating the code in question may take a few minutes. If I don't recall the source file it is in, then I may need to search the source for a unique method name, variable name, source comment, etc. If I don't recall any unique key words in code, then I may search the change request system for the change request I used when applying the programming task I'm trying to recall. If I'm really desperate, I may check old e-mail to determine the approximate date that I made the change and then review source code history to find the changes I applied previously. The first strategy is not painful at all, the last is obviously tedious and time consuming.*

> *This tool can help me to start coding better. I think if I was about to start an incomplete task, [***CODE NARRATIVES** *] would be very helpful. It would help me to write the necessary code faster than I would write it without using the tool.*

Long-term task-switching. Some developers reporting finding **CODE NARRATIVES** particularly helpful for resuming a task that had been put on hold for a few days while working on another task. Developers found the combination of spending several days away from a task while working on something completely different as very disruptive. But when resuming these long-term interrupted tasks, developers found the process of performing the episodic review very restorative.

Recalling Events Several developers noted that reviewing changes with **CODE NARRATIVES** helped them reflect on their programming activity.

> *When I try to remember changes, all that I can remember is the changes that led to the final result, to the current test, but when writing code usually programmers tend to write something, delete it, rewrite it, and actual code written is maybe five times more than final result. [***CODE NARRATIVES** *] helps me to remember changes*

that I added but then changed my mind, or redone something, or refactored something.

One developer describes the information he found really important to recall and document:

Details about why the code was refactored or modified and the rationale. The main rationale of the technical decision should be documented and those people participating should also be mentioned, thereby leading to a history of decisions.

Code reviews. First-level management had recently received orders from upper management to perform more code reviews. Because code reviews might take place two to four weeks after the original programming tasks were completed, several developers and management found using **CODE NARRATIVES** a useful aid for developers to recall details of the programming task and prepare for the code review.

Future Possibilities. In general, most developers were excited about the future possibilities enabled by future iterations of the memlet and code history infrastructure. Some developers proposed the idea of hooking into the code history via programatic access and other external events. This would allow developers to replay or replicate the changes to other different projects or consolidate code history after checking in code into a source control repository. One developer asked for more analytics on the code history be provided in order to suggest possible lessons learned and highlight interesting events. Another developer took this further and suggested that the tool include real-time analysis that would help prevent future mistakes based on the team's code history.

Barriers There were several barriers to using **CODE NARRATIVES** that were reported by developers.

History Selection Locating the range of history relevant to an episodic recall task was a barrier to using the tool, especially as more history accumulated over time. Because developers

preferred to view history in a chronological order, they would need to scroll through past events and locate the start of history that they were interested in.

One feature that helped alleviate this barrier was the introduction of collapsible regions of events. This allowed developers to more quickly scroll through the history and locate the region of history relevant to them. This worked particularly well if the developer was indexing into the history based on temporal information, but not as well if she was trying to locate a specific event.

Temporal Disorientation Similar to how developers can experience spatial disorientation when navigating graphical user interfaces, when reviewing **CODE NARRATIVES** developers sometimes experienced *temporal disorientation*, a sense of disorientation that occurred when a developer can no longer associate the events in **CODE NARRATIVES** with her mental timeline of events.

Developers suggested two things that could help. First, temporal landmarks could help orient developers as they scrolled through the view. Second, simply reducing the number of events could reduce the number of steps that developers would have to mentally take through their mental timeline.

Finally, one developer pointed out how recall could be better supported by providing other frame of references, such as commits to the main source code repository:

> *While reviewing the changes, I often found myself saying, "that's when I was doing X!" At the same time, I often wanted to know when the automark commits were in relation to my actual commits. For instance, it would be nice to know "this stream of commits was my work between commits X and Y."*

Save Bursts and Save Twitching In the implementation of GANJI's code history, updates to code history occurred on every file save. However, some developers demonstrated an unconscious behavior that I call a *save twitching*, which is manifested as frequent saves of a change that is being made to the same code line. Developers were often not self-aware of save twitching. For other developers, there was also a tendency for edits to occur in

save bursts. That is, over a span of a few minutes, developers may make many edits and subsequent saves, and then cool-off to reflect for a few minutes.

Save twitching resulted in **CODE NARRATIVES** presenting many events of the same edit in progress. Save bursts resulted in redundant information being displayed. Because each change event displays the surrounding contextual lines above and below a change, if two or more change events occurred close together, they displayed as two different events, that would duplicate the surrounding context. If they were considered one event, then there would be more opportunities to share context.

Both save twitching and save bursts introduced considerable noise for those developers and made comprehension of the events more difficult.

One feature that helped alleviate this barrier was the introduction of *temporal fuzzing*. Temporal fuzzing is a preprocessing step applied to change events that merges multiple change events if they occurred in a small window of time. For example, when temporal fuzzing was introduced using a three minute window on two developer's code history, the number of events was reduced by 50% and 80% respectively. The developers were highly satisfied with the reduction of events and of redundant context. Although the developers did point out that too strong of a temporal fuzz may affect recall, they found the new result much more clean. Further improvements to the temporal fuzzing may be needed to find the right balance.

Performance As code history accumulated, some developers began to experience delay in generating **CODE NARRATIVES**, taking as long as two minutes. Any small delay greatly hampers developers resuming programming tasks, as they would rather just get back to the code, instead of waiting around for a tool to support them.

There are several possible ways to optimize the implementation of **CODE NARRATIVES** to reduce performance overhead, including caching and pre-calculation. But performance can be a barrier even before a tool is used. In one conversation with a developer, he was concerned with using any new developer tool and its potential impact on his programming environment's performance.

Will the tool take up lots of space, will it slow things down?

Being able to explain the tool architecture and implementation and its performance was an important barrier to overcome to get this developer to even install the tool.

Awareness and Opportunity Some developers reported that they forgot to use the tool to help resume their task. They would resume their task using resumption strategies such as navigation or viewing source code history, and then realize half-way through that it might have been easier to use **CODE NARRATIVES**. Awareness is a problem of habit, training, and workflow integration.

Other developers were not aware of opportunities of using the tool in other contexts, such as helping them remember what they did before writing a commit message, until it was explicitly pointed out. Opportunity is partially a problem of education, exposure, and encouragement.

Awareness and opportunity barriers have parallels in other tool usage scenarios, such as barriers to using refactoring tools [136]. Some of the research ideas for encouraging tool usage might apply to memlets.

Filtering Changes and Summarizing Systematic Changes Several developers expressed surprise on how much changes they generated in even just one morning of coding. This was especially the case for developers that mixed refactoring efforts with other coding efforts. For example, if a developer renamed a program element, she would have to replace every occurrence of that program element in the code, which would produce many changes to view in the **CODE NARRATIVES**. Other developers felt that some types of changes did not have as high priority for them if there were already other more interesting changes to present.

All developers were strongly interested in automatically recognizing and summarizing systematic changes.

The brain is very good at filtering out things, I can see these diffs and know it would be renames. Still it would be good if they could be done automatically.

[With these renames clogging up the view], there was only about 30% meat to review. It was helpful to go through the changes, but there was a lot of noise to go through too.

Another feature that developers were strongly interested in was the ability to summarize some changes as natural language or hide other changes, such as deleted code changes if there was already too much history to be viewed.

8.2.3 Limitations

There are several limitations and threats to validity of the **CODE NARRATIVES** study.

One limitation of the episodic memory study is that I only studied recollection of events that had occurred recently. The same results may not generalize to longer interruption time-frames, such as several days or weeks after the programming session has occurred. Another limitation is that the version of **CODE NARRATIVES** that participants used in the study was simplified to focus on change events. Including other types of events may have changed the types of event recalled and the amount of effort required.

There are several threats that limit applicability of the results. One external threat is that the data collected in the study only refers to field conditions I observed and may not generalize to other companies. These type of events may not be as frequent or occur in the same fashion in other contexts. An internal threat is that the data collected in the ethnographic study only come from one experimenter. As a result, although there were differences observed in several measures, I cannot say if those differences are significant or not without further establishing the consistency and reliability of my event coding. In future work, new observations need to be validated by having multiple experimenters independently observe the same phenomena and consistently producing the same codes in multiple settings.

8.2.3.1 Summary

CODE NARRATIVES were evaluated to measure if there was any differences in recall created with the memlet, and understand if any relationships existed with recall effort or event type. Secondly, to address Research Question 7, *What are the observable benefits and disadvantages with*

using memlets for interruption recovery in software development?, data from surveys and field observations was analyzed to identify barriers and benefits to the tool.

Performing episodic recall tasks with **CODE NARRATIVES** enabled developers to recall more events when compared with free recall. Second, the nature of recalled events were different. Based on an analysis of tool logs, developers using **CODE NARRATIVES** demonstrated a quicker resumption time on average—but there was not enough data to reach statistical significance.

Developers using free recall mainly discussed what features they were adding and the specifications that they were given to complete the task. This suggests that when developers are unaided, they have trouble recollecting details about their programming task and instead drew upon task goals. This would help explain why specifications and features were frequently discussed. Even though developers had access to source control differences, the lack of temporal presentation of change events reduced the effectiveness of the tool for recall.

Developers using **CODE NARRATIVES**, discussed features and specifications, but also included more details about problem-solving and brought up interesting events and stories related to the task even though they weren't explicitly captured by the tool. This suggests that when developers had access to **CODE NARRATIVES**, they could use the information to recall contextual details about the events such as problems they encountered, structural changes they made to the code, and rationale behind certain changes.

The effort involved with performing the episodic recall task took three to five minutes for most cases. There was no significant difference in performing the task with free recall or the memlet, and there was no strong correlation between time spent and the amount of events recalled. This result suggests using **CODE NARRATIVES** can result improved recall without imposing significant resumption costs and that the quality of recall does not necessarily depend on how long developers spent using the tool.

Finally, six benefits and six barriers were identified. The identified benefits highlight additional advantages that can be gained from using the tool and suggest possible features that may further improve developer's productivity. The identified barriers pose risks to tool adoption and usage among practitioners. The identified benefits and barriers are opportunities and challenges for future research.

The results are consistent with my claim derived from my conceptual framework. Episodic memory has two memory failures that need support: recollection failures and source failures. The performance of free recall is consistent with poor support of recollection failures and source failures. Developers using free recall experience more source memory failures because they forget contextual details about events, as evidenced by lower recall of problem-oriented and narrative-oriented events. Developers using free recall aided by source control differences experience recollection failures because they cannot recall the proper sequence of events that occurred, as evidenced by lower recall of code-oriented events such as refactoring and refinement events. Developers are able to overcome these memory failures by using **CODE NARRATIVES**.

8.3 Smart Reminders: Prospective Reminders Study

To evaluate **Claim** 1, **SMART REMINDERS** *reduce monitoring failure by introducing mechanisms for conditional triggers and reduce prompting failure by introducing mechanisms for enhancing awareness of reminders.*, the following study was conducted. Because prospective memory often involves remembering to perform actions several hours or days later, an experimental design that can take place over the course of several days is appropriate. Additionally, an experimental design that allows the study of real reminders instead of artificially imposed ones, enables participants to be self-directed without active instruction by an experimenter.

In this section, I provide both qualitative and quantitative evidence for my claim, based on measures of the completion speed and completion rate of reminders created with and without the smart reminder memlet, and how design variations of the smart reminder result in measures that can be explained by my conceptual framework.

8.3.1 Method

The developers in my study were asked to create reminders with and without the aid of **SMART REMINDERS**. The developers worked on programming tasks that were part of their daily jobs. My decision to study developers in the context of developers' actual work instead of in the lab was driven by the difficulty of deriving artificial tasks that professional programmers would be willing to work on for weeks at a time.

The study is divided into two phases, each lasting one week (this typically included 8 days of activity as programmers worked weekends too). The first phase (Baseline) is used to establish a baseline of experimental measures. In the second phase (Intervention), the participants were instructed to use **SMART REMINDERS**.

In the first phase, I examined the types of reminders created using the memlet, the code history associated with the reminder, the amount of exposure to the reminder, and the time when the reminder was mark completed, if at all. In the second phase, I examined the code history to find when reminder messages in TODO notes were created and deleted. I also collect measures of cognitive load imposed by the tool. After the phases completed, I analyzed the survey results to extract qualitative feedback about the tool.

8.3.1.1 Participants

We recruited eight developers with experience with at least three years experience using C#. All reported having previously used Visual Studio and were involved in active software development projects for work. The eight developers we studied were all male, aged from 19 to 38, with an average age of 29. Several participants reported working on several projects at the same time, which resulted in them having to often switch and suspend work on a task.

8.3.1.2 Tasks

Cognitive Load Task To measure cognitive load, participants were asked to perform a NASA-TLX cognitive load assessment test [79] before a programming session started. The test measures the six different factors related to cognitive load on a twenty point scale. A final 100 point score is derived from six ratings. After a reminder is created, the test is given again to measure differences in cognitive load.

Participants perform the cognitive load task twice for each of the three reminder mechanisms (Attach Here, Attach Everywhere, and Due By), meaning that each participant performs six cognitive load tasks. Instructions for the cognitive load task are given to participants after they had at least one day to gain familiarity with **SMART REMINDERS**.

Reminder Task Participants are given instructions on how to create reminders with the smart reminder tool. When participants were in the first phase of the study, they were instructed to create TODO notes that was consistent with their normal development practices. Two developers reported not frequently using TODO notes, but adopted the practice for the study. When participants were in the second phase of the study, they were instructed to create an attachable reminder or Due By reminder using the memlet in addition to any TODO note they had created. Participants were self-directed in determining which type of reminder to create.

8.3.1.3 Tools and Instrumentation

The developers used Visual Studio 2010 or 2012. The smart reminder memlet was installed on the system and verified to work. A test data export trial was performed to make sure data collection

was in order.

Because my analyses would involve understanding when reminders for tasks were created, seen, and potentially completed, the developers also installed GANJI to record their full interaction history.

8.3.1.4 Procedure

The developers worked on existing freelancing or personal projects on their own equipment. There were given instructions in how to install the tools and given a tutorial to explaining tool usage. I then communicated with developers to verify that they could get the tools to work, that they understood the different features offered by the tools, and that they could export data logged by GANJI and the tools.

There were then told that they would spend one week programming with the reminder tools disabled and one week with the tools enabled.

When a developer wanted to create a reminder, they wrote the reminder message using the code editor. They could then decide what type of reminder they wanted to receive: attachable or Due By reminder. When they completed the reminder, they were instructed to mark the reminder complete.

Afterward, I collected developers' logged data, administered a survey for feedback about the tool, and then answered any questions about the study.

8.3.2 Results

54 unaided reminders and 76 **SMART REMINDERS** were collected from the eight participants. 48 cognitive load tasks were reported by participants. After both phases of the study, eight participants returned surveys.

The data collected from participants allowed me to understand the different type of **SMART REMINDERS** that were created with the memlet, determine if there was any differences in completion speed and completion rate of reminders, and understand if any relationships existed between exposure to reminders and cognitive load. The same participants are compared in the baseline and memlet phase. More details are available in Appendix D.

8.3.2.1 Reminder Types

An analysis was performed on the collected reminders. When in the memlet phase, participants had complete control over the selection of the the reminder mechanism used. To understand which types of reminder mechanisms developers selected, reminders were divided up by type of reminder mechanism used. The following Table 13 summarizes the resulting division of types.

Table 13: Reminders as divided by reminder mechanism.

REMINDER MECHANISM	REMINDERS CREATED
Attach Everywhere	19
Due By	22
Attach Here	35
Unaided	54

8.3.2.2 Completion Speed

Reminders that occurred in the unaided phase were compared with reminders that occurred in the memlet phase. Completion speed provides a possible measure of the strength and effectiveness of a reminder mechanism. To measure completion speed, the time between creation of the reminder and the time it was marked completed was computed. The results are summarized in Table 14.

For unaided reminders, the time when a reminder was added in the code history was compared with the time it was deleted in the code history. To determine differences in completion speed, reminders that were completed were compared between the two phases. Attachable reminders (Attach Here and Attach Everywhere) on average were completed nearly three days faster than either Due By reminders or unaided reminders. This difference is significant by 2-tailed unpaired t-test: Due By versus Attach Here ($p < 0.002$, effect size 1.04), Due By versus Attach Everywhere ($p < 0.001$, effect size 1.39), unaided versus Attach Here ($p < 0.0001$, effect size 1.21), and unaided versus Attach Everywhere ($p < 0.0001$, effect size, effect size 1.52).

Table 14: Completed reminders and mean completion speed as divided by reminder mechanism.

MECHANISM	# COMPLETED	MEAN COMPLETION SPEED
Attach Everywhere	15	1.49 days
Attach Here	22	1.55 days
Due By	17	4.39 days
Unaided	24	4.86 days

8.3.2.3 Completion Rate

The completion rate was measured and compared between the different reminder mechanisms. To determine completion rate, the reminders were analyzed to see if they had been marked complete. For unaided reminders, reminders that were added in the code history were checked against a corresponding deletion in the code history. To compare the difference in completion rate, a Fischer's exact test was used, which is appropriate for measuring differences in frequency of categorial data (completed vs. non-completed).

The differences in completion rate are displayed in Table 15.

Table 15: Rate of completed reminders as divided by reminder mechanism.

MECHANISM	COMPLETION RATE
Unaided	44%
Attach Here	62%
Due By	77%
Attach Everywhere	79%

The developers on average completed 65% of the reminders during the timeframe of the study. Completion rate varied with reminder mechanism. Unaided reminders had the lowest completion rate followed by Attach Here. Due By and Attach Everywhere reminders had the highest completion rates. The differences in completion rate between unaided reminders and Due By and Attach Everywhere reminders were significantly different by the Fischer's exact test ($p < .015$, effect size .303) and ($p < .011$, effect size .298) respectively.

8.3.2.4 Exposure

For attachable reminders, particularly Attach Here reminders, the amount of exposure to a reminder was measured. To measure exposure, the number of visit events to a file was measured by analyzing the code history when the reminder was in effect. To provide a normalized way of comparing exposure, a file was categorized as having low, medium, and high levels of exposure based on a percentile rank of all file exposures. Finally, the file's normalized exposure was then compared between Attach Here reminders that were completed and those that were not completed. To compare the difference in normalized exposure, a chi-squared test was used, which is appropriate for measuring differences in frequency of multi-dimensional data (low, medium, high).

Incomplete Attach Here reminders appeared more often in low exposure files, whereas completed Attach Here reminders more often appeared in high and medium exposure files. This difference is significantly different by a chi-squared test with Yates correction ($p < 0.016$, effect size .485).

One developer noted why Attach Here worked well only some of the times.

> *Attach here would work if I am constantly going to that source file.*

8.3.2.5 Cognitive Load

The cognitive load associated with using specific reminder mechanisms was measured (Attach Here, Attach Everywhere, and Due By). To measure cognitive load, the NASA-TLX assessment test was used. Sixteen cognitive load differentials were collected for each reminder type. To calculate the differential, the raw TLX was compared between the pre-test and post-test. For each reminder mechanism, an average differential was then calculated for each participant. To compare the differences in cognitive load between reminder mechanisms, the average differential given by a participant was compared across conditions using a paired t-test.

Based on a 0-100 scale, the average differential for cognitive load decreased 16 points for Due By reminders, increased .2 points for Attach Here reminders, and increased 1 point for Attach Everywhere reminders.

Comments from the surveys helped explained the differences in cognitive load. One developer explained why they felt cognitive load was reduced by Due By reminders.

> *I really forget about tasks, so they allow me to focus.*

On the other hand, attachable reminders received mixed reaction. Some developers felt they had less to worry about since they could use it to control their attention.

> *It actually relieved some stress. Since I did not have to remember or look for what I had to do next.*

However, if reminders started to pile up or they were too frequently exposed to the reminder, they felt it caused additional cognitive load.

> *It reminds me of all I still have to do.*

8.3.2.6 Individual Factors

There were differences in the frequency that participants used TODO reminders and the type of reminder mechanisms they selected as reminders.

Reminder Breakdown Programmer A heavily used Due By reminders, completing seven. He also completed two Attach Here reminders and one Attach Everywhere reminder. He left one Attach Here reminder incomplete. During his baseline activity, he completed three reminders and left four reminders incomplete.

Programmer B mostly used Attach Everywhere reminders, completing six. He also completed one Attach Here reminder and three Due By reminders. He left three Attach Everywhere reminders, one Attach Here reminder, and one Due By reminder incomplete. During his baseline activity, he completed three reminders and left five incomplete.

Programmer C mostly used Attach Here reminders, completing three. He also completed one Due By reminder and left another one incomplete. During his baseline activity, he completed one reminder and left two incomplete.

Programmer D mostly used Attach Everywhere reminders, completing four. He left one Attach Everywhere reminder and one Attach Here reminder incomplete. During his baseline activity, he completed one two reminders and left three incomplete.

Programmer E mostly used Attach Here reminders, but left three of them incomplete. He completed one Attach Here reminder and one Attach Everywhere reminder. During his baseline activity, he completed one reminder and left two incomplete.

Programmer F mostly used Attach Everywhere reminders, completing three. He also completed two Attach Here reminders. During his baseline activity, he completed three reminders and left five incomplete.

Programmer G mostly used Attach Here reminders, completing eight. He also completed four Due By reminders. He left two Due By and two Attach Here reminders incomplete. During his baseline activity, he completed six reminders and left three incomplete.

Programmer H mostly used Attach Here reminders, completing five. He also completed two Due By reminders. He left several five Attach Here reminders incomplete and one Due By reminder incomplete. During his baseline activity, he completed five reminders and left six incomplete.

Observations Attachable reminders were the most popular type of reminder used by developers. Four programmers mostly used Attach Here reminders, three mostly used Attach Everywhere reminders, and one mostly used Due By reminders. Some reminders may have been more appropriate for particular reminder mechanisms. For example, Due By reminders may have been more applicable for reminders that had specific deadlines in mind or were waiting for other work to be completed.

Some programmers may have been influenced to create more reminders than they typically do. All but one programmer created a higher number of reminders when they had extra reminder mechanisms available as compared to their baseline. All programmers improved their completion rate of reminders when compared to their baseline activity. After the baseline, the developers may have been able to use the different mechanisms to better understand their expectations when creating a reminder. That is, they may have been less likely to create a reminder that they did not believe they could complete given the consequence: a pending build error or cluttering of their

screen real estate.

8.3.2.7 Temporal Factors

Developers created reminders uniformly throughout their coding efforts in the first 5 days (ranging 19–22), with a reduced but uniform number of reminders created day 5 to day 8 (ranging 10–13). However, individually, programmers created several reminders together in batches, and there were sometimes stretches where no reminders were created. Developers may have been leaving reminders before suspending a task. The patches may have resulted from switching between different projects for several days.

There are several factors related to the creation time of a reminder that can influence outcome. Incompletion rate may have been influenced by time of creation and the nature of the phase. Due By reminders may have had a longer completion time because programmers set reminders dates several days later than other types of reminders that could be completed more quickly.

During the baseline phase, in examining reminders created during the first half of the phase (early), 80% of the completed reminders were created during this time versus 50% of the incomplete reminders. This suggests that reminders that get created later in the baseline phase (late) were more likely to be categorized as incomplete. Attach Here reminders had the highest incompletion rate of the reminder mechanisms introduced during the second phase. 60% of the completed reminders were created early versus 53% of the incomplete reminders. Comparing when baseline reminders and Attach Here reminders are created, there is a comparable distribution of incomplete reminders created early on. However, there is a higher rate of Attach Here reminders that were created late and then completed than baseline reminders that were created late.

Reminders may very in priority and applicability. Due By reminders had a high completion time when compared to other reminder mechanisms. Due By reminders provide an explicit note of creation time, expected due time, and actual completion time. This allows us to characterize observed completion time in terms of expected due dates. There is an average 2.5 days offset between the creation date and expected due date for a reminder. This result explains why completion time is typically high for a Due By reminder. The observed difference between the

due by date and completion date can help characterize the necessity of the reminder (are all the reminders completed early?) and the strength of the reminder (how soon the reminder is completed therefore after). Due By reminders were completed an average 1.3 days after the due date. Only four of the seventeen Due By reminders were completed a day early. Nine out of the seventeen were completed on the same day or the day after of the due date. Finally, three reminders were completed three, seven, and eight days after the due date (The three day delayed reminder was snoozed). The results suggest that most Due By reminders are completed within a day of their due date. When switching between projects, there is still a possibility for a long delay. Finally all incomplete Due By reminders had due dates set after the second phase.

8.3.3 Observations

The surveys were analyzed for explanations and feedback about how developers used **SMART REMINDERS**. Overall, developers had positive feedback about using **SMART REMINDERS**. Most feedback centered on the reminder mechanisms.

Several developers explained why they thought Attach Everywhere was very effective for helping them remember to do an action.

> *I used Attach everywhere so I remember where I need to go to add/repair code; it helped remind me the most.*
>
> *Attach everywhere, sometime if we need to fix a issue which reference to many other part, I want using attach everywhere to reminder me. If I try to change other part, I need more careful to avoid affect this issue.*

Most developers reported that they would like some way to manage how often they were exposed to a reminder and provided a variety of different recommendations to reduce exposure. One possible strategy is to only display the reminder initially when resuming a programming session, then fade it after a few minutes. Another strategy is to use a sinusoidal function that periodically brings a reminder in and out of focus. Finally, another strategy includes using hot keys or ambient hints to help developers toggle the display of reminders.

On the other hand, some developers also explained that after overexposure, they would start to ignore looking at the reminders themselves because they were already aware of them.

One barrier developers mentioned was that attachable reminders could start to crowd the viewport once more than ten were active. Finally, one developer did mention an awareness barrier that sometimes prevented him from creating a reminder with the tool.

> *I just need to remember to use it at times since I forget that its there. I enjoy the fact that its hidden until you need it, but sometimes it slips your mind and you fall back into the same patterns you followed before using [***SMART REMINDERS** *].*

8.3.4 Limitations

There are several limitations and threats to validity of the **SMART REMINDERS** study.

One limitation is that the prospective memory study examined reminders being completed in the time frame of one week. If participants had a longer time of evaluation, more reminders messages may have been completed. For a comparison, researchers have previously found in a study of open-source projects that after one month, there is a wide variance in the completion rate of TODO messages [188]. It is not until six months that the percentage of incomplete TODO messages finally drops to between 3% and 15%. Even with this limitation, incomplete reminders in my study ranged between 22% and 56%, which is not out-of-step of the ranges that happen in other software development projects.

There are some limitations related to developer behavior. One limitation was that developers choose their own reminder mechanisms. For example, developers may have chosen Due By reminders for tasks that could not be done immediately without a few days passing, whereas they may have chosen attachable reminders for tasks that could be done opportunistically. While this may affect comparisons between different reminder mechanisms, it still allows for a comparison against unaided reminder messages. Another limitation is that the study recruited mostly experienced developers. Expert developers may use reminders differently than novices, which would need to be observed by sampling from a broader population.

Finally, another limitation was that the developers may been influenced in how they created reminders with the memlets. They may have wanted to demonstrate that they were using the new tools even if they were not naturally inclined to do so. Another influence is that they may have been less likely to create a reminder that they did not believe they could complete given the

consequence: a pending build error or occupying their screen real estate.

One external threat is that the practice of using TODO messages in code, although common, is not uniformly practiced. There are many variations in rules and conventions for using TODO messages in code, which may not generalize to other companies. For example, the Due By reminder introduces an error message when compiling the program. Although this error does not actually prevent execution and testing of the program, some developers can have negative reactions to the mechanism based on their individual principles and practices. Similarly, although developers were engaged in their own programming tasks based on their work settings, the extent to which other programmers engage in the same types of tasks is unknown. An internal threat is that reminders for programming tasks may have not been sufficiently random between the two phases selected. Another internal threat is related to how completion of unaided reminders was measured. It is possible that a participant may have completed a task, but forgot to mark the task as complete (by deleting the TODO message). There may therefore be a subsequent lag in when the unaided reminder was deleted. Second, the nature of the reminder mechanisms may have made it more likely for a participant to mark the reminder complete.

8.3.5 Summary

SMART REMINDERS were evaluated to measure if there was any differences in completion speed and completion rate of reminders created with the memlet, and understand if any relationships existed with exposure to reminders and cognitive load.

Due By reminders had slower completion speeds, but high completion rates. This suggests that the reminder mechanism was effective in prompting the user to act on the reminder when they needed to act on it, but did not cause them to act sooner on it.

Attach Here had fast completion speed, but relatively low completion rates. Exposure helps explain why. Reminders that were only visible in files that had low exposure had the tendency to not be completed when compared with reminders only visible in high-traffic files. If a file is not visited, then the reminder mechanism has no influence, making it not much different than an unaided reminder.

Attach Everywhere reminders had fast completion speed and high completion rates, but

imposed slightly higher cognitive load on developers. The constant exposure to the reminders helped get them done and fast, but at the cost of occasionally overwhelming the developer.

The results are consistent with my claim derived from my conceptual framework. Prospective memory has two processes that need support: prompting and monitoring. The performance of Due By reminders is consistent with strong prompting support but weak monitoring support. Developers can "set and forget" the reminders because they are confident they will not forget the reminder. This, in turn, does not engage prospective memory's monitoring process.

The performance of Attachable reminders is consistent with moderate prompting support and strong monitoring support. This is supported by the difference in performance of Attach Here and Attach Everywhere reminders—Attach Here reminders have significantly lower completion rates than Attach Everywhere reminders because of lower exposure. By constantly exposing the developers to the reminders, they allowed developer's monitoring processes to be engaged as they were more self-aware of having to do the reminders even without having to visually process the content of the reminders. Still, engaging a monitoring processes is not without consequence; in other studies of prospective memory, higher cognitive load is associated with reminders that are supported by monitoring processes rather than by prompting processes [209]. Prompting support is moderate because it works only as well as the frequency of exposure, and overexposure can even lead to the reminders being ignored.

8.4 Limitations

The two performed studies have additional limitations and threats to validity in common. There are several unevaluated claims.

In both studies, there are threats to construct validity. The conceptual framework and the resulting tool designs are dependent on sound construction from theory. In the process of framing and presentation of the cognitive neuroscience of memory, there are several limitations present. Literature on reasoning and problem solving is not included in the design of the framework, which may have additional implications for tool designs. Interactions among memory types is not described. For example, prospective memory cooperates with associative memory to hold long-term intentions that will be prompted. Finally, there may be additional information needs

and alternative visual designs that could be derived.

Even though both studies took place in the context of workplace environments, threats to ecological validity still exist. For example, the design and implementation of the memlets may not scale to the size or constraints imposed by software in industry. Developers in the studies were already experiencing problems with performance for **CODE NARRATIVES** and with screen real estate and overexposure in **SMART REMINDERS**. Finally, there were several barriers that were identified that would have to be addressed before memlets could be applied in practice.

8.4.1 Remaining Claims

Finally, three claims remain unevaluated. I provide a basic outline of an experiment for evaluating the claims in future research.

To evaluate Claim 2, **TOUCH POINTS** *reduce the number of errors and omissions made by a programmer during an interrupted task by increasing the ability for a programmer to attend to multiple programming locations during the programming task,* the following study is proposed.

Because a programming task involving many code changes scattered throughout a code base will heavily exercise attentive memory. Interruption to such as task should increase the number of errors and omissions made by the programmer. A study examining the differences in error rates made by the programmer could be performed. The study could determine if less errors occur with **TOUCH POINTS**. When performing the experimental task, participants will be interrupted and instructed to perform an unrelated task. Participants in the Touch Points group, will have **TOUCH POINTS** available when resuming the task. One expected outcome is that participants in the control group will have a higher correction rate than the Touch Point group. Interruptions to the control group will increase the likelihood that participants omit a necessary change and must make a later correction.

To evaluate Claim 3, *Associative links reduce the navigational effort of a programmer recovering from an interruption by improving the recognition and recall of program elements visited during the task,* one possible approach is to use a study based on cued recall. With *cued recall,* partial information is given to a participant, and she is asked to correctly recall the appropriate answer. Cued recall is an effective method of evaluation because it allows for a uniform comparison across

participants and for understanding the effects of different cues on recall.

A study examining differences in navigation performance with and without **ASSOCIATIVE LINKS** could be performed. The study could determine if more navigational knowledge is retained by participants with the memlet. To perform the study, the participant is either in a condition of being aided with the **ASSOCIATIVE LINKS** or in a control condition when performing the navigation tasks. After a distraction task, the programmer is given a cued recall test. The cued recall test involves answering a set of questions that ask the participant to identify which item correctly corresponds to each answer. Each question requires information that would have been learned during a navigation task that was previously performed. When answering such a question, it should be possible to respond by recalling from associative memory and not by logically deducing the answer from the partial information. One expected outcome is that when participants are asked to recall information from the navigation tasks, they will have a higher cued recall score and a faster recall time when in the associative links condition than in the control condition.

To evaluate Claim 5, **SKETCHLETS** *reduce the effort of priming concepts necessary for understanding code*, the following study is proposed.

Because abstractions that have not been used in a while are not likely to be primed, a programming task involving revisiting old code should heavily exercise the priming processes of conceptual memory. In long-term interruptions, the amount of time needed to prime abstractions should increase. A study examining the differences in resumption time by a programmer could be performed. The study could determine if less resumption time is required when using **SKETCHLETS**. When performing the experimental task, participants will be asked to revisit code they have not seen in several months. One expected outcome is that participants in the control group will have a higher resumption time than those aided with **SKETCHLETS**.

8.5 Summary

In this chapter I described the evaluation of research questions and claims related to my thesis. There were aspects of research questions that had not yet been evaluated. Two experiments were conducted to evaluate the effectiveness of **CODE NARRATIVES** and **SMART REMINDERS** in reducing the negative effects of interrupted programming tasks. Evidence was found for both memlets

that was consistent with the predictions of the conceptual framework discussed in Chapter 5. However, several limitations, threats to validity, and barriers to tool usage were discussed. Finally, three claims related to the **TOUCH POINTS**, **ASSOCIATIVE LINKS**, **SKETCHLETS** memlets remain unevaluated.

Chapter IX

DISCUSSION

In this chapter, I discuss several factors, observations, and implications related to interruptions, individuals, ecology, programming tasks, and generalization. Finally, I touch upon some future research directions. The discussion is meant to guide other researchers in interpreting and in applying my conceptual framework in specific contexts, such as novice users, pair programmers, or different kinds of programming tasks and understanding how it might apply in future research efforts.

9.1 Interruption Factors

There are several factors such as the length of an interruption, the brain's ability to draw on background processes, the complexity of an interrupted task and the brain's ability to process concurrent goals that influence how a developer recovers from an interruption.

9.1.1 Length

The length of an interruption has several implications on recovery costs and influences on cognitive processes. A common set of questions revolve around short-term interruptions: If someone is interrupted for a minute, what are the consequences? If someone stopped working for 15 minutes on a programming task to check email, should it be considered an interruption?

In answer to the first question, most psychology experiments study interruptions on the orders of seconds and minutes and find clear evidence of disruption to the primary task from short-term interruptions as measured by increases of resumption lag.

In regards to the second question, previously during my exploratory research, I measured activity that occurred within the IDE, and for the purpose of the study, considered an interruption to occur when there was no activity in the programming environment for 15 minutes or more. There are two concerns here: First the length of the interruption and second the nature of the

work occurring outside of the IDE.

For the first concern, based on my definition of interruption, any break in work is considered an interruption. Once a minute has passed without any interaction in the IDE, the chances of no more work occurring in the next few hours increases significantly. As observed in the study, most events are clustered tightly in sessions: 98% of the events are within one minute of another. Once a developer is interrupted, he may pursue other activities before returning to work on the interrupted task, increasing the length of the interruption. This behavior has been observed in the workplace; O'Conaill's study [140] found 40% of interrupted tasks are not immediately resumed after an interruption.

For the second concern, knowledge workers have been observed to perform micro task-switching [15]. That is, they often rotate between several active tasks and windows every few minutes (programming, checking email, adjusting a music player, etc.) [95]. As these types of task-switches often occur within a time span of a few minutes, they were not measured in my study. Although tasks performed outside of the IDE can be related to the programming task, they can often be considered a different task, such as pursuing an information retrieval task by doing a web search related to a programming task or in the case of email, a communication or monitoring task.

Another way to characterize these types of questions is to consider what are the differences in short-term and long-term effects due to an interruption? Using my framework, it is possible to characterize some of the effects of interruption on different types of memory based on length of an interruption. In Table 16, I outline some memory effects related to short-term and long-term interruptions.

For short-term interruptions, different kinds of memory are affected differently. For attentive memory, the ability to maintain focus on items decays immediately, especially if attention must be paid to similar types of items. Prospective and associative memory can be resilient to short-term interruption over the course of the day. Episodic memory can be disrupted by short-term interruptions because the interruption may disturb processes maintaining contextual information about events that have occurred. Perceptions that have been primed in conceptual memory fade with disuse.

Table 16: Table illustrating the effects of interruption length on different memory types.

MEMORY	SHORT TERM	LONG TERM
Attentive	Focus decays.	Items need to be restored into focus.
Prospective	Resilient.	Monitoring processes discontinue.
Associative	Resilient.	Weak associations have faded.
Episodic	Source memory disruptions.	Consolidation reduces event fidelity.
Conceptual	Perceptions need to be primed.	Abstractions need to be primed.

With respect to effects on long-term memory, the different memory types are affected in different ways. Attentive memory is devastated. For prospective memory, monitoring processes discontinue the maintenance of prospective actions. Weak associations held in associative memory are irretrievably lost. For events held in episodic memory, details and specifics are lost as the events are consolidated. Abstractions that have been primed in conceptual memory fade with disuse.

Based on these different effects, research tools can consider using the length of the interruption to predict which memory type needs the most support. Future interfaces designs can consider displaying memlets as recommendations when there is the most need.

9.1.2 Background Processes

The nature of the brain supports background processing of information. If a person stopped working to go on a walk, or solved a bug while taking a shower, what implication does this have about interruptions?

Simply going for a walk increases creativity, even prior to starting the creative task [143]. So going for a walk in itself has positives consequence as a direct result of the biochemistry of the brain. What if the person was programming, then decided to go on a walk? The *default-mode network* [29], is a region of the brain that is active during moments of rest or when a person is not engaged in an a explicit task. Default-mode network processes involve reflection over low priority tasks or unsolved problems. These tasks are often selected from episodic memory [75]. However, these processes are interrupted and disengaged when a person must attend to a demanding or

externally cued task.

Although background processes can alleviate some of the problems associated with interruption, the tasks reflected on are processed in an indeterminate manner and not always directed toward the goals we intend. The background processes only engage when the programmer is not doing explicit work on a task, which may not always be an option. Finally, having an unsolved problem pop in and out of attention can be unpleasant when a programmer has returned home for the day and wishes to focus on true rest, friends, and family.

When considering interruption management strategies, background processes can be used as another channel for preserving mental state. Understanding when background processes are engaged, or not, can enable researchers to predict which interrupted tasks can incur the most resumption cost.

9.1.3 Complexity and Concurrency

The complexity of an ongoing task or interrupted task has several implications on recovery costs and influences on cognitive processes. How can we understand resource conflicts between tasks, especially if the goals are concurrent?

The right hemisphere of the brain can take on a secondary task if both the primary task and secondary task are simple and do not require access to the same types of information [36]. In this way, it is possible for the brain to process two concurrent tasks but no more. However, both hemispheres are recruited in complex tasks [13]. There is also a general overhead introduced with shifting attention between tasks [92].

Although the brain can support temporary shifts in attention and even process two concurrent goals, it does so at the cost of disrupting complex tasks which require the same resources that the secondary task would consume. This helps explain why interruptions such as instant messages related to a task are less disruptive [45]. The brain may be keeping the primary task in mind, while using the secondary processes—already primed for handling secondary information related to the primary task—to integrate and handle the instant message. Unfortunately, not all interruptions are related to the task at hand.

When considering interruption management strategies, complexity can be used to understand

the mental resources required to support a task. Future research that improves the measure of task complexity can enable researchers to predict which interrupted tasks can incur the most resumption cost. Future interfaces can discourage developers from taking on tasks that are too complex and suggest ways to refactoring the task into more manageable units of work.

9.2 Individual and Ecological Factors

There are several individual factors, such as the expertise of developer and the user's preferences in using memlets, and several ecological factors such as developers that work with pair programming or in open-office floor plans that may influence how a developer may recover from an interruption.

9.2.1 Expertise

A person's expertise has several implications on recovery costs and influences on cognitive processes. In general, experts are viewed as being more resistant to interruptions than novices [59].

Experts have more experience in dealing with interruptions. For example, experienced nurses devise more effective strategies for dealing with interruptions than inexperienced nurses [31]. The strategies are more effective because experts constantly self-monitor, keep track of what they are doing, and check for errors in their reasoning [59]. Finally, experts have access to more abstract representation in their conceptual memory allowing them to attend to more information [38]. Unfortunately, the path to becoming an expert is not easily walked: For a novice, evidence suggests this can be a 10 year journey [40].

Studies examining the difference between an expert and novice find that performance differences arise from differences in brain activity. Not only do experts require less brain activity than novices, they also use different parts of their brains as evidenced by EEG and fMRI studies [127, 80]: Experts use conceptual memory whereas novices use attentive memory. That is, experts are able to exploit abstractions in conceptual memory, whereas novices must hold primitive representations in attentive memory.

When examining the strategies programmers used to manage interruptions, many of the programmers studied were experienced developers. By understanding these strategies and providing tools in support of them, not only do novices have access to expert strategies, but

experts have tools that more directly support their workflow than ad-hoc practices. There may be differences in how memlets are designed based on expertise. For example, the design may encourage reminder types that are less prone to misuse: Novices may be less capable at estimating due dates, which can make Due By reminders less suitable for novices. There are elements of design that experts may be better suited to leverage. For example, experts might use higher levels of abstraction in events viewed with **CODE NARRATIVES** or locations viewed with **TOUCH POINTS**.

9.2.2 User Preferences

There may be a conflict with how a memlet was designed and a user's existing habits and preferences. During my evaluation of **CODE NARRATIVES** and **SMART REMINDERS**, I observed several differences in participant's tool preferences.

There are several ways to resolve misalignment of preferences and memlet design. Sometimes a misalignment can be resolved by understanding what underlying goal a user wants to achieve using the tool. For example, one user preferred using attachable reminders, but felt that for some reminders they were not catching enough of his attention. It turned out he did not realize that there was an option to view the attachable reminders in all code windows. Once he realized this was an option, he no longer felt the tool did not misalign with his goal.

Other times the misalignments point to an opportunity to redesign or refine features. For example, one user felt that he he did not want to always see attachable reminders in the top of the viewport, and would prefer to place them relative to the region of code where they were originally created. It turns out after further questioning that he simply thought he was seeing the attachable reminders too often. Because there was higher cognitive load associated with more frequent exposure of the same reminder (when programmers were concentrating heavily in one file), this user's preference could lead to a justified redesign that reduced the amount of exposure.

Of course, options work well for preferences that follow a discrete set of choices. For **CODE NARRATIVES**, *where* to store the history associated with code changes differed among participants. The current design associated the local history with the current solution in Visual Studio. For some developers this is a good option because they use solutions as logical units of work. Other developers maintained multiple solutions for the same work. Finally, some developers wanted to

move the history outside of the solution folder structure (because the items were showing up in their search tools), but still wanted to keep it associated with the solution. Based on a follow-up survey, I was able to identify a sensible set of options that would handle most user preferences.

If there is a diverging set of interests and no common ground can be found, it may be necessary to divide a tool. For **CODE NARRATIVES**, some developers enjoyed viewing **CODE NARRATIVES** rendered in HTML in a separate browser window. Other developers wanted the tool to focus more on summarization and to keep it as a view within Visual Studio. The best way forward may be to split the functionality of **CODE NARRATIVES**.

Finally, another way to manage user preferences is to consider the profiles of different types of programmers in selecting common groups of memlets and features. For example, for someone like Charles who often needs to switch between different projects, **SMART REMINDERS** can be an appropriate tool for managing reminders. However, if the time between switching projects is too long, as Ana often experiences in her work, then certain reminder mechanisms, such as Attach Here, would be highly ineffective. In her case, connecting with additional reminder mechanisms that can work externally to the programming environment might be necessary. Bob, who mainly works on one system that he is an expert in, may find **ASSOCIATIVE LINKS** distracting as he has already built his own conceptual knowledge for how to best navigate the system.

These profiles can even be dynamically adjusted. For example, a developer may experience different frequencies of interruption during different phases of work. During periods of frequent interruption, **CODE NARRATIVES** might be better tailored for preserving and reviewing fine-grain information, versus periods of lower interruption where less detail may needed to be displayed. Different kinds of interruptions such as event-based breakpoints (leaving for the day) versus task-based breakpoints (switching a task) may influence memlet use and may drive preferences such as automatic display of a memlet.

9.2.3 Pair Programming

Pair programming is a software development practice where two developers work side-by-side on a single computer during programming tasks. Cognitively, it touches upon the idea of distributed cognition [87] and transactive memory [208]. *Distributed cognition* is a theory that describes

how cognition and knowledge can be distributed in groups and pairs. In one study of transactive memory [208], it was found that couples outperformed randomly assigned pairs in memory tasks, because the couples were better than strangers at self-selecting what to remember and anticipating what their partner would not remember.

Some developers have suggested that pair programming could assist developers in recovering from interruption:

> *Interesting article. I was surprised to discover when I first started pair programming 13 years ago, how one person in a pair can hold the context while the other person in the pair gets interrupted. We could get back to work in seconds.*
>
> *I would guess a pair gets interrupted less than an individual. Societal graces about interrupting people talking and all that. In addition, when a pair *does* get interrupted, their minds (with their different grasps of the context) will get back on task quicker through cooperation. Don't knock it until you've tried it.*

The first developer suggests that pair programming could provide a cognitive structure that enables one developer to handle an interruption, while allowing the other developer to continue working. The second developer suggests that there would be minimal interruption between the pairs because a developer could use contextual and social cues to avoid interrupting the partner at moments of high mental load. While there is no scientific confirmation of the first point, the second point is supported in the literature: In a lab study of 7 pairs of students, there was no qualitative indicators of interruptions imposed by pair programming partners [94].

Although there is some evidence that pair programming could help developers with interruption, it would not completely address the issue of interruption. Pair programmers may still need memory aids to deal with long-term interruptions or partner swaps. While studies have identified benefits to pair programming, those benefits are not uniformly beneficial when considering overall economic factors [78]. In my professional experience, the prevalence of pair programming may be overstated, as many organizations may state they are practicing pair programming, while programmers rarely perform it.

9.2.4 Open Offices

Some developers work in office environments that expose them to a constant stream of interruptions. *Open-offices*, where people work in close proximity without any rooms and sometimes no dividers, is popular for reduced cost and increased opportunity for collaboration.

There are several sources of evidence that point out the negative effects of interruptions in open-offices. Open-offices can lead to lower productivity and increased stress in employees [27]. Workers are more like to experience both internal and external interruptions in open-offices [47].

As a result of working in open offices, workers may be forced to develop alternative work strategies. Some might delay complex work until working at night or at home. In experiments with frequent interruptions, subjects naturally adapted more aggressive suspension strategies [129]—open-office workers will likely have to do the same. During the summer of 2013, I had the opportunity to visit several startup offices with open-office floor plans. I observed that programmers had developed a set of cultural norms to help deal with interruptions: For example, wearing headphones signalling a request for no interruptions or self-segregation into quiet areas and collaboration spaces.

The design of the office environment will have long-lasting consequences on the productivity of the workers who work in them. Managers need to consider more than the initial cost savings offered by a particular office design, but also factor in long-term productivity costs. Finally, many of the factors that need to be considered are not static, for example, developer's age, project maturity, and team-sizes. Which means what works well for today's office may not work well for tomorrow's office.

9.3 Programming Tasks

Memlets are designed to support specific types of programming tasks. However, programming tasks vary in importance, frequency, and disruptiveness.

9.3.1 Attentive Task: Refactoring and Systematic Changes

TOUCH POINTS were developed to support developers with refactoring tasks due to limitations in attentive memory. Refactoring and other systematic changes generally require many locations of

code to be changed in order to improve code quality or restructure the design of the program. But how frequent and important is the task of refactoring to developers?

Previously [136], I found in an analysis that I did with Murphy-Hill that in a two-year span of four developers' refactoring-tool histories, only two weeks did not contain any refactoring tool usage. Additional, in a recorded history of 41 developers, 41% of programming sessions contained refactoring tool usage. In this view, we can see refactoring as a habitual task that occurs at recurring intervals throughout the development cycle and not just during a cleanup phase prior to a software release.

During my field study when evaluating **CODE NARRATIVES**, I also observed that there was an entire team devoted to handling systematic changes to the code base. For example, one task that they were assigned was to convert their code base, which targeted 32-bit machines to also support 64-bit machines. The team was deeply interested in the idea of **TOUCH POINTS**, as they were using just simple text search to track changes. A similar team was observed at Microsoft handling large-scale systematic changes to code [99]. Additionally, code that was refactored was found to have lower defect rates [99].

Refactoring can vary in its disruptiveness. Some refactoring changes can be fairly formulaic, and, as a result, resumption may be straight-forward. Other times, refactoring can involve many cascading changes, with each have many pending chains of change tasks. Programmers commonly refer to this state as, yak shaving [1], and may often not know they will be getting into this state beforehand.

9.3.2 Prospective Task: Blocked tasks and using task annotations

SMART REMINDERS were designed to support developers with remembering to complete ephemeral and blocked tasks due to limitations in prospective memory. Developers need support for writing down and tracking reminders in order to complete ephemeral and blocked tasks and later being prompted to perform them.

In an observational study of seventeen developers at Microsoft, within a 90 minute observation

[1] `http://www.hanselman.com/blog/YakShavingDefinedIllGetThatDoneAsSoonAsIShaveThisYak.aspx`

window, developers experienced a median of four blocked tasks which they had to defer work on until they could get answers from another colleague or devote time to performing further investigations. More generally, developers frequently apply task annotations in code using TODO comments in support of short-term tasks, subtasks, and future work [188].

During my field study, when evaluating **CODE NARRATIVES**, I also observed a developer with a *scrummaster role* (a person who oversees the status of programming tasks assigned to the team during a code sprint), found himself having to frequently switch between his management role and coding. He made extensive use of TODO notes and intentional compile errors. Programmers that have to frequently switch roles or switch between projects may use more informal representations such as task annotations in code, as programmers associate creation of a formal task representations with high overhead costs [188].

The most disruptive aspect of blocked tasks is forgetting to resume them in a timely manner. How likely this is depends on several factors. Some organizations invest heavily in tracking the status of all developer tasks, whereas other organizations allow developers to freely roam among the tasks they are assigned.

9.3.3 Episodic Task: Recall and Review of Code Changes

CODE NARRATIVES were developed to support developers recalling recent code changes that they had forgotten due to limitations in episodic memory. Developers need support for remembering details about how they changed code.

There are many tasks involving the recall of past code changes. Developers often have to recall changes to share them with teammates, document how they did a task, refresh their memory before resuming work, or verify if there are any incomplete changes. High quality code commit messages are viewed as good software engineering practice and involves remembering the changes that were recently performed; however, developers in practice create sparse and incomplete commit messages. Previously [136], I found in an analysis that I did with Murphy-Hill that only in about half of the commits we inspected where refactoring was performed, did developers indicate they had refactored code.

The most disruptive aspect of recalling past code changes is omission of important or incomplete changes. Incomplete changes can result in bugs, which may even spread disruption to other team members and customers of the product or service. The importance of recalling important changes varies with the recall task. For documenting a code change, they are very important; however for just refreshing their memory, not reviewing a particular change in the worst case would lead to a slightly slower task resumption time.

9.3.4 Associative Task: Navigating code

ASSOCIATIVE LINKS were developed to support developers with disorientation when navigating code due to limitations in associative memory. Developers need support for improving memory cues when visiting different locations in code.

Developers spend a large amount of time navigating source code. In a laboratory study of 10 experienced Java developers, Ko [102] observed that developers lost track of relevant code because they had to temporally navigate to another location in code and forgot how to find the relevant code location again. Developers spent, on average, 35 percent of their time performing the mechanics of navigation within and between source files. Previously, I studied data [149] from twelve programmers collected over several months of development activity in the professional workplace and found that developers work with 57–83 methods in a day (25% and 75% percentile respectively) and frequently switched between those methods as they worked. 60% of transitions were to different classes.

Navigational errors are not usually particularly disruptive on an individual basis, but they are quite frequent. This problem may be worse for large or unfamiliar code. I have observed where navigational errors in unfamiliar code resulted in participants not completing tasks in a timely manner [148]. However, this may be less problematic when navigating in familiar code.

9.3.5 Conceptual Task: Program Understanding

SKETCHLETS were developed to support developers with problems in forming and repriming concepts due to limitations in conceptual memory. Developers need support for learning and reactivating programming concepts in order to keep pace with new technology or to revisit previous projects or modules they have not worked on in a while.

Interruption may reduce the effect of priming of concepts needed for a programming task, requiring that programmers refresh their memories. The disruptiveness of this effect depends on the frequency of interaction and rate of decay of primed concepts. This effect may be especially pronounced for programmers onboarding onto new projects or new technologies. There is limited research exploring this space, however Fritz et al. has recently explored measuring frequency of interaction and authorship of code as a degree of knowledge of the code [66].

9.4 Generalization to Other Environments

My conceptual framework can generalize to other environments. In particular, the five memory types I describe in my framework can be applied to designing memory aids in other environments, specifically because the process in which the five memory types were derived was not specific to programming.

Memlets can generalize to other computing environments. For example, I have created a prospective-based memlet that allows reminders to be posted to specific URLs and a episodic-based memlet showing the timeline of visits to developer-specific websites for Chrome. More generally, my conceptual framework can be used to reason about design elements and their memory support in other computing environments. For example, in Figure 40, I illustrate how different design elements map to different memory support in the Chrome browser.

When browsers were first introduced, they had no way to allow people to view more than one page within one browser window. The introduction of tabs resulted in the first step of supporting attentive memory, by allowing more than one page to be visited. But once too many pages are visited, it becomes difficult to track locations in the browser. The introduction of icons associative with the page aids associative memory, making it easier to recall particular locations. However, this effect is only short-term—often if a person revisits her browser the next day, associative memory has faded and the resulting clutter results in the user closing all the tabs. Currently browsers have little prospective memory support. Using my prospective-based memlet, users can now post a reminder to a page. Finally, if a person knows she had visited a particular page in the past (e.g. I found a useful Stack Overflow question last Friday), then that person could use the episodic-based memlet to recall the relevant Stack Overflow question.

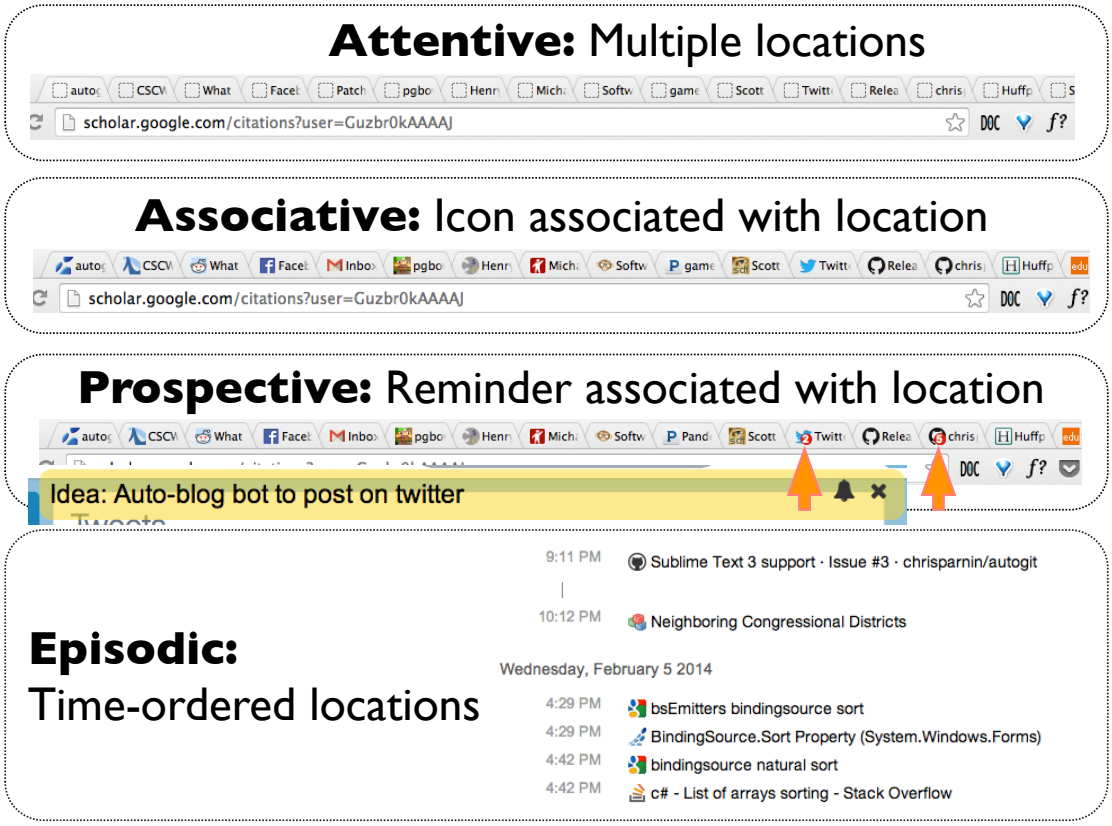


Figure 40: Reasoning about memory support in browsers.

My conceptual framework can generalize to memory aids outside of the desktop environment. For example, researchers used my framework to help them understand how to support prospective memory issues in the design of a smart bag [60].

My infrastructure can generalize to other programming environments and possibility other work environments. For example, I have had success in migrating **CODE NARRATIVES** into a popular text editor, Sublime Text. There are things that are specific to programming and may stay specific to programming unless additional effort is made in migrating certain concepts over into other environments. Programmers' strategies for dealing with interruptions have evolved from factors such as the medium of work (text) and taking advantage of tools that arise due to the nature of programming: source control and static checking. Infrastructure may not be necessarily available in other professions, for example writers may also work with text, but they may not use source control or have automatic tools that identify errors.

Finally, there are many opportunities to better understand how cognition involved with programming tasks overlaps with other knowledge worker tasks. The conceptual framework may work with tasks that are not programming tasks, but still involve the same types of memory types. However, we do not have a good model for suggesting these relationships. This is a good reason to further examine and understand the cognitive nature of programming, which is described in the next section.

9.5 Future Work

There are several research directions for future work.

9.5.1 Future Models and Techniques

Our understanding of the human brain increases everyday. There are many promising results that will help refine future versions of my conceptual framework. For example, there is new emerging research that could help create a better understanding of the mechanisms and processes of attentive memory. This in turn allows new design considerations to be found. Second, my framework provides a limited set of concepts and does not handle other cognitive processes, such as attentional processes, which are distinct from memory processes. As our understanding of the brain increases, so will our ability to understand how different types of memory interact with each other and how other cognitive processes play a role.

In future work, I would like to take advantage of more physiological techniques for assessing cognitive load. I have used electromyography (EMG) in studying the different cognitive loads associated with programming. Several colleagues and I have already starting exploring how more techniques such as functional magnetic resonance imaging (fMRI) [179], electroencephalography (EEG) [65], functional near-infrared spectroscopy (fNIR) [138], eye-tracking and other techniques can yield insights into the inner workings of a programmer's mind. For example, eye-tracking can help better highlight, model, and predict what information developers were actively paying attention to and what information they may have not seen but want to know.

Previously, I was able to demonstrate that certain programming activities correlated with higher levels of cognitive load as measured by subvocalization, and more difficult tasks could

be contrasted with easier tasks by total levels of subvocalization. I also recently worked with Janet Siegmund in a controlled experiment using fMRI. In the experiment, 17 participants were observed inside an fMRI scanner while they were comprehending short source-code snippets, which was contrasted with locating syntax errors. We found a clear, distinct activation pattern of five brain regions, which are related to language processing, attention, and working memory. The experiment demonstrated that this can be a viable method of research for exploring cognitive aspects of programming.

9.5.2 Research Vision

In this section, I present a research vision and a list of future research questions.

Programming is a deeply complex but relatively new human activity. Its young age has lead to countless battle for which, despite the many compelling arguments presented, there are not definitive answers. My research aims to change that by using brain imaging studies to understand the programmer's mind. Understanding the brain offers us the chance to distill these complex issues into fundamental answers.

One recent debate was prompted by legislation in Kentucky that offers foreign language credits for students learning programming languages [2]. On the surface, this is a premature proposition, mainly because we do not know enough either way. But a more interesting question arises: What exactly would introducing programming at a younger age change in the brain? For people that are fluent in a second language, studies have shown distinct developmental differences in language processing regions of the brain. How might computational fluency at an early age restructure our future minds?

People need to deal with complexity, and software is what we often use to do that. Not only do domains, such as simulating the climate or understanding flash crashes in automated financial systems, involve immense complexity, software is also increasingly involving a wide range of expertise and skills (e.g. end-user programming, cognitive support of quadriplegics and the memory impaired). Neuroscience will become essential in understanding how people deal with complexity. When designing software, models of cognitive complexity based on neuroscience

[2] `http://tinyurl.com/kjhr39d`

principles will be incorporated into software for domains mired in complexity.

Performing research on how programmer's brain's work has more than just theoretical value, but can have real downstream effects in improving education, training, and the design and evaluation of tools, languages, and programming environments for programmers.

Finally, developers will have access to augmented programming environments, which include devices that can sense verbal formations, mental imagery, mental load, alertness, and general brain state. The sensed signals further serve as feedback and input into the programming process. Furthermore, devices can deliver localized pulses (via transcranial magnetic stimulation) that can prime and enhance particular brain regions needed for programming. Not only might developers use these devices, the devices will support other knowledge workers too.

The following is a list of future research questions:

- What does "programmer's flow" (a state of complete immersion in a programming task) look like from a neuroscience perspective, and how can we quantify its effects on the programmer and creation of software?
- Programming tools require the concerted use of many cognitive facilities, including symbolic memory, spatial memory, keyboard-based text entry, and visual feedback. How can we design them to take advantage of the cognitive capabilities we are good at and alleviate the challenges of using the ones we are bad at?
- The adoption of end-user software development environments, such as Excel, has expanded the pool of programmers tenfold [173]. What kinds of programming tasks, tools, and contexts will it take to democratize programming to reach the next power of 10? For example, how should initiatives like the "Hour of Code" be designed to account for the cognitive skills and levels of those involved?
- Many philosophies and pedagogies influence how we teach novices to program. How does a teacher's particular curricular choices affect the cognitive aspects of how students understand, write, and maintain software? How can we use neurological imaging to evaluate what and how well they are learning? For example, does teaching particular concepts such as design patterns change the way people fundamentally understand code?

Chapter X

RELATED WORK

In this chapter, I review several areas of related work. Both inside and outside of the programming domain, there are several research directions that have focused on strategies and tools for interruption management, working-state organization, mental recovery. There are also other cognitive theories of programming that have been proposed, and approaches related to logging of interaction history.

Finally, I compare other related research tools with memlets and discuss differences in memory support in terms of the memory types described in my conceptual framework. Note that related work about memory is included in the literature review available in Appendix A.

10.1 Interruption Management

Several approaches support interruption management. The approaches range from *preventive* (avoiding the interruption) to *elaborative* (preserving working state) and differ in applicability and to a certain extent, philosophy. An example of preventative measures is described in Peopleware [53], where interruptions from officemates were less frequent after the introduction of private offices and resulted in an improvement in the productivity of programmers. Another approach focused on preventing interruptions from computer messages by predicating how busy a person was based on interruptibility heuristics. With the approach, computer alerts could be delayed until a time where the user was less busy [89].

Several elaborative approaches involve writing down information in anticipation of an interruption. Elaborative approaches in their purest form would have the programmer record everything, including audio, screenshots, diagrams, and personal notes. In practice, the observed process is much simpler: mainly writing down keywords on sticky notes or sending reminder emails [20]. Dekel [49] describes a tool enabling developers to express the task structure of their work and associate comments and audio recordings with task elements. Unfortunately,

in practice, research approaches like these that have attempted even partial elaboration of task information have had limited buy-in from developers [151]. This is largely regulated by the nature of interruptions: They are often not anticipated or offer little time to record information, and even when programmers do write down information, it is incomplete and still results in resumption failures [148].

Improvements in elaborative approaches will likely follow advances in technology that improve the note-taking experience using devices such as smart pens, tablets, and surfaces (see Parnin et al. [150]).

10.2 Task and Working Context Management

Several tool designs support organization of working context. Group Bar [155] enables 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 [163], do this automatically by clustering related work items based on interaction coupling. Rather than organize the documents and windows into definitive task categories, another organizational approach involves tools that can associate personal tags with documents and activities [141] or tag shared work items [194].

Successful organization of work items through tools can reduce the friction between switched tasks; however, it does not fully address the problems of recovering lost working state for programmers. These memory aids do not take 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.

10.3 Desktop and Programming Environments

Another direction of research has investigated various ways of augmenting temporal, spatial, and contextual cues in the computing environment. Scalable Fabric [168] using a zoomable organization of windows and documents to leverage spatial memory to facilitate task switching. A similar effort has been made for viewing source code through Code Thumbnails [50]. Window Scape [191] introduced temporal snapshots of windows where users can return to view recent contexts. In a study in which subjects searched open documents and windows [142], 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.

For programmers, Mylyn [98], an Eclipse tool, attempts to improve cues by first recording the frequency of activity in documents, and then bolding the text of the document and method names of more frequently accessed items in tool windows, such as Package Explorer (a tree of documents). Further, Mylyn can reduce clutter by filtering out items not recently worked on. A limitation of Mylyn is that a more detailed history, such as changes, cannot be reviewed. Another system, Code Bubbles [26], redesigns the IDE to be based on fragments rather than files. Instead of editing a file, the developer views and edits small fragments of the source code. Using the system, a programmer could quickly accumulate and simultaneously view fragments. The fragments provide alternative associative and spatial cues that would not be available in a traditional programming environment.

10.4 Visualization of Developer Activity

Researchers have proposed several approaches for understanding and visualizing human activity in software development. Initial work focused on the activity in source code repositories and problems associated with coordinating development between teams [190, 23], which has been shown to reduce conflicts [172]. Similarly, Ellis and colleagues have visualized bug repositories in support of prioritizing development tasks [77, 55].

More recent work has shifted the focus to include other aspects of the development process [146], including awareness of personal plans and activities. `Spyware` [166, 167] collects developer interaction history and provides an interface for visualizing and exploring development sessions. Its visualization provides a good overview for characterizing development sessions, but it is not designed to provide the more in depth information that would be necessary when recalling development activity.

Safer and Murphy [170] investigated various designs of user interfaces for identifying when a recent task was completed. One interface displayed a timeline of screenshots captured for each action of the performed task. The other used a timeline of a filtered treeview indicating the files and methods visited or edited. Using a screenshot-based cue, users were able to identify details

about recently performed tasks quicker and with less errors than when using semantic-based cues. In a similar vein, a later version of `Spyware` introduced a mechanism for replaying a development session [167].

A few approaches have tried to socialize development activity. Replay [82], focuses on replaying other developer's changes in order to better understand those changes. Fine-grain changes are captured from the environment and can be shared with other developers. The tool's main focus is on social activity awareness and not personal interruption recovery. Using Codebook [19], developers can subscribe to their work artifacts, keep track of dependencies, and discover connections using a web interface. Again, the focus of this tool is on maintaining awareness of others' activities, but not on a developer's own personal history.

Although these approaches have provided strong guidance in visualizing activities, several issues remain. Concerning the benefit of instant replay, psychology researchers have doubted its effectiveness for resuming interrupted activities and cite an overall negative effect [93]. Developers may have preferred the screenshots over the task context view because it contained more details about the source code. Indeed, although a screenshot can be a rich source of cues that trigger recall of performed activities, choosing the granularity of screenshots for every action may impose an interface that is too fine-grained and too difficult to form a collective image of what took place—especially for a development session lasting a few hours. Furthermore, the interface based on screenshots is not well-suited toward extensions such as aggregation. Conversely, an interface displaying the file names and method names modified and visited within a small segment of time provides too little context to properly identify the activities that occurred.

10.5 Cognitive Theories in the Psychology of Programmers

Several theories of cognitive theories for programmers have been proposed to inform tool design.

In top-down comprehension [28] a programmer formulates a hypothesis about a program that reflects the hierarchical structure of the program's code. The programmer is guided by using cues called *beacons* that are similar to *information scents* in information foraging theory [158]. In bottom-up comprehension [175, 156], a programmer gradually understands code by *chunking*

her understanding into syntactic and semantic knowledge units. In opportunistic and systematic strategies [110], programmers either systematically examine the program behavior or seek boundaries to limit their scope of comprehension on an as-needed basis. Von Mayrhauser and Vans offered an integrated metamodel [205] to situate the different comprehension strategies in a single model scheme.

Storey et al. devised an information-needs framework based on cognitive theories of programmers [186] and used it to design a new programming environment, *Shrimp* [186]. However, Storey et al.'s framework is limited to exploration tasks and does not incorporate memory failures and the resulting information needs reflected in everyday programming tasks.

Researchers in the information-foraging field have taken a broader view of how people seek information. Unlike program comprehension, these theories are generalized from several different domains. However, O'Brien and Buckley [139] have retrofitted several models taken from different information-foraging theories to explain program comprehension. The newly synthesized model proposes that programmers switch between different stages of focus formulation and information seeking—progressing and evaluating their state during each stage.

A different direction taken by Murray and Lethbridge is their work on micro processes for cognitive patterns [137]. Their approach maps different problem scenarios to the cognitive processes that solves them. Strategies are constructed to reflect specific problem types rather than conforming to a general framework (e.g., the strategy needed for understanding the cultural practices of a team is distinguished from getting the big picture of a program). Similarly, other researchers have tried uncovering strategies that programmers use in specific situations. Fleming and colleagues examined the strategies for understanding parallel programs [63].

In contrast with other theories, my research directly considers the constraints of the human memory. Although some of the above cognitive theories consider memory, many researchers devised theories limited by notions of memory that were available at the time of their work. For example, Shneiderman, who published several influential articles on programmer memory and comprehension, once likened the ability of musicians to memorize every note of thousands of songs or long symphonies to programs and suggested programmers would obtain the same ability to commit the entire program to memory [176]. Rather the opposite has occurred: Programs

are not unvarying sacred tomes, but organic and social documents that are understood and navigated with the assistance of abstract memory cues such as search keywords and spatial associations [102].

10.6 Other Logging Frameworks

Although tools exist for recording events in programming environments, they have are several limitations that influenced me to develop my own infrastructure.

The most ubiquitous framework for recording programming events is the Usage Data Collector (UDC) that ships with the Eclipse programming environment[1]. The strength of this framework is that it is widely deployed, providing potential access to interaction data from over Java 100,000 developers. However, it has since been shutdown. E. Murphy-Hill and I along with his advisor, Andrew Black, have used such data in research related to studying refactoring tool usage by developers [135]. However, this infrastructure was intended to collect simple frequency statistics of tool commands in order to identify which views or commands within Eclipse are popular among developers. This framework misses many essential bits of context such as file names.

The Mylyn Monitor API [2] is also available as a plugin for Eclipse. The framework supports creating event listeners that record various events in the IDE. The Mylyn Monitor plugin is more capable than the UDC framework because it offers more programmatic control and customizations, such as tracking access to files and code elements. But there are still several limitations with the framework. One limitation is that the framework was primarily built for supporting the type of views that Mylyn uses: aggregating a simple count of frequency for files or code elements. The framework (out of the box) maintains neither context of events nor snapshots of artifacts.

Having the capability for taking snapshots of artifacts and analyzing their evolutions is essential for creating the views needed for memlets. This information allows an analysis to not only understand *when* and *where* a programmer was working, but also *what* he or she was doing. As a result, visualizations can display snippets of code related to an event or provide detailed change histories. Researchers have proposed systems for recording change history at the atomic-level

[1] `http://www.eclipse.org/epp/usagedata/`

[2] `http://wiki.eclipse.org/index.php/Mylyn_Integrator_Reference#Monitor_API`

for artifacts. Systems like SpyWare [167] take into account the actual *changes* to the software when analyzing the activity of programmers by recording and replaying each keystroke. Similarly, distributed version control systems such as git [3], allow a developer to make personal versions of files without committing to a public branch. Unfortunately, these systems are limited to changes to source code and do not record interaction data.

The goal for developing my infrastructure has been to expand the richness of the context recorded with events while including the capability to capture snapshots of artifacts such as the change history of source files.

10.7 Tool Support Coverage of Memory Types

Considering the different approaches and tools proposed by other researchers, there are considerable gaps in their coverage of different memory types. In Table 17, I summarize the coverage and discuss the limitations.

Table 17: Cognitive support for different tools. A half-circle indicates partial support and a full-circle indicates full support.

TOOL	ATTENTIVE	PROSPECTIVE	ASSOCIATIVE	EPISODIC	CONCEPTUAL
Scalable Fabric	◑		◑		
Window Scape	◑		◑	◑	
Group Bar	◑				◑
Shrimp	●				
Mylyn	●		◑		
Code Bubbles	◑		●		◑
Code Canvas	◑				◑
Replay				◑	
TagSea		◑			◑

Few tools support episodic memory. WindowScape [191] differs from Scalable Fabric [168] and Group Bar [155] in that it supports episodic memory by including temporal snapshots of window state. Replay [81] supports episodic memory by playing back source code changes in their original order. However, none of these tools includes support for summarizing or structuring events into narrative structures.

[3] `http://git-scm.com/`

Even fewer tools support prospective memory well. TagSea [189] supports the ability to define tags for the purpose of creating a reminder, but does not include facilities to prompt or monitor the conditions related to the reminder.

Most tools support attentive memory by providing a way to focus on multiple items. With the exception of Mylyn [98], few tools support the ability to filter items, and none support annotating the items, which greatly limits their ability to handle ephemeral explosions of attention common in programming.

Several tools incorporate design elements that support associative memory. Code Bubbles [26] does this well. It provides multiple modalities to support association, including spatial modalities and enhancing visual modality through additional icons such as bug markers. Otherwise, few tools incorporate more than one modality in support of associative memory, for example, Scalable Fabric [168] is limited to a spatial modality.

Several tools have limited conceptual memory support. Most allow grouping of items into new categories, and some support organizing items hierarchically. However, these tools do not integrate richer forms of concept development, such as developer's sketching-like behavior, nor consider which organized knowledge may need to be primed. Code Canvas [52] provides an interesting contrast to Code Bubbles. Both tools support a similar modality: spatial. But being similar on the surface, the tools actually support different memory types. Code Bubble's fluid and dynamically changing landscape promotes temporary spatial associations that last a few hours (associative memory); whereas the stable layout of Code Canvas promotes a longer-term, collection of spatial abstractions (conceptual memory).

Chapter XI

CONCLUSION

The impact of interruptions on software development is staggering. Solingen [203] 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 each interruption. Long-term interruptions are also common: Ko et al. [103] observed 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. In my exploratory research of interruptions [151], I analyzed interaction logs of 10,000 programming sessions from 86 programmers and found that in a typical day, developers rarely program in long continuous sessions. Instead, a developer's day is fragmented into many short sessions (15-30 minutes) with an additional one or two longer sessions (1-2 hours). Further, at the start of each of the longer sessions, a programmer often spends a significant amount of time (15-30 minutes) rebuilding working context before resuming coding. Interruption is a wide-spread and costly problem faced by programmers.

Ultimately, we cannot eliminate interruptions. But we can find ways to reduce the impact on the memory failures that often result from interruption.

In this dissertation, based on a series of empirical studies and reviews of cognitive neuroscience literature, I derive a set of information needs for recovering from interrupted programming activities. My conceptual framework structures the information needs in terms of five memory types particularly relevant to programmers: attentive, prospective, associative, episodic, and conceptual.

Attentive memory holds conscious memories that can be freely attended to. Attentive memory is easily disrupted by short-term interruptions. *Prospective memory* holds reminders to perform future actions in specific circumstances (e.g., to buy milk on the way home from work). Prospective memory failures often involve forgetting to perform an action in a timely manner.

Associative memory holds a set of non-conscious links between manifestations of co-occurring stimuli. Associative memory failures stem from poor interface design, especially when insufficient environmental cues are provided. *Episodic memory* holds the recollection of past events. Episodic memory failures often involve forgetting contextual details about events and omission of events. *Conceptual memory* holds representations in a continuum of perceptions and abstractions. Conceptual memory is often disrupted by long-term interruptions, which in turn requires that conceptual memory be restored and primed before becoming useful.

My thesis states, *Memlets, conceptual models and visualizations that support limitations in programmers' memories, reduce the negative effects of interrupted programming tasks by overcoming memory failures experienced.*

To perform research on my thesis, I examined seven research questions:

- What memory deficiencies arise from interruption of programming tasks?
- What strategies and sources of information do programmers use to recover from interruptions?
- What memory aids can programmers use to recover from interruptions?
- To what extent can memory deficiencies be linked to resumption strategy failures or lack of information?
- To what extent can the innate properties of human memory be used to *derive* more effective resumption aids?
- Which new algorithms and concepts are needed for building effective resumption aids?
- What are the observable benefits and disadvantages with using memlets for interruption recovery in software development?

The thesis statement states that the negative effects of interrupted programming tasks, such as increased errors and time-to-task completion, can be reduced if programmers are supported by appropriate memory aids in the form of memlets. The dissertation argues that the memlets must satisfy a programmer's cognitive needs (see Chapter 2) and explains that these information needs

are dependent on the various constraints and memory failures that arise in different areas of memory during specific programming tasks (see Chapter 5). My conceptual framework structures these constraints and failures as information needs in terms of five memory types particularly relevant to programmers: attentive, prospective, associative, episodic, and conceptual. Based on these information needs and constraints, the design and implementation of memlets was undertaken (see Chapter 5 and 6). The implementation is realized in an extension built for Visual Studio (see Chapter 7).

Based on my conceptual framework, the following memlets were derived: **TOUCH POINTS**, **SMART REMINDERS**, **CODE NARRATIVES**, **ASSOCIATIVE LINKS**, and **SKETCHLETS**. *Touch points* support attentive memory by allowing a programmer to focus on programming elements in the environment at a much longer duration and number than she would be able to do if relying solely on her own memory. **TOUCH POINTS** are designed for programming tasks that involve refactoring, which may involve having to attend to many locations in code to apply a change. *Smart reminders* support prospective memory by allowing a programmer to designate reminders that trigger based on an external conditions. **SMART REMINDERS** are designed for situations such as blocked programming tasks, which may delay when a programmer can resume a task. *Code narratives* support episodic memory by allowing a programmer to review a narrative of the events and experiences related to their programming task. **CODE NARRATIVES** are designed for programming tasks that involve recall of past events, such as code reviews. *Associative links* support associative memory by providing spatial, temporal, and activity-related links between various programming elements. **ASSOCIATIVE LINKS** are designed for programming tasks that involve code navigation, in which developers frequently experience disorientation. *Sketchlets* support conceptual memory by allowing the representation and naming of new concepts derived from existing programming elements. Sketchlets are designed for programming tasks that involve learning new concepts or restoring concepts that need to be need to be refreshed.

To evaluate my thesis statement, I focused on evaluating two specific claims:

- **CODE NARRATIVES** reduce source failures by maintaining and presenting contextual details of programming events and reduce recollection failure by maintaining and presenting the ordering and fidelity of programming events.

- **SMART REMINDERS** reduce monitoring failure by introducing mechanisms for conditional triggers and reduce prompting failure by introducing mechanisms for enhancing awareness of reminders.

Two studies were conducted to evaluate the claims.

A field study with **CODE NARRATIVES** was performed in order to determine if differences in recall could be observed when resuming an interrupted programming task. To measure the differences in recall, a content analysis of events recalled with and without **CODE NARRATIVES** was performed when developers were resuming interrupted programming tasks. To ensure that differences in recall are not simply explained by increases in recall effort, I also measure the time used to perform the episodic review tasks. Further, by allowing participants to use existing source difference tools, they had access to the same information provided by **CODE NARRATIVES**, but without temporal support and with less fidelity.

A study with **SMART REMINDERS** was performed in order to determine if differences in completion rate and speed could be observed when remembering to complete interrupted programming tasks. The developers in the study were asked to create reminders with and without the aid of **SMART REMINDERS** for programming tasks that were part of their daily jobs. The study showed how design variations of the smart reminder resulted in differences in completion rates and speed that can be explained by the conceptual framework.

Based on the studies, my thesis is supported by the following findings (Chapter 8):

1. Developers can recall nearly twice as many events using **CODE NARRATIVES** over traditional tools, with comparable recall effort.

2. Developers can perform nearly twice as many prospective actions using **SMART REMINDERS** over a TODO note, with limited impact to cognitive load.

Further discussion on factors, implications, and observations related to interruptions was given in Chapter 9. An overview of related work and its relation to the conceptual framework were described in Chapter 10. Details on my literature review and exploratory experiments can be found in the following appendices.

Appendix A

LITERATURE REVIEW FOR COGNITIVE NEUROSCIENCE AND PSYCHOLOGY OF MEMORY

When faced with frequent interruptions and task-switching, programmers have difficulty keeping relevant knowledge in their mind. An understanding of how programmers actively manage this knowledge provides a foundation for evaluating cognitive theories and building better tools. Recently, advances in cognitive neuroscience and brain imaging technology have provided new insight into the inner workings of the mind; unfortunately, theories such as program understanding have not similarly advanced. In this appendix, I review recent findings in cognitive neuroscience and examine the impacts on our theories of how programmers work and on the design of programming environments. This literature review has been published in the proceedings of the Psychology of Programmers Interest Group [147].

A.1 Motivation

Researchers have long been perplexed in understanding how programmers can make sense of millions of lines of source code text, extract meaningful representations, and then perform complex programming tasks, all within the limited means of human memory and cognition. To perform a programming task, a programmer must have the ability to read code, investigate connections, formulate goals and hypotheses, and finally distill relevant information into transient representations that are maintained long enough to execute the task. Amazingly, programmers routinely perform these mental feats across several active programming projects and tasks in fragmented work sessions fraught with interruptions and external workplace demands.

In coping with these demands and limitations, the programmer must have mental capacity for dealing with large workloads for short periods of time and cognitive mechanisms for maintaining and coordinating transient representations. As of yet, we have no cognitive model that adequately

explains how programmers perform difficult programming tasks in the face of constant interruption. As a consequence, we have a limited basis for predicting the effects of interruption or evaluating different tools that may support task-switching for programmers.

New perspectives on memory and programmers are needed. Early models of memory, which I review below, have identified several key processes and provided many fruitful predictions. However, when pressed with explaining programmer behavior on more strenuous tasks, such as dealing with an interruption, these models have difficulty accounting for sustained performance [57]. Further, new results continue to emerge from studies of patients with novel brain lesions (injuries to specific brain regions after a stroke or accident) who display behaviors that undermine many of the assumptions of early memory models [184]. Likewise, early perspectives on programmers now seem dated. Shneiderman, who has published several influential articles on programmer memory and comprehension, once likened the ability of musicians to memorize every note of thousands of songs or long symphonies to that of programmers and suggested programmers would obtain the same ability to commit entire programs to memory in exact detail [176]. Rather the opposite seemed to have occurred: Programs are not untouched sacred tomes, but organic and social documents that are understood and navigated with the assistance of abstract memory cues such as search keywords and spatial memory of scrollbar positions [102].

The methods available to researchers have expanded greatly. For example, it is now possible to administer drugs that interfere with memory formation and even genetically engineer rats (whose basic brain structure for memory is remarkably similar to humans) without the genes for neurotransmitters necessary for consolidating short-term memories into long-term memories. Additionally, fMRI machines provide the ability to measure changes in blood oxygenation levels associated with increased brain activity within 1-2 seconds to regions of brain with 1-3 mm^3 precision [204]. These methods, which have not been previously available, have lead to the founding of a new interdisciplinary field. *Cognitive neuroscience*, a term coined by George Miller and Michael Gazzaniga, is "understanding how the functions of the physical brain can yield the thoughts and ideas of an intangible mind" [70]. For researchers studying the cognitive aspects of programmers, never have more opportunities been available to expand our understanding of the inner workings of the programmer's mind.

A.2 Psychological Studies of Memory

Memory research has had a long and rich history in the psychology community. Here, we briefly cover some of the key findings.

One of the earliest contributions to memory was Miller's work in 1956 on limitations on information processing. Regardless of what item a participant was being asked to memorize, Miller observed that the capacity for short-term memory was 5-9 items [124]. Recent research has suggested the actual limit is closer to 4 items [43].

In 1968, Atkinson and Shiffrin presented an influential model of memory called the *modal model of memory* [9]. 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 then 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 information 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 idea of *working memory* [10] to help explain how items could be manipulated and processed in separate modalities (e.g., visual versus verbal). The original model included separate storage of verbal (phonological loop) and visual–spatial memory with a central executive process that guides attention and retrieval from the stores. In 2000, Baddeley added an episodic buffer that allowed temporary binding of items [11].

Chase and Simon proposed that experts such as chess players can manage larger mental workloads by learning how to effectively *chunk* information after extensive practice and study [38]. The chunking theory proposes that it takes about 8 seconds to learn a new chunk, and that only about seven chunks can be held in short-term memory. For example, a chess master can out-maneuver an expert player because he or she can attend to more plausible moves and better assess positions of the chess board.

Several researchers have raised concerns about limitations with the chunking theory. First, information for tasks such as playing chess did not appear to be stored in short-term or working memory (or at least was transferred to long-term memory faster than predicted by chunking

theory). Charness found when chess players interpolated playing chess with other tasks long enough to eliminate short-term memory, no or minimal effect on recall was found [35]. Second, chunking theory has a hard time explaining how people [56] or experts [57] performing everyday tasks could handle unpacking and shifting between multiple chunks with such a limited store.

An important alternative to the chunking theory was articulated over a series of papers by Chase, Ericsson and Staszewski [37, 58], who observed mental strategies used by mnemonists and experts. The resulting *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 [198]; and (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, and later the expert could retrieve associated items by mentally visiting locations within the room).

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 [57].

A.2.1 Conflation of Working Memory

Working memory has become a conflated term among researchers.The term working memory was first introduced in 1960 in Miller and colleagues' influential book [125] on plans and reasoning as a system for temporary retention of plans and goals. The context of usage was for reasoning processes in cognitive models without strong considerations with concerns such as memory duration or task switching.

When Baddeley and Hitch first introduced their working memory model, they were concerned with the unitary view of short-term memory [11]. What they wanted to emphasize was that short-term memory was not a passive store, but was interleaved with distinct modality-based processing of sensory input (separate verbal and visual processing). Further, they wanted to consider the low-level attentional processes needed to maintain those memories (over the time scale of seconds). What Baddeley describes can best be attributed to perceptual systems located

in posterior regions of the brain. For example, patients with lesions to the left supermarginal gyrus (phonological loop) have difficulty holding words in memory [202]; whereas patients with a lesions in the right parieto-occiptial region have difficulty with spatial locations (for example, remembering a series of spots pointed to by another person).

Unfortunately, researchers across multiple disciplines have conflated the different usages of these terms to mean the same thing—that a perceptual store is the same as an "temporarily indefinite" pool of memory. That is, when referring to something such as the *working memory* of a programmer, a researcher most likely means it in the sense of Miller and not Baddeley. Baddeley's working memory theory cannot account for how people perform complex tasks or recover from interruptions. Theories such as *long-term working memory* attempt to reconcile this difference, but offer no neuroscience basis for the theory [57], leaving some researchers to suggest the standard model of working memory has outlived its usefulness [160].

Finally, recent evidence confirms that activity in neurons of the prefrontal cortex is related to attended items and not remembered ones [106].

A.3 Memory in Cognitive Neuroscience

A.3.1 Building Blocks of Memory: Long-term Potentiation (LTP)

Like physicists who seek to understand the building blocks of atoms in order to understand the world, we seek to understand the building blocks of the brain, especially those that contribute to memory. Nearly a century after scientists recognized the atom as a fundamental unit of matter, neuroscientists followed by recognizing that the neurons play a similar role. Certainly, when examining the neuron in depth today, the picture has much changed from the simple view of passive integration of incoming signals, into the view comprising a complex interplay of voltage-gated ion channels with local synaptic regulation. Here, we focus on the fundamental aspects of a neuron that explains how a brief stimulus from the world can have long-lasting effects on the brain.

The neurological basis for memory is widely believed to be the long-term potentiation (LTP) of neuron synapses. After a synapse undergoes LTP, subsequent stimulus of the synapse will display a stronger response than prior to undergoing LTP. In 1973, Bliss and Lomo [24] first observed

LTP after repeatedly stimulating rabbit brain cells and found responses to increase 2-3 times and persist for several hours. Some researchers consider LTP to be a neurobiological codification of the Hebbian learning process: "Neurons that fire together, wire together" [83].

Interesting aspects of LTP are its various forms of persistence and its connection with memory consolidation. It is now understood that LTP occurs in at least two stages: *early LTP* and *late LTP*. In early LTP, increased response is achieved for a few hours by temporarily increasing the sensitivity and number of receptors at a given synapse [111]. In late LTP, more long-lasting changes involve production of proteins to signal changes to the synapse's surface area and additional dendritic spines associated with stimulation [111].

But how long do these changes last? In general, synapses undergoing early LTP return to baseline within three hours. Late LTP, however, has a much longer duration, in certain circumstances, lasting years (See Abraham's review on LTP duration [1] for a more in-depth coverage). LTP in the rat hippocampus lasting months and in one instance, one year, has been observed in the laboratory simply after applying four instances of high frequency stimulation spaced by five minutes [2]. In the human brain, newly formed memories are only expected to persist in the hippocampus for a few months until system memory consolidation into the neocortex is complete. This is consistent with amnesia patients who have difficulty recalling long-term memories a few months prior to their accidents [184].

Further neurological processes of memory are of interest, such as long-term depression (LTD) and neurogenesis. Whereas LTP increases the efficacy of synaptic transmission, LTD unravels those improvements to make it more difficult for two neurons to fire. The interactions between LTD and LTP are not yet entirely understood; however, it is known that during initial phases of LTP, reversal is more easily accomplished but becomes less so as time passes [1]. If LTP is a mechanism for rapid memorization, are there other possible mechanisms for changes in the brain? In short, the answer is yes. With neurogenesis it is possible to grow new neurons and form new growths of white matter. Brain cells were once considered to be like teeth, once lost we could not regrow new brain cells. Now, it has been demonstrated that brain cells routinely die and new ones grow throughout our lives [177]. One of the most striking examples is a study of taxi drivers in London (who need to know very detailed spatial and contextual representations

such as street intersections, routes, and traffic conditions) that found when comparing the size of the hippocampus (an area of the brain responsible for remembering associations and spatial memory) with that of the general population, a significant increase in size was observed and was correlated with time on the job [113].

A.3.2 Role of Hippocampus in Rapid Memorization

Few medical cases both arrest the imagination and have made as profound an impact on memory research as has the story of H.M. [174]. H.M. was a man suffering from severe seizures who elected to have most of his medial temporal lobe bilaterally removed in an attempt to reduce the occurrence of the seizures. Although the surgery was successfully in reducing the seizures, an unforeseen consequence was that H.M. now suffered from anterograde amnesia, a condition where a patient cannot recall or form new memories but can otherwise recall past life events and facts and operate normally. H.M., with very few exceptions, could not learn new semantic facts, such as new words, or remember recent events, such as meeting a person. For H.M., retention of new memories generally only lasted a few minutes. If H.M. was having a conversation with a person for the first time, who then left the room and reentered after a few minutes, afterward H.M. would not have recollection of having met the person or even of having a conversation. A detailed analysis of the surgery performed on H.M. indicates that virtually all of the entorhinal cortex and perhinal cortex were removed, about half of hippocampal cortex remained although severely atrophied, and the parahippocampal cortex remained largely intact [42]. Since H.M., numerous cases have emerged demonstrating how different lesions result in loss of different memory abilities. However, the case of H.M. illustrates the essential role of the hippocampus in forming long-lasting memories.

The hippocampal network is responsible for many specialized memory activities such as remembering item familiarity, spatial location, temporal order, contextual details, and general associations. The hippocampal network includes the hippocampus, parahippocampus, and entorhinal cortex. When a stimulus reaches the hippocampus, several stages of processing and memory formation occur. In the earliest stage, the hippocampus determines the familiarity of the stimulus and, if deemed interesting enough, reinforces pathways that form basic associations. In

later stages, events are formed into experiences, higher-level episodic formations, by integrating with information held in the prefrontal cortex.

Morris and Frey postulate that the hippocampus provides the ability for an "automatic recording of attended experience" [131]. They argue that many important events cannot be anticipated nor recur. Hence traces and features of experiences must be recorded as they happen. Further, Morris makes the argument based on neuroanatomical studies that the hippocampus does not store sensory stimuli directly, but rather associates indices in other cortical regions [132]. For example, the memory of eating a new food at a restaurant is associated with various stimuli (the visual appearance, aroma, taste), contextual details such as the movements of other patrons, and semantic details such as the name of the restaurant. The hippocampus is well equipped for its role of automatic association. It contains very plastic and deeply branched neurons called *pyramidal cells*. Further, it serves as a central hub integrating multiple sensory pathways from perceptional regions with interpretations from the prefrontal cortex.

Although studies of amnesia patients provide insight into loss of ability, they cannot account for how the hippocampus operates for healthy people. However, imaging studies of people performing memorization tasks have provided understanding of the hippocampus. In one study, subjects memorized a list of words and then were asked to recall the studied words [54]. What was unique about this study was that fMRI images were taken while the subjects were studying and recalling the words. The researchers found that failure to recall a word was linked to weaker activity in the hippocampus during memorization. In contrast, success in recalling words was linked to stronger activity. From this study, one could conclude that if a stimulus failed to induce LTP in hippocampal cells at the time of the event, then no conscious memory is likely to persist. Another study has found a similar effect in the entorhinal cortex for items judged to be familiar but not recalled [130].

Research has also found evidence suggesting that specific subareas (e.g., perirhinal or parahippocampal cortices) and specific lateralization (left or right) appear to be associated with different functions (e.g.„ familiarity recognition or encoding) and different modalities (e.g.„ spatial vs. verbal). However, it is still not entirely clear how well we can localize function. For example, the parahippocampus was associated with encoding and recall of spatial memories [159], but activity

in the parahippocampus was also found to be highly associated with recognizing objects with unique contextual associations (e.g., A hardhat invokes a specific context of dusty construction yards and is therefore associated with higher parahippocampal response; whereas a book has a less specific context and thus less activity [16]). One view put forward by Mayes, proposes that rather than operating on specific modalities of a hard-coded domain, such as verbal specific processing, the hippocampus supports different types of associations—inter-item, within-domain, and between-domain associations—and requires different computations for these associations types [115]. This domain dichotomy view explains why a process such as familiarity recognition may be associated with different regions because recognizing a familiar object would require different processing (and thus different regions) than recognizing a familiar object and location association.

A.3.3 PFC: Goal Memory and Executive Processes

The frontal lobes are the most modern addition to the human brain, providing facilities for planning and reasoning about everyday tasks and social situations (see Figure 41). Within the prefrontal cortex (PFC) lies amazing circuity for maintaining attention on important items during a task and for prompting attention when we need to perform a prospective action.

Humans fluidly perform and seamlessly switch among different tasks in a day. Routine activities, such as ordering a cup of coffee, can be performed without much cognitive effort. The rules are readily apparent: selecting a cup size, specifying a brew, paying the cashier—yet routine activities can be highly dynamic and even arbitrary. People have no problem adapting the rule to buy a cup of tea instead, or, given a rule never before encountered, *clap if you hear a phone ring*, most people would have no problem performing the task. However, if the rule was instead, *clap if you hear a phone ring in a coffee shop*, then people may fail to remember to apply the rule. Such forgetting would be a failure of *prospective memory*, remembering to remember. How does the brain store and manage prospective memories and other supporting memories needed for performing tasks?

The prefrontal cortex (PFC) is a region situated in the most anterior (toward forehead) portion of the frontal lobe. The PFC, a recent evolutionary addition, extends from the motor control

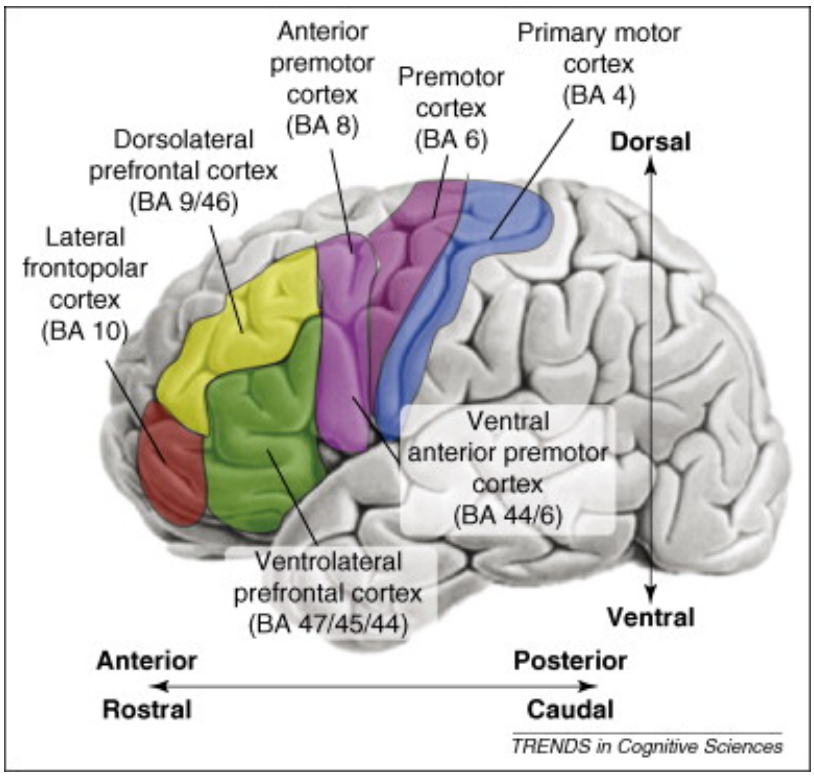


Figure 41: Frontal memory regions [12].

regions of the frontal lobe to provide cognitive and executive control. E. Miller and Cohen [122] provide a compelling and influential account of the PFC. They argue the PFC provides the ability to bias a particular response from many possible choices. For example, when crossing a street, a person may be accustomed to looking left to check for oncoming traffic. However, if that person were an American tourist visiting London, then top-down control would be required to override the typical response and bias it toward a response for looking right first. In this theory, rules, plans, and representations for tasks are learned via highly plastic PFC neurons (a view also shared by Fuster [67]), but may migrate over time. One apt metaphor offered by Miller and Cohen is a railroad switch:

> *"The hippocampus is responsible for laying down new tracks and the PFC is responsible for flexibly switching between them."*

The PFC also plays an important role in top-down attention: In early studies of monkey brains, when a food reward was shown to a monkey and then subsequently hidden for a delay period, persistent firing of neurons in the PFC was sustained during the delay period. Despite distracting

stimuli, the monkey could recall the location of the food reward. However, monkeys with PFC lesions could not maintain attention and performed poorly at recalling the food [68].

More recent work has uncovered a possible mechanism for how the PFC can simultaneously maintain several active items in mind. When examining the firing patterns of ensembles of neurons, rhythmic oscillations can be observed. These oscillations encode the attributes of an attended item. Siegel and colleagues [178] observed that when multiple items need to be attended to, distinct items were maintained in distinct phase orientations of the oscillating signal. Our limited ability to attend to multiple items are constrained by the speed and space it takes to encode waves within a limited frequency spectrum. An interesting benefit emerging from phase coding of items is the "free" temporal order of those items. In the same experiment, when the order of items were misremembered, there was a correlation with inadequate phase separation of the encoded items: The signal still preserved enough information to represent the items, but not enough information was available to determine their order. This view offers an interesting alternative to the concept of *working memory*. The prefrontal cortex can maintain many representations for tasks (especially if the representations refer to associations within the hippocampus), but can only attend to a few at a time.

A.3.4 Memory Consolidation and Schemas

Consolidation of memory from the hippocampus to the neocortex (also called system consolidation) was traditionally viewed as a slow process taking months. However, some research suggests schemas, such as those predicted by Morris [132], allow this consolidation to happen more rapidly. In one recent animal study, rats were trained for one month on how to associate cues with certain types of food in hidden sand caches (the locations of cues and types of food in caches are rearranged between trials) [195]. The pairings of the cues and caches provided a rule for the rat to reason on how to find more food (e.g., if the blue marker has a bacon pellet, then the red marker will have one too) which provided a schema for performing the task. In the study, rats that learned the pairings in one trial were given hippocampal lesions several hours later. Lesioned rats with task schemas demonstrated system consolidation into the neocortex within a few hours (as opposed to months or years) whereas rats without schema learning had

no memory of the food cache because memory was slower to consolidate without a task schema. Amazingly, lesioned rats with task schemas retained this knowledge even when tested 4-5 months later.

A.3.5 Memory Organization and Architectures

A.3.5.1 Understanding Memory Types

As previously mentioned, researchers distinguish between sensory, short-term, working, and long-term memories. For long-term memory, Squire proposed a taxomony [185] 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, an influential memory researcher publishing for over 50 years, describes semantic memory as knowledge of facts and episodic memory as a recollection of past events. Tulving's experience with an amnesiac patient E.P., who could learn new facts but not remember how he came to learn about them, lead Tulving to distinguish between our ability to *know* (to recall that the sky is blue) and *remember* (to relieve a past experience via mental time travel) [196].

Studies of patients with newly acquired amnesia have revealed further subtypes of memories. This includes familiarity, recency, and source memories. *Familiarity memory* involves the "feeling of knowing" that an object in a particular context has been encountered before without necessarily recalling the context (e.g., seeing a face in the crowd that seems familiar but does not trigger a name). Familiarity memory is not to be confused with priming. First, in priming, a person previously exposed to an item is more likely to recall that item in the future, however, without a conscious recollection of having been primed. With familiarity, a person is aware that something seems familiar. Second, priming and familiarity have doubly dissociated brain regions: Familiarity is supported in the entorhinal cortex; priming is believed to involve *modification* of the object representations within perceptual memory (for example, H.M. could be primed only for words he had learned prior to his accident) [161].

Some tasks involve recalling how long ago an event occurred, called *recency memory*. Milner studied patients who underwent surgery affecting the frontal lobes and found certain patients

would have difficulty recalling how recently they have seen a word [126]. This suggests the prefrontal cortex plays a role in maintaining and binding a temporal context to memories. Further research has uncovered the importance of top-down involvement of the prefrontal cortex in episodic memory. Although many associations can be remembered in a bottom-up fashion as part of episodic memory, certain types of memories require top-down control and thus direct involvement of the prefrontal cortex.

Often when we learn facts we can associate the initial experience with where we learned that fact. These types of memories are called *source memory*. Activation of the prefrontal cortex is necessary for forming source memories [73].

The daunting array of different types of memory was my primary motivation to introduce an simpler framework for understanding the results of cognitive neuroscience consisting of five types of memory: prospective, attentive, associative, episodic, and conceptual.

A.3.5.2 Memory Systems

Since the modal model of memory was proposed in 1968, numerous findings have challenged many of basic premises of the model and, accordingly, several researchers have sought to put forth their own account. Here, we review a few of these models.

In Tulving's serial-parallel-independent (SPI) model [197], rather then providing a mechanistic model of memory, Tulving provides a few guiding principles and generalizations. Simply, he believes the process of encoding a memory to be a serial process (output of one system provides the input for another), the process of storage to be distributed and parallel (traces of a single stimulus exists in multiple regions of the brain with the potential for later access), and the retrieval of memory to be independent (different systems do not depend on others—once a memory is formed it is available within that system even if others are damaged). This view highlights the importance of viewing memory as a network of coordinating systems rather than as a unified store.

In Fuster's account of memory systems [67], he uses a two-stage model derived from anatomical and neuropsychological studies of the brain. In general, raw senses are processed at progressively higher and higher levels of analysis starting from the most posterior region of the brain to

the most anterior region of the brain. Fuster divides this journey into two major components: perceptual memory and executive memory. Within the perceptual region, processes and memory start from phyletic sensory memory, then being integrated into polysensory, forming into episodic memory, generalizing into semantic memory, and abstracting into conceptual memory. After a brief hop over the motor system, the executive memory region involves concept, plan, program, act and phyletic motor memories. Fuster's account again highlights the specialization and localization of memory, but emphasizes the importance of top-down and bottom-up processes in memory, and how processing of a stimulus is collocated with its memory.

In Anderson (of ACT-R fame [8]) and colleagues' account of memory [7], a cognitive architecture is composed of several modules responsible for specialized processing of information. Modules have access to buffers which include: a goal buffer, a retrieval buffer, a visual buffer, and a motor buffer. Interestingly, the model also includes a production system for learning based on the basal ganglia. The basal ganglia is mainly responsible for motor control; however, it also has been "recruited" by the prefrontal cortex for reward-based learning of rules [123]. As such, the strength the ACT-R model is in simulating learning and problem solving; however, the model is less effective in modeling memory retention (for an exception, see Altmann and Trafton's work on memory for goals [4]).

The following memory types are described in my taxonomy of memory types and summarized in Figure 42:

PROSPECTIVE MEMORY

Prospective memory holds reminders to perform an action in the future under specific circumstances [209] (e.g., setting up a mental reminder to buy milk on the way home from work). Prospective memory is located in the anterior prefrontal cortex (lateral Brodmann area 10) of the brain and is supported by memory processes distinct from those supporting other types of memory [165].

When forming a prospective memory, both an intended action and a retrieval cue are stored. Subsequently, perceptual processes monitor the environment for the cue, retrieve the memory, and bring cognitive attention to the intended action.

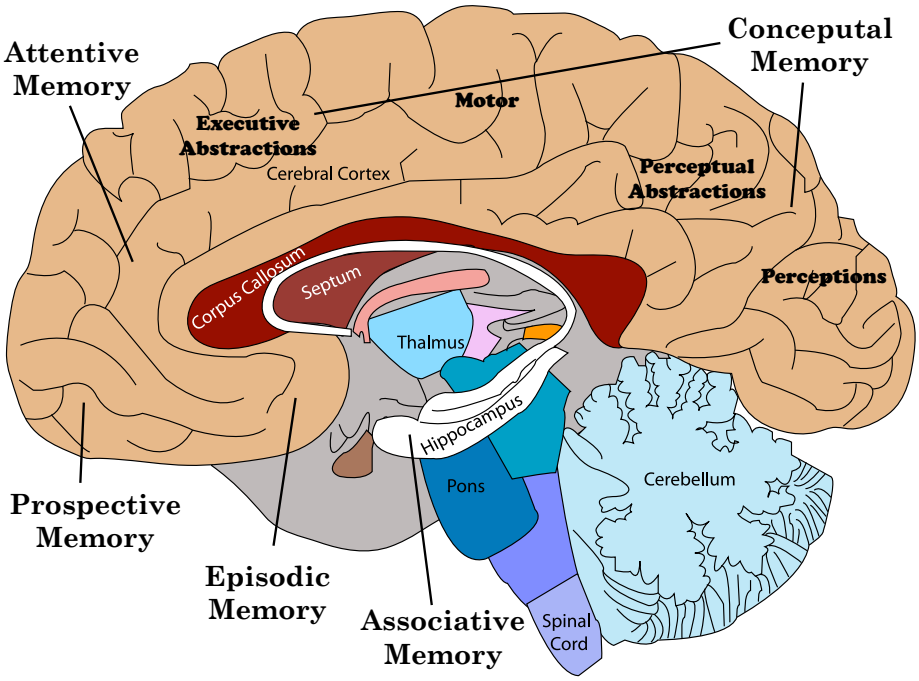


Figure 42: Memory types in sagittal view of brain.

Understanding how cognitive control occurs in the prefrontal cortex in relation to prospective memory is still an ongoing research question. Some researchers believe prospective memory requires some form of attentional resources [183], whereas other researchers believe if reminder cues are readily available then the process could be automatic [116]. A recent fMRI study has found that depending on the nature of the intention, prospective memory could involve both strategic monitoring and automatic retrieval from cues [165].

Given the complexity of the processes for forming and recalling prospective memories, naturally there are several points of failure [104]: When an intention is held in prospective memory, monitoring process continually scan for the conditions necessary for acting upon the intention. These monitoring processes compete with other cognitive resources, leaving prospective memory susceptible to *monitor failure*, failure to act on an applicable intention. When a condition is realized, prompting processes must also compete against active goals in order for the intention to receive conscious attention. Therefore, prospective memory is also susceptible to *engage failure*, a failure to acquire conscious attention.

ATTENTIVE MEMORY

Attentive memory holds conscious memories that can be freely attended to. Within it, goals, plans, and task-relevant items can be sustained for substantial periods of time. Attentive memory is found in the ventrolateral and dorsolateral prefrontal cortex (Brodmann areas 8, 9, 44, 45, 46, and lateral 47), a region situated in the most anterior (forehead) portion of the brain's frontal lobe. Attentive memory has two complementary operations with corresponding neural mechanisms: focusing and filtering.

The ability to maintain focus on items has been well studied in the PFC of primates and humans [106]. In early studies of monkey brains, when a food reward was shown to a monkey and then subsequently hidden for a delay period, persistent firing of neurons in the PFC was sustained throughout the delay period. Despite distracting stimuli, the monkey could recall the location of the food reward. However, monkeys with damage to the PFC could not maintain attention and performed poorly at recalling the location of the food [68]. A human equivalent of these monkeys has been found in one patient, Clive Wearing. After a viral infection that destroyed much of his PFC and hippocampus, the patient could no longer attend to any memory longer than 30 seconds [207].

Humans can fluidly filter and switch between attended items via selective attention. *Selective attention* is the ability to conditionally attend to an item based on attributes of the item. For example, in a crowded room, attention can be switched from people with red shirts to people with hats. This ability derives from highly plastic neurons that can become tuned to firing when certain attributes, such as a particular color, are present [69]. Selective attention complements the ability to focus on multiple items by allowing distinct groups of items to be attended to. Thus, via selection attention, more items can be concurrently attended given that the items have distinct attributes and are distinctly recalled [144].

Attentive memory is highly volatile and prone to frequent failures. When a programmer is actively engaged in a programming task, attentive memory allows a programmer to maintain focus on particular programming elements or goals that are relevant to a programming task. Although residuals of previously attended items can be found after switching attention [162], task switching often results in a *concentration failure*, a failure to maintain focus on an item.

Attentive memory can only provide reliable focus on a few consciously accessible items at a time. Constraints imposed by phase coherence and modality separation frequently induce a *limit failure*, a failure to hold the required number of items. Interruption is very likely to disrupt a programmer's maintenance of attended items, such as the programming locations being edited.

HIPPOCAMPAL NETWORK

The next two types of memory make use of the same pathways of the brain, called the hippocampal network.

The hippocampal network is one of the most important components of human memory. It is responsible for many specialized memory activities such as remembering familiarity, spatial location, temporal ordering, contextual details, and general associations. The hippocampal network includes the hippocampus, parahippocampus, and entorhinal cortex.

This network comprises extremely plastic neurons that are widely connected to multiple processing centers in the brain. Without the hippocampus, most memories would not exist. This has been demonstrated in several ways. Rats have been genetically engineered to not retain any memory longer than a few minutes by suppressing neurotransmitters used in the hippocampus [62].

Several amnesia patients who have had their hippocampus surgically removed, have demonstrated similar symptoms. One such patient, H.M.[174], with very few exceptions, could not learn new facts such as vocabulary or remember recent events such as meeting a person. For H.M., retention of new memories generally only lasted a few minutes. Otherwise, H.M. could recall past life events and facts and operate normally. Unlike Clive Wearing, H.M. had an intact prefrontal cortex, and thus could still attend to existing memories for many minutes at a time, even holding long and intelligent conversations.

This complicated structure is used by two main memory components: associative memory and episodic memory. When a stimulus reaches the hippocampus, several stages of processing and memory formation occur. In the earliest stage of processing, the hippocampus determines the familiarity of the stimulus and, if deemed interesting enough, reinforces pathways that form basic associations, leading to eventually higher-level episodic formations.

HIPPOCAMPAL (ASSOCIATIVE MEMORY)

Associative memory holds a set of non-conscious links between manifestations of co-occurring stimuli. Associative memory is located within the limbic system, in the perforant pathway of the hippocampus. Associative memories are essential for the "automatic recording of attended experience" [131]. The reason why the brain evolved the ability to record such activity is that many important events cannot be anticipated nor recur, and therefore traces and features of experiences must be recorded in real-time.

Despite the raw power of associative memory, it has many weaknesses. When an associative memory is born in the hippocampus, it is still fragile, and its expected lifetime is only a few hours. Formation of an association is determined by uncontrollable factors such as novelty, or interest. For example, in brain imaging studies of subjects memorizing words, the experimenters could predict which words would be forgotten based on their activation strength in associative memory [54]; that is, forgotten words did not produce a strong enough response to engage associative memory during the memorization period. In such cases, the result is a *retention failure*. To combat this failure, people often form intentional associative memories though internal speech (activation of speech motor systems and speech comprehension [84]), which is nearly equivalent to hearing ourselves speak aloud and may subsequently excite auto-associative mechanisms [119].

Other times, an associative memory is formed, but with weak or missing associations. For example, it is common to associate with the visual features of an item, but fail to associate with other attributes such as its name, limiting our ability to recall it. This phenomenon is like when someone says, "I'll recognize it when I see it". As a result, *association failure*, a failure to form complete or strong associations, frequently occurs.

HIPPOCAMPAL (EPISODIC MEMORY)

Tulving describes *episodic memory* as the recollection of past events [196]. Whereas associative memory provides the facility for soaking up raw experiences, episodic memory involves a much more complex network of memory processes. Located in the entorhinal cortex (Brodmann area 28 and 34), episodic memories involve highly processed input from every sensory modality, as

well as input relating to ongoing cognitive processes. Additionally, as the brain develops over time, the ability to learn and anticipate complex forms of episodic structures enables more concise representations of experiences to be retained [100].

In order to form episodic memories, cognitive resources are required, and when those resources are otherwise engaged (e.g., on a hard programming task), memory failure can occur. Episodic memory requires processes in the lateral prefrontal cortex for maintaining information about recency and ordering of events retained in the hippocampus [73]. As a result, when learning new experiences it is common to experience a *recollection failure*, a failure to recall an orderly and complete sequence of events. However, research has shown that episodic cues can assist in improving episodic memory performance (even in memory-impaired patients [85]).

Episodic memory is not as fully automatic as the spatio-temporal and perceptual components of associative memory and can be easily disrupted [96]. For example, a person may remember the experience of hearing sentences being read aloud; however, they commonly forget details such as whether the voice was male or female or the specific order of the sentences. Therefore, experiences requiring heavy cognitive load are susceptible to *source failure*, a failure to recall contextual details associated with an experience.

CONCEPTUAL MEMORY

How does the brain remember objects such as a hammer and concepts such as *tool*? The brain first learns basic features of encountered stimuli such as the wood grains and metal curves of a hammer and then organizes those features into progressively higher levels of abstraction. In this way, *conceptual memory* is best understood as a continuum between perceptions and abstractions. There is no one place in the brain that stores the concept of a hammer, but rather the representation is distributed throughout the brain.

In Fuster's account of the brain [67], the continuum of conceptual memory is ingrained in the physical organization of the brain. Fuster divides the brain into two major components: the perceptual region and the executive region. Within the perceptual region, starting in the most posterior (rear) region of the brain, banks of highly-tuned neurons fire in response to basic perceptions, such as color or lines. Continuing forward, more complex perceptions such as

orientation or movement are processed. Eventually, perceptions such as a bouncing sphere give way to concepts such as a *ball* and so on.

The executive region exhibits the same pattern of abstractions as its perceptual counterpart. It contains abstractions over action: acts, plans, programs, and goals. Starting in the premotor area, with responsibility for planning basic actions such as phyletic motor movements, the level of abstraction increases as one moves onward, ending at the most frontal regions of the brain containing the most abstract concepts such as goals.

When a stimulus is encountered, that stimulus is likely to stimulate neurons and memory processes throughout the brain. Although conceptual memory can be thought as a single continuum, there are several dynamics and differences at play.

A process called *repetition suppression* enables the brain to retain memory of previously seen perceptions by slowing the firing rate of the neurons related to those perceptions. This effect can last for days as perceptions become more abstract. Repetition suppression causes certain perceptions to be primed. *Priming* occurs when suppression of certain areas short-circuit the processing of information, allowing the response to become more probable.

There are several failures possible when remembering perceptions. As a result of priming perceptions and abstractions in conceptual memory, associative and attentive memory become more effective over time in successfully forming associations and increasing focusing capability. However, perceptions must be continually refreshed due to the short duration of repetition suppression. Further, this information is exceedingly low-level and non-conscious. To recall a forgotten perception, one would have to rely on residual effects such as priming, which is error-prone and involuntary. Memory of spatial information can be additionally supported by associative memory. However, it is fickle and easily disturbed by unstable arrangements of spatial and visual information as often encountered when navigating computer interfaces. In general, the state of unprimed memory results in an *activation failure*, an inefficient state of conceptual memory. Interruption may reduce the effect of priming of concepts needed for a programming task, requiring that the programmer refresh his memory.

There are also weaknesses possible when remembering abstractions. One weakness is that several exposures may be required before an abstraction can be formed. That is, a person may not

be able to incorporate an abstraction directly into the processing and remembering of instances of that abstraction until after systematic consolidation has situated that abstraction in the processing pathway. Although effects such as priming also apply to abstractions and can ease the recall of information, these effects are not sufficient mechanisms for maintaining a conscious memory as seen in patients such as Clive Wearing or H.M. Because the formation of abstractions in conceptual memory rely on systemic consolidation, it is common to experience *formation failure*, a failure to form an abstraction. Interruption may reduce the ability for a programmer to hold together newer ideas that do not yet have conceptual memory support.

Appendix B

DETAILS ON EMPIRICAL STUDIES OF PROGRAMMER INTERRUPTION STRATEGIES

In this appendix, I describe a collection of studies I performed to understand how programmers manage interruptions: a survey of programming professionals and an analysis of the activities performed to resume a programming task from recording programming sessions. I describe the datasets from these studies, report on the observed activities, and describe insights and scenarios emerging from the data. The studies have been published in the Proceedings of the International Conference on Program Comprehension [151] and extended for publication in the Springer Software Quality Journal [152].

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. I describe an exploratory analysis performed on 10,000 recorded programming sessions of 86 programmers and a survey conducted on 414 programmers to understand the various strategies and coping mechanisms developers use to manage interrupted programming tasks. For example, Rugaber and I found that task resumption is a frequent and persistent problem for developers. We found that only in 10% of the sessions does programming activity resume in less than one minute after an interruption, and 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.

B.1 Datasets and Measures

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

B.1.1 Interaction History

I draw upon a variety of data sets in the analysis. The first data set I will call the *Eclipse data set*. It was originally collected in the latter half of 2005 by Murphy et al. [133]. The researchers used the Mylyn Monitor tool to capture and analyze fine-grained usage data from volunteer programmers using the Eclipse development environment [1].

The *Visual Studio data set* was collected in 2005 by Parnin and Görg [149] at an industrial site. The data covers ten developers working for 30 work days (1.5 months). Since the original publication, the data reported here also covers two more developers and a few more months of data.

The *UDC dataset* is publicly available from the Eclipse Usage Data Collector [192] 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 minutes or more. This segmentation is well supported by the nature of the data. The interval between events follows a Poisson distribution: for 98% of the 4.5 million events, the time between those events is less than a minute. This means tight clusters can be formed with any threshold above a minute. Similar thresholds have been reported in other studies [214, 166].

Events collection is performed solely by the IDE. It should be noted that what appears to us as a break in programming activity, may be the developer engaging in a related task such as checking in source code or searching for online code examples. If I included window focus events [164], then I could better explain these breaks. Nevertheless, I believe many breaks from programming activity are due to diversions [14] that often derail programming efforts, as observed by Ko [103].

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);

[1] `http://eclipse.org`

Table 18: Summary of data in the *UDC*, *Visual Studio*, and *Eclipse* data sets. The column Filtered* indicates the remaining number of sessions after removing sessions with a duration less than one minute and the column Edits+ indicates the number of filtered sessions with at least one edit event.

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

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

In the analysis, I accounted for these differences by running the experiments separately to check for any discrepancies between the data sets. A summary of the data sets can be seen in Table 18.

B.1.2 Survey

I 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 gaming companies. I initially conducted the external study and, encouraged by the feedback, distributed the survey with slight modifications to employees at Microsoft. Overall, I found general agreement with the external survey and internal survey, but only report the internal results here.

As I was primarily interested in professional software development, I selected the data from full-time software developers—excluding interns or people in roles such as tester or project manager. I also excluded developers that had been solicited for other surveys by the research group, to avoid oversampling. Respondents were compensated by a chance to win a $200 gift certificate. From the participant pool, I 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. I drew the selection of future tools from current research proposals for improvements to programming environments. I 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.

B.1.3 Measures

B.1.3.1 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 leading to a recovery period on the order of minutes [203]. Therefore, I 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 to it (see Figure 43).

In this study, I focus most of the 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 activity is related to resumption costs. For this reason, I only examine sessions with coding activity.

B.2 Analysis

When a programmer is interrupted from a programming task, what activities need to be performed in order to resume work? In this section, I 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 any resumption delay at all?

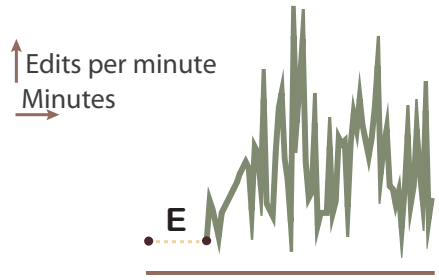


Figure 43: Edit lag (E). When starting a new programming session, a latency can be typically observed before coding activity (edits to code) begins. I believe activities during this time period are used to prepare for the coding activity. Typically, once coding activity has been initiated, it strongly persists for the rest of the session (see graph line).

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

In Figure 44, I display the distribution of edit lags across all sessions having editing activity. In 10% of the sessions, edit lag was less than a minute. For the rest of the sessions, several minutes pass until the first edit event occurs. In about 30% of sessions, the edit lag is over 30 minutes. In these sessions, I 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 19, I 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.

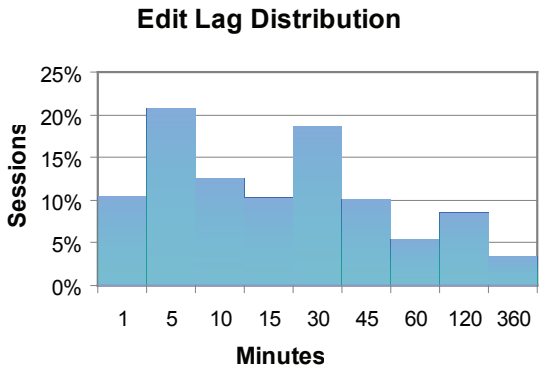


Figure 44: Developers often do not make their first edit of the session until at least several minutes have past.

SESSIONS IN A TYPICAL DAY						
sessions	1-3	1-2	1-2	1	0-1	rarely
duration	15m	30m	1h	2h	4h	8h

Table 19: The frequency of sessions of various durations is shown. In a typical day, developers program in several short sessions with an additional one or two longer sessions. That is, in a typical day a developer might have three 15 minute sessions, two 30 minute sessions, a one hour session, and another two hour session.

B.2.1 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, I make several measurements. First, I measure how often the first change is made without navigating to other methods or classes. Second, I measure how often the first change is eventually made in the last edited method even with intervening navigations to other methods or classes. Finally, I measure the edit lag for both cases. Only the *Visual Studio* data set had sufficient information about navigation for this analysis.

The analysis finds that in 91 of 1213 sessions (7.5%) the programmers were able to make changes without navigating to other methods. Second, in 209 out of 1213 sessions (17%) the programmers do eventually return from visiting other locations and make changes to the last

Table 20: Developers are able to resume coding within one minute for 35% of sessions and within 30 minutes for most sessions when the work involves completing the last edited method.

	RESUMPTION COST			
Sessions	35%	22%	23%	12%
Edit Lag	1m <	1-5m	5-15m	15-30m

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 20. When the programmers do continue work in the same method, they can often resume work quickly. When resuming coding in the last method, the programmer does this within one minute 35% of the time. I 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 task to next 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.

B.2.2 Navigate to Remember

In the previous subsection, I 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, I separate the analyses and use two different

metrics. With the *Visual Studio* data set, to measure the number of places visited, I record the set of all methods or classes visited before making the change. I then measure the navigational cost as determined by the distance between code elements. For the *Eclipse* data set, I measure the number of selection events prior to the edit.

Table 21: Developers typically visit several locations of code before beginning a change. The range of elements covering 75% of values is shown with the mean in parentheses.

	VISUAL STUDIO	ECLIPSE
Locations	2-12 (7)	
Navigation Distance	4-40 (27)	
Selection Events		15-150 (135)

The results can be seen in Table 21. In general, the developers navigated within the code to several locations before beginning to edit. The developers also took considerably longer to start coding when navigation was involved. As shown in Table 22, 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 20).

B.2.3 Execute the Program

Debugging is an expensive but comprehensive way of obtaining knowledge about program behavior. I 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, I examined commands used in the *Eclipse* data set related to launching a debugging session or execution of programs. I 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 23, I 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 suggests that many of these sessions were continuations of previous debugging sessions. Another explanation is that running the program through the debug command provides advantages such as breaking on an exception or retrospectively setting a breakpoint. Also of interest is the frequency of program execution:

Table 22: Developers take a little longer to start coding when the work involves navigating to other locations first. In contrast with Table 20, less sessions are quickly resumed (35% vs. 16%).

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

Table 23: Frequency of programmers debugging or running the program before making their first edit.

	Pre-Edit Lag	Post-Edit Lag	% Sessions
debug	521	1666	13%
run	84	693	2%
total	605	2359	15%

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 progress and correctness.

Table 24: Participants reported their frequency of program execution for resumption.

frequency	5%	25%	50%	75%	95%	weighted average
participants	126	89	68	48	22	33%

From the survey results (see Table 24), 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, 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.*

B.2.4 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, I measure how often programmers leave behind these types of reminders.

To measure the use of roadblock cues reminders, I examine the *Eclipse* data set for 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 I 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. Also, I could not directly measure the use of prospective notes in the code through the interaction histories data sets, but have results concerning this possibility from the survey.

Table 25: Viewing problems left over from the last session is common.

	PRE-EDIT LAG	POST-EDIT LAG	% SESSIONS
problem view	301	265	9%

The results show that the Problem View was used in 9% of sessions (see Table 25). From the 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 43% of resumptions.
Several participants described how they used compile errors. When first resuming, participants often 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 need 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.

B.2.5 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, I measure the activity related to viewing details about tasks stored in a task repository. In addition, I also examine how developers use task-tracking software to understand how intentionally developers are tracking subtasks or small tasks with task tracking software.

To measure the use of task information, I examined commands used in the *Eclipse* data set related to accessing a bug repository such as Bugzilla, the Mylyn Task List, or the Eclipse Task List.

To measure the use of task tracking for management of subtasks, I examined the *UDC* dataset and *Eclipse* dataset. The *UDC dataset* includes commands recorded from users using Mylyn to track software tasks. I 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 tracking tools for managing small tasks.

The results for viewing task details are shown in Table 26. 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 minutes. 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 26: 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%

In Table 27, 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 doers of magnitude less than for managing assigned tasks.

From the 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 which 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.*

Table 27: 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

B.2.6 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, I count the number of sessions that have CVS or SubVersion history commands during the edit lag. To obtain the activity related to history, I separate the actions concerned with retrieving information from the managerial repository commands. I was primarily concerned with comparing revisions, viewing the history of a revision, viewing the commit log, and viewing a revision annotation. I also measure the occurrence of commands in the rest of the session to gain insight into the significance of commands occurring during the edit lag. Finally, I am only able to measure the *Eclipse* data set for revision history usage.

Table 28: 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%

The results can be seen in Table 28. From this measurement, I observe occasional use of revision history during the edit lag (only 4% of sessions). However, relative to commit commands, the history commands are used fairly frequently and are as likely to be used during the edit lag as during normal coding activity. Finally, edit lags were longer than 15 minutes for 59% of sessions when using revision history commands.

From the survey results (see Table 29), participants indicated they view a source difference

Table 29: 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%

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 I have collected may be conservative. Not all users always use the Eclipse plug-ins for accessing revision history. Future studies should expand the instrumentation collection to directly measure access to source control.

B.3 Discussion

Several analyses were described in Section B.2. In Table 30, a summary is given of the frequency of various activities developers performed when starting a programming session. Although results from the survey and interaction history logs gives different insights, it is interesting to note that both sources agree in the order of frequency among the resumption strategies. The survey results for strategy frequency may give the impression of small differences among strategies employed. 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 strategy B with 25% frequency and another 100 people using strategy B with 75% frequency). However, examining the underlying distributions show that they are significantly different ($p < 0.05$) by a Kolmogorov-Smirnov test for every distribution except between the View Problem List and the View Task List.

Table 30: 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%

From the survey results, I 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 to recover task knowledge. One participant describes using several of the strategies I 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.*

What do the presented results suggest about future research and tools? In the following, I review the findings and provide hypotheses that might explain the observed behavior and implications for tool research. Finally, researchers should be able to experimentally apply latency measures when studying their proposed tools.

B.3.1 Edit and Navigation Behavior

Relying solely on the last edit location is often insufficient for resumption. In some situations it is very effective: 35% of these sessions were resumed in less than a minute, which is much higher than the 10% of sessions overall that were resumed in less than a minute. However, this scenario was only applicable in 17% of the sessions and still required navigation to other locations. In other cases, developers experienced 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 that 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 [213, 120]. It is unknown how much contextual detail programmers lose after task completion. There is some evidence that gives direction. In Bailey's study [3], 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 returned to baseline levels. This suggests that memory access ceases after task completion, which may effect the retention of task-related information that was in attentive memory.

Finally, developers are spending a considerable amount of time navigating to several locations in the 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. Research should continue to evaluate alternative navigation interfaces such as CodeThumbnails [50], Relo [181], and SHriMP [187] and their affect on resumption costs.

B.3.2 Other Sources of Task Knowledge

I 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 other sessions. This 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 they impacts their tasks. This would be consistent with other research: Status awareness was one of the top information needs found in Ko's study [103] 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 sessions. Surprisingly, this process is still manual and could explain its 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 might include the ability to automatically summarize and highlight code differences between programming sessions. This would provide additional cues for programmers to trigger reminders on any incomplete tasks or unresolved bugs.

B.3.3 Toward Supporting Task Resumption

Having seen the variety of activities developers perform at the beginning of a programming session, I identify better tool support 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 largely depends on the manner in which the developers were interrupted and the current state of their

programming tasks. However, the variety of suspended task states and types of knowledge programmers need to recall poses many challenges to developing tool support.

With involuntary interruptions, developers have limited time to properly suspend their programming tasks and as a result will most likely select strategies that 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 insufficient cues contained at the last site of work.

To gain more insight to direct research efforts, I surveyed developers about what features they would like in future tools. The first set of features they described were approaches for enhancing environmental cues and 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 changes.
- **Code Thumbnails.** Thumbnails of recent places navigated or edited.
- **Instant Diff.** Highlighted code showing how a method body changed 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 past work.

Continuing from the last state of the programming task can happen in a variety of ways: The developer might resume coding at 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. 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 stages by including perspectives for different task stages or introduce light-weight mechanisms for annotating programming artifacts. A list of some measures that support task state include:

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

Table 31: 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

The developers' ratings in Table 31 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 into the code (Instant Diff), in a separate window (Change Summary), or 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 is the ability to recover fine-grained changes rather than just 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.*

B.3.4 Limitations of Analysis

I have presented an analysis of developers' activity before recommencing coding and proposed hypotheses explaining their behaviors. The nature of the datasets I have analyzed give us a good overview of behavior but understandably 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 the current study, I could use the structure of the experimental activity to infer properties about task resumption, but ultimately the effects may be compound. 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 tracking 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 described above establish an upper bound on the activity corresponding with resuming a programming task and set the basis for future confirmatory studies.

Differences both among 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 the ability to analyze the questions thoroughly (i.e., Visual Studio dataset was applicable for Section B.2.1, B.2.2). Within a dataset, differences between users, such as experience, and differences between tasks, such as investigation or refactoring, all influence factors surrounding resumption beyond my control. As such, any attempt to draw general conclusions about the impact of interruption from these results (i.e., resumption time) should be avoided. Instead, I recommend researchers use these results as a baseline for future study.

Finally, the interaction history data does not capture a complete representation of all possible activities that were performed. I 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 I have corroborated this evidence with results from the survey study, other types of studies would need to be performed to measure the frequency and effectiveness of these techniques.

B.4 Conclusions

In this appendix, I presented a set of empirical studies that I used to theorize about suspension and resumption strategies. I have presented an analysis of three sets of data that provides new insight into how programmers resume a programming task—particularly characterizing which activities are performed at the beginning of a resumed session. I conducted 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.

Appendix C

DETAILS ON CONTROLLED EXPERIMENT FOR EVALUATING RESUMPTION OF INTERRUPTED PROGRAMMING TASKS

In this appendix, I present a experiment that was performed to study the effect of cues on the resumption of an interrupted programming task. The experiment has been published in the Proceedings of the International Conference on Computer Human Interaction [148]. Developers, like all modern knowledge workers, are frequently interrupted and blocked in their tasks. Before performing the experiment, a contextual inquiry into developers' current strategies for resuming interrupted tasks was perfomed. DeLine and I ran a controlled lab study to compare the effects of two different kinds of automated note taking cues 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 well: Developers using either cue completed their tasks with twice the success rate as those using note-taking alone. Despite the similar performance of the cues, developers strongly preferred the cue that presents activities chronologically as code snippets.

C.1 Experiment

Based on the results of the preliminary survey, DeLine and I 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. The base condition consisted 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 [98], the most widely used research tool for developer task management; the

second is based on developers' rating of task resumption features from the preliminary study. The 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.

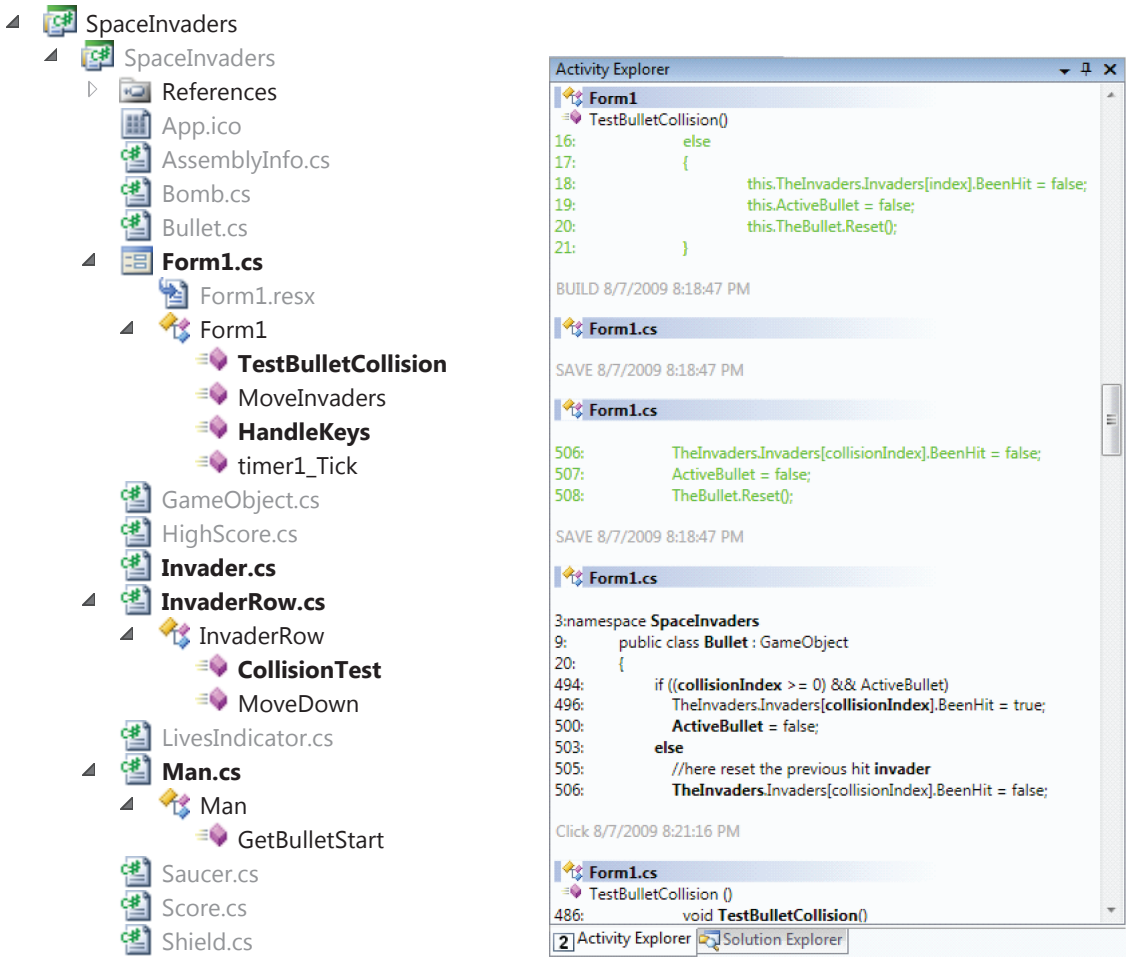


Figure 45: The experimental resumption aids: DOI treeview (left) and my content timeline (right).

The two experimental cues fed from the same interaction log and were shown in separate windows in the Microsoft Visual Studio 2008 development environment. The two cues, shown in Figure 45, differ only in how the developer's activities are presented. The first, called the *degree-of-interest (DOI) treeview*, consisted of a treeview of names of the program's parts, namely, its projects, files within projects, types within files, and 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 used three levels: unvisited parts of the code in gray, occasionally visited parts in

normal font, and often visited parts 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 text selected during programming is presented in black boldface, a code insertion is shown with the new code in green, and a code change is shown as a pair of before-and-after texts. Snippets with both kinds of cues, clicking on a program part caused the editor to navigate to the corresponding part. Mylyn also includes the ability to decay the DOI model as time passes and segregate different task contexts. For the study, I did not implement these aspects because the task duration was not long enough to trigger the time-decay, and I was already isolating workspaces for the different tasks.

The two kinds of cues differed in two dimensions. First, the DOI treeview presented program parts by name (project names, file names, type names, member names); the content timeline presented program parts by content (the program text). Second, the DOI treeview organized the developer's activities by the code's structure; the content timeline organized 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 I tested opposite corners of this two-by-two design quadrant in the hopes of maximizing the measured effects.

C.2 Research Methodology

C.2.1 Participants

Fifteen professional programmers (one female), average age of 39 years (range 31 to 56), participated in this study for receipt of two copies of Microsoft software. The programmers 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, I screened developers for experience with debugging and modifying the code of other developers and working in teams of at least 3 people.

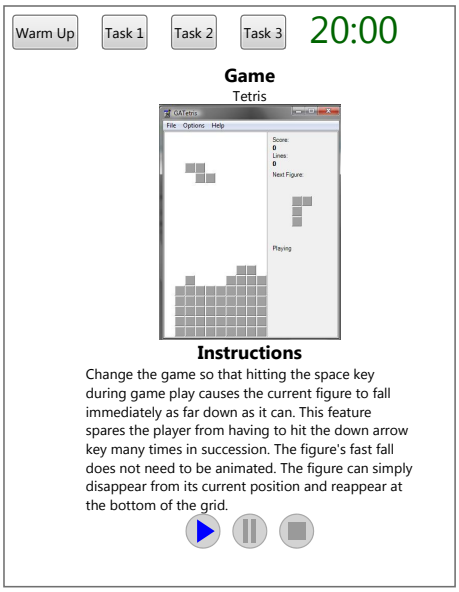


Figure 46: Application used for controlling interruptions and task switches in the experiment.

C.2.2 Methods and Procedure

Each subject was run individually in a user-study laboratory. An experimenter was present for the entire session. The experimenter informed the subjects of the goals of the study. Subjects were told that I was interested in how they multi-tasked among programming tasks and how they used the different tools given to them. In addition, I stressed that I was not interested in a perfect solution, but in completion of as much of the experimental task as possible.

For the tasks, I used source code from three simple games (Tetris [1], Pacman [2], and Space Invaders [3]). The games are C# implementations of classic arcade games. The motivation for using these games was based on the familiarity the games offered and the nontrivial complexity of their source code. Further, I had confidence from previous user studies 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 I provided to coordinate the subject's activities over the three programming tasks, shown in Figure 46. For

[1] http://www.codeproject.com/csharp/csgatetris.asp

[2] https://www.planet-source-code.com/vb/scripts/ShowCode.asp?txtCodeId=5669&lngWId=10#zip

[3] http://www.c-sharpcorner.com/UploadFile/mgold/SpaceInvaders06292005005618AM/SpaceInvaders.aspx

each task, the application automatically interrupted the subject several minutes after beginning the task, as described below. At the point of interruption, the subject reviewed the cue for that task (if any) and took notes, for up to one minute. The application then started 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 (in total), 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 Pacman, resuming with condition 2;
9. finish Space Invaders, resuming with condition 3;

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

To provide a consist and automated mechanism for interrupting the subjects, I used the following criteria: 15 seconds after a non-comment edit, after more than three non-comment edits, or after a twelve-minute timeout if no changes were made. I based these criteria on the factors Fogarty previously found to be associated with inopportune moments for interruption [64] and on the nature of the tasks. The solutions for the programming tasks typically required changes to multiple parts of the source code. By setting the interruption point several moments after making a change, upon returning, the developer must ensure the change is completed correctly and then recall the steps and relevant parts of code needed for the next change.

C.2.3 Tasks

Each programming task was to add a small feature to a 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 a 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 enemies, filling in an empty position if possible. If all enemies positions are filled, no new enemy is created.

I 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.

C.3 Results

C.3.1 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.

C.3.1.1 Task completion

Subjects completed tasks more frequently when one of the cues was available when resuming a task with a weak significant difference (chi-square, p <0.1, effect size .04).

Table 32: When developers had any form of activity history available, they were about twice as likely to complete a task than otherwise.

	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	

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 where 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 [51, 50].)

C.3.1.2 Error Rates

To better understand why some subjects failed to complete the tasks in the allocated time, I 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 a misunderstanding about the program or the changes he or she made.

Note that I 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 33 categorizes the failures of the seven subjects completing at least one task.

Table 33: When developers only had notes, they were more likely to make errors or to take longer to relocate necessary task context.

	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	

As one example of a programming error, one subject was interrupted in the middle of hooking the space key event for the Tetris task. When the subject later resumed the task, he completed defining the space key event, but forgot to connect the event with the handler. The subject moved on to the next part of the task, making the tetris block fall down, and quickly implemented the functionality much faster than other subjects. However, when the subject tested the solution, the missing association 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. Next, the subject needed to find out how to create a new invader, but was interrupted during the search. The subject took notes for the search, but did not jot down the code location for resetting the bullet. After resumption, the subject found how to create an invader, and needed to return to the previously found code for resetting the bullet. Unfortunately without sufficient notes or cues to rely on, the subject took 5 minutes to relocate this code. The subject was frustrated that she knew what code she 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 her time and could not complete the task within the allocated time. Experts could protect 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.

C.3.1.3 Lag Measures

To understand the cost of resuming tasks in the three conditions, I 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, I 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.

C.3.1.4 Note Taking

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

The contents 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 numbers). During task suspension, subjects wrote down various types of information. Several subjects summarized the task description into a simple one-line sentence ("Modify tetris so piece falls to bottom") or summarized what they had previously worked on. Most subjects tended to write notes about the most immediate subtask and neglected to record other locations needed later for the task. That is they recorded 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 referred to code symbols, while only 14% of the lines of situated notes referred to code symbols. This suggests that when subjects wrote unsituated notes on paper, they need to spend several lines of text recreating the

context. 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).

C.3.1.5 Subjective Ratings

After the tasks were over, I 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 the tool they used helped them resume their tasks (speed); (2) how well it reminded them of details from the interrupted task; (3) how useful they felt it would be after a week (durability); and (4) how well they liked it overall. Their average ratings are shown in Figure 47. 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 two-tailed t-test ($p < 0.001$, effect size 1.51).

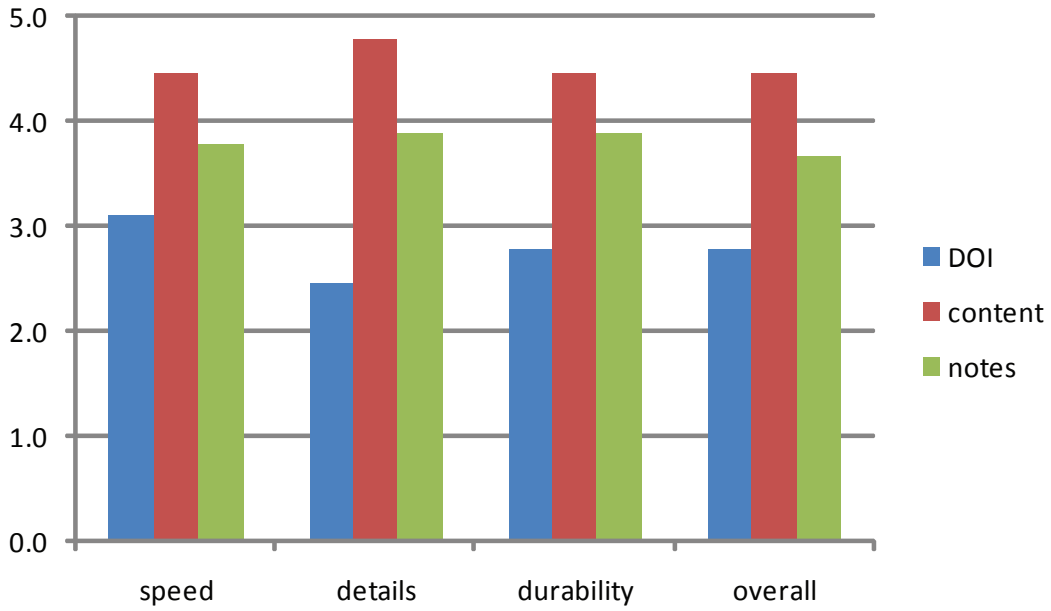


Figure 47: Subjects consistently preferred the content timeline over notes and notes over the DOI treeview.

C.3.1.6 User Feedback

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 triggering their memory about tasks. 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 the 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. 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 the ability to 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 work.

C.3.2 Discussion

Subjects used the cues in different ways. Subjects treated the DOI treeview like tabs in a multi-document interface. That is, when they could not remember which method contained the code they had in mind, they used the names as *information scents* [105] and clicked 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 returned to the activity timeline to rewind further back to review their previous actions before their last stopping point. From this review, subjects often saw the next place to return to.

Although the study found no difference between the DOI treeview and content timeline, subjects rated the DOI treeview 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 have been 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. This suggests notes remain an important mechanism for suspending interrupted tasks.

C.3.3 Threats to Validity

I choose a time limit for tasks that made it possible to conduct the experiment within a 2 hour time frame without exhausting participants. However, this time limit may have excluded less experienced participants who would need much longer time to complete the tasks. The study has focused on experts, many of whom completed the tasks within a few minutes, and may not generalize to novice users.

I believe that the minute to prepare for interruption that I gave each subject makes the study unrepresentative of interruptions without warning but adequately representative of interruptions

with warnings and self-interruptions, both of which frequently occur among professional developers [103]. I included this minute to prepare, despite the threat, because previous psychology studies have shown that having a period for studying an environmental cue during interruption makes the cue valuable during resumption [5]. The focus in this study was to measure differences in the cues.

Finally, an additional threat to external validity is the nature of programming tasks I selected. Programmers encountering new code may not be as able to effectively use memory cues such as the DOI treeview because they are not as familiar with the specific names within the program and they have little opportunity to explore. However, the 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.

C.4 Conclusions

I have presented an exploratory study that provides new insight into how programmers recover from interruptions and a controlled study that evaluates how cues assist resumption of programming tasks. The results provide a strong motivation for tool support to help 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 kinds of cues performed well, but subjects preferred the content timeline over the DOI treeview. This has important implications for other kinds of knowledge workers. 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 effective 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 a 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 may not want a coworker to see his mistakes when viewing his timeline. Yet, there is a fine line between showing a 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.

Appendix D

ADDITIONAL DATA

Table 36: Tasks and Backgrounds of Participants in Prospective Memory Study.

Age	38
Experience	As a certified Microsoft Developer (MCPD) I have been coding since 90's. My areas of expertise are desktop, mobile and web applications, that are build on the .NET platform.
Work	Latest project I have been working on was accounting package made for Dutch market. It was build using C#, DevExpress controls, WCF, MS SQL, ClickOnce and was deployed on Windows Azure platform. Other one is Android app made in Basic 4 android. No. of concurrent projects : 3.
Age	34
Experience	15 years programming experience in the following languages: C/C++, Delphi, C#, Java, ML, Prolog, Lisp, CUDA, OpenCL. Parallel Computing, GPGPU, Machine Learning.
Work	I usually work on several projects at once. Either internal or for clients.
Age	19
Experience	I started programming with Visual Basic when I was around 14. Since then, I've used C# and C++ and C#. C# being my primary language.
Work	Normally, I work on one or two projects at a time. Unless I am freelancing, then it could be quite more.
Age	33
Experience	6 years experience in C# and ASP.net.

Work	I usually work on several projects at once.
Age	27
Experience	4 Years experience
32 Work	Generally Web project (php or asp.net) Now i'm working on three different projects on freelance + study projects every month.
Age	24
Experience	I have 3 years experience in the IT field, I prefer the object-oriented programming, the MVC concept and News Latest Technologies of Distributed Systems.
Work	I am a student, so at this moment we have a lot of project to do.
Age	30
Experience	I have more than 7 yearsÕ work experience in software development and project management. I am focus on .Net programming, I like using .net skills to build high user experience and performance applications.
Work	
Age	24
Experience	I started with Windows Forms application and worked on several projects with it for 1 year after which I shifted my focus on the Web and started with ASP.NET WebForm. Later found ASP.NET MVC to be more interesting and since then have been working with ASP.NET MVC and front-end application Javascript frameworks for 1.5 years now.
Work	I work as a .NET Developer so most of my work is based on .NET and for the past year all on Web e.g. Web Application (ASP.NET MVC) or API (WebAPI). In my spare time I do freelancing so I might have to work on at least 2-3 projects concurrently.

Table 40: Programmers A–H Baseline reminders.

Owner	Day Created	Completion (Days)
Completed		
A	4	0.04
A	7	0.50
A	8	0.65
B	7	0.74
D	2	1.97
E	5	2.27
C	3	2.59
B	5	2.99
F	4	3.27
F	2	4.52
F	4	4.61
G	3	4.83
D	3	5.07
G	3	5.36
G	1	5.46
G	2	6.16
B	2	6.82
G	2	7.51
G	1	8.16
H	1	8.25
H	1	8.59
H	1	8.59
H	1	8.94
H	1	8.95

	Incomplete
A	1
A	5
A	4
A	5
B	6
B	2
B	3
B	7
B	8
C	5
C	3
D	7
D	3
D	1
E	4
E	6
F	1
F	4
F	3
F	4
F	3
G	5
G	8
G	1
H	8
H	6
H	7

H	1
H	5
H	5

Table 34: Programmers A–K number of recall events as for tasks in free recall condition.

Free Recall	A	B	C	D	E	F	G	H	I	J	K
Code-Oriented											
Add Feature	7	2	3	3	2	2	2	7	3	4	2
Remove Feature	0	1	1	0	0	1	0	3	2	1	0
Mechanics	0	0	0	0	0	0	0	0	0	0	0
Refinement	0	0	0	0	0	0	0	0	0	0	0
Restructing	0	0	0	0	0	0	0	0	1	0	0
Total	7	3	4	3	3	3	2	10	5	5	2
Domain-Oriented											
Logic	3	4	1	4	0	2	1	5	3	3	1
Rationale	0	2	0	0	0	1	0	0	1	0	0
Purpose/Role	2	1	1	3	2	1	1	3	2	2	2
Program Behavior	0	0	0	1	0	1	1	2	2	1	3
Program/Data Flow	2	0	0	0	0	0	0	0	0	0	1
Data	1	0	0	1	0	0	0	0	0	0	0
Total	8	7	2	9	2	5	3	10	8	6	7
Narrative-Oriented											
Context of Work	1	0	0	1	1	1	0	0	1	0	1
Transition	0	0	0	0	0	0	0	0	0	0	0
Sterotype/Pattern	0	0	0	0	0	0	0	0	0	0	0
Milestone	0	0	0	0	0	0	0	0	0	0	0
Side Note	0	0	0	0	0	0	0	0	0	0	0
Incomplete Work	0	0	0	0	0	0	0	0	0	0	0
Total	1	0	0	1	1	1	0	0	1	0	1
Problem-Oriented											
Problem	0	0	1	1	2	1	0	1	0	2	1
Approach	1	0	0	0	0	0	0	0	0	0	0
Resolution	0	0	1	1	2	1	0	0	0	2	1
Mistake	0	0	0	0	0	0	0	0	0	0	0
Incorrect Code	0	0	0	0	0	0	0	0	0	0	0
Backtracking	0	1	0	0	0	0	0	0	0	0	0
Experimenting	0	0	0	0	0	0	0	0	0	0	0
Total	1	1	2	2	4	2	0	1	0	4	2

Table 35: Programmers A–K number of recall events as for tasks in memlet condition.

MEMLET	A	B	C	D	E	F	G	H	I	J	K
CODE-ORIENTED											
Add Feature	4	2	2	2	1	2	2	3	2	1	1
Remove Feature	0	1	1	0	0	0	0	1	0	0	0
Mechanics	0	0	0	4	0	1	3	1	1	0	0
Refinement	0	0	0	2	1	0	0	2	0	0	0
Restructing	3	0	0	3	2	1	7	9	4	0	0
Total	7	3	3	11	4	4	12	16	16	1	1
DOMAIN-ORIENTED											
Logic	3	4	2	6	2	1	1	1	1	3	0
Rationale	3	3	1	1	1	1	1	2	0	1	0
Purpose/Role	2	1	1	1	1	1	2	1	1	2	0
Program Behavior	0	0	0	1	1	1	2	1	1	0	0
Program/Data Flow	2	0	1	3	2	0	0	2	1	0	0
Data	1	0	0	6	1	0	1	1	0	0	0
Total	11	8	5	18	8	4	7	8	8	6	0
NARRATIVE-ORIENTED											
Context of Work	1	1	1	0	1	1	1	2	1	1	1
Transition	2	0	1	0	0	2	2	0	0	0	2
Sterotype/Pattern	0	0	1	4	1	0	3	3	2	0	0
Milestone	1	0	0	0	0	0	2	1	0	1	0
Side Note	0	0	0	0	0	1	0	5	3	0	0
Incomplete Work	0	0	0	0	1	0	1	0	0	0	0
Total	4	1	3	4	3	4	9	11	11	2	3
PROBLEM-ORIENTED											
Problem	1	1	0	0	1	0	2	1	0	1	0
Approach	1	1	0	0	1	0	1	2	0	1	0
Resolution	0	1	0	0	1	0	2	1	0	1	0
Mistake	0	0	2	2	1	0	1	1	1	0	1
Incorrect Code	0	0	0	0	0	0	0	0	0	0	1
Backtracking	2	1	0	0	1	1	2	2	1	0	0
Experimenting	3	3	0	0	1	0	3	0	1	0	0
Total	7	7	2	2	6	1	11	7	7	3	2

Table 37: Programmers A–H Attach Here reminders.

Owner	Day Created	Completion (Days)	Normalized Exposure
Completed			
G	1	0.03	LOW
E	8	0.04	MED
G	4	0.08	MED
H	4	0.09	MED
F	2	0.10	MED
H	5	0.36	MED
C	6	0.42	MED
H	7	0.54	HIGH
A	4	0.56	HIGH
G	3	0.56	HIGH
C	4	0.56	HIGH
G	2	0.56	HIGH
A	2	0.78	HIGH
G	4	0.79	HIGH
D	5	0.86	HIGH
G	6	1.11	HIGH
F	7	1.45	HIGH
H	3	1.69	HIGH
G	5	2.75	HIGH
B	5	3.20	HIGH
G	1	8.77	HIGH
C	1	8.94	HIGH
Incomplete			
A	1		LOW
D	2		LOW
E	2		LOW
E	3		LOW
E	3		LOW
H	4		LOW
B	4		LOW
H	5		MED
G	5		MED
H	5		HIGH
G	6		HIGH
H	7		HIGH
H	8		HIGH

Table 38: Programmers A–H Due By reminders.

Owner	Day Created	Completion (Days)	Due Day	Finish Day	Due Offset
Completed					
A	3	0.04	3	3	0
A	8	0.10	8	8	0
A	6	0.99	7	6	-1
A	5	1.05	5	6	1
C	3	1.73	4	4	0
A	2	2.94	5	4	-1
A	2	3.67	5	5	0
A	2	4.47	7	6	-1
B	3	4.53	6	7	1
B	1	4.81	4	5	1
G	3	5.26	5	8	3
H	2	5.56	7	7	0
B	2	6.82	3	8	5
G	1	8.04	9	9	-1
G	1	8.05	8	9	1
H	1	8.13	2	9	7
G	1	8.49	1	9	8
Incomplete					
C	4		5		
B	6		8		
H	8		10		
G	8		11		
G	8		9		

Table 39: Programmers A–H Attach Everywhere reminders.

Owner	Day Created	Completion (Days)
Completed		
F	4	0.09
F	7	0.09
A	4	0.43
B	1	0.53
D	8	0.71
D	7	1.20
B	2	1.21
D	2	1.30
B	3	1.60
D	4	1.76
D	5	1.76
B	6	2.61
D	6	2.95
E	5	3.02
F	5	3.19
Incomplete		
D	3	
B	6	
B	8	
B	8	

REFERENCES

[1] ABRAHAM, W. C., "How long will long-term potentiation last?," *Philosophical transactions of the Royal Society of London. Series B, Biological sciences*, vol. 358, pp. 735–744, April 2003.

[2] ABRAHAM, W. C., LOGAN, B., GREENWOOD, J. M., and DRAGUNOW, M., "Induction and Experience-Dependent Consolidation of Stable Long-Term Potentiation Lasting Months in the Hippocampus," *J. Neurosci.*, vol. 22, no. 21, pp. 9626–9634, 2002.

[3] ADAMCZYK, P. D. and BAILEY, B. P., "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*, (New York, NY, USA), pp. 271–278, ACM, 2004.

[4] ALTMANN, E. M. and TRAFTON, J. G., "Memory for goals: an activation-based model," *Cognitive Science*, vol. 26, pp. 39–83, 2002.

[5] ALTMANN, E. M. and TRAFTON, J. G., "Task interruption: Resumption lag and the role of cues," in *Proceedings of the 26th annual conference of the Cognitive Science Society*, 2004.

[6] ALTMANN, E. M. and TRAFTON, J. G., "Timecourse of recovery from task interruption: Data and a model," *Psychonomic Bulletin and Review*, vol. 14, pp. 1079–1084, 2007.

[7] ANDERSON, J. R., BYRNE, M. D., DOUGLASS, S., LEBIERE, C., and QIN, Y., "An integrated theory of the mind," *Psychological Review*, vol. 111, no. 4, pp. 1036–1050, 2004.

[8] ANDERSON, J. R. and LEBIERE, C., *The Atomic Components of Thought*. Mahwah, NJ: Erlbaum, June 1998.

[9] ATKINSON, R. C. and SHIFFRIN, R. M., *The psychology of learning and motivation (Volume 2)*, ch. Human memory: A proposed system and its control processes, pp. 89–195. Academic Press, 1968.

[10] BADDELEY, A. and HITCH, G., *The psychology of learning and motivation: Advances in research and theory*, ch. Working memory, pp. 47–89. New York: Academic Press, 1974.

[11] BADDELEY, A., "Working memory: looking back and looking forward.," *Nature reviews. Neuroscience*, vol. 4, pp. 829–839, October 2003.

[12] BADRE, D., "Cognitive control, hierarchy, and the rostro-caudal organization of the frontal lobes.," *Trends in cognitive sciences*, vol. 12, pp. 193–200, May 2008.

[13] BANICH, M. T., "The missing link: the role of interhemispheric interaction in attentional processing.," *Brain and cognition*, vol. 36, pp. 128–157, Mar. 1998.

[14] BANNON, L., CYPHER, A., GREENSPAN, S., and MONTY, M. L., "Evaluation and analysis of users' activity organization," in *CHI '83: Proceedings of the SIGCHI conference on Human Factors in Computing Systems*, (New York, NY, USA), pp. 54–57, ACM, 1983.

[15] BANNON, L., CYPHER, A., GREENSPAN, S., and MONTY, M. L., "Evaluation and analysis of users' activity organization," in *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '83, (New York, NY, USA), pp. 54–57, ACM, 1983.

[16] BAR, M. and AMINOFF, E., "Cortical analysis of visual context.," *Neuron*, vol. 38, pp. 347–358, April 2003.

[17] BARNES, J. M. and UNDERWOOD, B. J., "Fate of first-list associations in transfer theory.," *Journal of experimental psychology*, vol. 58, pp. 97–105, Aug. 1959.

[18] BARRY, D. and STANIENDA, T., "Solving the java object storage problem," *Computer*, vol. 31, pp. 33–40, 1998.

[19] BEGEL, A., KHOO, Y. P., and ZIMMERMANN, T., "Codebook: discovering and exploiting relationships in software repositories," in *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 1*, ICSE '10, (New York, NY, USA), pp. 125–134, ACM, 2010.

[20] BELLOTTI, V., DALAL, B., GOOD, N., FLYNN, P., BOBROW, D. G., and DUCHENEAUT, N., "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*, (New York, NY, USA), pp. 735–742, ACM, 2004.

[21] BENAQUISTO, L., "Open coding," *The Sage encyclopedia of qualitative research methods*, vol. 2, pp. 581–582, 2008.

[22] BEYER, H. R. and HOLTZBLATT, K., "Apprenticing with the customer," *Commun. ACM*, vol. 38, pp. 45–52, May 1995.

[23] BIEHL, J. T., CZERWINSKI, M., SMITH, G., and ROBERTSON, G. G., "Fastdash: a visual dashboard for fostering awareness in software teams," in *CHI '07: Proceedings of the SIGCHI conference on Human factors in computing systems*, (New York, NY, USA), pp. 1313–1322, ACM, 2007.

[24] BLISS, T. V. and LOMO, T., "Long-lasting potentiation of synaptic transmission in the dentate area of the anaesthetized rabbit following stimulation of the perforant path.," *The Journal of physiology*, vol. 232, pp. 331–356, July 1973.

[25] BRAGDON, A., REISS, S. P., ZELEZNIK, R., KARUMURI, S., CHEUNG, W., KAPLAN, J., COLEMAN, C., ADEPUTRA, F., and LAVIOLA, JR., J. J., "Code bubbles: rethinking the user interface paradigm of integrated development environments," in *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 1*, ICSE '10, (New York, NY, USA), pp. 455–464, ACM, 2010.

[26] BRAGDON, A., ZELEZNIK, R., REISS, S. P., KARUMURI, S., CHEUNG, W., KAPLAN, J., COLEMAN, C., ADEPUTRA, F., and LAVIOLA, JR., J. J., "Code bubbles: a working set-based interface for code understanding and maintenance," in *Proceedings of the 28th international conference on Human factors in computing systems*, CHI '10, (New York, NY, USA), pp. 2503–2512, ACM, 2010.

[27] BRENNAN, A., CHUGH, J. S., and KLINE, T., "Traditional versus open office design a longitudinal field study," *Environment and Behavior*, vol. 34, no. 3, pp. 279–299, 2002.

[28] BROOKS, R., "Towards a theory of the comprehension of computer programs," *International Journal of Man-Machine Studies*, vol. 18, pp. 543–554, 1983.

[29] BROYD, S. J., DEMANUELE, C., DEBENER, S., HELPS, S. K., JAMES, C. J., and SONUGA-BARKE, E. J., "Default-mode brain dysfunction in mental disorders: A systematic review," *Neuroscience & Biobehavioral Reviews*, vol. 33, no. 3, pp. 279–296, 2009.

[30] BRUSH, A. B., MEYERS, B. R., TAN, D. S., and CZERWINSKI, M., "Understanding memory triggers for task tracking," in *CHI '07: Proceedings of the SIGCHI conference on Human factors in computing systems*, (New York, USA), pp. 947–950, ACM, 2007.

[31] BURGER, J. L., PARKER, K., CASON, L., HAUCK, S., KAETZEL, D., O'NAN, C., and WHITE, A., "Responses to work complexity: the novice to expert effect.," *Western journal of nursing research*, vol. 32, pp. 497–510, June 2010.

[32] BY GUY M. WHIPPLE), M. O. T., *Mental Fatigue*. Warwick & York, 1911.

[33] CANFORA, G., CERULO, L., and DI PENTA, M., "Ldiff: An enhanced line differencing tool," in *Proceedings of the 31st International Conference on Software Engineering*, ICSE '09, (Washington, DC, USA), pp. 595–598, IEEE Computer Society, 2009.

[34] CANFORA, G., CERULO, L., and PENTA, M. D., "Identifying changed source code lines from version repositories," in *Proceedings of the Fourth International Workshop on Mining Software Repositories*, MSR '07, (Washington, DC, USA), pp. 14–, IEEE Computer Society, 2007.

[35] CHARNESS, N., "Memory for chess positions: resistance to interference," *Journal of Experimental Psychology: Human Learning and Memory*, vol. 2, pp. 641–653, 1976.

[36] CHARRON, S. and KOECHLIN, E., "Divided representation of concurrent goals in the human frontal lobes.," *Science (New York, N.Y.)*, vol. 328, pp. 360–363, Apr. 2010.

[37] CHASE, W. G. and ERICSSON, K. A., *The psychology of learning and motivation*, vol. 16, ch. Skill and working memory, pp. 1–58. New York: Academic Press, 1982.

[38] CHASE, W. and SIMON, H., "Perception in chess," *Cognitive Psychology*, vol. 4, pp. 55–81, 1973.

[39] CHERUBINI, M., VENOLIA, G., DELINE, R., and KO, A. J., "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*, (New York, NY, USA), pp. 557–566, ACM, 2007.

[40] CHI, M. T. H., GLASER, R., and REES, E., *Expertise in problem solving*, vol. 1, pp. 7–75. Erlbaum, 1982.

[41] CORBI, T. A., "Program understanding: Challenge for the 1990s," *IBM Systems Journal*, vol. 28, no. 2, pp. 294 –306, 1989.

[42] CORKIN, S., AMARAL, D. G., GONZÁLEZ, R. G., JOHNSON, K. A., and HYMAN, B. T., "H. m.'s medial temporal lobe lesion: findings from magnetic resonance imaging.," *The Journal of neuroscience : the official journal of the Society for Neuroscience*, vol. 17, pp. 3964–3979, May 1997.

[43] COWAN, N., "The magical number 4 in short-term memory: a reconsideration of mental storage capacity.," *The Behavioral and brain sciences*, vol. 24, February 2001.

[44] CUTRELL, E., CZERWINSKI, M., and HORVITZ, E., "Notification, disruption and memory: Effects of messaging interruptions on memory and performance," in *Proceedings of Interact 2001*, 2001.

[45] CUTRELL, E. B., CZERWINSKI, M., and HORVITZ, E., "Effects of instant messaging interruptions on computing tasks," in *CHI '00 Extended Abstracts on Human Factors in Computing Systems*, CHI EA '00, (New York, NY, USA), pp. 99–100, ACM, 2000.

[46] CZERWINSKI, M., HORVITZ, E., and WILHITE, S., "A diary study of task switching and interruptions," in *CHI '04: Proceedings of the SIGCHI conference on Human factors in computing systems*, (New York, NY, USA), pp. 175–182, ACM Press, 2004.

[47] DABBISH, L., MARK, G., and GONZÁLEZ, V. M., "Why do i keep interrupting myself?: environment, habit and self-interruption," in *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, pp. 3127–3130, ACM, 2011.

[48] DE TOLNAY, C., *Michelangelo, Vol. IV: The Tomb of Julius II*. Princeton University Press, 1945.

[49] DEKEL, U., "Designing a prosthetic memory to support software developers," in *ICSE Companion '08: Companion of the 30th international conference on Software engineering*, (New York, NY, USA), pp. 1011–1014, ACM, 2008.

[50] DELINE, R., CZERWINSKI, M., MEYERS, B., VENOLIA, G., DRUCKER, S., and ROBERTSON, G., "Code thumbnails: Using spatial memory to navigate source code," in *VLHCC '06: Proceedings of the Visual Languages and Human-Centric Computing*, (Washington, DC, USA), pp. 11–18, IEEE Computer Society, 2006.

[51] DELINE, R., CZERWINSKI, M., and ROBERTSON, G., "Easing program comprehension by sharing navigation data," in *VLHCC '05: Proceedings of the 2005 IEEE Symposium on Visual Languages and Human-Centric Computing*, (Washington, DC, USA), pp. 241–248, IEEE Computer Society, 2005.

[52] DELINE, R. and ROWAN, K., "Code canvas: zooming towards better development environments," in *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 2*, ICSE '10, (New York, NY, USA), pp. 207–210, ACM, 2010.

[53] DEMARCO, T. and LISTER, T., *Peopleware: Productive Projects and Teams (Second Edition)*. Dorset House Publishing Company, Incorporated, 2nd ed., February 1999.

[54] ELDRIDGE, L. L., KNOWLTON, B. J., FURMANSKI, C. S., BOOKHEIMER, S. Y., and ENGEL, S. A., "Remembering episodes: a selective role for the hippocampus during retrieval.," *Nature neuroscience*, vol. 3, pp. 1149–1152, November 2000.

[55] ELLIS, J. B., WAHID, S., DANIS, C., and KELLOGG, W. A., "Task and social visualization in software development: evaluation of a prototype," in *CHI '07: Proceedings of the SIGCHI conference on Human factors in computing systems*, (New York, NY, USA), pp. 577–586, ACM, 2007.

[56] ERICSSON, K. A. and DELANEY, P. F., *Models of Working Memory: Mechanisms of Active Maintenance and Executive Control*, ch. Long-term working memory as an alternative to capacity models of working memory in everyday skilled performance, pp. 257–297. Cambridge, UK: Cambridge University Press, 1999.

[57] ERICSSON, K. A. and KINTSCH, W., "Long-term working memory," *Psychological Review*, vol. 102, no. 2, pp. 211–245, 1995.

[58] ERICSSON, K. A. and STASZEWSKI, J. J., *Complex information processing: The impact of Herbert A. Simon*, ch. Skilled memory and expertise: Mechanisms of exceptional performance, pp. 235–267. Hillsdale, NJ: Lawrence Erlbaum, 1989.

[59] ERICSSON, K. A., *The Cambridge handbook of expertise and expert performance*. Cambridge University Press, 2006.

[60] FARION, C. and PURVER, M., "Did you pack your keys?: Smart objects and forgetfulness," in *CHI '14 Extended Abstracts on Human Factors in Computing Systems*, CHI EA '14, (New York, NY, USA), pp. 539–542, ACM, 2014.

[61] FIELDING, R. T., *Architectural styles and the design of network-based software architectures*. PhD thesis, 2000. AAI9980887.

[62] FLEISCHMANN, A., HVALBY, O., JENSEN, V., STREKALOVA, T., ZACHER, C., LAYER, L. E., KVELLO, A., RESCHKE, M., SPANAGEL, R., SPRENGEL, R., WAGNER, E. F., and GASS, P., "Impaired long-term memory and NR2A-type NMDA receptor-dependent synaptic plasticity in mice lacking c-Fos in the CNS.," *The Journal of neuroscience : the official journal of the Society for Neuroscience*, vol. 23, pp. 9116–9122, Oct. 2003.

[63] FLEMING, S., KRAEMER, E., STIREWALT, R., DILLON, L., and XIE, S., "Refining existing theories of program comprehension during maintenance for concurrent software," pp. 23 –32, June 2008.

[64] FOGARTY, J., KO, A. J., AUNG, H. H., GOLDEN, E., TANG, K. P., and HUDSON, S. E., "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*, (New York, NY, USA), pp. 331–340, ACM, 2005.

[65] FRITZ, T., BEGEL, A., MÜLLER, S. C., YIGIT-ELLIOTT, S., and ZÜGER, M., "Using psycho-physiological measures to assess task difficulty in software development," in *Proceedings of the 36th International Conference on Software Engineering*, ICSE 2014, (New York, NY, USA), pp. 402–413, ACM, 2014.

[66] FRITZ, T., OU, J., MURPHY, G. C., and MURPHY-HILL, E., "A degree-of-knowledge model to capture source code familiarity," in *Proceedings of the 32Nd ACM/IEEE International Conference on Software Engineering - Volume 1*, ICSE '10, (New York, NY, USA), pp. 385–394, ACM, 2010.

[67] FUSTER, J. M., "The prefrontal cortex–an update: time is of the essence.," *Neuron*, vol. 30, pp. 319–333, May 2001.

[68] FUSTER, J. M. and ALEXANDER, G. E., "Neuron activity related to short-term memory.," *Science (New York, N.Y.)*, vol. 173, pp. 652–654, August 1971.

[69] FUSTER, J. and JERVEY, J., "Neuronal firing in the inferotemporal cortex of the monkey in a visual memory task," *J. Neurosci.*, vol. 2, no. 3, pp. 361–375, 1982.

[70] GAZZANIGA, M. S., IVRY, R. B., and MANGUN, G. R., *Cognitive neuroscience: the biology of the mind*. Norton, 3rd ed., 2009.

[71] GE, X. and MURPHY-HILL, E., "Reconciling manual and automatic refactoring," in *ICSE '12: Proceedings of the 34th International Conference on Software Engineering*, (Washington, DC, USA), p. to appear, IEEE Computer Society, 2012.

[72] GILLIE, T. and BROADBENT, D., "What makes interruptions disruptive? a study of length, similiarity, and complexity," *Psychological Research*, vol. 50, pp. 243–250, 1989.

[73] GLISKY, E. L., POLSTER, M. R., and ROUTHIEAUX, B. C., "Double dissociation between item and source memory," *Neuropsychology*, vol. 9, pp. 229–235, 1995.

[74] GODFREY, M. W. and ZOU, L., "Using origin analysis to detect merging and splitting of source code entities," *IEEE Trans. Softw. Eng.*, vol. 31, pp. 166–181, Feb. 2005.

[75] GREICIUS, M. D. and MENON, V., "Default-mode activity during a passive sensory task: uncoupled from deactivation but impacting activation.," *Journal of cognitive neuroscience*, vol. 16, pp. 1484–1492, nov 2004.

[76] GUZZI, A., PINZGER, M., and VAN DEURSEN, A., "Combining micro-blogging and ide interactions to support developers in their quests," in *Software Maintenance (ICSM), 2010 IEEE International Conference on*, pp. 1 –5, sept. 2010.

[77] HALVERSON, C. A., ELLIS, J. B., DANIS, C., and KELLOGG, W. A., "Designing task visualizations to support the coordination of work in software development," in *CSCW '06: Proceedings of the 2006 20th anniversary conference on Computer supported cooperative work*, (New York, NY, USA), pp. 39–48, ACM, 2006.

[78] HANNAY, J. E., DYBÅ, T., ARISHOLM, E., and SJØBERG, D. I. K., "The effectiveness of pair programming: A meta-analysis," *Inf. Softw. Technol.*, vol. 51, pp. 1110–1122, July 2009.

[79] HART, S. G. and STAVENLAND, L. E., "Development of NASA-TLX (Task Load Index): Results of empirical and theoretical research," in *Human Mental Workload* (HANCOCK, P. A. and MESHKATI, N., eds.), ch. 7, pp. 139–183, Elsevier, 1988.

[80] HATFIELD, B. D., HAUFLER, A. J., HUNG, T.-M. M., and SPALDING, T. W., "Electroencephalographic studies of skilled psychomotor performance.," *Journal of clinical neurophysiology*, vol. 21, no. 3, pp. 144–156, 2004.

[81] HATTORI, L., D'AMBROS, M., LANZA, M., and LUNGU, M., "Software evolution comprehension: Replay to the rescue," in *Program Comprehension (ICPC), 2011 IEEE 19th International Conference on*, pp. 161 –170, june 2011.

[82] HATTORI, L., LUNGU, M., and LANZA, M., "Replaying past changes in multi-developer projects," in *Proceedings of the Joint ERCIM Workshop on Software Evolution (EVOL) and International Workshop on Principles of Software Evolution (IWPSE)*, IWPSE-EVOL '10, (New York, NY, USA), pp. 13–22, ACM, 2010.

[83] HEBB, D., *The organization of behavior*. New York: Wiley, 1949.

[84] HICKOK, G., OKADA, K., and SERENCES, J. T., "Area Spt in the Human Planum Temporale Supports Sensory-Motor Integration for Speech Processing," *Journal of Neurophysiology*, vol. 101, pp. 2725–2732, May 2009.

[85] HODGES, S., BERRY, E., and WOOD, K., "SenseCam: A wearable camera that stimulates and rehabilitates autobiographical memory.," *Memory (Hove, England)*, vol. 19, pp. 685–696, Oct. 2011.

[86] HODGETTS, H. M. and JONES, D. M., "Contextual cues aid recovery from interrruption: The role of associative activitation," *Journal of Experimental Psychology: Learning, Memory, and Cognition*, vol. 32, no. 5, pp. 1120–1132, 2006.

[87] HUTCHINS, E., *Cognition in the wild*. MIT Press, Feb. 1995.

[88] IQBAL, S. T. and BAILEY, B. P., "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*, (New York, NY, USA), pp. 1489–1492, ACM, 2005.

[89] IQBAL, S. T. and BAILEY, B. P., "Leveraging characteristics of task structure to predict the cost of interruption," in *Proceedings of the SIGCHI conference on Human Factors in computing systems*, CHI '06, (New York, NY, USA), pp. 741–750, ACM, 2006.

[90] IQBAL, S. T. and HORVITZ, E., "Disruption and recovery of computing tasks: field study, analysis, and directions," in *CHI '07: Proceedings of the SIGCHI conference on Human factors in computing systems*, (New York, NY, USA), pp. 677–686, ACM, 2007.

[91] JACCARD, P., "Étude comparative de la distribution florale dans une portion des Alpes et des Jura," *Bulletin del la Société Vaudoise des Sciences Naturelles*, vol. 37, pp. 547–579, 1901.

[92] JERSILD, A. T., "Mental set and shift.," *Archives of psychology*, 1927.

[93] JOHN, M. S., SMALLMAN, H. S., and MANES, D. I., "Recovery from interruptions to a dynamic monitoring task: the beguiling utility of instant replay," in *Proceedings of the Human Factors and Ergonomics Society 49th Annual Meeting*, 2005.

[94] JONES, D. and FLEMING, S., "What use is a backseat driver? a qualitative investigation of pair programming," in *Visual Languages and Human-Centric Computing (VL/HCC), 2013 IEEE Symposium on*, pp. 103–110, Sept 2013.

[95] JUDD, T. and KENNEDY, G., "Measurement and evidence of computer-based task switching and multitasking by net generation students," *Computers & Education*, vol. 56, no. 3, pp. 625–631, 2011.

[96] KAHN, I., PASCUAL-LEONE, A., THEORET, H., FREGNI, F., CLARK, D., and WAGNER, A., "Transient disruption of ventrolateral prefrontal cortex during verbal encoding affects subsequent memory performance.," *Journal of neurophysiology*, vol. 94, pp. 688–698, July 2005.

[97] KEOGH, E., CHU, S., HART, D., and PAZZANI, M., "Segmenting Time Series: A Survey and Novel Approach," in *In an Edited Volume, Data mining in Time Series Databases. Published by World Scientific*, pp. 1–22, 1993.

[98] KERSTEN, M. and MURPHY, G. C., "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*, (New York, NY, USA), pp. 1–11, ACM, 2006.

[99] KIM, M., ZIMMERMANN, T., and NAGAPPAN, N., "A field study of refactoring challenges and benefits," in *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*, FSE '12, (New York, NY, USA), pp. 50:1–50:11, ACM, 2012.

[100] KIM, S.-Y., "The Effects of Storytelling and Pretend Play on Cognitive Processes, Short-Term and Long-Term Narrative Recall.," *Child Study Journal*, vol. 29, no. 3, pp. 175–91, 1999.

[101] KLINGNER, J., *Measuring Cognitive Load During Visual Tasks by Combining Pupillometry and Eye Tracking*. Ph.d. dissertation, Stanford University, Department of Computer Science, May 2010.

[102] KO, A. J., COBLENZ, M. J., and AUNG, H. H., "An exploratory study of how developers seek, relate, and collect relevant information during software maintenance tasks," *IEEE Trans. Softw. Eng.*, vol. 32, no. 12, pp. 971–987, 2006. Senior Member-Myers, Brad A.

[103] KO, A. J., DELINE, R., and VENOLIA, G., "Information needs in collocated software development teams," in *ICSE '07: Proceedings of the 29th international conference on Software Engineering*, (Washington, DC, USA), pp. 344–353, IEEE Computer Society, 2007.

[104] KONDO, K., MARUISHI, M., UENO, H., SAWADA, K., HASHIMOTO, Y., OHSHITA, T., TAKAHASHI, T., OHTSUKI, T., and MATSUMOTO, M., "The pathophysiology of prospective memory failure after diffuse axonal injury - lesion-symptom analysis using diffusion tensor imaging," *BMC Neuroscience*, vol. 11, pp. 147–157, November 2010.

[105] LAWRANCE, J., BURNETT, M., BELLAMY, R., BOGART, C., and SWART, C., "Reactive information foraging for evolving goals," in *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '10, (New York, NY, USA), pp. 25–34, ACM, 2010.

[106] LEBEDEV, M. A., MESSINGER, A., KRALIK, J. D., and WISE, S. P., "Representation of attended versus remembered locations in prefrontal cortex.," *PLoS biology*, vol. 2, Nov. 2004.

[107] LETHBRIDGE, T. C., SIM, S. E., and SINGER, J., "Studying software engineers: Data collection techniques for software field studies," *Empirical Softw. Eng.*, vol. 10, pp. 311–341, July 2005.

[108] LEWIS-PEACOCK, J. A. and POSTLE, B. R., "Temporary Activation of Long-Term Memory Supports Working Memory," *J. Neurosci.*, vol. 28, pp. 8765–8771, August 2008.

[109] LITTMAN, D. C., PINTO, J., LETOVSKY, S., and SOLOWAY, E., "Mental models and software maintenance," in *Papers presented at the first workshop on empirical studies of programmers on Empirical studies of programmers*, (Norwood, NJ, USA), pp. 80–98, Ablex Publishing Corp., 1986.

[110] LITTMAN, D. C., PINTO, J., LETOVSKY, S., and SOLOWAY, E., "Mental models and software maintenance," in *Papers presented at the first workshop on empirical studies of programmers on Empirical studies of programmers*, (Norwood, NJ, USA), pp. 80–98, Ablex Publishing Corp., 1986.

[111] LYNCH, M. A., "Long-term potentiation and memory.," *Physiological reviews*, vol. 84, pp. 87–136, January 2004.

[112] MAALEJ, W., "Task-first or context-first? tool integration revisited," in *Proceedings of the 2009 IEEE/ACM International Conference on Automated Software Engineering*, ASE '09, (Washington, DC, USA), pp. 344–355, IEEE Computer Society, 2009.

[113] MAGUIRE, E. A., GADIAN, D. G., JOHNSRUDE, I. S., GOOD, C. D., ASHBURNER, J., FRACKOWIAK, R. S., and FRITH, C. D., "Navigation-related structural change in the hippocampi of taxi drivers.," *Proceedings of the National Academy of Sciences of the United States of America*, vol. 97, pp. 4398–4403, April 2000.

[114] MARK, G., GONZALEZ, V. M., and HARRIS, J., "No task left behind? Examining the nature of fragmented work," in *CHI '05: Proceedings of the SIGCHI conference on Human factors in computing systems*, (New York, NY, USA), pp. 321–330, ACM Press, 2005.

[115] MAYES, A., MONTALDI, D., and MIGO, E., "Associative memory and the medial temporal lobes.," *Trends in cognitive sciences*, vol. 11, pp. 126–135, March 2007.

[116] MCDANIEL, M. A. and EINSTEIN, G. O., "Strategic and automatic processes in prospective memory retrieval: A multiprocess framework," *Applied Cognitive Psychology*, vol. 14, pp. 127–144, 2000.

[117] MCFARLANE, D., "Comparison of four primary methods for coordinating the interruption of people in human-computer interaction," *Hum.-Comput. Interact.*, vol. 17, no. 1, pp. 63–139, 2002.

[118] MCGEE-LENNON, M. R., WOLTERS, M. K., and BREWSTER, S., "User-centred multimodal reminders for assistive living," in *Proceedings of the 2011 annual conference on Human factors in computing systems*, CHI '11, (New York, NY, USA), pp. 2105–2114, ACM, 2011.

[119] MCGETTIGAN, C., WARREN, J. E., EISNER, F., MARSHALL, C. R., SHANMUGALINGAM, P., and SCOTT, S. K., "Neural correlates of sublexical processing in phonological working memory.," *Journal of cognitive neuroscience*, vol. 23, pp. 961–977, April 2011.

[120] MCKINNEY, F., "Studies in the retention of interrupted learning activities," *Journal of Comparative Psychology*, vol. 19 (2), pp. 265–296, 1935.

[121] MERRIAM-WEBSTER ONLINE, "Merriam-Webster Online Dictionary," 2009.

[122] MILLER, E. K. and COHEN, J. D., "An integrative theory of prefrontal cortex function.," *Annual review of neuroscience*, vol. 24, no. 1, pp. 167–202, 2001.

[123] MILLER, E. and BUSCHMAN, T., *The Neuroscience of Rule-Guided Behavior*, ch. Rules through recursion: How interactions between the frontal cortex and basal ganglia may build abstract, complex, rules from concrete, simple, ones, p. (in press). Oxford University Press., 2007.

[124] MILLER, G. A., "The magical number seven, plus or minus two: some limits on our capacity for processing information. 1956.," *Psychological review*, vol. 101, pp. 343–352, April 1994.

[125] MILLER, G., GALANTER, E., and PRIBRAM, K., *Plans and the Structure of Behavior*. New York: Holt, Rinehart & Winston, 1960.

[126] MILNER, B., CORSI, P., and LEONARD, G., "Frontal-lobe contribution to recency judgements.," *Neuropsychologia*, vol. 29, no. 6, pp. 601–618, 1991.

[127] MILTON, J., SOLODKIN, A., HLUSTÍK, P., and SMALL, S. L., "The mind of expert motor performance is cool and focused.," *Neuroimage*, vol. 35, pp. 804–813, Apr. 2007.

[128] MIYATA, Y. and NORMAN, D. A., "Psychological issues in support of multiple activities," in *User Centered System Design: New Perspectives on Human-Computer Interaction* (NORMAN, D. A. and DRAPER, S. W., eds.), pp. 265–284, Hillsdale, NJ: Erlbaum, 1986.

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

[130] MONTALDI, D., SPENCER, T. J., ROBERTS, N., and MAYES, A. R., "The neural system that mediates familiarity memory.," *Hippocampus*, vol. 16, no. 5, pp. 504–520, 2006.

[131] MORRIS, R. G. and FREY, U., "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*, vol. 352, no. 1360, pp. 1489–1503, 1997.

[132] MORRIS, R. G. M., "Elements of a neurobiological theory of hippocampal function: the role of synaptic plasticity, synaptic tagging and schemas," *European Journal of Neuroscience*, vol. 23, no. 11, pp. 2829–2846, 2006.

[133] MURPHY, G. C., KERSTEN, M., and FINDLATER, L., "How are Java software developers using the Eclipse IDE?," vol. 23, (Los Alamitos, CA, USA), pp. 76–83, IEEE Comp. Soc. Press, 2006.

[134] MURPHY-HILL, E. and BLACK, A. P., "An interactive ambient visualization for code smells," in *Proceedings of the 5th international symposium on Software visualization*, SOFTVIS '10, (New York, NY, USA), pp. 5–14, ACM, 2010.

[135] MURPHY-HILL, E., PARNIN, C., and BLACK, A. P., "How we refactor, and how we know it," in *Proceedings of the 31st International Conference on Software Engineering*, ICSE '09, (Washington, DC, USA), pp. 287–297, IEEE Computer Society, 2009.

[136] MURPHY-HILL, E., PARNIN, C., and BLACK, A. P., "How we refactor, and how we know it," *IEEE Transactions on Software Engineering*, vol. 99, no. PrePrints, 2011.

[137] MURRAY, A. and LETHBRIDGE, T. C., "Presenting micro-theories of program comprehension in pattern form," in *IWPC '05: Proceedings of the 13th International Workshop on Program Comprehension*, (Washington, DC, USA), pp. 45–54, IEEE Computer Society, 2005.

[138] NAKAGAWA, T., KAMEI, Y., UWANO, H., MONDEN, A., MATSUMOTO, K., and GERMAN, D. M., "Quantifying programmers' mental workload during program comprehension based on cerebral blood flow measurement: A controlled experiment," in *Companion Proceedings of the 36th International Conference on Software Engineering*, ICSE Companion 2014, (New York, NY, USA), pp. 448–451, ACM, 2014.

[139] O'BRIEN, M. P. and BUCKLEY, J., "Modelling the information-seeking behaviour of programmers - an empirical approach," in *IWPC '05: Proceedings of the 13th International Workshop on Program Comprehension*, (Washington, DC, USA), pp. 125–134, IEEE Computer Society, 2005.

[140] O'CONAILL, B. and FROHLICH, D., "Timespace in the workplace: dealing with interruptions," in *CHI '95: Conference companion on Human factors in computing systems*, (New York, NY, USA), pp. 262–263, ACM Press, 1995.

[141] OLEKSIK, G., WILSON, M. L., TASHMAN, C., MENDES RODRIGUES, E., KAZAI, G., SMYTH, G., MILIC-FRAYLING, N., and JONES, R., "Lightweight tagging expands information and activity management practices," in *CHI '09: Proceedings of the 27th international conference on Human factors in computing systems*, pp. 279–288, ACM, 2009.

[142] OLIVER, N., CZERWINSKI, M., SMITH, G., and ROOMP, K., "Relalttab: assisting users in switching windows," in *IUI '08: Proceedings of the 13th international conference on Intelligent user interfaces*, (New York, NY, USA), pp. 385–388, ACM, 2008.

[143] OPPEZZO, M. and SCHWARTZ, D. L., "Give your ideas some legs: The positive effect of walking on creative thinking.," *Journal of experimental psychology. Learning, memory, and cognition*, Apr. 2014.

[144] PARK, S., KIM, M.-S. S., and CHUN, M. M., "Concurrent working memory load can facilitate selective attention: evidence for specialized load.," *Journal of experimental psychology. Human perception and performance*, vol. 33, pp. 1062–1075, Oct. 2007.

[145] PARNIN, C. and GORG, C., "Design guidelines for ambient software visualization in the workplace," in *Visualizing Software for Understanding and Analysis, 2007. VISSOFT 2007. 4th IEEE International Workshop on*, pp. 18 –25, june 2007.

[146] PARNIN, C. and GÖRG, C., "Design guidelines for ambient software visualization in the workplace," in *Visualizing Software for Understanding and Analysis, 2007. VISSOFT 2007. 4th IEEE International Workshop on*, pp. 18–25, June 2007.

[147] PARNIN, C., "A cognitive neuroscience perspective on memory for programming tasks," in *In the Proceedings of the 22nd Annual Meeting of the Psychology of Programming Interest Group (PPIG)*, 2010.

[148] PARNIN, C. and DELINE, R., "Evaluating cues for resuming interrupted programming tasks," in *Proceedings of the 28th international conference on Human factors in computing systems*, CHI '10, (New York, NY, USA), pp. 93–102, ACM, 2010.

[149] PARNIN, C. and GÖRG, C., "Building usage contexts during program comprehension," in *ICPC '06: Proceedings of the 14th IEEE International Conference on Program Comprehension*, pp. 13–22, 2006.

[150] PARNIN, C., GÖRG, C., and RUGABER, S., "Codepad: interactive spaces for maintaining concentration in programming environments," in *Proceedings of the 5th international symposium on Software visualization*, SOFTVIS '10, (New York, NY, USA), pp. 15–24, ACM, 2010.

[151] PARNIN, C. and RUGABER, S., "Resumption strategies for interrupted programming tasks," in *ICPC '09: Proceedings of the 17th IEEE International Conference on Program Comprehension*, (Washington, DC, USA), IEEE Computer Society, 2009.

[152] PARNIN, C. and RUGABER, S., "Resumption strategies for interrupted programming tasks," *Software Quality Journal*, vol. 19, pp. 5–34, 2011. 10.1007/s11219-010-9104-9.

[153] PARNIN, C. and RUGABER, S., "Programmer information needs after memory failure.," in *ICPC* (BEYER, D., VAN DEURSEN, A., and GODFREY, M. W., eds.), pp. 123–132, 2012.

[154] PARNIN, C. and TREUDE, C., "Measuring api documentation on the web," in *Proceedings of the 2nd International Workshop on Web 2.0 for Software Engineering*, Web2SE '11, (New York, NY, USA), pp. 25–30, ACM, 2011.

[155] PATRICK, G. S., BAUDISCH, P., ROBERTSON, G., CZERWINSKI, M., MEYERS, B., ROBBINS, D., and ANDREWS, D., "Groupbar: The taskbar evolved," in *Proceedings of OZCHI 2003*, pp. 34–43, 2003.

[156] PENNINGTON, N., "Stimulus structures and mental representationp in expert comprehension of computer programs," *Cognitive Psychology*, vol. 19, pp. 295–341, 1987.

[157] PETERSEN, M. G., "Interactive spaces: towards a better everyday?," *interactions*, vol. 12, pp. 44–45, July 2005.

[158] PIROLLI, P. and CARD, S. K., "Information foraging," *Psychological Review*, vol. 106, pp. 643–675, 1999.

[159] PLONER, C. J., GAYMARD, B. M., RIVAUD-PÉCHOUX, S., BAULAC, M., CLÉMENCEAU, S., SAMSON, S., and PIERROT-DESEILLIGNY, C., "Lesions affecting the parahippocampal cortex yield spatial memory deficits in humans.," *Cerebral cortex (New York, N.Y. : 1991)*, vol. 10, pp. 1211–1216, December 2000.

[160] POSTLE, B. R., "Working memory as an emergent property of the mind and brain.," *Neuroscience*, vol. 139, pp. 23–38, April 2006.

[161] POSTLE, B. R. and CORKIN, S., "Impaired word-stem completion priming but intact perceptual identification priming with novel words: evidence from the amnesic patient h.m.," *Neuropsychologia*, vol. 36, pp. 421–440, May 1998.

[162] PROVOST, J.-S., PETRIDES, M., SIMARD, F., and MONCHI, O., "Investigating the Long-Lasting residual effect of a set shift on frontostriatal activity," *Cerebral Cortex*, Dec. 2011.

[163] RATTENBURY, T. and CANNY, J., "Caad: an automatic task support system," in *CHI '07: Proceedings of the SIGCHI conference on Human factors in computing systems*, (New York, USA), pp. 687–696, ACM, 2007.

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

[165] REYNOLDS, J. R., WEST, R., and BRAVER, T., "Distinct neural circuits support transient and sustained processes in prospective memory and working memory.," *Cerebral cortex (New York, N.Y. : 1991)*, vol. 19, pp. 1208–1221, May 2009.

[166] ROBBES, R. and LANZA, M., "Characterizing and understanding development sessions," in *ICPC '07: Proceedings of the 15th IEEE International Conference on Program Comprehension*, (Washington, DC, USA), pp. 155–166, IEEE Computer Society, 2007.

[167] ROBBES, R. and LANZA, M., "Spyware: a change-aware development toolset," in *Proceedings of the 30th international conference on Software engineering*, ICSE '08, (New York, NY, USA), pp. 847–850, ACM, 2008.

[168] ROBERTSON, G., HORVITZ, E., CZERWINSKI, M., BAUDISCH, P., HUTCHINGS, D. R., MEYERS, B., ROBBINS, D., and SMITH, G., "Scalable fabric: flexible task management," in *AVI '04: Proceedings of the working conference on Advanced visual interfaces*, (New York, NY, USA), pp. 85–89, ACM, 2004.

[169] ROBILLARD, M. P. and WEIGAND-WARR, F., "Concernmapper: simple view-based separation of scattered concerns," in *Proceedings of the 2005 OOPSLA workshop on Eclipse technology eXchange*, eclipse '05, (New York, NY, USA), pp. 65–69, ACM, 2005.

[170] SAFER, I. and MURPHY, G. C., "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, ACM, 2007.

[171] SALVUCCI, D. D., TAATGEN, N. A., and BORST, J. P., "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*, (New York, NY, USA), pp. 1819–1828, ACM, 2009.

[172] SARMA, A., REDMILES, D., and VAN DER HOEK, A., "Empirical evidence of the benefits of workspace awareness in software configuration management," in *SIGSOFT '08/FSE-16: Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of software engineering*, (New York, NY, USA), pp. 113–123, ACM, 2008.

[173] SCAFFIDI, C., SHAW, M., and MYERS, B., "Estimating the numbers of end users and end user programmers," in *VLHCC '05: Proceedings of the 2005 IEEE Symposium on Visual Languages and Human-Centric Computing*, (Washington, DC, USA), pp. 207–214, IEEE Computer Society, 2005.

[174] SCOVILLE, W. B. and MILNER, B., "Loss of recent memory after bilateral hippocampal lesions. 1957.," *The Journal of neuropsychiatry and clinical neurosciences*, vol. 12, no. 1, pp. 103–113, 2000.

[175] SHNEIDERMAN, B. and MAYER, R., "Syntactic semantic interactions in programmer behavior: a model and experimental results," *International Journal of Computer and Information Sciences*, vol. 8, pp. 219–238, June 1979.

[176] SHNEIDERMAN, B., *Software psychology: Human factors in computer and information systems (Winthrop computer systems series)*. Winthrop Publishers, 1980.

[177] SHORS, T. J., MIESEGAES, G., BEYLIN, A., ZHAO, M., RYDEL, T., and GOULD, E., "Neurogenesis in the adult is involved in the formation of trace memories.," *Nature*, vol. 410, pp. 372–376, March 2001.

[178] SIEGEL, M., WARDEN, M. R., and MILLER, E. K., "Phase-dependent neuronal coding of objects in short-term memory.," *Proceedings of the National Academy of Sciences of the United States of America*, vol. 106, pp. 21341–21346, December 2009.

[179] SIEGMUND, J., KÄSTNER, C., APEL, S., PARNIN, C., BETHMANN, A., LEICH, T., SAAKE, G., and BRECHMANN, A., "Understanding understanding source code with functional magnetic resonance imaging," in *Proceedings of the 36th International Conference on Software Engineering*, ICSE 2014, (New York, NY, USA), pp. 378–389, ACM, 2014.

[180] SINGER, J., ELVES, R., and STOREY, M.-A., "Navtracks: Supporting navigation in software maintenance," in *ICSM '05: Proceedings of the 21st IEEE International Conference on Software Maintenance*, (Washington, DC, USA), pp. 325–334, IEEE Computer Society, 2005.

[181] SINHA, V., KARGER, D., and MILLER, R., "Relo: Helping users manage context during interactive exploratory visualization of large codebases," in *VLHCC '06: Proceedings of the Visual Languages and Human-Centric Computing*, (Washington, DC, USA), pp. 187–194, IEEE Computer Society, 2006.

[182] SMITH, J., "Wpf apps with the model-view-viewmodel design pattern," Feb. 2009.

[183] SMITH, R. E., "The cost of remembering to remember in event-based prospective memory: investigating the capacity demands of delayed intention performance.," *Journal of experimental psychology. Learning, memory, and cognition*, vol. 29, pp. 347–361, May 2003.

[184] SPIERS, H., MACGUIRE, E., and BURGESS, N., "Hippocampal amnesia," *Neurocase*, vol. 7, pp. 352–382, 2001.

[185] SQUIRE, L. R., "Memory systems of the brain: a brief history and current perspective.," *Neurobiology of learning and memory*, vol. 82, pp. 171–177, November 2004.

[186] STOREY, M.-A. D., FRACCHIA, F. D., and MÜLLER, H. A., "Cognitive design elements to support the construction of a mental model during software exploration," *J. Syst. Softw.*, vol. 44, pp. 171–185, January 1999.

[187] STOREY, M.-A., BEST, C., MICHAUD, J., RAYSIDE, D., LITOIU, M., and MUSEN, M., "Shrimp views: an interactive environment for information visualization and navigation," in *CHI '02: CHI '02 extended abstracts on Human factors in computing systems*, (New York, NY, USA), pp. 520–521, ACM, 2002.

[188] STOREY, M.-A., RYALL, J., BULL, R. I., MYERS, D., and SINGER, J., "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*, (New York, NY, USA), pp. 251–260, ACM, 2008.

[189] STOREY, M.-A., RYALL, J., SINGER, J., MYERS, D., CHENG, L.-T., and MULLER, M., "How software developers use tagging to support reminding and refinding," *IEEE Trans. Softw. Eng.*, vol. 35, pp. 470–483, July 2009.

[190] STOREY, M.-A. D., ČUBRANIĆ, D., and GERMAN, D. M., "On the use of visualization to support awareness of human activities in software development: a survey and a framework," in *SoftVis '05: Proceedings of the 2005 ACM symposium on Software visualization*, (New York, NY, USA), pp. 193–202, ACM, 2005.

[191] TASHMAN, C., "Windowscape: a task oriented window manager," in *UIST '06: Proceedings of the 19th annual ACM symposium on User interface software and technology*, pp. 77–80, ACM, 2006.

[192] THE ECLIPSE FOUNDATION, "Usage Data Collector Results," January 5th, 2009. Website, `http://www.eclipse.org/org/usagedata/reports/data/commands.csv`.

[193] TRAFTON, J. G., ALTMANN, E. M., and BROCK, D. P., "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.

[194] TREUDE, C. and STOREY, M.-A., "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*, (Washington, DC, USA), pp. 12–22, IEEE Computer Society, 2009.

[195] TSE, D., LANGSTON, R. F., KAKEYAMA, M., BETHUS, I., SPOONER, P. A., WOOD, E. R., WITTER, M. P., and MORRIS, R. G. M., "Schemas and Memory Consolidation," *Science*, vol. 316, no. 5821, pp. 76–82, 2007.

[196] TULVING, E., *Organization of memory*, ch. Episodic and semantic memory, pp. 381–403. New York: Academic Press, 1972.

[197] TULVING, E., *The Cognitive Neurosciences*, ch. Organization of memory: Quo vadis?, pp. 839–847. Cambridge, MA: MIT Press, 1995.

[198] TULVING, E. and THOMSON, D. M., "Encoding specificity and retrieval processes in episodic memory," *Psychological Review*, vol. 80, pp. 352–373, 1973.

[199] UNDERWOOD, B. J., "Interference and forgetting," *Psychological Review*, vol. 64, no. 1, pp. 49–60, 1957.

[200] VAKILIAN, M., CHEN, N., NEGARA, S., RAJKUMAR, B. A., BAILEY, B. P., and JOHNSON, R. E., "Use, disuse, and misuse of automated refactorings," in *ICSE '12: Proceedings of the 34th International Conference on Software Engineering*, (Washington, DC, USA), p. to appear, IEEE Computer Society, 2012.

[201] VALIPOUR, M. H., AMIRZAFARI, B., MALEKI, K. N., and DANESHPOUR, N., "A brief survey of software architecture concepts and service oriented architecture," in *Computer Science and Information Technology, 2009. ICCSIT 2009. 2nd IEEE International Conference on*, pp. 34–38, Aug. 2009.

[202] VALLAR, G. and BADDELEY, A. D., "Fractionation of working memory: Neuropsychological evidence for a phonological short-term store," *Journal of Verbal Learning and Verbal Behavior*, vol. 23, no. 2, pp. 151 – 161, 1984.

[203] VAN SOLINGEN, R., BERGHOUT, E., and VAN LATUM, F., "Interrupts: Just a minute never is," *IEEE Software*, vol. 15, no. 5, pp. 97–103, 1998.

[204] VILLRINGER, A., *Functional MRI*, ch. Physiological Changes During Brain Activation, pp. 3–13. Springer, 2000.

[205] VON MAYRHAUSER, A. and VANS, A. M., "From code understanding needs to reverse engineering tools capabilities," in *CASE '93: The Sixth International Conference on Computer-Aided Software Engineering, (Institute of Systems Science, National University of Singapore, Singapore; July 19-23, 1993)*, pp. 230–239, July 1993.

[206] WALNY, J., HABER, J., DORK, M., SILLITO, J., and CARPENDALE, S., "Follow that sketch: Lifecycles of diagrams and sketches in software development," in *Visualizing Software for Understanding and Analysis (VISSOFT), 2011 6th IEEE International Workshop on*, pp. 1 –8, sept. 2011.

[207] WEARING, D., *Forever Today: A Memoir of Love and Amnesia (Charnwood Large Print)*. Ulverscroft Large Print Books Ltd, new ed ed., Jan. 2006.

[208] WEGNER, D., ERBER, R., and RAYMOND, P., "Journal of personality and social psychology," vol. 61, pp. 923–929, 1991.

[209] WINOGRAD, E., *Practical Aspects of Memory: Current Research and Issues*, vol. 2, ch. Some observations on prospective remembering, pp. 348–353. Chichester: Wiley, 1988.

[210] WIXTED, J. T., "The psychology and neuroscience of forgetting.," *Annual review of psychology*, vol. 55, pp. 235–269, 2004.

[211] WOLFE, M. B. W. and MIENKO, J. A., "Learning and memory of factual content from narrative and expository text," *British Journal of Educational Psychology*, vol. 77, no. 3, pp. 541–564, 2007.

[212] ZACKS, J. M., TVERSKY, B., and IYER, G., "Perceiving, remembering, and communicating structure in events," *Journal of Experimental Psychology: General*, vol. 130, pp. 29–58, 2001.

[213] ZEIGARNIK, B., "Das behalten erledigter und unerledigter handlungen," *Psychologische Forschung*, vol. 9 (1), pp. 1–85, 1927.

[214] ZOU, L. and GODFREY, M. W., "An industrial case study of program artifacts viewed during maintenance tasks," in *WCRE '06: Proceedings of the 13th Working Conference on Reverse Engineering*, (Washington, DC, USA), pp. 71–82, IEEE Computer Society, 2006.