Programmer Information Needs after Memory Failure

Chris Parnin
Georgia Institute of Technology
Atlanta, Georgia USA
chris.parnin@gatech.edu

Spencer Rugaber
Georgia Institute of Technology
Atlanta, Georgia USA
spencer@cc.gatech.edu

***Abstract*—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 program development environments have begun to recognize this problem, none of these tools take into account the brain's structure and limitations. In this paper, we present a conceptual framework for understanding the strengths and weaknesses of human memory, particularly with respect to it ability to deal with work interruptions. The framework explains empirical results obtained from experiments in which programmers were interrupted while working. Based on the framework, we discuss programmer information needs that development tools must satisfy and suggest several memory aids such tools could provide. We also describe our prototype implementation of these memory aids.**

I. Introduction

Despite human memory's remarkable abilities, memory limitations inhibit programmer productivity. In particular, work interruption devastates memory and makes tasks take twice as long to perform and have twice as many errors [1]. Unfortunately, such interruptions are common—developers rarely are able to program in long continuous sessions [2]. Instead, a developer's day is fragmented into many short sessions (15-30 minutes) interspersed with occasional longer ones (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) recovering before resuming coding.

Yet, almost no current programming tool is built based on a modern understanding of the strengths and weaknesses of human memory. Current software development environments and their accompanying tools are based on tool design frameworks decades old, founded on psychology research that is even older. As one of the authors of a prominent framework recently stated, "These models are being used long-past their shelf-life" [3].

The overall thesis of our 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, we must demonstrate the effects of interruption on a programmer and casually relate them to limitations of human memory, *i.e.*, explain programmer performance problems in terms of memory limitations. Further, we need to devise strategies for overcoming these memory limitations and contextualize them in terms of programmer information needs.

In this paper, we describe programmer information needs in terms of the cognitive neuroscience of human memory. In so doing, we abstract out key concepts and principles of human memory from modern neuroscience literature. Finally, we describe how these information needs can be realized with new tools and relate them to shortcomings in existing programming tools.

II. The Cognitive Neuroscience of Memory

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 many years later? Previously, we provided a general review of the psychological and cognitive neuroscience research on the brain and memory [4]. In the current paper, we synthesize our findings in terms of five different types of human memory that are heavily used during programming. Our categories, derived Fuster's [5] and Morris and Frey's [6] accounts of memory, are the following: prospective, attentive, associative, episodic, and conceptual (summarized in Figure 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) [7]. 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 [8]. 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 [9]: 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

978-1-4673-1216-5/12/$31.00 © 2012 IEEE

ICPC 2012, Passau, Germany

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 (PFC) (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: 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 [10], 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.

Hippocampal Network

The next two types of memory make use of the same pathway of the brain, called the hippocampal network. 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.

The hippocampus network 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, 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.

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" [6]. 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 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 uniqueness, 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 [11]; 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 [12]), which is nearly equivalent to hearing ourselves speak aloud and may subsequently excite auto-associative mechanisms [13].

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 occur.

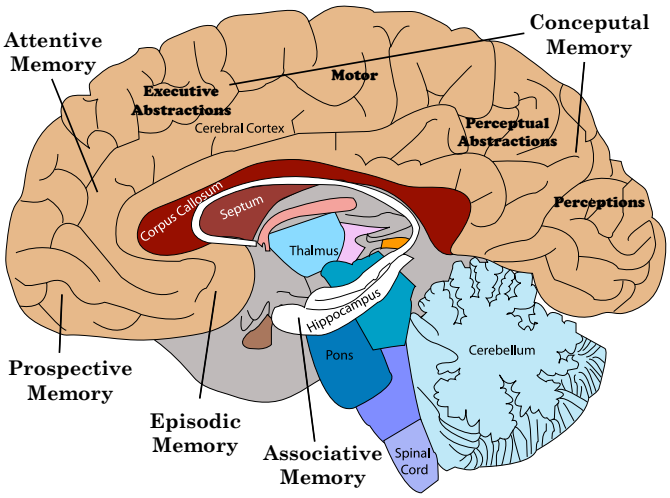

Fig. 1: Memory types in sagittal view of brain.

Episodic Memory

Tulving, an influential memory researcher, describes *episodic memory* as the recollection of past events [14]. 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 areas 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 [15].

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 in the lateral prefrontal cortex for maintaining information about recency and ordering about events retained in the hippocampus [16]. 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 [17].

Episodic memory is not as fully automatic as the spatio-temporal and perpetual components of associative memory can be easily disrupted [18]. 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.

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.

In Fuster's account of the brain [5], 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.

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 brain 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. 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. 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. 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.

III. Programmer Information Needs

In this section, for each memory type, we first describe a programming activity and how it greatly stresses that type. We then demonstrate how various memory failures affect developers in practice, and the mechanisms they use to cope with those memory failures. Finally, we then generalize these behaviors as information needs that each correspond to a particular memory failure. In Table I, we give a summary of the information needs that result after memory failure. The table displays the different memory types and their memory failures. In support of the memory failures and corresponding information needs, memory aids are derived.

A. Prospective Memory Support

1) Task: Resuming a Blocked Programming Task: Developers often become blocked on a task: i.e., being 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 (waiting for a busy teammate to become available again or finish a task) [19]. Other reasons include holding off on a task due to an unexpected shift in scheduling priority or server/database downtime [2]. Regardless of how the developer became blocked, the consequence is the same: A blocked developer must

TABLE I: 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	Learning new API	Source failure Recollection failure	Store context Support narrative	*code narratives*
conceptual	Forming concepts	Activation failure Formation failure	Support priming Support abstraction	*memlets*

remember to perform a task after a potentially lengthy interval. Unfortunately, prospective memory's ability to prompt us at the appropriate time can be quite unreliable.

2) Developer Studies: Various studies have described how developers have tried to cope with prospective memory failures. For example, developers often leave TODO comments throughout code [20]. 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 [2]. 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 [21]: leave sticky notes and emails to themselves [2].

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 polling 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, we introduce the concept of a *smart reminder*, which compensates for prospective memory failures by providing facilities for monitoring and polling external conditions and for modulating levels of engagement.

B. Attentive Memory Support

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.

2) Developer Studies: Studies examining refactoring practices of programmers have found several deficiencies in tool support [22].One essential deficiency is the lack of ability to track the statuses of many locations in code. As a workaround, 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 the task well in an automated fashion: A correct change removes the compile error, whereas an incorrect change arising from a complication 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 [23].

3) Information Needs: Concentration failures arise when programmers need to shift attention away from a programming task. Interruptions to tasks can cause programmers to lose

track of the status of previously attended locations of code.

Information Need 3 - Developers need support for persistent 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, we introduce the concept of *touch points*, which supports maintaining status across many locations in code.

C. Supporting Associative Memory

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 quickly absorb and familiarize himself with the code. This includes learning new identifiers, locations, relationships, conventions, and behaviors. Such a task deeply taxes associative memory.

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. [24] observed that programmers used 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 flip through open tabs because they fail to associate the tabs with desired code locations [25].

3) Information Needs: Retention failures occur when environmental stimuli do not offer sufficient features to trigger associative mechanisms in the hippocampus. 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 for tabs).

Information Need 5 - Developers need support for diverse and distinguishable 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 its functionality, but fail to form strong associations with details such as its name or location [26]. As a result, developers often spend significant time locating code after an interruption [2].

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

To address these needs, we provide *associative links*, which are memory aids that provide distinguishable features and indexing by multiple modalities. Specially, we give an example of how a code tab can be made more associable with alternative modalities.

D. Episodic Memory Support

1) Task: Learning a New API: Developers must often learn how to use new programming language features or APIs. For example, if a developer wanted to plot tweets obtained from the Twitter API onto a map using the Google Maps API, she would have to learn how to deal with the many concepts and quirks associated with both of the APIs. The difficulty of learning new APIs is often compounded by the fact that documentation, when it exists, is often of poor quality and lacks sufficient examples and explanations [27]. As a result, programmers can become derailed from their original task, when unresolved understanding [28] blocks progress. They must often piece together their learning experiences from many hours or days of frustrating coding attempts and false starts, sprinkled with occasional moments of triumph.

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 previously made changes [2]. However, developers complain of the problems they have with using existing diff tools including: The information provided is unordered, verbose, time-consuming, and cognitively demanding.

Studies of programmers have found that presenting information about a past programming session in an episodic manner [29], [26], [30], improves recall of a past programming task. Similarly, studies that examined recall of life experiences have shown that when a sequential presentation of events (pictures) from a past experience is given, that presentation can be more effective at stimulating recall than when other contextual details (names or locations) are given [29], [17]. 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 an optimal learning strategy for intermediate learners [31].

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 example.

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. For example, developers occasionally need to relate their programming experiences to colleagues who want to perform similar tasks. In both 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, we introduce the concept of a *code narrative*, which support developers in retaining and recollecting contextual details and narratives about their learning experiences.

E. Supporting Conceptual Memory

1) Task: Forming Concepts: Developers are expected to maintain expertise in their craft throughout their careers. Unfortunately, the path to becoming an expert is not easily walked: For a novice, evidence suggests this can be a 10 year journey [32]. And for experts trying to become experts in new domains, like the desktop developer becoming a web developer, there are many concepts that must be put aside and new ones learned.

Studies examining the difference between an expert and a 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 [33]: 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.

2) Developer Studies: Studies suggest that sketching, diagramming, 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 [34]. Diagrams are used to form and retain early concepts [35]. Note-taking is a also common strategy to retain information about a programmed task when interrupted; however, such notes can often be incomplete and lead to resumption failures [26].

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

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

Activation failures occur when a concept has not been used recently, lessening a programmer's ability to use that learned concept. Developers that have been interrupted during a programming task need to refresh the concepts associated with the task before resuming work.

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

To address conceptual memory failures, we introduce the concept of a *memlet*, which support developers in abstracting and refreshing concepts in source code.

IV. Tools for Prevention of and Recovery from Memory Failure

In this section, we describe tools we have devised that address the information needs articulated in the previous section. Each tool is presented in terms of the information needs served and the way in which it address the needs.

A. Smart Reminders for Prospective Memory

1) Information Needs: 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. A smart reminder is composed of three parts: 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 in which the reminder is conveyed to the user. The *reminder message* is a textual notification.

To support Information Need 1, a smart reminder can be created with a reminder condition that monitors 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 [36]. To support such strategies, we have created *proximity conditions*, which condition the display of a reminder based on proximity to relevant locations such as a class or namespace path. To support monitoring of external conditions, we have devised several domain-specific conditions that check on things such as task completion in a task tracker and checkins of source files into source control systems. Ultimately, a rich space of reminder conditions are possible, tailorable to different types of programming environments, team compositions, personal preferences, and software development processes.

To support Information Need 2, a smart reminder can be created with a notification mechanism that varies in strength. *Passive notifications* do not force attention, but remain passive until dismissed. For example, we 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 they are explicitly 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 an activity such as a checkin, program build or program execution. Finally, it is possible to design notifications that blend these different levels.

2) Related Devices:

- *TagSea*: a set of hierarchical tags on annotated source code lines [37].
- *Roadblock*: an intentional compile error that must be addressed before compiling a program.

Todo comments often get treated as documentation, with its known limitations, and not as prospective reminders. TagSea support representing and organizing reminder messages, but do not support engaging a user's attention or conditioning the display of the reminder. Roadblocks can be viewed as constrictive notifications but not as the other various configurations of a smart reminder.

B. Touch Points for Attentive Memory

1) Information Needs: Touch points are attentive memory aids that enables a programmer to maintain persistent and stateful attention to programming elements. A *programming element* is a named entity such as a class or a method. 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 keep track, a touch point maintains internal state about recency of edits and visits. This internal state enables programmers to 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 on the 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 the recently recorded programming activity.

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 [38].

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 sets of a tree-like collection of programming elements, including statements indicating activity, annotations, and issues. That is, they support managing multiple locations and tracking progress.

C. Associative Links for Associative Memory

1) Information Needs: An *associative link* is a memory aid that helps a programmer form and recall associations by providing distinctive features and multimodal 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. Some examples of modalities include:

- *lexical*: alphabetic combinations, i.e., identifiers.
- *structural*: position in program element organization.
- *spatial*: visible position in programming interface.
- *operational*: user action taken at the source code location.
- *syntactical*: grammatical role of the source code element.

For supporting navigation within unfamiliar code, we demonstrate how associative links can be used to improve the accessibility of 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 name might be a method name or a file name. To improve access, 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 a 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 are more likely to encourage the formation of associative memories, as there are more distinctive elements present during the act of navigation.

To support Information Need 6, 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 when stepping through a program while 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, the associative links allow multiple modes of indexing into code locations to improve access.

2) Related Devices:

- *NavTracs*: a set of files associated by frequent co-visitation [25].
- *Code Canvas*: a fixed layout of source code content, associating each file with a spatial position on a zoomable plane [39].
- *Code Bubbles*: a dynamic layout of source code fragments, associating each fragment with a spatial position on a scrollable plane [40].

There are several related devices that exhibit characteristics similar to associative links. NavTracs provide ways of indexing into code locations via operational modality, specificly naviga-

tion 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 or provide multiple modalities for improving access.

D. Code Narratives for Episodic Memory

1) Information Needs: 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, such as an edit, a search or a run-time exception. A *narrative structure* provides a schema for anticipating and organizing the events of a story. Narrative structures are socially constructed [15], 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, we found two common narrative structures used by developers: *overcoming obstacles* and *teaching tutorials* [41].

To support Information Need 7, a programming environement 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 is 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 the obstacle narrative structure, 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 discover the problem. The *resolution* is the series of programming events that solved the conflict.

For the tutorial narrative structure, 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. We have prototyped algorithms that semi-automatically populate a series of programming events into a tutorial-style narrative and publish it as a blog post.

Finally, a distinction is made between personal and shared narratives. When a narrative is shared, more 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, personal narratives can leverage existing episodic memories, enabling a programmer to move fluidly through his own experiences. In support of personal narratives, we allow coding details to be organized hierarchically, by clustering and grouping programming events into programming activities, supporting improved indexing.

2) Related Devices:

- *information quests*: are a collection of files visited, annotated with an information seeking goal and shared with others [42].
- *code replays*: are a stream of change events that can be replayed and shared with others [30]

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.

E. Memlets for Conceptual Memory

1) Information Needs: A *memlet* is an conceptual memory aid that helps a programmer form and prime concepts by supporting abstraction and reviewing concepts that need to be refreshed. A memlet is composed of a programming element, an overlay, and a set of workspaces. A *overlay* is a visual plane that contains a set of annotations projected onto the programming element. An *workspace*, is a visual plane that contains 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 the programming element. Workspaces can be shared with other programming elements, enabling a programmer to represent abstractions between programming elements.

To support Information Need 10, code that has not been recently viewed can be toggled to auto-display annotations. Additionally, visual cues indicating non-recency can be used to encourage reviewing relevant workspace.

2) Related Devices:

- *ConcernMapper*: a set of concerns organizing projections of programming elements [43].
- *Code folding*: a set of hierarchical compiler directives organizing source code lines.

In ConcernMapper, a concern allows a single abstraction to be built over many programming elements; whereas memlets allow abstractions to happen at a finer granularity. Code folding interleaves organization with source code; whereas memlets provide overlays and alternative workspaces. In contrast with memlets, these devices do not integrate developer's sketching-like behavior, nor consider which organized knowledge may need to be primed.

V. Discussion

A look at the mechanisms of the brain and its capacity for memory gives us a renewed perspective into our existing theories about programmer cognition. Consider, the classic work of Shneiderman and Mayer: When programmers were asked to recite recently viewed programs, they describe semantic and not syntactic contents [44]. However, considering the underlying mechanisms of the brain, an alternative explanation is that semantics (abstractions) are more easily primed in conceptual memory than syntactics (perceptions), and are therefore easier to freely recall. Further, the conclusion that syntactic information is not retained can also be given an alternative explanation: Syntactics *are* retained, but not directly; instead an association forms between semantic and syntactic information, explaining the difficulty in freely recalling the syntactics. This also suggests that in future viewings of the code, syntactics plays a role in associatively recalling semantics related to the block of code without a renewed comprehension effort.

Other frameworks of programming information needs, such as the one described by Storey et al. [45], build on theories such as Shneiderman and Mayer's to provide means of supporting program comprehension and reducing cognitive overload. Overall, the guidance from such frameworks is sound, but ultimately limited. For example, Storey et al.'s information needs are limited to exploration tasks and do not incorporate memory failures and resulting information needs reflected in everyday programming tasks.

Developers use many coping mechanisms to ward off memory failures. For example, developers send email messages to themselves as prospective reminders, use compile errors as attentive points of focus and use source code history to reconstruct a narrative of their work. Our aim is not to discount the value of these mechanisms, but to instead use their existence as evidence for the importance of understanding the corresponding memory failures. We propose that by using this understanding, we can build a tool framework that supports the many fragile facets of human memory, ultimately leading to better tools for software development.

Understanding memory types may also help us better understand our research tools. For example, two research tools, Code Canvas and Code Bubbles, support a similar modality: spatial. But what appears similar on the surface may actually support different memory types. Code Bubble's fluid and dynamically changing landscape likely promotes temporary spatial associations that last a few hours; whereas the stable layout of Code Canvas likely promotes a longer-term, conceptual memory of spatial abstractions.

Our framing and presentation of the cognitive neuroscience of memory has several limitations. We, do not include literature on reasoning or problem solving. We also do not discuss interactions among memory types. For example, prospective memory cooperates with associative memory to hold long-term intentions. Finally, there are other information needs yet to be found that the community can seek.

VI. Conclusion

In this paper, we have examined the previously explored literature of cognitive neuroscience of human memory and structured the results in terms of five memory types particularly relevant to programmers: attentive, prospective, associative, episodic, and conceptual memory. We describe how failures in these memory types can be related to empirical evidence of programmers information seeking and preservation needs. Finally, we present five memory aids—touch points, smart reminders, associative links, code narratives, and memlets—that address these information needs and can inspire future tool development.

We have prototyped a set of tools, called worklets, including the five examples in the paper, as extensions for Visual Studio. More details on implementation of code narratives [1] and associative links [2] can be found online. We are in the process of refining the tools for evaluation and designing a series of laboratory and field experiments for evaluating the effectiveness of the tools in managing interruptions.

References

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

[2] C. Parnin and S. Rugaber, "Resumption strategies for interrupted programming tasks," *Software Quality Journal*, vol. 19, pp. 5–34, 2011, 10.1007/s11219-010-9104-9.

[3] M.-A. Storey, "an interactive visualization environment for exploring java programs," 2011, plenary presentation: International Conference on Program Comprehension. [Online]. Available: http://www.slideshare.net/mastorey/icpc-2011-storey-8471063

[4] C. Parnin, "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.

[5] J. M. Fuster, "The prefrontal cortex–an update: time is of the essence." *Neuron*, vol. 30, no. 2, pp. 319–333, May 2001. [Online]. Available: http://view.ncbi.nlm.nih.gov/pubmed/11394996

[6] R. G. Morris and U. Frey, "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. [Online]. Available: http://dx.doi.org/10.1098/rstb.1997.0136

[7] E. Winograd, *Practical Aspects of Memory: Current Research and Issues*. Chichester: Wiley, 1988, vol. 2, ch. Some observations on prospective remembering, pp. 348–353.

[8] J. R. Reynolds, R. West, and T. Braver, "Distinct neural circuits support transient and sustained processes in prospective memory and working memory." *Cerebral cortex (New York, N.Y. : 1991)*, vol. 19, no. 5, pp. 1208–1221, May 2009. [Online]. Available: http://dx.doi.org/10.1093/cercor/bhn164

[9] K. Kondo, M. Maruishi, H. Ueno, K. Sawada, Y. Hashimoto, T. Ohshita, T. Takahashi, T. Ohtsuki, and M. Matsumoto, "The pathophysiology of prospective memory failure after diffuse axonal injury - lesion-symptom analysis using diffusion tensor imaging," *BMC Neuroscience*, vol. 11, no. 1, pp. 147–157, November 2010.

[10] J.-S. Provost, M. Petrides, F. Simard, and O. Monchi, "Investigating the Long-Lasting residual effect of a set shift on frontostriatal activity," *Cerebral Cortex*, Dec. 2011. [Online]. Available: http://dx.doi.org/10.1093/cercor/bhr358

[1] http://blog.ninlabs.com/2011/11/auto-blogging-publishing-a-coding-task-to-wordpress-5/

[2] http://blog.ninlabs.com/2011/10/napkin-idea-code-tabs/

[11] L. L. Eldridge, B. J. Knowlton, C. S. Furmanski, S. Y. Bookheimer, and S. A. Engel, "Remembering episodes: a selective role for the hippocampus during retrieval." *Nature neuroscience*, vol. 3, no. 11, pp. 1149–1152, November 2000. [Online]. Available: http://dx.doi.org/10.1038/80671

[12] G. Hickok, K. Okada, and J. T. Serences, "Area Spt in the Human Planum Temporale Supports Sensory-Motor Integration for Speech Processing," *Journal of Neurophysiology*, vol. 101, no. 5, pp. 2725–2732, May 2009.

[13] C. McGettigan, J. E. Warren, F. Eisner, C. R. Marshall, P. Shanmugalingam, and S. K. Scott, "Neural correlates of sublexical processing in phonological working memory." *Journal of cognitive neuroscience*, vol. 23, no. 4, pp. 961–977, April 2011.

[14] E. Tulving, *Organization of memory*. New York: Academic Press, 1972, ch. Episodic and semantic memory, pp. 381–403.

[15] S.-Y. Kim, "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. [Online]. Available: http://www.eric.ed.gov/ERICWebPortal/detail?accno=EJ605419

[16] E. L. Glisky, M. R. Polster, and B. C. Routhieaux, "Double dissociation between item and source memory," *Neuropsychology*, vol. 9, pp. 229–235, 1995.

[17] S. Hodges, E. Berry, and K. Wood, "SenseCam: A wearable camera that stimulates and rehabilitates autobiographical memory." *Memory (Hove, England)*, vol. 19, no. 7, pp. 685–696, Oct. 2011. [Online]. Available: http://dx.doi.org/10.1080/09658211.2011.605591

[18] I. Kahn, A. Pascual-Leone, H. Theoret, F. Fregni, D. Clark, and A. Wagner, "Transient disruption of ventrolateral prefrontal cortex during verbal encoding affects subsequent memory performance." *Journal of neurophysiology*, vol. 94, no. 1, pp. 688–698, July 2005.

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

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

[21] V. Bellotti, B. Dalal, N. Good, P. Flynn, D. G. Bobrow, and N. Ducheneaut, "What a to-do: studies of task management towards the design of a personal task list manager," in *CHI '04: Proceedings of the SIGCHI conference on Human factors in computing systems*. New York, NY, USA: ACM, 2004, pp. 735–742.

[22] E. Murphy-Hill, C. Parnin, and A. P. Black, "How we refactor, and how we know it," *IEEE Transactions on Software Engineering*, vol. 99, no. PrePrints, 2011.

[23] X. Ge and E. Murphy-Hill, "Reconciling manual and automatic refactoring," in *ICSE '12: Proceedings of the 34th International Conference on Software Engineering*. Washington, DC, USA: IEEE Computer Society, 2012, p. to appear.

[24] A. J. Ko, M. J. Coblenz, and H. H. Aung, "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.

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

[26] C. Parnin and R. DeLine, "Evaluating cues for resuming interrupted programming tasks," in *Proceedings of the 28th international conference on Human factors in computing systems*, ser. CHI '10. New York, NY, USA: ACM, 2010, pp. 93–102. [Online]. Available: http://doi.acm.org/10.1145/1753326.1753342

[27] M. P. Robillard, "What makes apis hard to learn? answers from developers," *IEEE Softw.*, vol. 26, pp. 27–34, November 2009.

[28] A. J. Ko, B. A. Myers, and H. H. Aung, "Six learning barriers in end-user programming systems," in *Proceedings of the 2004 IEEE Symposium on Visual Languages - Human Centric Computing*, ser. VLHCC '04. Washington, DC, USA: IEEE Computer Society, 2004, pp. 199–206. [Online]. Available: http://dx.doi.org/10.1109/VLHCC.2004.47

[29] I. Safer and G. C. Murphy, "Comparing episodic and semantic interfaces for task boundary identification," in *CASCON '07: Proceedings of the 2007 conference of the center for advanced studies on Collaborative research*. ACM, 2007, pp. 229–243.

[30] L. Hattori, M. D'Ambros, M. Lanza, and M. Lungu, "Software evolution comprehension: Replay to the rescue," in *Program Comprehension (ICPC), 2011 IEEE 19th International Conference on*, june 2011, pp. 161 –170.

[31] M. B. W. Wolfe and J. A. Mienko, "Learning and memory of factual content from narrative and expository text," *British Journal of Educational Psychology*, vol. 77, no. 3, pp. 541–564, 2007. [Online]. Available: http://dx.doi.org/10.1348/000709906X143902

[32] M. T. H. Chi, R. Glaser, and E. Rees, *Expertise in problem solving*. Erlbaum, 1982, vol. 1, pp. 7–75. [Online]. Available: http://www.public.asu.edu/~mtchi/papers/ChiGlaserRees.pdf

[33] J. Milton, A. Solodkin, P. Hlustík, and S. L. Small, "The mind of expert motor performance is cool and focused." *Neuroimage*, vol. 35, no. 2, pp. 804–813, Apr. 2007. [Online]. Available: http://dx.doi.org/10.1016/j.neuroimage.2007.01.003

[34] J. Walny, J. Haber, M. Dork, J. Sillito, and S. Carpendale, "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*, sept. 2011, pp. 1 –8.

[35] M. Cherubini, G. Venolia, R. DeLine, and A. J. Ko, "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: ACM, 2007, pp. 557–566.

[36] M. R. McGee-Lennon, M. K. Wolters, and S. Brewster, "User-centred multimodal reminders for assistive living," in *Proceedings of the 2011 annual conference on Human factors in computing systems*, ser. CHI '11. New York, NY, USA: ACM, 2011, pp. 2105–2114. [Online]. Available: http://doi.acm.org/10.1145/1978942.1979248

[37] M.-A. Storey, J. Ryall, J. Singer, D. Myers, L.-T. Cheng, and M. Muller, "How software developers use tagging to support reminding and refinding," *IEEE Trans. Softw. Eng.*, vol. 35, pp. 470–483, July 2009. [Online]. Available: http://dl.acm.org/citation.cfm?id=1591903.1592342

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

[39] R. DeLine and K. Rowan, "Code canvas: zooming towards better development environments," in *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 2*, ser. ICSE '10. New York, NY, USA: ACM, 2010, pp. 207–210. [Online]. Available: http://doi.acm.org/10.1145/1810295.1810331

[40] A. Bragdon, S. P. Reiss, R. Zeleznik, S. Karumuri, W. Cheung, J. Kaplan, C. Coleman, F. Adeputra, and J. J. LaViola, Jr., "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*, ser. ICSE '10. New York, NY, USA: ACM, 2010, pp. 455–464. [Online]. Available: http://doi.acm.org/10.1145/1806799.1806866

[41] C. Parnin and C. Treude, "Measuring api documentation on the web," in *Proceedings of the 2nd International Workshop on Web 2.0 for Software Engineering*, ser. Web2SE '11. New York, NY, USA: ACM, 2011, pp. 25–30. [Online]. Available: http://doi.acm.org/10.1145/1984701.1984706

[42] A. Guzzi, M. Pinzger, and A. van Deursen, "Combining micro-blogging and ide interactions to support developers in their quests," in *Software Maintenance (ICSM), 2010 IEEE International Conference on*, sept. 2010, pp. 1 –5.

[43] M. P. Robillard and F. Weigand-Warr, "Concernmapper: simple view-based separation of scattered concerns," in *Proceedings of the 2005 OOPSLA workshop on Eclipse technology eXchange*, ser. eclipse '05. New York, NY, USA: ACM, 2005, pp. 65–69. [Online]. Available: http://doi.acm.org/10.1145/1117696.1117710

[44] B. Shneiderman and R. Mayer, "Syntactic/semantic interactions in programmer behavior: A model and experimental results," *International Journal of Parallel Programming*, vol. 8, pp. 219–238, 1979, 10.1007/BF00977789. [Online]. Available: http://dx.doi.org/10.1007/BF00977789

[45] M.-A. D. Storey, F. D. Fracchia, and H. A. Müller, "Cognitive design elements to support the construction of a mental model during software exploration," *J. Syst. Softw.*, vol. 44, pp. 171–185, January 1999. [Online]. Available: http://dx.doi.org/10.1016/S0164-1212(98)10055-9