

Mobile Devices in Programming Contexts: A Review of the Design Space and Processes

Poorna Talkad Sukumar
University of Notre Dame
Notre Dame, USA
ptalkads@nd.edu

Ronald Metoyer
University of Notre Dame
Notre Dame, USA
rmetoyer@nd.edu

ABSTRACT

"What design innovations can the ubiquity and features of mobile devices bring to the programming realm?" has been a long-standing topic of interest within the human-computer interaction community. Yet, the important design considerations for using mobile devices in programming contexts have not been analyzed systematically. Towards this goal, we review a sample of the existing research work on this topic and present a design space covering (i) the target contexts for the designed programming tools, (ii) the types of programming functionality supported, and (iii) the key design decisions made to support the programming functionality on the mobile devices. We also review the design processes in the existing work and discuss objectives for future research with respect to (i) the trade-offs in enabling programming support given the constraints of mobile devices and (ii) applying human-centered methods particularly in the design and evaluation of programming tools on mobile devices.

Author Keywords

Programming; touchscreen; source code; IDE; smartphones; tablets; mobile devices; software development; survey;

CCS Concepts

•General and reference → Surveys and overviews;
•Human-centered computing → Touch screens; Interaction design process and methods; •Software and its engineering → Software development methods; Please use the 2012 Classifiers and see this link to embed them in the Tex: https://dl.acm.org/ccs/ccs_flat.cfm

INTRODUCTION

The confluence of human-computer interaction (HCI) and programming exists to improve the design of programming tools through the use of HCI methods because "programmers are users too" [45]. In the same vein, we look at the programming tools designed for use on mobile devices from an HCI perspective to capture their design rationales and provide a road map for further exploration.

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. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

DIS '19, June 23–28, 2019, San Diego, CA, USA

© 2019 Copyright held by the owner/author(s). Publication rights licensed to ACM. ISBN 978-1-4503-5850-7/19/06...\$15.00

DOI: <https://doi.org/10.1145/3322276.3322323>

Mobile touchscreen devices, with their ubiquity and unique features, make new programming avenues possible. For example, mobile devices are being considered for use in classrooms for computer science programming courses [28, 68] and in combination with desktops for collaborative and co-located programming tasks such as code review and pair programming [12, 36, 52]. Furthermore, mobile phones will be the *sole* computing devices available to a majority of the population in many emerging countries where desktops and laptops are far less prevalent; hence creating opportunities for designing programming applications for small screens [17].

Prior work done in this area, however, is somewhat sparse and consists of isolated designs. There is a lack of connectedness and coherency in the existing work and very few articles appear to build on previous designs.

The variety in the prior work could be in part due to the numerous possibilities and variables in using mobile devices for programming. Each of the previous articles can be seen as focusing on one facet of this problem and designing, for example, for a particular type of mobile device, context, and set of programming tasks. We also suspect the pessimism surrounding the perceived drawbacks of mobile devices for programming, namely, the absence of a physical keyboard, small screen space, and limited processing capabilities [2, 58, 61], to have hindered the research advances in this direction.

In this paper, we review the diverse existing work to structure the important design considerations to both guide practitioners and encourage research efforts in this area. In addition to presenting the design space, we review the design processes and methods in the papers and discuss (i) making trade-offs between the *complexity* of the enabled programming functionality and their *usability* on the mobile devices and (ii) employing HCI methods in addition to and also different from those used in traditional software development for designing programming tools for use on mobile devices. Although on a different topic, this paper adopts the method and structure of the paper by Rasmussen et al. [59] presenting a design space of shape-changing interfaces.

SCOPE OF THE REVIEW

This topic has been addressed in various ways in the existing work. For example, some papers have looked at using mobile devices primarily for programming and have focused on suitable programming languages and design tactics towards this goal [19, 23, 38, 68]. Some others make use of mobile devices

to provide auxiliary functionality in larger interactive programming environments [12, 52]. Context-driven approaches are also used wherein mobile-device-based programming tools are designed for particular target users and settings [25, 28, 38, 61]. There are also papers whose relevance to the topic is not directly apparent, for example, by being visual-programming-based [14] or entailing a modest programming functionality such as code annotation [36]. This paper coalesces and organizes the key design decisions in these different papers.

We review papers that discuss the use of mobile touchscreen devices enabling direct touch and/or pen input, i.e. smartphones and tablets, in programming contexts. We also discuss the touch-based programming support designed on tablet PCs that do not require the use of a physical keyboard. Our review is based on 18 papers (with multiple papers covering certain system designs) and additional example papers illustrating visual programming and code annotation concepts. We also draw on extensive literature on programming environments and mobile interaction in general.

We found the relevant papers by searching in the ACM digital library, IEEE Xplore digital library, and Google Scholar and by using snowballing, i.e. using forward and backward citation tracking from the selected papers. The collection of papers is included in Table 8.

We structure the design space by analyzing the following characteristics:

- *Context Choices*: who are the target users and what are the target physical or social settings for the programming functionality designed?
- *Types of Programming Functionality*: what types of programming functionality have been addressed?
- *Design Decisions*: what are the key design decisions made to enable the programming support on the mobile devices?

We identify these three characteristics to represent the main design questions for this topic. We abstract from the papers and categorize the design alternatives for each of these characteristics in sections 3, 4, and 5, respectively. We refer the readers to the respective papers to find more details on the rationales around individual designs.

Tables 4, 5, and 6 each consider a pair of characteristics to further illustrate how they interact with one another and are populated with references that incorporate the categories corresponding to each column and row. Unfilled cells in these tables present "research gaps" or combinations that are yet to be explored.

CONTEXT CHOICES

This section presents an overview of the target contexts found in the reviewed papers. The capturing of design rationales for artifacts ideally proceeds from recording their contexts of use which largely determine the usability of the artifacts and hence can be a valuable guide to practitioners [16](Ch. 6). However, the target contexts were not well articulated in many of the reviewed papers. We discuss three predominant context

choices and present a summary of these context choices with examples in Table 1.

Learning environments: Smartphones and tablets are increasingly being considered for use in teaching environments for computer science courses. Applications have been developed for use on mobile touchscreen devices to support college students in learning new programming languages [25, 28, 38]. Stylus-based *static* code annotations supported on tablets facilitate classroom interaction [27], lecture presentation [4, 18, 27, 41], and code review and grading [27, 41, 55].

Visual-programming-based environments, similar to and inspired by those commonly used in learning contexts for children and youngsters (such as *Scratch* [60] and *Blockly* [20]), have been implemented or adapted for use on smartphones and tablets; examples include *ScratchJr* [65], *MakeCode* [15], *YinYang* [40] and *Catroid* [64].

Professional programming environments: We observed a clear delineation of professional programmers in the reviewed papers. Professional programmers are seen to consist of those who program or write software extensively. They are also defined as those with a computer science degree or equivalent [45]. Users belonging to this category are seen as being less likely to use tablets or smartphones as their *primary* devices for programming. Hence mobile devices are mostly used to provide auxiliary (such as task-specific functionality [52] or code annotation in integrated development environments (IDEs) [56, 66]) and collaborative programming support [12, 36] in their work environments.

Generic programming: This category applies to many types of programmers including students, hobbyists, mobile application (app for short) developers, end users, and people who program for fun or as a small part of their work. Mobile devices, which are present with users at almost all times, are seen as good candidates to support *self-contained* programming anywhere and for simple program entry and editing tasks [19, 61, 67].


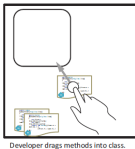

Learning environments	Professional programming environments	Generic programming
Learning applications; classroom use	Auxiliary support, e.g., programming tasks; collaborative programming	Self-contained programming for end-users, students, mobile app developers
 MakeCode [15]	 CodePads [52]	 TouchDevelop [39]

Table 1. Context choices.

TYPES OF PROGRAMMING FUNCTIONALITY

We discuss the programming functionality addressed in the papers under six categories and present a summary of these programming-functionality types with examples in Table 2.

App development on mobile devices: App development environments have been developed on smartphones and tablets for enabling the creation of customized apps using the device resources and for deployment on the devices themselves. For example, *TouchDevelop* [67] is tailor-made to use the phones’ content, sensors, and features to create apps. Newer versions of *TouchDevelop* can be used on a variety of devices including tablets [7]. Similarly, *MobiDev* [53, 61] enables the creation of web-based mobile apps using only smartphones. App-debugging capabilities are also enabled on smartphones [48] since testing and debugging mobile apps in general are complex due to emulators lacking necessary features of mobile devices [26, 71].

Programming in specific languages: There are papers focusing on providing programming support in particular languages on mobile devices. Examples include the proposed programming environment for the *Factor* language [23] and the support provided to program in *Java* [2, 38].

Visual-programming-based applications: Visual Programming (VP) languages enable programming by means of interacting with graphical elements, such as blocks, symbols, and arrows, rather than text. We found several examples of VP-based environments on mobile devices [14, 15, 21, 25, 28, 64, 65]. VP-based applications are predominantly used either in educational or end-user contexts.

Concurrent and collaborative programming: Auxiliary and collaborative support are provided by multiple interconnected mobile devices used as part of larger programming environments. For example, in *CodePads* [52], mobile devices augment desktops to enable the distribution of programming artifacts on the devices which can be simultaneously manipulated using the task-specific functionality the devices come equipped with. In *Code Space* [12], mobile devices facilitate collaborative tasks and sharing of content among developers’ laptops and a large shared display during co-located developer meetings.

Support for programming tasks: Support for programming tasks, such as refactoring, has been designed on tablet devices by means of touch- and pen-based gestures [6, 11, 52, 58]. For example, in *CodePads* [52], task-specific functionality, such as gestures for performing refactoring, navigating documents, and recording code histories, is designed for use on tablet devices.

Aids for code comprehension: Tablet devices have been used to provide supplemental programming support in the form of aids for code comprehension and problem solving. These aids include pen-based *dynamic* annotations (the annotations remain attached to the respective code segments despite dynamically changing code) in the IDE [36, 54, 66] and linking camera images containing (brainstormed) sketches to code in the IDE [9]. In *CodeGraffiti* [36], code annotation and sketching on a peripheral tablet is used to enable pair-programming,


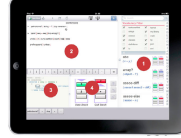
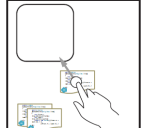



App development on mobile devices	Programming in specific languages	Support for programming tasks
Creating general-purpose apps on and for mobile devices	Programming in Factor, Java	Support for refactoring, navigation, recording code histories
 TouchDevelop [39]	 Touching Factor [23]	 CodePads [52]
Visual-programming based applications	Concurrent and collaborative programming	Aids for code comprehension
Programming with graphical elements for beginners and end-users	Sharing and concurrent manipulation of artifacts; collaborative programming tasks	Code annotation; linking sketches to code
 MakeCode [15]	 Code Space [12]	 CodeGraffiti [36]

Table 2. Types of Programming Functionality.

where two programmers collaborate to solve programming tasks.

DESIGN DECISIONS

In surveying the design decisions in the papers, we observed interaction mechanisms employed to overcome the drawbacks and leverage the interaction affordances of mobile devices. We discuss these mechanisms under eight categories and present a summary of these design decisions with examples in Table 3.

VP-like interactions: VP may be a somewhat natural fit for use on touchscreen devices because it inherently leverages their interaction style and makes minimal or no use of the keyboard [22, 29]. VP elements, such as blocks, arrows, and other graphical symbols including the typical user interface (UI) components on mobile devices, are easily manipulable on mobile touchscreen devices. Hence VP-like interactions are used even in text-based programming interfaces designed on mobile devices [25, 28, 61].

For example, block-based programming exercises in *Python* are designed on mobile devices to help computer science students learn programming [25, 28]. *MobiDev* [61] provides a VP-like means to create the front-end of mobile apps by

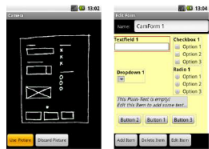

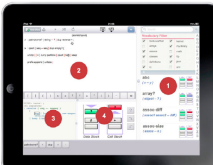

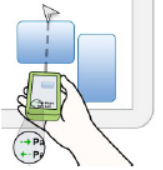


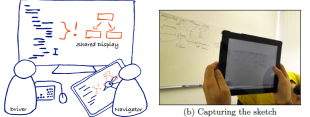
Visual-programming-like interactions	Structured editing	Programming-language-centered designs	Alternatives to the on-screen keyboard
Touchscreen-friendly VP-like interactions used even for text-based programming	Structured editing enabled by means of built-in constructs and adaptive keypads for text-based programming	Finding language characteristics (such as paradigm, syntax and semantics) to facilitate their use on mobile devices	Use of alternative methods for text entry/editing including speech, adaptive keypads, and gestures
 <p>MobiDev [61]</p>	 <p>TouchDevelop [39]</p>	 <p>Touching Factor [23]</p>	 <p>Custom keyboard for Java [2]</p>
Multi-device interactions	Gestures for programming tasks	Compact code representations	Multimodal interactions
Leveraging the connectedness of multiple mobile devices for collaboration and sharing of programming content	Tasks, performed using menus and hotkeys on IDEs, supported through touch- and pen-based gestures	Code represented using space-conserving techniques such as code folding, transparent overlays, and visual abstractions	Leveraging mobile-device modalities, e.g., annotation and camera and speech input
 <p>Code Space [12]</p>	 <p>RefactorPad [58]</p>	 <p>Deverywhere [19]</p>	 <p>CodeGraftiti [2] SketchLink [9]</p>

Table 3. Key design decisions for using mobile devices for programming.

selecting and arranging UI widgets using drag-and-drop while requiring the back-end code to be typed in.

Structured editing: The syntax-directed or structured editing enabled by adaptive keypads and templates for program constructs in *TouchDevelop* [7, 67] and the custom *Java* keyboard [2] provide a middle ground between the space-inefficient VP [44] and editing of "raw text" as done on desktops. They also leverage selection using finger taps, which is the most familiar touchscreen interaction known to users and also considered as a unitary act [72].

Compact code representations: We found design decisions concerning representing code compactly on the space-constrained mobile devices. These decisions include both mechanisms to display code in its actual "raw text" form [48, 52] as well as changing its intrinsic appearance [19]. Design tactics for the former include using expandable fields i.e. code folding, transparent overlays displaying additional related content [28, 48], and switching between low- and high-fidelity representations of the program content depending on the task being performed [52]. For example, low-fidelity representations are used for navigating through the programming artifacts whereas high-fidelity representations are used for manipulating the program content [52].

Design suggestions for the latter mechanism gleaned from *Deverywhere* [19] include using different colors, fonts, and symbols to distinguish among the program elements, vertical text boxes to show names and extents of functions, background colors to denote scopes, and circular watermarks to denote loops. Additionally, bold horizontal lines are used as delimiters and some elements such as braces, types, and certain keywords, such as `then`, `else`, `class`, `return`, are omitted by making their connotations apparent in the code graphically.

Programming-language-centered designs: We found design decisions centered on programming languages in the surveyed work which include selecting languages that are mobile-device-friendly [23] and leveraging characteristics of languages to facilitate their use on mobile devices [2, 19].

For example, a tablet-based development environment has been designed for *Factor*, a concatenative programming language, since it has a minimalistic and compact representation, unlike imperative programming languages which are text heavy [23]. *Java*'s syntax and semantics are leveraged through the design of a custom keyboard [2] and templates for voice input [19] to enable efficient code entry.

		App-development on mobile devices	Programming in specific languages	Visual-programming-based applications	Concurrent and collaborative programming	Support for programming tasks	Aids for code comprehension
	Learning environments	[7], [53], [61], [67]	[2], [25], [28], [38], [67]	[21], [25], [28], [64], [65]			
	Professional programming environments				[12], [36], [52]	[6], [11], [52]	[9], [36], [54], [66]
	Generic programming	[7], [48], [53], [61], [67]	[2], [19], [23], [38]	[14]		[58]	

Table 4. Combinations of the context choices (rows) and types of programming functionality (columns) found in the reviewed papers. Unfilled cells present "research gaps" or combinations that are yet to be explored.

		VP-like interactions	Structured editing	Compact code representations	Programming-language-centered designs	Multi-device interactions	Alternatives to the on-screen keyboard	Gestures for programming tasks	Multimodal interactions
	Learning environments	[25], [28], [61]	[2], [67]	[28]	[2]		[2], [67]		[61]
	Professional programming environments			[52]		[12], [36], [52]		[6], [11], [52]	[9], [36], [52], [54], [66]
	Generic programming	[61]	[2], [67]	[19], [48]	[2], [19], [23]		[2], [19], [58], [67]	[58]	[19], [58], [61]

Table 5. Combinations of the context choices (rows) and design decisions (columns) found in the reviewed papers. Unfilled cells present "research gaps" or combinations that are yet to be explored.

		VP-like interactions	Structured editing	Compact code representations	Programming-language-centered designs	Multi-device interactions	Alternatives to the on-screen keyboard	Gestures for programming tasks	Multimodal interactions
	App-development on mobile devices	[61]	[67]	[48]			[67]		[61]
	Programming in specific languages		[2]	[19], [28]	[2], [19], [23]		[2], [19]		[19]
	Visual-programming-based applications	[25], [28]		[28]					
	Concurrent and collaborative programming			[52]		[12], [36], [52]		[52]	[52]
	Support for programming tasks			[52]		[52]	[58]	[6], [11], [52], [58]	[52], [58]
	Aids for code comprehension					[36]			[9], [36], [54], [66]

Table 6. Combinations of the types of programming functionality (rows) and design decisions (columns) found in the reviewed papers. Unfilled cells present "research gaps" or combinations that are yet to be explored.

Multi-device interactions: Interconnected mobile devices are used either by themselves or in combination with conventional programming setups to provide collaborative programming support, such as pair-programming [36] or sharing of program content [12], or to enable distribution of programming tasks as in *CodePads* [52].

Alternatives to the on-screen keyboard: Using the soft keyboard on mobile devices can be cumbersome and it takes up a significant portion of the already limited screen space. Design decisions which reduce or eliminate the need for using the on-screen keyboard include the use of voice input for program entry [19], adaptive keyboards designed to easily input the needed program primitives and constructs [2, 67], and gestures designed for text editing [58].

Gestures for programming tasks: Touch- and pen-based gestures have been formulated and/or elicited for programming tasks, such as refactoring [6, 11, 52, 58]. These programming tasks are typically performed using the long menus or keyboard shortcuts in IDEs. For example, gestures designed in *CodePads* [52] for refactoring purposes include drawing a square for creating a new placeholder class into which methods can be dragged, a "check" gesture to test if a refactoring can be performed, and a "flick" gesture to perform the refactoring and apply it in the IDE.

Multimodal interactions: Certain mobile-device modalities have been leveraged in programming contexts for typical uses as well as to provide novel functionality. Pen input is often used as digital ink enabling annotation [36, 52, 54, 66] or for performing touch-like gestures because of their greater precision [58] or even for handwriting code [74]. Speech is intended for inputting source code [19] and camera (or image) input is used both to link images containing sketches to code in the IDE [9] and for building the graphical front-end of mobile apps [61].

In *Code Space* [12], hybrid "touch + air" gestures are used, with touch gestures performed on developers' personal devices and the in-air gestures sensed by two Microsoft Kinect sensors. The gestures enable interactions between the developers' devices and a multitouch shared display during developer meetings, e.g., using a mobile device to point and manipulate digital objects on the display.

A REVIEW OF THE DESIGN PROCESSES

Design involves making decisions among various options and the last three sections present a post hoc structuring of the design options in the reviewed papers to help practitioners incorporate these ideas into their own designs. The following two sections highlight aspects of the design processes themselves in the reviewed papers which led to their respective design decisions and based on which we draw important considerations for designing such tools in the future. These aspects include the tradeoffs made and the HCI methods used in the various stages of the design process [16](Ch. 5).

"Programming on mobile devices" combines the disciplines of software development and mobile interaction. These disciplines can be said to conflict with one another considering the elaborate traditional programming practices on laptop/desktop

systems and the apparent inability of mobile devices to adequately support these practices. Hence any programming tool being designed for use on mobile devices has to take this conflict into account and make trade-offs between the tool's programming *complexity* and its *usability* on mobile devices.

Furthermore, while substantial research exists on the use of suitable HCI methods to improve the design of developer tools [24, 45], these methods may be insufficient for the design of programming tools on mobile devices. Designing and evaluating mobile interactions require methods different from those used in the traditional software development domain which are strongly tied to the WIMP paradigm.

TRADE-OFFS: COMPLEXITY VS. USABILITY

Design is a decision-making process and inherently involves making trade-offs to accommodate goals within the specified constraints [16](Ch. 5). In other words, design entails making sub-optimal choices for one aspect so that another can be met. While design can also produce radical solutions satisfying many conflicting constraints as suggested in the section on design decisions, we note the trade-offs made in the reviewed papers in this section. Specifically, we summarize the decisions made where the *usability* of the designed tools and techniques is prioritized limiting the *complexity* of the enabled programming functionality and vice versa and then discuss their implications.

Usability > Complexity

Here we list examples where limited or less complex programming support is enabled as a result of prioritizing usability and facilitating their use on the mobile devices.

Built-in code blocks used in *TouchDevelop* [67] and the *Java* learning application [38] require users to enter/edit code in a certain order limiting their "flow" and flexibility. Concatenative languages such as *Factor* have been chosen over imperative languages for use on mobile devices because they are less text-heavy [23]. We observed that mobile devices are used in conventional programming environments only to provide auxiliary support [52] and/or to provide indirect support through gestures and annotation [11, 12, 36, 54, 66]. In particular, mobile touchscreen devices are incorporated into programming environments in *Code Space* [12] and the papers enabling aids for code comprehension [9, 36, 54, 66] solely to leverage their interaction affordances and do not enable programming per se.

While VP may be more suitable for use on touchscreen devices, they are known to use screen space inefficiently when displaying more content or complex program structures [44, 47]. Hence mobile applications based on VP have generally been developed for use either in educational or end-user contexts [14, 21, 25, 28, 40, 64, 65] since they may not be well-suited for enabling complex programming applications.

Complexity > Usability

Here we list examples of cumbersome or sub-optimal interactions resulting from enabling the desired (complex) programming functionality.

MobiDev [61] requires the entire code for the back-end of the apps being created to be typed in using the on-screen keyboard which can be both cumbersome and time-consuming. In *Deverywhere* [19], the authors intend for users to program on mobile devices mainly using voice input which may not be an ideal choice for all situations, especially those where the users are in the presence of other people or noise.

We have seen several new input techniques designed, such as the keyboard extension for *Java* [2], speech templates for entering code in *Deverywhere* [19], gestures for refactoring tasks [6, 11, 58], and task-specific functionality and gestures in *CodePads* [52]. These new input techniques will have to be learned and the papers do not sufficiently address the usability and learnability aspects of their proposed techniques.

Discussion

Even if limited programming functionality is supported by mobile devices, they can still be useful in certain contexts, such as for learning and end-user programming. However, sub-optimal or cumbersome interaction mechanisms can render the tools unusable. *Hence usability of the tools should be prioritized in the design processes.*

There is an inherent assumption that touchscreens are "natural" user interfaces compared with the WIMP paradigm and gestures are a "natural" means of interaction; however, touchscreens and gestures by themselves are not natural [50]. New touchscreen input techniques such as gestures and speech and keyboard input have to be designed carefully ensuring that they are learnable, intuitive, memorable, and guessable as well as standardized over time to be usable. For example, while it can be time-consuming to parse the linear menus of IDEs, the menus are designed to make all the actions discoverable and learnable through exploration whereas the utility of gestures predominantly lies in how easy they are to remember [50].

Additionally, design principles concerning the usability differences between novice and expert users (i.e. knowledge in the world vs. knowledge in the head) will have to be incorporated so that the system is both easy for novices to use and allows experts to be more efficient [49](Ch. 3). Novice experience, especially, is considered critical to the successful usage of new input methods since users may be reluctant to expend the effort required to become experts if they are dissatisfied with their initial experiences with these methods [37].

APPLYING HUMAN-CENTERED METHODS

Involving users right from the beginning of and throughout the design process is crucial and not only leads to the development of more usable designs but also has other benefits such as being more economical and producing new insights [16](Part 2). We observed a general scarcity in the application of HCI methods in the reviewed papers and many of them only present design sketches or prototypes and don't include evaluation studies. This failing to adequately capture user needs and experiences makes it difficult to not only ascertain the usefulness and usability of the individual designs but also those of programming on mobile devices as a whole.

We summarize below the HCI methods that have been used in the papers under four categories, namely, user requirements, design, evaluation, and long-term use, and then present a discussion. We use the terms HCI and human-centered methods interchangeably and these refer to any of the several methods, drawn from HCI or other fields such as the social sciences, describing the involvement of users in different stages of the design-thinking process. A summary of the HCI methods used in the reviewed papers is presented in Table 7.

User requirements

Understanding user needs is a key aspect of interaction design. This stage typically employs practices such as contextual inquiry [10] where users are observed in their actual work environments to understand what they do currently and to formulate design goals.

Numerous exploratory studies have been done to study the behaviors of programmers and the functionality and features of conventional IDEs [8, 13, 34, 43, 51, 62, 70]. These studies, by making apparent the shortcomings in and opportunities for new functionality in conventional programming setups, have served as starting points for much of the existing work proposing the use of mobile devices in these environments.

These studies report, for example, that programmers spend more time reading and comprehending code as part of maintenance tasks than they do writing code [31]. It has also been observed that programmers often use elements external to the IDE such as paper or whiteboards for taking notes or drawing sketches to complete programming tasks both when working independently and during tasks involving collaboration [13, 34, 51]. While traditional IDEs provide support for tasks such as refactoring by incorporating numerous tools accessible through long linear menus or hotkeys, it has been observed that programmers are usually either unaware or do not make optimal use of many of those tools [62] and additionally, using these tools is a slow process for novice and expert users alike [43].

These findings have been the basis for the design goals in the reviewed papers that focus on augmenting the IDEs with mobile touchscreen devices [52], formulating gestures for refactoring [11, 58], and providing annotation and code comprehension support [9, 36, 54, 66].

Design

After understanding user needs or investigating a problem, designers typically brainstorm ideas and HCI methods are used in this stage for assessing and obtaining feedback on the early design ideas and prototypes. We found three such methods used in the reviewed papers.

A focus group consisting of computer science students was conducted to gather user perceptions on the concept developed for *MobiDev* [53]. A formative study was performed with 9 professional developers to assess the concept and gestures formulated in *Code Space* [12] to inform design changes and improvements to the system. For example, among the "touch + air" gestures devised in *Code Space* [12], the participants did not find the gestures for peer-to-peer content transfer, where

HCI methods	Stage in the design process	Purpose	Benefits to mobile-device-programming
Contextual Inquiry	User requirements	Observations and interviews conducted in the work contexts of users to understand their work habits and practices	Understanding programmer practices in conventional setups can help in identifying uses for mobile devices in these setups
Exploratory lab studies	User requirements	Observing users in the lab as they perform given tasks	Understanding programmer approaches and their procedural knowledge to perform tasks can suggest ways to enable such task-support on mobile devices
Focus group	Design	Gather user perceptions about an initial concept developed	Usefulness and user attitude feedback can be obtained for programming application ideas for mobile devices from "target" users
Formative study	Design	Preliminary evaluation of prototypes to see if they are acceptable and if they can be improved	Useful especially for gestures and new input methods designed to assess how acceptable and suitable they are and make changes to designs accordingly
Gesture elicitation study	Design	Elicit "intuitive" and "memorable" gestures from users	Useful for designing gestures for programming tasks considering both mobile device interaction and logic of programming tasks
Think-aloud usability study	Evaluation	Observing how participants actually use a system to obtain details of participant interactions and uncover usability issues	Useful for gleaning user mental models and the <i>learnability</i> of the proposed programming applications
Quantitative empirical methods	Evaluation	Controlled experiments to evaluate designs and test hypotheses by measuring objective and subjective attributes of participant behaviors	Useful for comparing participant performances and preferences for proposed mobile device applications with existing alternatives
Log analyses	Long-term use	Analyzing large quantities of data gathered from real world use of a system to study usage patterns	Useful for studying user behaviors with programming applications affording the authoring and generation of custom code
Longitudinal study	Long-term use	The interactions of one or more users with a system are studied periodically over several weeks or months	Useful for assessing the learning progression for gestures and new input methods

Table 7. A summary of the HCI methods used in the reviewed papers, the stage in the design process where they are generally applied, their purpose, and the benefits they can bring to the domain of mobile-device-programming. The highlighted methods are also used in the traditional software development domain [45].

they were required to point at a colleague, to be socially acceptable. This implies that a less obtrusive mechanism was more preferable.

Gesture elicitation studies, such as the one employed in *RefactorPad* [58] to build a user-defined gesture set for refactoring tasks, provide a means to capture users' thinking and preferences. The user preferences are used to inform the design choices for the system at hand and hence may result in more intuitive and memorable interaction decisions. For example, the gestures elicited in *RefactorPad* showed that more participants preferred to use the pen for performing the gestures and devised multitouch gestures for only a few tasks. Additionally, their interaction behaviors appeared to be influenced by the mobile operating systems (*Android* or *iOS*) they were most familiar with.

Evaluation

Evaluation typically consists of empirical studies with target users using tasks reflective of real-world activities. We found both informal evaluations employing think-aloud protocols

to gain insights into participant interactions as well as more formal evaluations using standard measures and statistical methods. The participants in most of these studies were made up of computer science students.

An evaluation using think-aloud with 8 participants was done by Biegel et al. [11] to compare the gestures formulated for refactoring with their IDE counterparts. They found that most of the participants did not remember the shortcuts for the tasks and spent more time to find the respective commands in the IDE menu but they were able to quickly learn and use the gestures on the touch devices.

Evaluation studies comparing proposed tools with alternatives using formal measures have also been conducted. The measures include binary task completion and completeness for tasks containing programming exercises [38], task completion times for app-creation tasks requiring program copy and creating predefined UIs [61], and number of keystrokes, words per minute and error rate for program entry/copy tasks [2].

In addition to the objective measures, subjective measures using Likert scales [61] and the NASA TLX questionnaire [2] and informal participant impressions and preferences [38, 61] have also been collected. The subjective feedback provides additional insights into the usability of the tools. For example, in the evaluation of *MobiDev*, although participants generally took longer to create apps by sketching the UI compared with arranging widgets on the screen, they preferred the sketching method stating that it was "fun" to work with [61].

Long-term use

Log analyses provide a means to study user behavior and the long-term usage patterns of the system [45]. Among the reviewed papers, we found that a longitudinal study has been done for the *Java* keyboard extension [3] and extensive log data has been collected for *TouchDevelop* [5, 7, 35].

The longitudinal study, was conducted mainly to gauge the learning progression for the *Java* keyboard extension [3] by collecting program input measures for ten participants for eight sessions spanning over two weeks. Thousands of *TouchDevelop* scripts have been gathered and analyzed to study various usage details, including the number of users and scripts generated per week, functional purposes and size of the created scripts, platforms used (mobile/browser), and code reuse ratios [5, 7, 35]. The types of users and their behaviors are also gleaned from the log analyses. For example, it was found that most users were novices and were active initially and later left or became less active [35].

Discussion

The use of HCI methods in the design of programming tools and even languages is important and a failing to do so may result in languages and tools that are not useful, usable, or learnable [45]. Much research has focused on identifying HCI methods that are particularly applicable in the software-development domain to serve various purposes including uncovering problems frequently faced by developers, evaluating new tools or features, and designing programming languages themselves [31, 32, 33, 45, 46]. *Can these methods also be employed in the design of programming tools on mobile devices and is there a need for using different and new methods as well?*

The HCI methods commonly used in the software development domain [45] can also be applied in the design of programming tools on mobile devices as we have seen in the reviewed papers (the highlighted methods in Table 7) for purposes such as uncovering developer problems and usability evaluations of the proposed tools. However, the interaction mechanisms of mobile touchscreen devices are *fundamentally different* from those of conventional laptop/desktop-based programming setups and these mechanisms primarily contribute to the use of the programming tools designed on them as outlined in the section on trade-offs. *As such, this domain also requires the use of HCI methods specific to mobile interactions and tailored for different types of users and possibly new methods for the design of languages and tools intended for use on mobile devices.* We discuss these categories of methods below.

HCI methods for mobile interactions:

The gesture elicitation study in *RefactorPad* [57], for example, draws from that described by Wobbrock et al. [73] which is predominantly concerned with touchscreen-interaction design. Similarly, the usability of the new program-entry techniques proposed, such as the *Java* keyboard extension [2], should ideally be evaluated using the methods and measures typically used for evaluating mobile text-entry techniques [37]. The evaluations should consider factors including *focus of attention* (FOA), qualitative vs. quantitative measures, the trade-off between speed and accuracy and comparison of performance between novices and experts [37].

The evaluations of text-entry techniques also distinguish between "text-creation" and "text-copying" tasks [37]. Programming is largely a "text-creation" task and entails behaviors such as taking time to think and the programs entered may not be well distributed in terms of characters and words used in the programming language. While "text-copy" tasks, as used in the reviewed papers [2, 61], are not reflective of typical usage, "text-creation" tasks make it hard to measure the aforementioned factors or generalize the findings.

HCI methods tailored for specific user types:

Methods suited to understand the requirements of specific programmer types should also be employed when designing programming tools on mobile devices. As we have seen, the proposed tools and techniques in this review are intended for various types of users including students, end-users, and professional programmers. These user types typically have different goals, expertise, and requirements. For example, while professional programmers generally program for others, end-users mostly program for themselves and hence the requirements of end-users can be more easily understood [30]. It is important to gather the needs of as well as evaluate with the target user population and overcome the tendency to use student participants to evaluate tools [63] designed for different target users.

New HCI methods for eliciting programming requirements specific to mobile devices:

HCI methods have also been used or developed for designing programming languages, e.g., natural-programming elicitation [46]. However, the interfaces or devices intended for the use of the languages will have a bearing on their designs. For example, conventional programming-language designs can be said to have strong affinities with the WIMP paradigm or the interaction style of desktops. On the other hand, the *Touchdevelop* language is specifically designed to promote ease of use on phones, such as being statically typed to enable autocompletion, and with deliberate limitations, such as no support for user-defined types [7, 67]. Hence there is a need to use or develop HCI methods to elicit programming mental models specific to the interactions afforded by mobile devices and mitigate the influences of legacy bias arising from user experiences with conventional systems [42].

CONCLUSION AND LIMITATIONS

We have provided an overview of the design possibilities for using mobile devices for programming based on reviewing

-
- CodePads [52]
 - Mobidev [53, 61]
 - TouchDevelop [5, 7, 35, 67, 69]
 - Code Space [12]
 - Touching Factor [23]
 - RefactorPad [58]
 - GROPG [48]
 - Touchifying an IDE [11]
 - Deverywhere [19]
 - Syntax-directed keyboard extension [2, 3]
 - Programming-specific gestures [6]
 - Java-learning application [38]
-
- Examples of Visual Programming [14, 15, 21, 25, 28, 40, 64, 65]
 - Examples of Static code annotation [4, 27, 41, 55]
 - Examples of aids for code comprehension [9, 36, 54, 66]
-

Table 8. Publications selected for review.

18 research papers (and additional example papers on visual programming and code annotation) to guide practitioners in this area. We also review two aspects of the design processes in the papers whose consideration is important for future research on this topic – the tradeoffs in enabling programming support on mobile devices and employing HCI methods particularly in the design of programming tools on mobile devices.

Our paper is the first of its kind to the best of our knowledge but it also has limitations. While we are confident that our sample of papers is representative of the state-of-the-art research on this topic, we did not follow a rigorous search and sampling process typical of systematic reviews. Furthermore, we focus only on research papers but there are also notable industry examples on the topic, such as the *Swift Playgrounds* by Apple [1]. While it can be hard to infer the design rationales of such industry-based apps, a review of the popular programming apps and their uses can also be beneficial and complement our contributions.

This research topic falls under both the software development and HCI disciplines but our paper is predominantly concerned with the HCI and interaction design aspects in the reviewed articles. An alternative would be to review the articles primarily from a software development perspective. For example, the programming functionality addressed can be mapped to the space of programming or software development activities to make apparent the gaps and opportunities.

While the distinction between tablets and phones were blurred in many of the reviewed designs, their respective form factors may present different design requirements and implications which need to be further explored.

Finally, we have captured and presented the key design decisions in the reviewed papers at a somewhat coarse-grained level. While we were constrained by the diverse functionality covered and lack of low-level details reported in many of the papers, it can be valuable to also look at the finer design aspects, such as the information architecture, layouts,

colors, aspects of visual communication, device orientation, as well as programming-related designs used in the applications and if they are designed for single-handed interactions, for left-handed/right-handed use, or optimized for use while sitting/standing/walking [37].

In summary, through this paper, we offer a reflection on the design space capturing the use of mobile devices in programming contexts and contribute to an understanding of the challenges involved and what is needed for progress in this potentially emerging research area.

ACKNOWLEDGMENTS

We thank Dr. Alan Dix for providing valuable suggestions to improve the paper. We are very thankful for the permissions to use example images from the respective papers.

REFERENCES

- [1] Accessed: 09-16-2018. *Swift Playgrounds* - Apple. <https://www.apple.com/swift/playgrounds/>
- [2] I. Almusaly and R. Metoyer. 2015. A syntax-directed keyboard extension for writing source code on touchscreen devices. In *Visual Languages and Human-Centric Computing (VL/HCC), 2015 IEEE Symposium on*. 195–202. DOI: <http://dx.doi.org/10.1109/VLHCC.2015.7357217>
- [3] I. Almusaly, R. Metoyer, and C. Jensen. 2017. Syntax-directed keyboard extension: Evolution and evaluation. In *2017 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*. 285–289. DOI: <http://dx.doi.org/10.1109/VLHCC.2017.8103480>
- [4] Richard Anderson, Ruth Anderson, Beth Simon, Steven A. Wolfman, Tammy VanDeGrift, and Ken Yasuhara. 2004. Experiences with a Tablet PC Based Lecture Presentation System in Computer Science Courses. In *Proceedings of the 35th SIGCSE Technical Symposium on Computer Science Education (SIGCSE '04)*. ACM, New York, NY, USA, 56–60. DOI: <http://dx.doi.org/10.1145/971300.971323>
- [5] Balaji Athreya, Faezeh Bahmani, Alex Diede, and Chris Scaffidi. 2012. End-user programmers on the loose: A study of programming on the phone for the phone. In *2012 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*. IEEE, 75–82.
- [6] Michaela Bačíková, Martin Marićák, and Matej Vančík. 2015. Usability of a domain-specific language for a gesture-driven IDE. In *Computer Science and Information Systems (FedCSIS), 2015 Federated Conference on*. IEEE, 909–914.
- [7] Thomas Ball, Sebastian Burckhardt, Jonathan de Halleux, Michał Moskal, Jonathan Protzenko, and Nikolai Tillmann. 2015. Beyond open source: the TouchDevelop cloud-based integrated development environment. In *Proceedings of the Second ACM International Conference on Mobile Software Engineering and Systems*. IEEE Press, 83–93.

- [8] Sebastian Baltes and Stephan Diehl. 2014. Sketches and diagrams in practice. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*. ACM, 530–541.
- [9] Sebastian Baltes, Peter Schmitz, and Stephan Diehl. 2014. Linking Sketches and Diagrams to Source Code Artifacts. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE 2014)*. ACM, New York, NY, USA, 743–746. DOI: <http://dx.doi.org/10.1145/2635868.2661672>
- [10] Hugh Beyer and Karen Holtzblatt. 1997. *Contextual design: defining customer-centered systems*. Elsevier.
- [11] Benjamin Biegel, Julien Hoffmann, Artur Lipinski, and Stephan Diehl. 2014. U Can Touch This: Touchifying an IDE. In *Proceedings of the 7th International Workshop on Cooperative and Human Aspects of Software Engineering (CHASE 2014)*. ACM, New York, NY, USA, 8–15. DOI: <http://dx.doi.org/10.1145/2593702.2593726>
- [12] Andrew Bragdon, Rob DeLine, Ken Hinckley, and Meredith Ringel Morris. 2011. Code Space: Touch + Air Gesture Hybrid Interactions for Supporting Developer Meetings. In *Proceedings of the ACM International Conference on Interactive Tabletops and Surfaces (ITS '11)*. ACM, New York, NY, USA, 212–221. DOI: <http://dx.doi.org/10.1145/2076354.2076393>
- [13] Mauro Cherubini, Gina Venolia, Rob DeLine, and Andrew J Ko. 2007. Let's go to the whiteboard: how and why software developers use drawings. In *Proceedings of the SIGCHI conference on Human factors in computing systems*. ACM, 557–566.
- [14] Jose Danado and Fabio Paternò. 2012. Puzzle: a visual-based environment for end user development in touch-based mobile phones. In *International Conference on Human-Centred Software Engineering*. Springer, 199–216.
- [15] James Devine, Joe Finney, Michał Moskal, Peli de Halleux, Thomas Ball, and Steve Hodges. 2018. MakeCode and CODAL: intuitive and efficient embedded systems programming for education. (2018). New York, NY, USA, 160–164. DOI: <http://dx.doi.org/10.1145/1029533.1029573>
- [16] Alan Dix, Janet E. Finlay, Gregory D. Abowd, and Russell Beale. 2003. *Human-Computer Interaction (3rd Edition)*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA.
- [17] Alan Dix, Ramesh Kozhissery, Ramprakash Ravichandran, and Dinoop Dayanand. 2009. Content development through the keyhole. *Proc. of EISE2009, Expressive Interaction for Sustainability and Empowerment* (2009), 67–78.
- [18] Stephen H. Edwards and N. Dwight Barnette. 2004. Experiences Using Tablet PCs in a Programming Laboratory. In *Proceedings of the 5th Conference on Information Technology Education (CITC5 '04)*. ACM, New York, NY, USA, 160–164. DOI: <http://dx.doi.org/10.1145/1029533.1029573>
- [19] Yishai A. Feldman, Ari Gam, Alex Tilkin, and Shmuel Tyszberowicz. 2015. Deverywhere: Develop Software Everywhere. In *Proceedings of the Second ACM International Conference on Mobile Software Engineering and Systems (MOBILESoft '15)*. IEEE Press, Piscataway, NJ, USA, 121–124. <http://dl.acm.org/citation.cfm?id=2825041.2825063>
- [20] N Fraser and others. 2013. Blockly: A visual programming editor. URL: <https://code.google.com/p/blockly> (2013).
- [21] Wesley Albert Fryer. 2014. Hopscotch Challenges: Learn to Code on an iPad! *Publications Archive of Wesley Fryer* 1, 1 (2014).
- [22] Michael Hackett and Philip T Cox. 2012. Touchscreen interfaces for visual languages. In *Proceedings of the 5th International Conference on Advances in Computer-Human Interactions (ACHI 2012)*. 176–179.
- [23] Marc Hesenius, Carlos Dario Orozco Medina, and Dominikus Herzberg. 2012. Touching Factor: Software Development on Tablets. In *Proceedings of the 11th International Conference on Software Composition (SC'12)*. Springer-Verlag, Berlin, Heidelberg, 148–161. DOI: http://dx.doi.org/10.1007/978-3-642-30564-1_10
- [24] Andreas Holzinger. 2005. Usability engineering methods for software developers. *Commun. ACM* 48, 1 (2005), 71–74.
- [25] Petri Ihantola, Juha Helminen, and Ville Karavirta. 2013. How to Study Programming on Mobile Touch Devices: Interactive Python Code Exercises. In *Proceedings of the 13th Koli Calling International Conference on Computing Education Research (Koli Calling '13)*. ACM, New York, NY, USA, 51–58. DOI: <http://dx.doi.org/10.1145/2526968.2526974>
- [26] Mona Erfani Joorabchi, Ali Mesbah, and Philippe Kruchten. 2013. Real challenges in mobile app development. In *Empirical Software Engineering and Measurement, 2013 ACM/IEEE International Symposium on*. IEEE, 15–24.
- [27] Sam Kamin, Michael Hines, Chad Peiper, and Boris Capitanu. 2008. A System for Developing Tablet PC Applications for Education. In *Proceedings of the 39th SIGCSE Technical Symposium on Computer Science Education (SIGCSE '08)*. ACM, New York, NY, USA, 422–426. DOI: <http://dx.doi.org/10.1145/1352135.1352279>
- [28] Ville Karavirta, Juha Helminen, and Petri Ihantola. 2012. A Mobile Learning Application for Parsons Problems with Automatic Feedback. In *Proceedings of the 12th Koli Calling International Conference on Computing Education Research (Koli Calling '12)*. ACM, New York, NY, USA, 11–18. DOI: <http://dx.doi.org/10.1145/2401796.2401798>

- [29] Claude Knaus. 2008. FEATURE: Interaction Design for Software Engineering: Boost into Programming Future. *interactions* 15, 4 (July 2008), 71–74. DOI: <http://dx.doi.org/10.1145/1374489.1374508>
- [30] Andrew J Ko, Robin Abraham, Laura Beckwith, Alan Blackwell, Margaret Burnett, Martin Erwig, Chris Scaffidi, Joseph Lawrance, Henry Lieberman, Brad Myers, and others. 2011. The state of the art in end-user software engineering. *ACM Computing Surveys (CSUR)* 43, 3 (2011), 21.
- [31] Andrew J. Ko, Htet Aung, and Brad A. Myers. 2005. Eliciting Design Requirements for Maintenance-oriented IDEs: A Detailed Study of Corrective and Perfective Maintenance Tasks. In *Proceedings of the 27th International Conference on Software Engineering (ICSE '05)*. ACM, New York, NY, USA, 126–135. DOI: <http://dx.doi.org/10.1145/1062455.1062492>
- [32] Andrew J Ko, Thomas D Latoza, and Margaret M Burnett. 2015. A practical guide to controlled experiments of software engineering tools with human participants. *Empirical Software Engineering* 20, 1 (2015), 110–141.
- [33] Andrew J Ko, Brad A Myers, Michael J Coblenz, and Htet Htet Aung. 2006. An exploratory study of how developers seek, relate, and collect relevant information during software maintenance tasks. *IEEE Transactions on software engineering* 32, 12 (2006), 971–987.
- [34] Thomas D LaToza, Gina Venolia, and Robert DeLine. 2006. Maintaining mental models: a study of developer work habits. In *Proceedings of the 28th international conference on Software engineering*. ACM, 492–501.
- [35] Sihan Li, Tao Xie, and Nikolai Tillmann. 2013. A comprehensive field study of end-user programming on mobile devices. In *2013 IEEE Symposium on Visual Languages and Human Centric Computing*. IEEE, 43–50.
- [36] Leonhard Lichtschlag and Jan Borchers. 2010. Codegraffiti: Communication by Sketching for Pair Programming.
- [37] I Scott MacKenzie and R William Soukoreff. 2002. Text entry for mobile computing: Models and methods, theory and practice. *Human-Computer Interaction* 17, 2-3 (2002), 147–198.
- [38] Chao Mbogo, Edwin Blake, and Hussein Suleman. 2016. Design and Use of Static Scaffolding Techniques to Support Java Programming on a Mobile Phone. In *Proceedings of the 2016 ACM Conference on Innovation and Technology in Computer Science Education*. ACM, 314–319.
- [39] Jedidiah McClurg, Sebastian Burckhardt, Michał Moskal, and Jonathan Protzenko. 2015. diffTree: Robust Collaborative Coding using Tree-Merge. (2015).
- [40] Sean McDirmid. 2011. Coding at the Speed of Touch. In *Proceedings of the 10th SIGPLAN Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software (Onward! 2011)*. ACM, New York, NY, USA, 61–76. DOI: <http://dx.doi.org/10.1145/2048237.2048246>
- [41] Kenrick Mock. 2004. Teaching with Tablet PC's. *J. Comput. Sci. Coll.* 20, 2 (Dec. 2004), 17–27. <http://dl.acm.org/citation.cfm?id=1040151.1040153>
- [42] Meredith Ringel Morris, Andreea Danielescu, Steven Drucker, Danyel Fisher, Bongshin Lee, Jacob O Wobbrock, and others. 2014. Reducing legacy bias in gesture elicitation studies. *interactions* 21, 3 (2014), 40–45.
- [43] E. Murphy-Hill, M. Ayazifar, and A. P. Black. 2011. Restructuring software with gestures. In *2011 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*. 165–172. DOI: <http://dx.doi.org/10.1109/VLHCC.2011.6070394>
- [44] Brad Myers and Andrew Ko. 2009. The past, present and future of programming in HCI. (2009).
- [45] Brad A Myers, Andrew J Ko, Thomas D LaToza, and YoungSeok Yoon. 2016. Programmers are users too: Human-centered methods for improving programming tools. *Computer* 49, 7 (2016), 44–52.
- [46] Brad A. Myers, John F. Pane, and Andy Ko. 2004. Natural Programming Languages and Environments. *Commun. ACM* 47, 9 (Sept. 2004), 47–52. DOI: <http://dx.doi.org/10.1145/1015864.1015888>
- [47] Bonnie A. Nardi. 1993. *A Small Matter of Programming: Perspectives on End User Computing*. MIT Press, Cambridge, MA, USA.
- [48] Tuan Anh Nguyen, Christoph Csallner, and Nikolai Tillmann. 2013. GROPG: A Graphical On-phone Debugger. In *Proceedings of the 2013 International Conference on Software Engineering (ICSE '13)*. IEEE Press, Piscataway, NJ, USA, 1189–1192. <http://dl.acm.org/citation.cfm?id=2486788.2486958>
- [49] Don Norman. 2013. *The design of everyday things: Revised and expanded edition*. Basic Books (AZ).
- [50] Donald A. Norman. 2010. Natural User Interfaces Are Not Natural. *interactions* 17, 3 (May 2010), 6–10. DOI: <http://dx.doi.org/10.1145/1744161.1744163>
- [51] Chris Parnin and Robert DeLine. 2010. Evaluating cues for resuming interrupted programming tasks. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. ACM, 93–102.
- [52] Chris Parnin, Carsten Görg, and Spencer Rugaber. 2010. CodePad: Interactive Spaces for Maintaining Concentration in Programming Environments. In *Proceedings of the 5th International Symposium on Software Visualization (SOFTVIS '10)*. ACM, New York, NY, USA, 15–24. DOI: <http://dx.doi.org/10.1145/1879211.1879217>

- [53] Bastian Pfleging, Elba del Carmen Valderrama Bahamondez, Albrecht Schmidt, Martin Hermes, and Johannes Nolte. 2010. MobiDev: a mobile development kit for combined paper-based and in-situ programming on the mobile phone. In *CHI'10 Extended Abstracts on Human Factors in Computing Systems*. ACM, 3733–3738.
- [54] Beryl Plimmer, John Grundy, John Hosking, and Richard Priest. 2006. Inking in the IDE: Experiences with Pen-based Design and Annotation. In *Visual Languages and Human-Centric Computing (VL/HCC'06)*. IEEE, 111–115.
- [55] Beryl Plimmer and Paul Mason. 2006. A Pen-based Paperless Environment for Annotating and Marking Student Assignments. In *Proceedings of the 7th Australasian User Interface Conference - Volume 50 (AUIC '06)*. Australian Computer Society, Inc., Darlinghurst, Australia, Australia, 37–44. <http://dl.acm.org/citation.cfm?id=1151758.1151762>
- [56] Richard Priest and Beryl Plimmer. 2006. RCA: Experiences with an IDE Annotation Tool. In *Proceedings of the 7th ACM SIGCHI New Zealand Chapter's International Conference on Computer-human Interaction: Design Centered HCI (CHINZ '06)*. ACM, New York, NY, USA, 53–60. DOI: <http://dx.doi.org/10.1145/1152760.1152767>
- [57] Felix Raab. 2016. *Source Code Interaction on Touchscreens*. Ph.D. Dissertation.
- [58] Felix Raab, Christian Wolff, and Florian Echtler. 2013. RefactorPad: Editing Source Code on Touchscreens. In *Proceedings of the 5th ACM SIGCHI Symposium on Engineering Interactive Computing Systems (EICS '13)*. ACM, New York, NY, USA, 223–228. DOI: <http://dx.doi.org/10.1145/2494603.2480317>
- [59] Majken K Rasmussen, Esben W Pedersen, Marianne G Petersen, and Kasper Hornbæk. 2012. Shape-changing interfaces: a review of the design space and open research questions. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. ACM, 735–744.
- [60] Mitchel Resnick, John Maloney, Andrés Monroy-Hernández, Natalie Rusk, Evelyn Eastmond, Karen Brennan, Amon Millner, Eric Rosenbaum, Jay Silver, Brian Silverman, and others. 2009. Scratch: programming for all. *Commun. ACM* 52, 11 (2009), 60–67.
- [61] Julian Seifert, Bastian Pfleging, Elba del Carmen Valderrama Bahamóndez, Martin Hermes, Enrico Rukzio, and Albrecht Schmidt. 2011. Mobidev: a tool for creating apps on mobile phones. In *Proceedings of the 13th International Conference on Human Computer Interaction with Mobile Devices and Services*. ACM, 109–112.
- [62] David C Shepherd and Gail C Murphy. 2008. A sketch of the programmer's coach: making programmers more effective. In *Proceedings of the 2008 international workshop on Cooperative and human aspects of software engineering*. ACM, 97–100.
- [63] Dag IK Sjøberg, Jo Erskine Hannay, Ove Hansen, Vigdis By Kampenes, Amela Karahasanovic, N-K Liborg, and Anette C Rekdal. 2005. A survey of controlled experiments in software engineering. *IEEE transactions on software engineering* 31, 9 (2005), 733–753.
- [64] Wolfgang Slany. 2012. Catroid: A Mobile Visual Programming System for Children. In *Proceedings of the 11th International Conference on Interaction Design and Children (IDC '12)*. ACM, New York, NY, USA, 300–303. DOI: <http://dx.doi.org/10.1145/2307096.2307151>
- [65] Amanda Strawhacker, Melissa Lee, Claire Caine, and Marina Bers. 2015. ScratchJr Demo: A Coding Language for Kindergarten. In *Proceedings of the 14th International Conference on Interaction Design and Children (IDC '15)*. ACM, New York, NY, USA, 414–417. DOI: <http://dx.doi.org/10.1145/2771839.2771867>
- [66] Craig J. Sutherland and Beryl Plimmer. 2013. vsInk: Integrating Digital Ink with Program Code in Visual Studio. In *Proceedings of the Fourteenth Australasian User Interface Conference - Volume 139 (AUIC '13)*. Australian Computer Society, Inc., Darlinghurst, Australia, Australia, 13–22. <http://dl.acm.org/citation.cfm?id=2525493.2525495>
- [67] Nikolai Tillmann, Michal Moskal, Jonathan de Halleux, and Manuel Fahndrich. 2011. TouchDevelop: Programming Cloud-connected Mobile Devices via Touchscreen. In *Proceedings of the 10th SIGPLAN Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software (Onward! 2011)*. ACM, New York, NY, USA, 49–60. DOI: <http://dx.doi.org/10.1145/2048237.2048245>
- [68] Nikolai Tillmann, Michal Moskal, Jonathan de Halleux, Manuel Fahndrich, Judith Bishop, Arjmand Samuel, and Tao Xie. 2012b. The Future of Teaching Programming is on Mobile Devices. In *Proceedings of the 17th ACM Annual Conference on Innovation and Technology in Computer Science Education (ITiCSE '12)*. ACM, New York, NY, USA, 156–161. DOI: <http://dx.doi.org/10.1145/2325296.2325336>
- [69] Nikolai Tillmann, Michal Moskal, Jonathan de Halleux, Manuel Fahndrich, and Tao Xie. 2012a. Engage your students by teaching computer science using only mobile devices with touchDevelop. In *2012 IEEE 25th Conference on Software Engineering Education and Training*. IEEE, 87–89.
- [70] Jagoda Walny, Jonathan Haber, Marian Dörk, Jonathan Sillito, and Sheelagh Carpendale. 2011. Follow that sketch: Lifecycles of diagrams and sketches in software development. In *Visualizing Software for Understanding and Analysis (VISOFT), 2011 6th IEEE International Workshop on*. IEEE, 1–8.

- [71] Anthony I Wasserman. 2010. Software engineering issues for mobile application development. In *Proceedings of the FSE/SDP workshop on Future of software engineering research*. ACM, 397–400.
- [72] Daniel Wigdor and Dennis Wixon. 2011. *Brave NUI World: Designing Natural User Interfaces for Touch and Gesture* (1st ed.). Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.
- [73] Jacob O Wobbrock, Meredith Ringel Morris, and Andrew D Wilson. 2009. User-defined gestures for surface computing. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. ACM, 1083–1092.
- [74] Qiyu Zhi and Ronald Metoyer. 2017. Recognizing Handwritten Source Code. In *Proceedings of Graphics Interface 2017 (GI 2017)*. Canadian Human-Computer Communications Society / Société canadienne du dialogue humain-machine, 163 – 170. DOI : <http://dx.doi.org/10.20380/GI2017.21>