

# CodePad: Interactive Spaces for Maintaining Concentration in Programming Environments

Chris Parnin, Carsten Görg, and Spencer Rugaber  
College of Computing, Georgia Institute of Technology, Atlanta, GA USA  
chris.parnin@gatech.edu, {goerg,spencer}@cc.gatech.edu

## ABSTRACT

When software developers work with a program's source code, the structure of the source code often requires that they split their attention simultaneously across several documents and artifacts. Disruptions to programmers' concentration caused by overwhelmed capacity can then lead to programming errors and increases in the time to perform a task. We suggest the addition of peripheral *interactive spaces* to programming environments for supporting developers in maintaining their concentration. We introduce the novel concept of a CodePad, a peripheral, multi-touch enabled display that allows developers to engage with and manipulate multiple programming artifacts. We illustrate visualizations built for a CodePad that support multiple development scenarios and we describe how developers can coordinate the interaction and communication between a CodePad and a programming environment in personal and collaborative tasks. Additionally, we propose a design space for other visualization tools and detail our initial prototype.

## Categories and Subject Descriptors

H.4 [Information Systems Applications]: Miscellaneous

## General Terms

Design

## 1. INTRODUCTION

Software development tasks typically require coordinating changes across multiple locations in a program's source code. Existing integrated development environments (IDEs) have provided limited support for maintaining working state associated with active artifacts relevant to the programming task. IDEs provide support for linking artifacts, including tabbed editors for editing multiple files, hierarchical file lists that can be collapsed and expanded, forward/backward navigation commands to return to previous locations, and lists of recently visited files. Most of IDE support relies on using

artifact names for linking; research comparing development interfaces using names or the *content* that a name refers to has shown that names are slower and less accurate [24] and content is strongly preferred [20]. Further, little support is given for managing working state such as plans, task progress, or recent actions associated with these artifacts.

Poor IDE support for programming tasks has resulted in developers having to still spend extra time managing working state and wasting effort in recovering lost artifacts. In a study of developers, Ko and colleagues found that developers spent 35% of their time navigating and recovering code [11]. Interestingly, developers were leveraging cues from the programming environment (such as Eclipse's package explorer, file tabs, and scroll bars) to encode their mental state of relevant code; however, the cues were highly unstable and could easily lose visibility (as they were erased by subsequent efforts). Other researchers have found that programmers need to frequently recover these lost artifacts: In a controlled study of program exploration tasks, 57% of program entities explored were frequently revisited [26]. In our previous field study of the work history of 10 professional programmers that spanned several weeks of development, we found that 60% of transitions between program entities were between different documents; furthermore, on a typical day the developers worked with 51 to 83 methods [21]. In general, programming environments do not make working state a first class entity that can be shielded from the constant shifting of focus, which contributes to an increased difficulty in maintaining concentration.

In response to this problem, research has attempted to either provide better *links* to relevant artifacts, or provide abstractions or visualizations capable of displaying multiple artifacts. When tools attempt to provide links to relevant artifacts, they have to predict or use a heuristic for what *relevant* means. Unfortunately, predictive guesses are rarely correct [21], and spatial visualizations break down when developers must transition to spatially distant locations (as they frequently do [11, 21, 26]) increasing the likelihood to become disoriented when panning and zooming.

In practice, programmers cope with the problem of maintaining concentration by adopting practices such as note taking, but they are given little support in linking their notes with a programming environment or situating their notes in a context. For programming tasks, text is not necessarily the dominant form of expression [3], nor is coding the dominant activity [13]. Our previous research on software developers shows that developers make extensive use of paper and other media distinct from the programming environment to man-

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

SOFTVIS'10, October 25–26, 2010, Salt Lake City, Utah, USA.  
Copyright 2010 ACM 978-1-4503-0028-5/10/10 ...\$10.00.

age programming tasks [20]. One developer we interviewed supports programming tasks by using a white board, a tablet for writing notes, several monitors and virtual machines, and a dedicated monitor with a document containing a stream of screen shots. In studying working style of hundreds of developers, it became clear that programmers vary in how they work depending on experience, task load, task types, and personal working styles; however, each is struggling to grapple with managing knowledge and attention across programming tasks and often resorts to using measures outside the programming environment to cope.

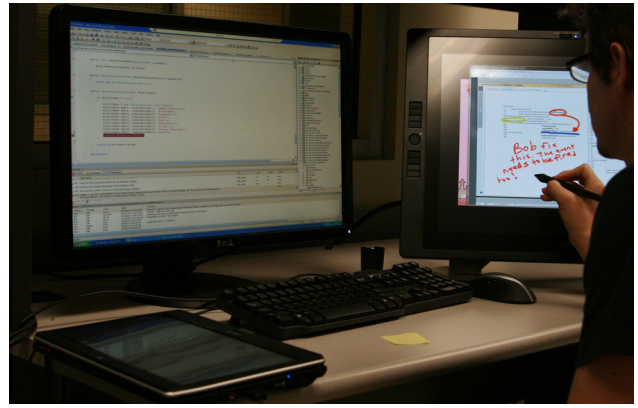
We propose a new approach—instead of trying to predict or assume what is relevant to a developer, we stress two important components for maintaining concentration: First, we provide an interactive space [23] distinct and physically separate from the programming environment that allows the developer to manage and see what is relevant. Second, this interactive space has additional modalities available for interacting with and annotating content. Together, these components create a mental playground separate from the programming environment, allowing developers to maintain concentration on their difficult and often interrupted work and provide an additional place to keep their thoughts. We call this interactive space a *CodePad*, a device that meets the following requirements: (a) displays content, (b) exists separate from a programming environment, (c) allows multi-modal input (pen and multi-touch), and (d) communicates and synchronizes with the programming environment.

To illustrate the problems with current approaches, we consider a programmer who would like to refactor code. In this particular case, she would like to consolidate several helper methods located throughout the source code. She does not have time to perform a *root canal* refactoring [17], but instead wants to interleave refactoring with her normal programming tasks. In current development systems, she would have little support: she would have to copy and paste the to-be-extracted methods and locations into a temporary text document or to individually perform the method extraction and relocation, constantly disrupting the main task. With a system such as Code Canvas [7] or Code Bubbles [2], the programmer could spatially arrange the code documents or fragments containing the prospective locations; however, she would still be forced to manage her attention between creating and organizing these fragments in the middle of her task, because they occupy the same space. Any re-design would have difficulty in addressing this issue without disrupting the main display and task. Using a CodePad, the programmer can maintain task or activity separation or simultaneously interact with the same code from multiple perspectives. In Section 3, we describe in more detail how a CodePad would assist the programmer with the code refactoring. We also detail how the CodePad would help with other tasks that involve disruption to concentration.

The main contributions of this paper are:

- A design space for interactive spaces that can coordinate with programming environments
- Visualizations for maintaining awareness of development artifacts, and
- Interactions for engaging and coordinating attention across these artifacts.

The human-computer interaction and the information visualization community have already adopted different types of interactive spaces to support users in executing demand-



**Figure 1: Workspace with a main development space (IDE on the large screen in the middle) and two additional interactive spaces: one dedicated CodePad (screen on the right) and one portable CodePad (tablet computer on the left).**

ing tasks [25]. Until now, the software visualization community did not take recent developments in this research domain into account, missing opportunities in the process. Thus, the main benefit of introducing the CodePad concept is to showcase to the software visualization community how advanced interactive spaces can support developers. Another benefit of our work is the foundation of conceptual and development frameworks that other researchers can extend to advance coordination designs, gestures, and visualizations for developers.

## 2. CODEPAD DESIGN SPACE

Creating interactive spaces for developers enables several new possibilities, but it also introduces several challenges. In this section, we present five different form factors for CodePads, discuss how CodePads can be linked to and communicate with IDEs, and illustrate design choices for content presented on a CodePad.

Figure 1 shows a developer’s work environment who is using two CodePads. The main development space is the IDE in the middle, there is one dedicated CodePad on the right, and one portable CodePad on the left.

### 2.1 Form Factor

Developers leverage different spaces and multiple forms of media in supporting programming tasks. Research on multiple monitor usage finds users with multiple monitors perform less window swapping than single-monitor users and position applications to have an affinity with a particular monitor [9]. In addition, users adopt strategies such as leaving open windows on secondary monitors as reminders or cues for tasks, and compartmentalizing tasks into separate containers. When displays are super-sized (filling the room), the additional space offers more opportunities for users to externalize memory and leverage spatial context in performing difficult tasks [1].

We now describe five different form factors that offer different affordances for a CodePad.

#### 2.1.1 Dedicated CodePad

A dedicated CodePad is useful for tasks requiring frequent and tactile manipulation of content. A dedicated CodePad

is generally large in size and situated close to the developer. The large size discourages repositioning, leaving the device mostly stationary. An example of a dedicated CodePad is included in Figure 1 on the right, a 21 inch interactive pen display.<sup>1</sup> This device is generally positioned at a tilt much like a drafting table, allowing comfortable pen interactions with the display content. A dedicated CodePad is well suited for capturing detailed and expressive notes associated with work artifacts.

### 2.1.2 Tabletop CodePad

A tabletop CodePad allows the surface of a desk used by a programmer to double as an interactive display. Through underlying infrared sensors or an overhead camera, content displayed on a tabletop CodePad can be manipulated through pen and touch gestures. The readily available space on a desk offers strategic placement of content, links, and quick notes within arms reach. An example of a tabletop CodePad device includes the Microsoft Surface,<sup>2</sup> a device capable of projecting content and sensing pen and multiple touch points based on infrared detection.

### 2.1.3 Portable CodePad

A portable CodePad is mid-sized but readily mobile, allowing a programmer to maintain a few items while enabling moderate interaction. The portable CodePad is ideal for sharing content with another developer, taking quick notes at a meeting, or even working in transit (*e.g.*, train into work). Tablet computers provide a starting point for portable CodePads; however, current tablet computers are bulky, have poor interaction support, and provide little integration with a programming environment. Besides tablets, a more apt device comes in the form of an iPad.<sup>3</sup>

### 2.1.4 CodePad Strip

For programmers working in a non-traditional office environment (*e.g.*, in a coffee shop), the lack of multiple monitors or interactive desks may seem like the programmer has been suddenly handicapped. However, in such scenarios, this problem can be addressed with the introduction of a *CodePad strip*, a thin horizontal interactive display. A CodePad strip allows simple thumbnails or visualizations of code artifacts to be displayed and interacted with. For example, a CodePad strip allows a programmer to easily switch between code documents by simply pressing a thumbnail of the code artifact. Laptop manufacturers are starting to include small touchable screens on the keyboard area of a laptop; standalone CodePad strips placed on the table are also possible. A design suggestion of a CodePad strip is show in Figure 2.

### 2.1.5 Paper-like CodePad

A final device we describe is a *codeprint*, a paper-like CodePad. A codeprint is light-weight, stackable, and easily portable. The ultra-thin form factor allows a developer to maintain several codeprints, easing the ability to switch between active codeprints and facilitating task-switching. Codeprints also expand the possibilities for handing-off work or delivering more interactive reports. The technology for codeprints is not far off: Electronic ink displays are rapidly

<sup>1</sup><http://www.wacom.com/cintiq>

<sup>2</sup><http://www.microsoft.com/surface>

<sup>3</sup><http://www.apple.com/ipad>



**Figure 2: A design suggestion showing how a thin CodePad strip could be integrated with a laptop computer (touchable code documents thumbnails are shown).**

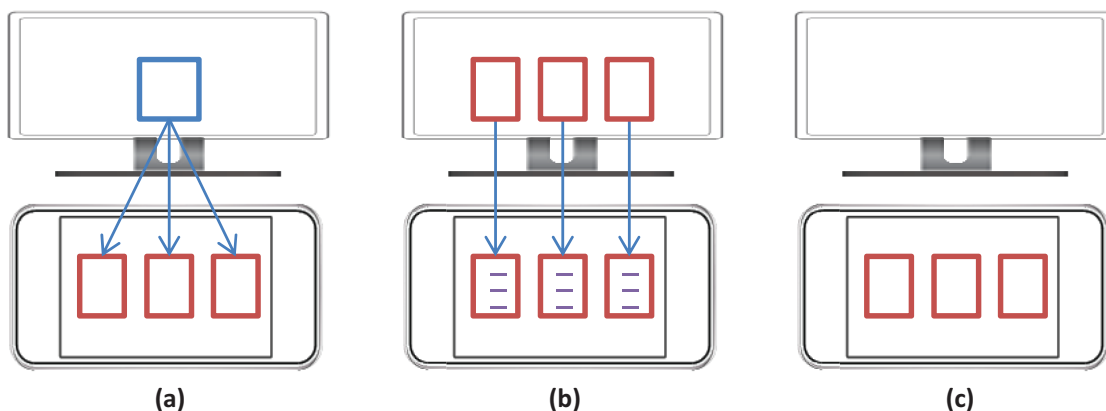
becoming reduced in weight and cost, and color versions have been developed. The main remaining challenge is to improve interaction capabilities of their displays.

## 2.2 Communication and Linkage

A challenge of having multiple interactive spaces is the lack of coordination and communication between the content of a programming environment and a CodePad. The issue of coordination is deeper than how to transfer content between the IDE and a CodePad. Transferring content can be solved by networking and in some cases it even is not a problem at all (*e.g.*, a dedicated CodePad is essentially an additional monitor connected to the same machine used for the programming task). The important and challenging problem is the decision of how much and what type of automation is offered for transferring content. We now briefly describe three linkage paradigms (see Figure 3) between a programming environment and any CodePad device. Keep in mind, these paradigms can be mixed and other paradigms certainly exist.

**Autoassociative or relational linkage:** a CodePad displays properties of a particular event or item whenever the item is focused or active (Figure 3(a)). There are numerous possibilities for how a CodePad could be bound in this manner: children of a parent class, emails and task descriptions associated with an active task, documentation of a method, code examples, or suggestions of relevant locations made by a recommendation system. If an item of interest is displayed on the CodePad, that item can be activated within the programming environment by touching the item or it can be saved by pinning the item with a specific gesture.

**Synchronized or mirrored linkage:** annotated content from a programming environment is projected onto a CodePad (Figure 3(b)). The nature of this projection may occur in various forms. We present three examples: overlay, abstraction, and detail. In an *overlay projection*, the CodePad displays information not visible in the programming environment with the content. For example, a code document is overlaid with notes from other developers or navigation trails from recent navigation. An *abstraction projection* presents the same content but in a simpler form (*e.g.*, an UML class diagram of current documents). A *detailed projection* pro-



**Figure 3: An IDE and a CodePad can be linked in various ways. Figure (a) shows an autoassociative link (the CodePad automatically displays related information to selected items in the IDE); Figure (b) shows a synchronized link (the CodePad mirrors and annotates content from the IDE); Figure (c) shows a manual link (the programmer manually sends content from the IDE to the CodePad).**

vides very specific but localized presentation of an item—for example, control flow or runtime values of variables.

**Send-to or manual linkage:** a programmer manually transfers an item to a CodePad (Figure 3(c)). The transfer operation can be initiated within the programming environment (from a context menu or keyboard short-cut) or from a grab gesture on the CodePad. Grabbing an item does not have to occur automatically but can be initiated from commands such as finding all references to a method, with the results appearing on the CodePad. Manually gathering content from a programming environment allows quick and ad-hoc segmentation of work artifacts. Finally, transfer operations are not limited to just being between a programming environment and a CodePad, but it is also possible to perform a send-to operation between CodePads (from a tabletop CodePad to a portable CodePad) or to non-CodePad devices such as another teammate’s electronic whiteboard.

## 2.3 Content Design

When designing applications for a CodePad, questions such as the following arise: How much of an item should be represented on the CodePad? Which operations should occur on a CodePad, and which ones should occur in the IDE? In the following, we discuss some guidelines.

A CodePad can serve two main purposes: either providing spatial contexts for salient cues that aid in the recall and restoration of appropriate items within a programming environment (a smarter tab) or novel manipulations of content not easily available or comfortable in normal contexts (instead of an outstretched arm to interact with distal monitors). Choosing one purpose or attempting to find a common ground somewhere in between has significant effects on the resulting design of visualizations and interactions.

When representing a code document on a CodePad, the design choices must consider with how much fidelity the document should be represented. Low fidelity representation (salient cues) can support linkage; manipulation of the document requires a high fidelity representation. Salient cues can include document thumbnails, frequent words within a document, actions performed on a document, people associated with previously editing a document, and relations to other parts of code. But for manipulating the document,

more and more of the document and interaction begins to migrate to the CodePad.

On the one hand, a CodePad can provide a haven for links into code undisturbed by the confused and exploratory activities of a programmer; on the other hand, a CodePad can provide rich annotations and casual manipulations of complex content not easily achieved in text form. The challenge is to avoid creating an incoherent and stale mess (computer desktop full of icons never clicked) coupled with ensuring interactions and content depictions do not detract from the benefits of the expanded interactive space. In the coming sections, we explore and elaborate further on this design tension in the context of specific programming scenarios.

## 3. PROGRAMMING SCENARIOS

In this section, we discuss three programming tasks and scenarios that involve maintaining concentration across multiple programming artifacts. We first provide background information from workplace studies that describe how particular programming tasks occur in practice. Next, we provide detailed explanations of the visualizations and interactions used by developers to complete the tasks. For brevity, we omit any initial configuration of the CodePad and programming environment and what steps would be necessary to change the mode of operation (refactoring mode or navigation mode). We conclude each scenario with a discussion.

### 3.1 Refactoring: Extracting and Moving Code

#### 3.1.1 Task

Refactoring is the process of changing the structure of a program without changing its behavior. A common refactoring is called EXTRACT METHOD, which first removes a subset of code from a long method, second creates a new method containing the extracted code, and finally configures the formerly long method to use the newly extracted method. EXTRACT METHOD is one of the most frequent refactorings developers want to perform [16], but developers use available refactoring tools in IDEs only 10% of the time [18]. One common reason for not using refactoring tools to perform an extract method is that developers want to batch several extractions together and then relocate the newly extracted methods to other locations. Current IDE tool support does

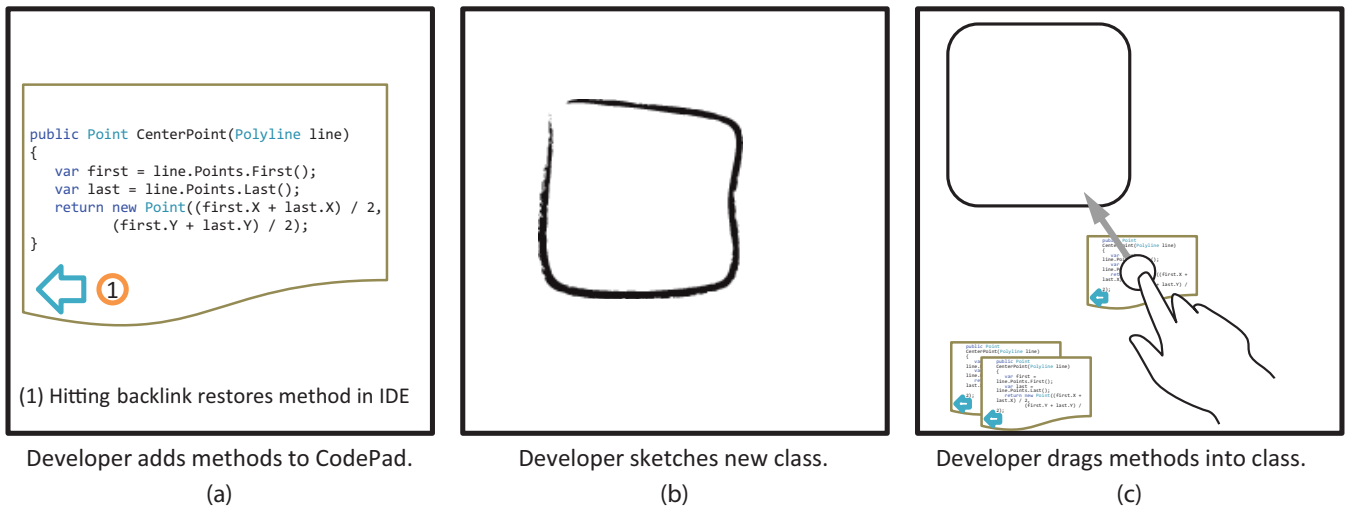


Figure 4: Workflow for performing a refactoring using a CodePad as an additional interactive space.

not make it easy to perform batch and compound refactorings because developers have to hold in memory all the locations involved with fragmenting, reassembling, and relocating code or rely on purposely breaking code to track code which can be problematic if unexpected issues arise and the refactoring needs to be rolled back.

### 3.1.2 CodePad Visualization and Interaction

How can a developer use a CodePad for refactoring? We first describe the workflow for supporting the refactoring (see Figure 4) and then detail the visualizations and interactions used in the process.

When a developer encounters a prospective code segment to relocate, she uses a send-to shortcut within the IDE which will transfer the currently selected method or code to the CodePad (Figure 4(a)); we are assuming only one CodePad is used for this example. After the developer has accumulated enough code fragments to gather together and form a new class, she initiates a square gesture on the CodePad to create a new placeholder (Figure 4(b)). Now she can begin to move methods into the class (Figure 4(c)). At this point, the developer may decide that the collection of methods actually involves two distinct concepts and should be divided into two classes. She uses a square gesture to create another placeholder class and then relocates some of the methods over to the other class.

The developer is confident that the batch refactoring can now be performed, but she would like to test it first. She uses a check gesture to test if the refactoring is feasible. Unfortunately, there is a problem with one of the extracted methods: A member variable was referenced in the code but is no longer available in the new class. The developer uses the back link on the problematic method extraction to bring the code into focus on the IDE. She manually fixes the problem at the original extraction site to address the issue (the method on the CodePad is updated as well). She confirms again with a check gesture and is informed that the refactorings can be now successfully applied. To apply the refactorings, the developer flicks the placeholder class toward the IDE (logical top of CodePad) and the refactoring is applied in the IDE where the developer can finalize details such as renaming the generated classes.

To represent a method on the CodePad, we display the method signature and the body in a scaled vector-based reduction. Because extracted methods tend to exist independently and programmers only need to recognize a method, we do not provide any other contextual cues (original file) for the extracted methods. However, we do provide a back link icon for returning to the method location within the programming environment. When a method is dragged into a class box, we display the method name without signature and provide a small thumbnail of the method body text. After a check gesture, problematic methods are unpacked from the contained class and marked to indicate the error. Otherwise, if successful, correct classes are highlighted in green.

Table 1 shows a summary of gestures used in this scenario.





GESTURE	ACTION
	Create new placeholder class
	Validate that a refactoring can be performed
	Perform batch refactorings of selected item
	Discard item

Table 1: Gestures supporting extract and move refactorings.

### 3.1.3 Discussion

Tabletop CodePads or portable CodePads would be the most suitable type of CodePads to perform refactorings. Two characteristics make these devices ideal: First, the medium-to-large amount of space available offers enough room to accommodate several methods. Second, both devices offer the ability to temporarily maintain items in the periphery while accumulating prospective methods to refactor and then easily bring into closer range when ready to perform the refactoring activity. For these types of refactorings, we believe manual linkage is the best way to coordinate content between the CodePad and IDE.

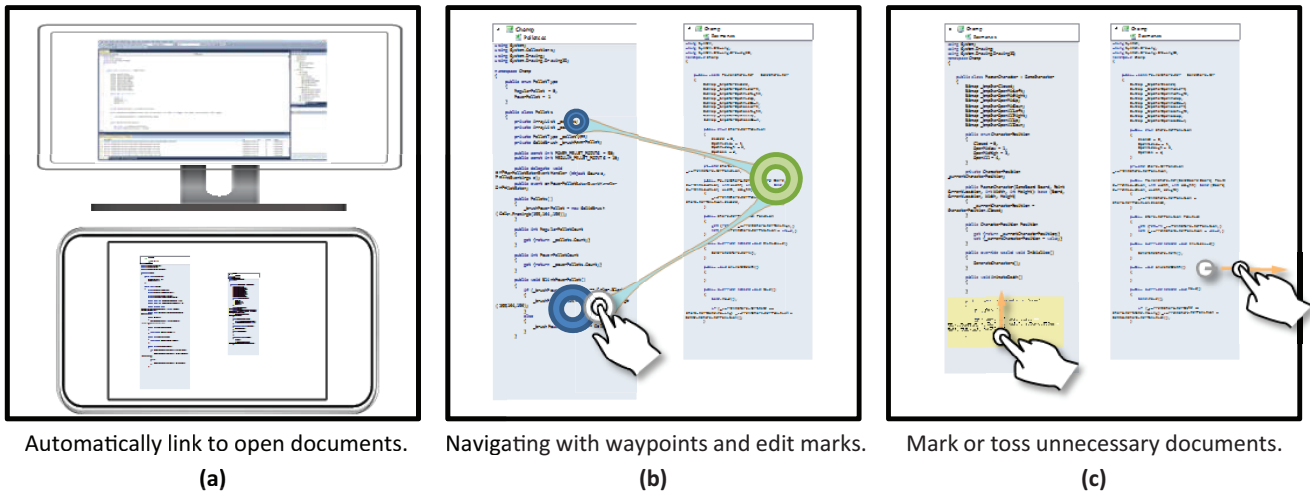


Figure 5: Workflow for navigating through document tabs using a CodePad as an additional interactive space.

There are several ways to improve workflow, gestures, and visualizations. Additional gestures for further extractions of code would offer more flexibility to a developer. Certainly, more strategies need to be developed for handling the fall out of refactoring errors and handling modality switches from a keyboard and focus in an IDE to a CodePad. Finally, improvements to the method representation (such as semantic zoom) can provide better aid for recognizing large methods that have been reduced on small CodePads.

### 3.2 Navigation through Document Tabs

When developers need to transition between documents, the main failing of document tabs are (1) they become spatially unstable and (2) document names are not very distinctive nor strongly associated with content (which is understandable considering a document can be reached via commands such as GOTO DEFINITION without ever being exposed to the document name).

In studies of developer navigation histories, a common finding is that developers frequently visit many locations in rapid succession in a phenomenon known as *navigation jitter* [27]. Navigation jitter has been commonly attributed to developers flipping through open tabs and file lists when trying to recall a location [21, 27]. If a fixed list of the four most recently used documents was maintained, 69% of navigations between documents could be satisfied [21]; however, developers often become frustrated with the number of open documents and choose to close all open tabs and start again. As a result, 74% of navigations not using document tabs occur by selecting it from the hierarchical file list (e.g., package explorer in Eclipse) [15].

#### 3.2.1 Task

In a web-based software project, a programmer has been assigned with the task of implementing labeling on a module within the program. Users of the software are able to customize what labels appear throughout the module, such as tabs, table columns, and headers. The programmer must be able to methodically track and maintain several locations among different items (database scripts, C# web-service, javascript, and html files) to ensure the proper customization of labels. The code impacted by this change does not necessarily have a concrete search path, instead requiring a careful

examination of the source code. Once all affected locations have been identified, additional care must be taken to ensure each location is properly changed. Document tabs offer poor support for representing the points of interest within the document and distinguishing relevant documents from intermediate stepping stones opened.

#### 3.2.2 CodePad Visualization and Interaction




When making a coordinated change across the program to support end-user labeling, the developer must track, in addition to the locations, the status of change (has it been changed yet?), special cases (how to handle customizing a label of grid column), and prospective actions (remember to add label refresh event).

With a CodePad, the developer can maintain a task overview as he makes progress. Every open document tab in the IDE is also mirrored on the CodePad (Figure 5(a)). If he closes a tab or uses a discard gesture on the CodePad, the document will be closed in both places. Each document is displayed in a scaled vector-based reduction to fit on the CodePad. The CodePad only needs to carry enough detail for coordinating attention (the resolution of code documents only needs to be sufficient for recognizing code), reads and edits can be done on the main screen. Rather than zooming and panning an entire virtual canvas, expand and shrink gestures can be easily applied to any target as needed.

As the developer visits and makes changes, he can see with waypoint markers (green for edits, blue for navigation) which parts of the code have been visited or edited (Figure 5(b)). Waypoint size and transparency are based on frequency and recency of interaction. To further provide the developer a sense of movements through space, we provide a bundled navigation trail between waypoints based on transition frequency. At any time, tapping waypoints will return the focus in the IDE to those points in the document. To declutter the work space, a wipe gesture (sideways hand gesture) can clear out any annotations over the affected region (Figure 5(c)).

The programmer has encountered several special cases that must be handled and wanted to mark these places to have a reminder. With a CodePad, the developer can mark the item in the IDE with a keyboard short cut, or use a select and then mark gesture on the CodePad to highlight the

code on the CodePad. Table 2 shows a summary of gestures used in this scenario.

GESTURE	ACTION
	Select a region of text
	Highlight a region of text
	Remove markers

**Table 2: Gestures supporting document navigation and annotation.**

### 3.2.3 Discussion

Numerous variations can be applied to this visualization. For smaller workloads, supporting extraction and pinning of methods would allow developers to keep small fragments available on hand without having to make more text visible. The ability to drag items together and form connections would support the creation of arbitrary relationships between code items and an easy way to bring them into focus. A magnify lens overlay would allow quick peeking on text without needing to expand or shrink it. Improved annotation support in the form of pen input and task-specific icons (found bug, TODO, good example, or task step) would enrich the developer’s ability to represent task knowledge.

The interaction of multiple CodePads could provide interesting possibilities. Multiple CodePad strips could be used to segregate content by affinity: A CodePad strip on the keyboard could provide a place to maintain a fixed list of code document thumbnails that do not change unless discarded, whereas another CodePad strip could be used to show recent documents or search results. Recommendations of other code locations can be placed in a separate CodePad strip. This segregation of content allows developers to trust and regulate attention between different types of links to content (a developer knows which code locations are recommendations and which code locations are tied to specific activity based purely on spatial context).

## 3.3 Task Hand-Offs

### 3.3.1 Task

Much of the knowledge needed for programming tasks is tacit in nature, leading to breakdowns in communication [12]. Some of this knowledge is internal to a specific developer or codified in tribal knowledge across a development team. Other bits of knowledge are never recorded or explicitly represented: design rationale or annotated history of recent code changes. This problem becomes apparent when a programmer is unable to complete a task and must hand-off an incomplete task to another developer or when a programmer is interrupted during a programming task and must rebuild lost context.

A large consulting company once attempted to achieve a 24-hour programming cycle. Developers in the USA would program for 12 hours and at the end of the day the developers then would wrap-up and hand-off any incomplete programming task to developers located overseas. The overseas developers picked up the incomplete tasks and attempted to make progress and then sent back the code to the USA developers. The project failed.

This story illustrates the difficulty programmers can have in communicating the knowledge needed for performing tasks—currently, developers have little support for capturing tacit knowledge with existing programming environments.

### 3.3.2 CodePad Visualization and Interaction

A programmer in the 24-hour development cycle scenario wants an effective way to review recent changes, make comments in the context of the changes, and suggest what needs to happen next. Asking the task recipient to review the code difference is impractical because it provides no logical way to navigate or annotate the structure. To support this programmer, we provide an interactive code history visualization that allows the developer to send an *annotated code history* to a task recipient. We first show the programmer a temporal breakdown of code changes and navigations as they happened and then allow the programmer to select and annotate which relevant items should be sent to the task recipient.

To obtain a finer temporal breakdown, we record a work history from the programming environment that includes navigation or click events, change activity, and snapshots of code documents at every save and build. From this work history, we segment the events into coherent groups called *code episodes* by using marker events such as switching documents or performing a build. We render each code episode with a degree of interest over the code document (that is only include code lines involved in events) and decorate the document with details from the work history (bold clicked words, color code differences). When rendering the code episode, we also include the following contextual details: file name, line numbers, timestamp, and class and method signatures (even if not in the original degree of interest).

Figure 6 illustrates how a developer can create an annotated code history. On the CodePad, the developer can bring up a history visualization. The developer can quickly thumb through the history with scroll gestures (Figure 6(a)): The scroll speed varies with the speed of the scroll gesture and terminates with a decelerating inertia after release of the finger. For longer histories or as a way to reach past events quicker, a pinch gesture adjusts the granularity of the history in staggered levels of temporal and semantic zoom. The first level of zoom removes save episodes, the second level removes the document switching constraint for grouping, the third level of zoom only displays only the first episode and last episode per build event. The highest levels of zoom provides a short summary of daily and weekly activity. Once finding the relevant timeframe, the developer can zoom back into more detail by using the expand gesture.

Developers can grab an item of interest by dragging a code episode laterally onto the annotation side (Figure 6(b)). The annotation side serves as a place to hold and arrange interesting code episodes. At the annotation side, the developer can then rearrange the order of episodes (drag up or down), merge them together (two finger merge) or split up an episode (two finger slice) into smaller episodes. When a code episode is placed at the annotation side, a back arrow will appear that, when clicked, will allow the developer to scroll and focus the timeline to the episode position. Double tapping a code episode will focus the IDE on the selected code document. With a pen stylus, code can be circled, underlined, and annotated with general sketches or writing (Figure 6(c)).

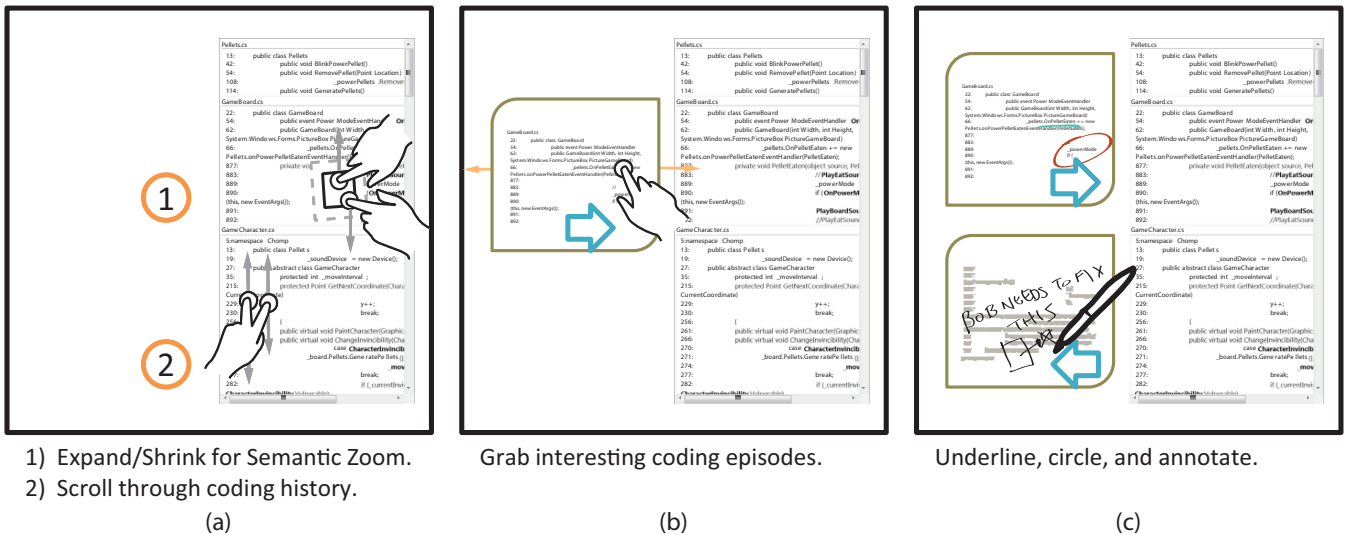


Figure 6: Workflow for creating an annotated change history using a CodePad.

With a few interesting code episodes extracted and annotated, the developer can send an exported html file or a task hand-off file to the task recipient. Task hand-off files can be loaded and used to navigate to locations in the IDE. Now, the task recipient has a better understanding of the task progress and remaining work. Table 3 shows a summary of gestures used in this scenario.

GESTURE	ACTION
	Merge two code episodes together
	Split a code episode

Table 3: Gestures supporting a task hand-off.

### 3.3.3 Discussion

More can be done with the annotated code history visualization. In our discussion with developers, code developers have mentioned other possible collaborative uses such as performing a peer review or asking a question, in addition, developers believed the annotated code history would provide an excellent resource for resuming a programming task that was interrupted or shelved. A stronger binding between code in the history and annotations could potentially feed into other visualizations (*e.g.*, display all references to underlined code). This speaks to a weakness of the current code history tool—the timeline may not hold code snippets that are of interest and must be separately fetched—but such interactions might alleviate this problem.

## 3.4 Reflection

We often still print out a paper on which to mark and annotate, leave paper sticky note reminders, or write on a whiteboard. The motivations vary from expressiveness to sociality of the medium, but what they have in common is interactions in a physical space. They also share the same failings: a static, fragile persistence aided by little more than strokes of ink. CodePads offer more than the supplantation of physical paper with digital paper: A CodePad can support multi-tasking by offering separate spatial contexts

for task content, annotating task goals and progress, maintaining simultaneous perspectives of code, and manipulating multiple code artifacts.

## 4. PROTOTYPE

### 4.1 Hardware

In our development with CodePads, we have used two devices: a Wacom Cintiq 21ux and an Acer Aspire 1420P. The Cintiq has a 21.3 inch LCD screen with a native 1600x1200 resolution. It has a swivel mount allowing the display to be adjusted to lay more flat or purely vertical, much like a drafting table or easel. Weighing in at 22 pounds, although movable, a developer is not likely to move this display often, but rather keep it as a dedicated CodePad. The pen provides 2048 distinct pressure levels and 60 degrees of tilt producing near pencil-like writing. Unfortunately, the current Cintiq does not support multi-touch; however, newer models are in production that do allow multi-touch interaction in addition to pen input.

The Acer Aspire 1420P is a multi-touch tablet notebook. The display is an 11.6 inch LED LCD with 1366x768 resolution. The size and weight (3.7 pounds/1.6 kilograms) best categorizes this device as a portable CodePad. The multi-touch display is still limited: Two simultaneous touch points can be tracked and the size of touch points is not exposed. We have performed most of our development on this device with support of the Windows 7 Multi-Touch API.

### 4.2 Implementation

We have detailed a vision of CodePads for altering how programmers interact with their software development environments. In exploring this vision, we have implemented a variety of features and prototypes; however, much work remains. Here, we present our current implementation and highlight some of our next steps.

Our main implementation goal has been to provide a development API and architecture for other researchers and developers to create CodePad applications. Our API centers around three components: an IDE interaction service provider, a service layer, and a gesture engine.



The IDE interaction service provider allows the CodePad to subscribe to events from the IDE or to send actions to the IDE. We have implemented providers for Visual Studio 2008 and 2010. Services include subscribing to the opening and closing of documents, and sending a document, method, or current selection to the CodePad. We also include services for collecting work history (IDE actions, save and build snapshots). To transfer items or send commands, we use an HTTP REST protocol running locally on the machine. We are currently in the process of providing an interface to refactoring commands.

The service layer provides access to resources such as code episode analysis for code history. In our services, we have limited support for semantic and temporal zoom: Besides Kim and Notkin’s work on systematic change summaries [10], little research has looked into how to create semantic summaries of coding activity. We are actively researching this topic in order to provide services for improving semantic and temporal zoom when browsing code history.

Our initial gesture framework used a naive implementation based on specifying matching line segments with relative thresholds on slope, length, and position. We distinguish a gesture from other possible gesture matches by maintaining a lattice of simultaneously possible gesture states. This implementation is not ideal; we are currently in the process of rewriting the gesture recognition framework to use a more robust template and projection scheme [14].

Improving the implementation and interactions of our visualizations is an ongoing effort. Error feedback and generation of batch refactoring is not yet developed. Bundling of navigation trails is still in development. We support creating an annotated code history; however, we are still in the process of supporting export and sharing. Finally, numerous improvements need to be made in providing more tactile feedback when manipulating content (*e.g.*, when extracting text, providing highlights around extracted text that makes clear when the text has been dragged enough to “break-free” from the document).

## 5. RELATED WORK

Helping developers maintain focus and concentration has received considerable attention from researchers (for a review of the cognitive issues involved in memory and attention for programmers see Parnin [19]). The pattern of research has pursued two goals. In the first goal, researchers have attempted to provide better links to relevant artifacts. In NavTracks [27], navigation loops are recovered from recent navigation paths, and the files related to the current method are displayed. In FAN [6], navigational history is analyzed to display a list of methods that are accessed next after visiting the current method. In our previous study of recommendation tools, we found that recency is the best predictor for recovering artifacts [21] (significantly beating any historical-based recommendation used by these systems), whereas predictive algorithms such as association rules are better for assisting exploration. Mylyn<sup>4</sup> uses the frequency of recent interaction to highlight and filter items from the document treeview to aid developers in identifying documents of interest.

Examples of research pursuing the second goal include Relo [28], which displays an abstracted representation of rel-

<sup>4</sup><http://www.eclipse.org/mylyn>

evant artifacts in a complementary UML view that is built from recently visited source code. SHriMP [29], uses a focus-and-context paradigm to allow multiple code locations to be viewed simultaneously. Code Thumbnails [5] represents source code documents in a SeeSoft-like [8] view and places the documents on a secondary monitor. An evaluation of the view found that comprehension was improved for participants that used the view in a coding task.

Code Canvas [7], an evolution of Code Thumbnails, seeks to maintain spatially stable locations for source code. In contrast to SHriMP, Code Canvas uses a semantic zoom and pan interface; depending on the zoom level, Code Canvas will vary what is displayed. For example, when the zoom level is too high to view the text of code, method names would still be visible, but when zoomed in further, more details would emerge. Instead of spatially stable documents, Code Bubbles [2] displays multiple code fragments (methods) on a virtual canvas. Initially starting with an empty canvas, Code Bubbles relies extensively on accumulating fragments by structural links (all references to a method) or search terms to place fragments onto the canvas. However, once fragments exist on the canvas, the navigation can occur by panning left and right.

Although some research has explored using other devices in support of software development, they have mainly focused on the design aspects, such as sketching UML [4]. Inside and outside of software development, research has explored how to use additional display devices to augment software development (which we previously reviewed in [22]). However, these monitors are either more for ambient awareness than for allowing direct interaction during active development, or have not focused on software engineering.

Few of these research developments have caught the attention of developers. We believe that many aspects about programming environments design work well: Often a programmer only wants to look at one large document and focus on coding within that space, and tabs work almost 70% of the time [21]. Also, most of the time, a treeview is a very quick way to collapse and jump through many documents (74% non-tab navigations occur with a treeview [15]). Other systems that provide visualizations or attempt to redesign the IDE make managing attention a first order process (programmers constantly pan, zoom, or filter within the programming environment to focus on what is relevant). In contrast, we make managing attention first class (real entities), but second order (not disturbing the main task) allowing these practices to recede into the background.

## 6. CONCLUSION

Understanding, navigating, and changing software systems still remains a daily challenge for software developers. Human capacity for attention and memory ground our designs; but poor design choices and increasing scale of development efforts grate against these capacities. The choices researchers have given developers so far is to choose between better links to software (while coding in a main document) or showing all the software (never escaping some form of view management). Both choices promise a better way for the developer, but either alternative seems to make a stark alignment toward just one aspect of development, when there are actually many.

Our motivation and argument is simple: Interaction and manipulation of physical space is a compelling but relatively

unexplored alternative to virtual space. Interactive displays of various sizes and form can greatly expand the mental workspace of developers. Pulling aside a particular item of interest can be far easier and less costly than managing everything as a fragment or hoping for an intelligent heuristic to provide access to the correct artifacts.

What may be ultimately flawed about software visualization is not the visualizations themselves but rather the interactions with them. At the heart of any visualization is interaction. Visualization systems, such as Code Bubbles or Code Canvas, provide wonderful visualizations but do not have a contingency plan for when interaction breaks down. We believe CodePad greatly expands how software visualization can be integrated into the daily activities of software developers in a way that spans the collaborative, exploratory, threaded, and interrupted nature of programming. CodePad does not supplant other visualizations—rather it provides an interactive space for managing and coordinating artifacts in the programming environment. For the developer, work will continue but with a little more space. For the researcher, numerous challenges and opportunities await.

## 7. REFERENCES

- [1] C. Andrews, A. Endert, and C. North. Space to think: large high-resolution displays for sensemaking. In *CHI*, pages 55–64, 2010.
- [2] A. Bragdon, R. Zeleznik, S. P. Reiss, S. Karumuri, W. Cheung, J. Kaplan, C. Coleman, F. Adeputra, and J. J. LaViola, Jr. Code bubbles: a working set-based interface for code understanding and maintenance. In *CHI*, pages 2503–2512, 2010.
- [3] 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*, pages 557–566, 2007.
- [4] R. Dachsel, M. Frisch, and E. Decker. Enhancing uml sketch tools with digital pens and paper. In *SOFTVIS*, pages 203–204, 2008.
- [5] R. DeLine, M. Czerwinski, B. Meyers, G. Venolia, S. Drucker, and G. Robertson. Code thumbnails: Using spatial memory to navigate source code. In *VLHCC*, pages 11–18, 2006.
- [6] R. DeLine, A. Khella, M. Czerwinski, and G. Robertson. Towards understanding programs through wear-based filtering. In *SOFTVIS*, pages 183–192, 2005.
- [7] R. DeLine and K. Rowan. Code canvas: Zooming towards better development environments. In *ICSE (New Ideas and Emerging Results)*, 2010.
- [8] S. G. Eick, J. L. Steffen, and E. E. Sumner, Jr. Seesoft—a tool for visualizing line oriented software statistics. *IEEE Trans. Softw. Eng.*, 18(11):957–968, 1992.
- [9] D. R. Hutchings, G. Smith, B. Meyers, M. Czerwinski, and G. Robertson. Display space usage and window management operation comparisons between single monitor and multiple monitor users. In *AVI*, pages 32–39, 2004.
- [10] M. Kim and D. Notkin. Discovering and representing systematic code changes. In *ICSE*, pages 309–319, 2009.
- [11] A. J. Ko, B. A. Myers, 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.*, 32(12):971–987, 2006.
- [12] H. Krasner, B. Curtis, and N. Iscoe. *Communication breakdowns and boundary spanning activities on large programming projects*. Ablex Publishing Corp., Norwood, NJ, USA, 1987.
- [13] T. D. Latoza, G. Venolia, and R. Deline. Maintaining mental models: a study of developer work habits. In *ICSE*, pages 492–501, 2006.
- [14] Y. Li. Protractor: a fast and accurate gesture recognizer. In *CHI*, pages 2169–2172, 2010.
- [15] G. C. Murphy, M. Kersten, and L. Findlater. How are Java software developers using the Eclipse IDE? *IEEE Softw.*, 23(4):76–83, 2006.
- [16] E. Murphy-Hill and A. P. Black. Breaking the barriers to successful refactoring: observations and tools for extract method. In *ICSE*, pages 421–430, 2008.
- [17] E. Murphy-Hill and A. P. Black. Refactoring tools: Fitness for purpose. *IEEE Softw.*, 25(5):38–44, 2008.
- [18] E. Murphy-Hill, C. Parnin, and A. P. Black. How we refactor, and how we know it. In *ICSE*, pages 287–297, 2009.
- [19] 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.
- [20] C. Parnin and R. DeLine. Evaluating cues for resuming interrupted programming tasks. In *CHI*, pages 93–102, 2010.
- [21] C. Parnin and C. Görg. Building usage contexts during program comprehension. In *ICPC*, pages 13–22, 2006.
- [22] C. Parnin and C. Görg. Design guidelines for ambient software visualization in the workplace. In *VISSOFT*, pages 18–25, 2007.
- [23] M. G. Petersen. Interactive spaces: towards a better everyday? *Interactions*, 12(4):44–45, 2005.
- [24] I. Safer and G. C. Murphy. Comparing episodic and semantic interfaces for task boundary identification. In *CASCON*, pages 229–243, 2007.
- [25] O. Shaer and E. Hornecker. Tangible user interfaces: Past, present, and future directions. *Found. Trends Hum.-Comput. Interact.*, 3(1–2):1–137, 2010.
- [26] J. Sillito, K. De Volder, B. Fisher, and G. Murphy. Managing software change tasks: an exploratory study. *Empirical Software Engineering, 2005. 2005 International Symposium on*, page 10 pp., nov. 2005.
- [27] J. Singer, R. Elves, and M.-A. Storey. Navtracks: Supporting navigation in software maintenance. In *ICSM*, pages 325–334, 2005.
- [28] V. Sinha, D. Karger, and R. Miller. Relo: Helping users manage context during interactive exploratory visualization of large codebases. In *VLHCC*, pages 187–194, 2006.
- [29] M.-A. Storey, C. Best, J. Michaud, D. Rayside, M. Litoiu, and M. Musen. Shrimp views: An interactive environment for information visualization and navigation. In *CHI (Extended Abstracts)*, pages 520–521, 2002.