



 **Opin vísindi**

This is not the published version of the article / Þetta er ekki útgefna útgáfa greinarinnar

Author(s)/Höf.: Sigurdur Gauti Samuelsson, Matthias Book
Title/Titill: Eliciting Sketched Expressions of Command Intentions in an IDE
Year/Útgáfuár: 2020
Version/Útgáfa: Post-print

Please cite the original version:

Vinsamlega vísið til útgefnu greinarinnar:

Sigurdur Gauti Samuelsson and Matthias Book. 2020. Eliciting Sketched Expressions of Command Intentions in an IDE. Proc. ACM Hum.-Comput. Interact. 4, ISS, Article 200 (November 2020), 25 pages. DOI: <https://doi.org/10.1145/3427328>

Rights/Réttur: © 2020 Copyright held by the owner/authors. This is the authors' version of the work. It is posted here for your personal use. Not for redistribution. The definitive Version of Record was published in Proceedings of the ACM on Human-Computer Interaction, November 2020, Article No.: 200, <https://doi.org/10.1145/3427328>.

Eliciting Sketched Expressions of Command Intentions in an IDE

SIGURDUR GAUTI SAMUELSSON, University of Iceland

MATTHIAS BOOK, University of Iceland

Software engineers routinely use sketches (informal, ad-hoc drawings) to visualize and communicate complex ideas for colleagues or themselves. We hypothesize that sketching could also be used as a novel interaction modality in integrated software development environments (IDEs), allowing developers to express desired source code manipulations by sketching right on top of the IDE, rather than remembering keyboard shortcuts or using a mouse to navigate menus and dialogs. For an initial assessment of the viability of this idea, we conducted an elicitation study that prompted software developers to express a number of common IDE commands through sketches. For many of our task prompts, we observed considerable agreement in how developers would express the respective commands through sketches, suggesting that further research on a more formal sketch-based visual command language for IDEs would be worthwhile.

CCS Concepts: • **Human-centered computing** → **Interaction techniques**; *Empirical studies in interaction design*; Touch screens; *User studies*; • **Software and its engineering** → *Integrated and visual development environments*.

Additional Key Words and Phrases: sketching; user interfaces; software development environments; elicitation study

ACM Reference Format:

Sigurdur Gauti Samuelsson and Matthias Book. 2020. Eliciting Sketched Expressions of Command Intentions in an IDE. *Proc. ACM Hum.-Comput. Interact.* 4, ISS, Article 200 (November 2020), 25 pages. <https://doi.org/10.1145/3427328>

1 INTRODUCTION

Like most designers and engineers, software developers routinely use sketches (i.e. informal, ad-hoc drawings [12]) to externalize thoughts and concepts [18, 43] and to visualize and communicate complex ideas [8]. Sketching works well as a communication tool [16] and in collaborative settings [34] and as an approach to solving complex design problems for those who practice it routinely [3, 4, 47]. Informal sketches take less time to create and require less cognitive effort to understand than textual specifications [20] and more formal visual notations such as the Unified Modeling Language (UML) [35].

Despite sketches' benefits as a "language for handling design ideas" [41] and their ubiquitous, casual use by software engineers to communicate visions of structures they would like to create [25], sketches as expressions of intent have so far been mostly absent from the tools in which software engineers' visions actually materialize: integrated development environments (IDEs) for the creation, navigation and manipulation of source code.

Modern IDEs tend to have very feature-rich graphical user interfaces (GUIs) with a vast variety of menus, dialogs, window panes, buttons, pop-ups and other widgets, all tailored to specific tasks,

Authors' addresses: Sigurdur Gauti Samuelsson, University of Iceland, Saemundargata 2, 102 Reykjavik, Iceland, [siggi-gauti@hi.is](mailto:siggigauti@hi.is); Matthias Book, University of Iceland, Saemundargata 2, 102 Reykjavik, Iceland, book@hi.is.

Publication rights licensed to ACM. ACM acknowledges that this contribution was authored or co-authored by an employee, contractor or affiliate of a national government. As such, the Government retains a nonexclusive, royalty-free right to publish or reproduce this article, or to allow others to do so, for Government purposes only.

© 2020 Copyright held by the owner/author(s). Publication rights licensed to ACM.

2573-0142/2020/11-ART200 \$15.00

<https://doi.org/10.1145/3427328>

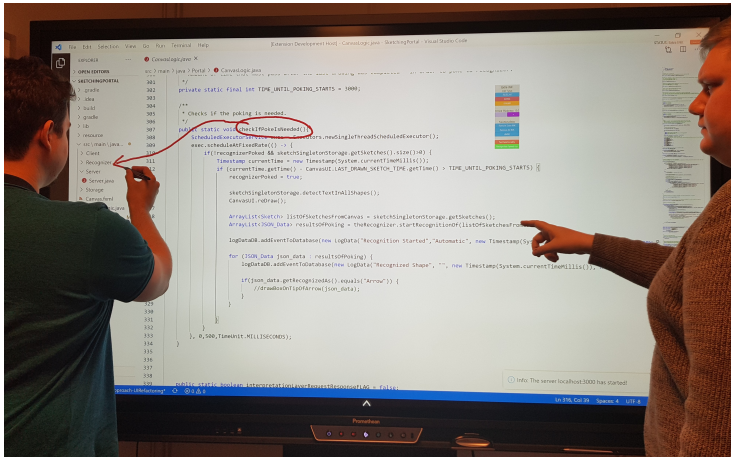


Fig. 1. Indicating the intent to move a method from one class to another in a hypothetical sketch-enabled IDE (simulation)

and many unique to this particular tool. Comprehending, navigating and editing the shown source code may require command sequences involving different menus, dialogs or keyboard shortcuts.

This lack of one of the most common ways of expressing software engineers' ideas in one of their most central tools sparked the vision of elevating sketches to a user interaction modality in software IDEs, enabling developers to perform operations by sketching directly on the source code to be manipulated, instead of expressing their intentions by navigating menus or remembering keyboard shortcuts [5].

While we believe that writing new source code from scratch is likely most efficiently done using a keyboard, we hypothesize that many common code comprehension and manipulation operations (such as moving code blocks, renaming identifiers, tracing definitions, inspecting variables, comparing code segments etc.) could be performed more swiftly and less disruptively by sketching one's intentions right on the source code.

Our rationale is that sketching often happens spontaneously, “in the moment” [25], as software engineers consider and discuss different aspects of a system's design. Refactoring [13] is a similarly spontaneous activity, performed ad hoc while a developer is pursuing other programming tasks [30]. We therefore hypothesize that sketch-based interaction could be particularly effective for performing refactorings and similar code editing operations, which developers should ideally be able to perform with minimal cognitive overhead, so as not to disrupt their primary train of thought, discussion, or coding objective.

As an example use case, we expect that code review sessions could be supported particularly well by sketching as an interaction modality:

Two software engineers standing at a digital whiteboard, using an IDE to review the code of a complex software project, might identify a need to refactor a method by moving it to a different class. Instead of having to leave the whiteboard to reach for a mouse and keyboard to cut and paste the code, or call up the necessary menus and dialogs, a developer could simply use a pen to encircle the method in the code editor, and draw an arrow to the desired target class in the project browser, to express the intended modification without having to interrupt the discussion, as envisioned in the simulated scene in Fig. 1.

Testing the hypothesis that sketching is a viable and effective auxiliary command modality for IDEs is our long-term research goal. Given the novelty of using sketches not just to produce content (as in common modeling tools, cf. Sect. 2.1), but to express commands, the first step in our research is to explore how software developers would use this new interaction modality. To examine this, we posed the following research questions:

- RQ1: Do different software engineers use sketching in similar ways to express commands?
- RQ2: What are software engineers' preferred ways of expressing common commands?

Our first research question tests the basic viability of the idea to use sketches as a command modality: Would prospective users struggle with it, would they all use it in highly individual ways, or would common, presumably universally intuitive usage patterns emerge? The first two outcomes would suggest that sketching may not be as intuitive of a modality as we expected, or that it might at least require explicit training. Only the third outcome would give us confidence that some IDE commands can indeed be expressed through sketches in ways that feel “natural” to many developers.

To answer this question, we designed an elicitation study that explores how software developers would express a number of code manipulation and comprehension commands by sketching on a simulated IDE.

If we can answer RQ1 positively, the next major research step should be the definition of a concrete visual language for expressing IDE commands through sketches. In preparation for this, RQ2 examines the observations from the elicitation study more closely and looks for patterns in the common answers that should inform the design of such a language.

This paper reports on the elicitation study and the answers to the research questions drawn from it. After an overview of related work in Sect. 2, Sect. 3 presents the setup of the elicitation study and introduces a taxonomy used to categorize sketches. Sect. 4 analyzes and discusses the study results to answer RQ1, and Sect. 5 presents and discusses the answers to RQ2. After a reflection on the limitations of the study and threats to the validity of the results in Sect. 6, Sect. 7 concludes the paper with an outlook on the next research steps.

2 RELATED WORK

Sketching – both on paper and on screen – has been suggested as a way to form, convey, and explore ideas in a broad variety of works in human-computer interaction (HCI) and software engineering [21]. In the subsections below, we discuss examples of works that use digital sketching to support software engineers in modeling, understanding and editing software, and we discuss how sketches are different from gestures as an input modality.

2.1 Sketching for Modeling Purposes

Numerous approaches focus on sketching as a way to create informal models of a software system. These tools primarily emulate the classic whiteboard experience but augment it with digital features, such as helping users to manage drawing space in Flatland [31], shape beautification in InkKit [9], and recording design histories in Calico [24], to name just a few. Other works have expanded the capabilities of traditional sketching tools by providing support for (remote) collaboration on sketches, such as Team Storm [15] or FlexiSketch Team [51].

Especially in the field of user interface modeling, tools have been proposed that interpret model sketches in order to derive interactive representations. For example, SILK [19] creates interactive UI mock-ups from storyboard sketches, and Doodle2App [29] converts mobile UI sketches into executable single-page apps.

2.2 Sketching for Code Annotation Purposes

In the works discussed above, sketching is the primary modality used to create or specify an artefact (a model, or an artefact derived from it). However, sketching can also be used to add supplementary information to existing artefacts – most notably in our context, source code.

For example, CodeAnnotator [7] adds support for sketching annotations on top of an IDE, motivated by the fact that programming environments do not natively support ink annotation, although annotations have proven to be helpful when actively reading and reviewing documents [26, 27, 39, 50].

CodeGraffiti aims to support developer communication in pair programming situations by enabling pair programmers to simultaneously write their code and annotate it with ephemeral or persistent sketches [22]. The tool also strives to enhance navigation of large code bases by linking code fragments to sketches [23].

2.3 Sketching for Code Editing and Refactoring Purposes

Moving further beyond code annotation and towards sketch-based interaction with code, CodePad [33] lets software developers navigate and manipulate code using gestures and sketches, e.g. to perform refactorings or generate artefacts. CodePad is distinct from other approaches in employing a separate, peripheral screen for the interaction rather than working directly on the IDE, with the aim to minimize users' need for context switches in complex operations involving several distinct code locations. While our approach aims to support similar types of tasks, we pursue a design philosophy of sketching directly on the code to be manipulated, as we believe this most closely matches users' experience of sketching on paper (e.g. when marking up a text document to express desired changes).

Raab [37] also explores ways of interacting with software IDEs in the absence of a mouse and keyboard, suggesting touch-based techniques for creating, selecting and editing source code with gestures. While related to our approach in the type of targeted displays and the ambition for intuitive, effortless interaction with source code, many of the techniques that Raab examines focus on optimizing more low-level code editing aspects such as cursor placement, selection bounding and virtual keyboard layout for touch-based coding.

2.4 Sketches vs. Gestures as a Command Modality

With the growing availability of touchscreens on devices of all sizes, performing surface gestures to trigger commands [54] is becoming increasingly common. Gestures however differ from sketches in some important ways: In most mainstream software tools, the primary uses of gestures that can be performed on trackpads or touchscreens (swipes, pivots, pinches, and zooms) seem to be for navigation purposes (e.g. a single finger emulates the mouse, two fingers are used for panning and scrolling, and three are used for system-level operations such as switching between applications). Such simple, rather generic gestures, along with drag & drop, seem to be the extent of gesture control available in most software tools. Applications that enable interaction with the user interface via more complex gestures appear to be quite rare. The majority of gestural interactions implemented in common software tools are single-stroke gestures related to the direct manipulation of on-screen objects [52].

Encouragingly, research comparing stroke shortcuts (i.e. simple gestures) and keyboard shortcuts showed that both modalities have the same level of performance when given enough practice, with the stroke shortcuts having substantial cognitive advantages in both learning and recall [2]. However, that study examined only gestures with relatively low complexity, and very little context.

Our long-term research goal is to improve on this state of the art by showing that sketching can transcend the limitations of gestures posited by Yee [52], because sketch strokes are visualized persistently as they are being drawn, which could help to avoid errors due to malformed strokes, increase memorability and precision, and make it easier to draw more complex multi-stroke shapes (to the point of including handwritten text as part of the sketch). Our hypothesis is that the higher visual complexity expressible with sketches, and the ability to more precisely relate to context (in the form of the underlying user interface widgets and source code), should help to broaden the possible command vocabulary and thus make sketches an even more effective modality than gestures or keyboard shortcuts.

Regarding the choice of sketching tool, Walny et al. showed that users have different expectations of the activities a pen should be used for (drawing), vs. those a finger should be used for (dragging and navigation) [48]. Tu et al. also showed that while using a finger to accomplish tasks can be slightly faster than using a pen, it requires a larger amount of surface and is less accurate for complex gestures (e.g. with regard to shape and axial symmetry) [42]. Given that sketching on an IDE is bound to involve complex shapes, high accuracy in relation to small background items, and possibly handwritten input, we chose to focus on a pen as the only input medium. Users should still be able to use a finger for touch gestures that e.g. scroll contents or open menus, but since that modality is already provided (and thus occupied) by existing tools and operating systems, it is not a subject of our research.

3 ELICITATION STUDY

Considering how common sketching is in software developers' practice, how common touch gestures are becoming as a command modality, and how closely digital gestures and sketches are related in terms of execution and processing, as argued in the preceding sections, we are curious to explore the potential of sketching as an additional command modality in software engineering tools. As a first step towards this goal, we conducted an elicitation study to examine how software engineers would use this modality if it was already available.

Elicitation studies in general examine users' preferences for how given tasks should be performed when not constrained by specific instructions or system capabilities. They can be conducted in a variety of ways, depending on the kind of user behavior to be explored, the level of completion of the intended system, and the data to be generated.

In one of the earliest examples, Good et al. [14] adapted the command-line interface of an e-mail system to novice users' needs by intercepting commands that the existing system could not parse, and considering how the system could be improved to accept them. In this type of elicitation study, there is no formal method for identifying a preferred way to accomplish a certain task; any generalization from the observations is instead left to the researcher.

Modanwal and Sarawadekar [28] followed a more formal approach to determine visually impaired users' preferred ways of interacting with a software system: In addition to proposing gestures to accomplish tasks, the participants rated their suggestions in terms of easiness, naturalness, learnability and reproducibility. From these factors, a performance metric was calculated to determine the most preferable responses to the prompts. A similar approach with different metrics was described by Nielsen et al. [32].

Our elicitation study closely follows the structure proposed by Wobbrock et al. [46, 49], who introduced the "agreement rate" as a metric for the similarity of different participants' responses to the same prompt, as described in more detail in Sect. 4.2 below. This method has been used in many elicitation studies focusing on gesture elicitation in particular [1, 6, 10, 11, 17, 36, 38, 40, 44, 45, 53], and was our method of choice due to the similarity of gestures and sketches for expressing commands, as discussed in Sect. 2.4.

In this elicitation study, we asked professional software engineers to perform sketches that should trigger a variety of operations in a hypothetical IDE, observed their responses, and analyzed to which degree their sketches for given prompts agreed. We describe the methodic details of the study in the following subsections, and then examine how the results answer our research questions in Sect. 4 and 5.

3.1 Study Demographic

A group of software engineers working at seven Icelandic software companies participated in our survey. Of the 19 participants, twelve self-identified as male and seven as female. All participants were right-handed. Their mean age was 32.6 years with a variance of ± 10.15 years, the oldest being 57 and the youngest 23 years old. Fourteen participants were between the ages of 25 and 35.

The participants' mean work experience was eight years with a variance of ± 10 years. This high variance is due to the few older participants who had 20, 26 and 38 years of experience, compared to the rest who had one to seven years of experience. The participants' specializations varied greatly and included e.g. Android development, C# back-end development, COBOL legacy software maintenance and computer security research.

3.2 Participation

Participation in the study was not incentivized monetarily or otherwise. We contacted university alumnae working in different IT-affine companies in Iceland and asked them to volunteer, and to ask their colleagues to volunteer as well. Each experiment was held on-site at the respective company, usually in a conference room. When there was more than one volunteer at a company, we conducted all of the experiments in a row.

3.3 Study Setup

Each experiment began with obtaining a signed informed consent form from the participant, letting them know that participation in the experiment was voluntary and all data would be anonymized. This was followed by a standard script describing the experiment, which the participant could choose to hear in Icelandic or English.

Each participant was provided with a digital pen (stylus) and a 15-inch laptop with a touch screen folded to hide the keyboard and create a large tablet-like device (a reenactment of the study environment is shown in Fig. 2). To help participants adopt comparable mindsets, each was asked to imagine being in a meeting room with one or more team members, discussing a software project or performing a code review while standing in front of a large touchscreen displaying source code in an IDE.

To ensure consistency of the experiment, we set up a slide show on the laptop, in which each of the slides contained a screenshot of an IDE and a written task prompt such as the one shown in Fig. 3. The participant was told that they would be shown screenshots of a common IDE, and asked to imagine that a hypothetical sketch interpretation interface could translate their sketches into the correct operations exactly the way they imagined. The participant was briefly trained on the device, allowing them to test the pen's draw and erase functions, as well as the slide show navigation. Once the participant was ready, screen and audio recording software were started, and the participant was encouraged to respond to the series of task prompts by sketching their commands while thinking out loud. Participants were able to skip a prompt if they had difficulties understanding it or could not come up with an appropriate sketch for it.

All experiments comprised the 16 prompts in Table 1. These prompts were chosen to represent a range of code editing, formatting, comprehension, navigation and refactoring operations with varying complexity.



Fig. 2. Reenactment of the elicitation study environment.

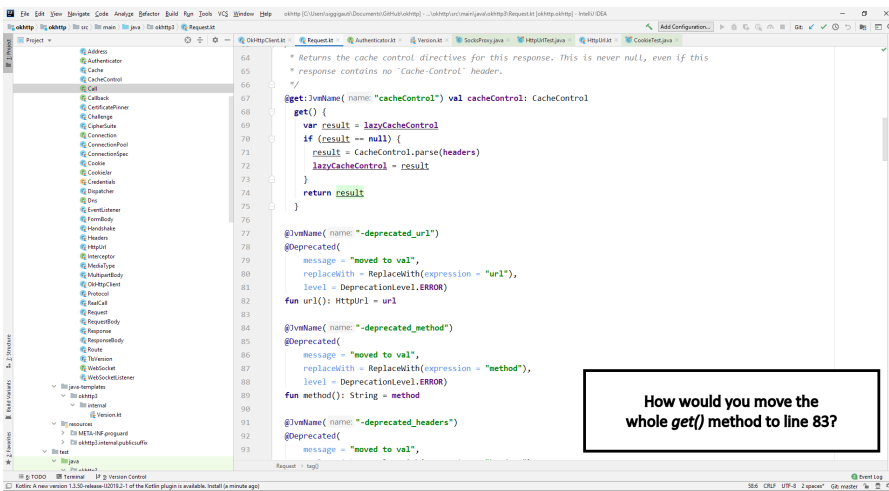


Fig. 3. Screenshot of an IDE with an example of a prompt, as shown to participants.

Each experiment began with the same three simple, presumably straightforward tasks (#1-#3 in Table 1) to get the participant used to and comfortable with the novel concept of sketching commands. The bulk of the prompts then followed in random order, to minimize any bias that the response to one prompt might have on responses to future prompts. The four prompts we found most challenging due to their complexity or abstractness (#13-#16) were always placed at the end of the experiment, to ensure the participant had gained as much experience as possible with the new modality before responding to these. During the experiment, the participant was encouraged to think aloud, voicing any concerns, frustrations and ideas, which were noted by the experimenter.

Each experiment took 15 to 25 minutes, depending on the speed of the participant. After a participant had responded to all prompts, the experiment was concluded with a questionnaire capturing anonymous demographic information such as age group, gender, left- or right-handedness, work domain and specialization.

Table 1. Elicitation study prompts

No.	Prompt	Operation Type
1	How would you indent the code lines 71 and 72 by one more level?	Formatting
2	How would you collapse the body of the <i>get()</i> method?	Visualization
3	How would you move the whole <i>get()</i> method to line 83?	Editing
4	How would you add a comment between lines 71 and 72?	Editing
5	How would you rename the <i>result</i> variable to “value”?	Refactoring
6	How would you rename the <i>parseTrimWS()</i> method to <i>parseTrimAsciiWhitespace()</i> ?	Refactoring
7	How would you undo an action?	IDE operation
8	How would you watch the <i>result</i> variable during a debug run?	Testing
9	How would you trace the source of the <i>url</i> variable definition? (i.e. where it is defined)	Comprehension
10	How would you create a virtual whiteboard area on top of the IDE?	IDE operation
11	How would you specify a path through the <i>queryParameterValues()</i> method to create a test case covering the <i>result.add</i> method?	Testing
12	How would you create a new empty method <i>PushNotif()</i> within this class?	Editing
13	How would you create a new empty method <i>PushNotif()</i> within the (currently not open) Dispatcher class?	Editing, Navigation
14	How would you ask to see a brace’s matching brace?	Comprehension
15	How would you keep the implementation of the <i>get()</i> method visible while switching to the OkHttpClient class?	Navigation
16	How would you remove the ink of a past sketch?	IDE Operation

3.4 Taxonomy of Command Sketches

The 19 participants’ responses to the 16 prompts yielded a multitude of sketches, with differences ranging from minute to total. In order to determine whether two sketches constitute the same visual expression, we derived a taxonomy from our participants’ responses that breaks a sketch down into three semantic elements: Each sketch expresses a **command**, which defines the operation the user intends to execute. Most commands need **parameters** in order to fully specify the desired operation (for example, a “move” command requires two parameters – what to move, and where to move it). Parameters are often expressed as (and commands are sometimes also implied by) **selections**. Selection of code elements can be achieved e.g. by circling or underlining a single word, drawing a box around multiple lines of code, drawing an arrow pointing to a line of code, or the like. The resulting taxonomy comprises the following categories:

Command (CMD): The way the participant’s intended command is expressed visually:

- Symbol (SYM): The command is symbolized by a hand-drawn letter, glyph, icon or other small shape to which the participant ascribes a particular significance (e.g. “M” or “//”).
- Text (TXT): The command is stated literally as a handwritten word or phrase (e.g. “move”, “add comment”).
- Selection (SEL): The command is implied by the type of selection (e.g. selecting a multi-line block and a single line outside the block to move a block).

- **Insufficient (INSUF):** The intended command cannot be identified in the participant's sketch (e.g. because the participant envisioned the command to be chosen from a context menu).

Parameter (PAR): The way a command's parameter is expressed visually (multiple parameter types can be associated with commands that require more than one parameter to be fully specified):

- **Text (TXT):** The parameter is stated explicitly as a handwritten literal (e.g. the new name of a variable).
- **Selection (SEL):** The parameter is provided through a selection (e.g. circling a code fragment to indicate what shall be moved).
- **Command location (C-LOC):** The parameter is implied by the location at which the command was sketched (e.g. inserting a comment at the location of a `"/"` symbol).
- **Unspecified (UNSPEC):** An optional parameter has not been provided by the participant.
- **Insufficient (INSUF):** A required parameter cannot be identified in the participant's sketch.

Selection (SEL): The scope of a selection, regardless of its visual expression (multiple selection types can be associated with commands that require more than one parameter to be fully specified):

- **Single word (S-WORD):** Visual expressions of selecting a single word in the source code, e.g.:
 - A brief tap (dot) with the stylus tip on a single word.
 - An underline, strike-through, circle or similar mark on a single word.
- **Single line (S-LINE):** Visual expressions of selecting a single line of source code, e.g.:
 - A line or arrow drawn to begin or end at a single line of code.
 - A vertical line ("`|`") or bracket (e.g. "`[`") drawn in the editor margin beside a single line of code.
- **Multi-line block (M-LINE):** Visual expressions of selecting multiple contiguous lines of source code, e.g.:
 - A vertical line or bracket drawn in the editor margin that spans several lines of code.
 - A circle, box or other closed curve drawn around several lines of code.
 - One or more lines drawn to strike through a block of several lines of code.

Note that this taxonomy categorizes commands only as symbols, text, selections etc., but does not distinguish individual symbols or words that participants use to express the commands. This is because the main purpose of the taxonomy is to let us calculate agreement rates in order to answer RQ1 (Sect. 4): To examine in principle whether different software engineers sketch commands in similar ways, determining agreement on the *type* level is sufficient. Reasoning about *concrete* commands would require the definition of a precise visual language, which is a major undertaking that we believe can only succeed as a follow-up activity to the study we are reporting on here: Without a positive answer to RQ1, the effort for defining a visual language would likely be in vain, and without the insights into common patterns from RQ2, the language would just reflect our own assumptions rather than many developers' preferred sketch expressions. Once we have determined that most users prefer to express a particular command e.g. symbolically, we should rather perform follow-up studies to investigate which concrete symbol most users would prefer for that command. We will therefore address the language definition in our future work, but already take preparatory steps toward it when answering RQ2 in Sect. 5.

For a similar reason, the taxonomy categorizes selections only by their scope, but not their style (visual expression): To answer RQ1, we are most interested in *what* users select, not *how* they do it. This stance is motivated by the usability consideration that a sketched command should work no matter which selection style users employed (e.g. whether they circled, underlined or tapped on a variable name they'd like to trace). Still, we recognize that particular selection styles carry semantics of their own: Striking through a code block vs. circling it both selects multiple lines of

```

64 * Returns the cache control directives for this response. This is never null, even if this
65 * response contains no "Cache-Control" header.
66
67
68 @get:@JvmName(name="cacheControl") val cacheControl: CacheControl
69 get() {
70     var result = lazyCacheControl
71     if (result == null) {
72         result = CacheControl.parse(headers)
73         lazyCacheControl = result
74     }
75     return result
76 }
77
78 @JvmName(name="--deprecated_url")
79 @Deprecated(
80     message = "moved to val",
81     replaceWith = ReplaceWith(expression = "url",
82     level = DeprecationLevel.ERROR)
83 fun url(): HttpUrl = url
84
85 @JvmName(name="--deprecated_method")
86 @Deprecated(

```

Fig. 4. Participant #18 response to prompt #3

```

64 * Returns the cache control directives for this response. This is never null, even if this
65 * response contains no "Cache-Control" header.
66
67
68 @get:@JvmName(name="cacheControl") val cacheControl: CacheControl
69 get() {
70     var result = lazyCacheControl
71     if (result == null) {
72         result = CacheControl.parse(headers)
73         lazyCacheControl = result
74     }
75     return result
76 }
77
78 @JvmName(name="--deprecated_url")
79 @Deprecated(
80     message = "moved to val",
81     replaceWith = ReplaceWith(expression = "url",
82     level = DeprecationLevel.ERROR)
83 fun url(): HttpUrl = url
84
85 @JvmName(name="--deprecated_method")
86 @Deprecated(

```

Fig. 5. Participant #2 response to prompt #3

code, but likely for different purposes. The semantics of such stylistic differences will have to be reflected in the future definition of a visual language for sketch-based IDE commands.

As an example of the diversity in responses we observed, the sketch in Fig. 4 shows participant #18’s response to prompt #3 (“How would you move the whole `get()` method to line 83?”). Here, we see a selection of a source code block by circling it, and the selection of a single line of code by an arrow pointing to it. The *move* command is not expressed explicitly but implied by these selections, and its two parameters (what to move, and where to move it) are defined by the selections as well.

In contrast, Fig. 5 shows participant #2’s response to the same prompt, which takes a radically different approach – selecting a source code block by drawing a bracket in its margin, and stating the command explicitly in textual form (“mv”). While the first parameter (what to move) is defined by the selection, the second parameter (where to move it) is stated textually as “L:83”.

In some of the participants’ responses, we classified the command as “insufficient” if the participant just talked about how to perform a command, but did not express their idea in sketch form. We also used this code when participants used the stylus to perform a “long press” gesture that they explained should open a context menu from which the intended command would be chosen. As we are focusing exclusively on sketch-based interaction, we categorized these gesture- and menu-based responses as “insufficient”.

We used the taxonomy to code responses to all prompts except prompt #16 (“How would you remove the ink of a past sketch?”) which was more of a question about a hypothetical IDE’s behavior than an operation on the source code. The responses to this prompt were almost exclusively verbal and included a suggestion for a dedicated “Remove ink” button as well as automatic removal of sketch strokes upon successful execution of a command. While these responses are not included in the analysis below, we will take them into account in our future design of a sketch recognition component for an IDE.

4 RQ1 RESULTS

4.1 Observed Sketching Styles

A breakdown of all 19 participants’ responses by their style of expressing commands, parameters and selections across all analyzed prompts (#1-#15) provides an initial impression of our participants’ preferences in answering our prompts, as shown in the bar charts in Fig. 6:¹

Chart a) shows a breakdown of the absolute numbers of different command expression styles across the 285 responses we analyzed in total. Commands implied by symbols and commands expressed by selections were observed in 35% and 32% of the responses, respectively, while commands

¹Since certain prompts possibly lend themselves more naturally to expression in certain styles, the breakdowns in this section should not be assumed to be generalizable, but merely descriptive of our observations for this experiment’s prompts.

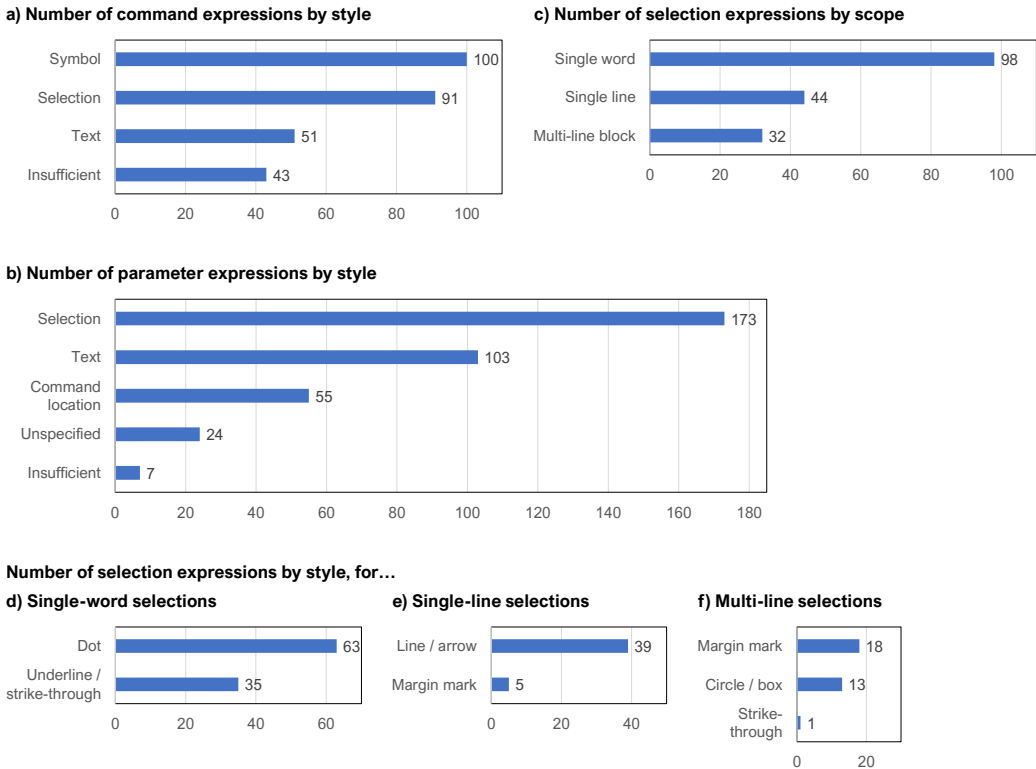


Fig. 6. Breakdown of responses by type and style of sketched expression.

were expressed textually in 18% of responses. In 15% of responses, the command was not expressed in a sketch-based form.

Chart b) shows a breakdown of the absolute numbers of different parameter expression styles across the 362 parameter expressions in the prompt responses we analyzed (this number is higher than that of the responses since some prompts required more than one parameter). 48% of the parameter expressions were by selection, 28% were stated literally, and 15% were implied by command location. 7% of optional parameter expressions were omitted, and only 2% of required parameters were not expressed in a sketch-based form.

Chart c) shows a breakdown of the absolute numbers of different selection scopes across the 174 selection expressions in the prompt responses we analyzed (this number is lower than that of the responses since not all responses utilized selections). Since the selection scope is largely determined by the prompt, this breakdown does not so much indicate participants' preferences but illustrates that our experiment prompted participants to consider all types of scopes to a significant extent, with 56% of selections denoting single words, 26% denoting single lines and 18% denoting multi-line blocks.

Between these scope categories, we however observed considerable differences in their style of expression: Chart d) shows that 64% of the single-word selections were via a single dot (i.e. tap with the stylus tip), while 36% used some form of symbolic marker (underline, strike-through) – interestingly though, not a single participant used an arrow to select a single word. Chart e) shows that the dominant style of selecting a single line of code in our experiment was a line or

Table 2. Classification of prompt #13 responses

Rank	CMD	PAR1	PAR2	SEL1	SEL2	Occurrences
1	TXT	TXT	UNSPEC	-	-	4
2	TXT	TXT	C-LOC	-	-	3
3	SYM	TXT	SEL	-	S-WORD	2
3	TXT	TXT	SEL	-	S-LINE	2
3	TXT	TXT	TXT	-	-	2
3	SYM	TXT	C-LOC	-	-	2
4	TXT	TXT	SEL	-	S-WORD	1
4	INSUF	TXT	INSUF	-	-	1
4	SEL	TXT	SEL	-	S-WORD	1
4	SEL	TXT	SEL	-	S-LINE	1

Table 3. Classification of prompt #3 responses

Rank	CMD	PAR1	PAR2	SEL1	SEL2	Occurrences
1	SEL	SEL	SEL	M-LINE	S-LINE	10
2	SEL	SEL	SEL	S-WORD	S-LINE	2
3	TXT	TXT	SEL	-	M-LINE	1
3	TXT	SEL	SEL	M-LINE	S-WORD	1
3	INSUF	SEL	INSUF	M-LINE	-	1
3	SYM	C-LOC	SEL	-	S-LINE	1
3	INSUF	INSUF	INSUF	-	-	1
3	SYM	C-LOC	TXT	-	-	1
3	SEL	SEL	SEL	M-LINE	S-WORD	1

arrow (i.e. some form of pointer), used in 89% of cases vs. an 11% usage of a vertical margin mark. Chart f) finally shows a 56%-to-40% split between vertical margin marks and circular or rectangular enclosures for selecting multiple lines, with a strike-through being used only for a single selection.

4.2 Agreement Rates

To answer RQ1, we calculate agreement rates for the responses to the various prompts. Agreement rates are a metric for the level of agreement among a group of people on e.g. how to perform a given task [46]. Here, we calculate the agreement rates based on the classification of participants' sketched responses to our task prompts, using the taxonomy introduced in Sect. 3.4. The agreement rate for a given prompt can be formally expressed as

$$a_p = \frac{\sum_{S_{i,p} \subseteq S_p} \frac{1}{2} |S_{i,p}| (|S_{i,p}| - 1)}{\frac{1}{2} |S_p| (|S_p| - 1)}$$

where p is the prompt, S_p is the set of sketched responses for prompt p , and $S_{i,p}$ is a subset of identical responses in S_p (with "identical" being defined as "classified identically using our taxonomy"). Then, a_p is the agreement rate for p , where a value of 0 denotes total disagreement between participants and 1 denotes total agreement.

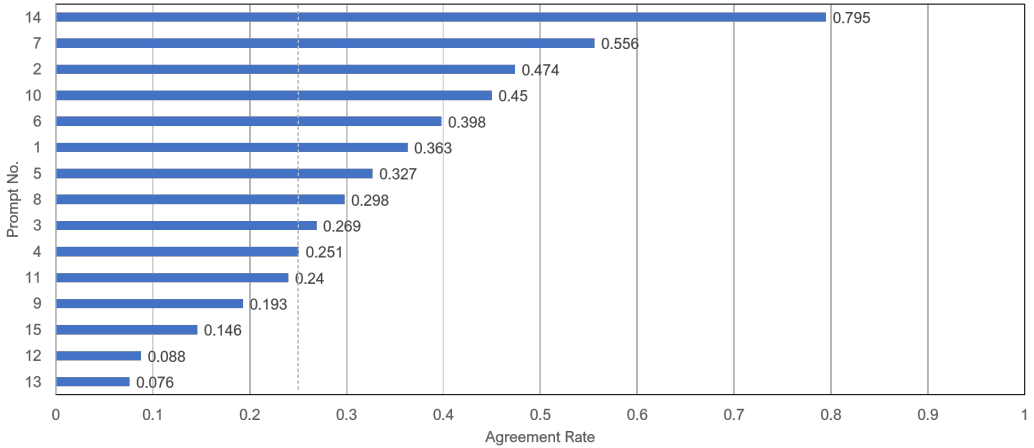


Fig. 7. Agreement rates for all prompts

For example, Table 2 shows the breakdown of classifications of responses to prompt #13 (creating a new method in a class that is currently not open), where CMD denotes the style of command expression, PAR1 denotes the style of expressing the first parameter (the new method's name), PAR2 denotes the style of expressing the second parameter (the target class), and SEL1 and SEL2 denote the style of selection expression, in case PAR1 or PAR2 are of type SEL, respectively. The table entries are ranked by the number of responses with this style combination.

We observed ten different sketching approaches to this prompt – some textual, some symbolic, some based on selections of different types. None of these approaches were chosen by more than three participants, and four had an unspecified parameter for a location. The prompt's agreement rate reflects this low level of consensus:

$$a_{13} = \frac{\binom{4*3}{2} + \binom{3*2}{2} + \binom{2*1}{2} + \binom{2*1}{2} + \binom{2*1}{2} + \binom{2*1}{2} \binom{1*0}{2} + \binom{1*0}{2} + \binom{1*0}{2} + \binom{1*0}{2}}{\binom{19*18}{2}} = 0.076$$

In contrast, Table 3 shows the classification breakdown of responses to prompt #3 (moving a method to a different line), coded in the same way as Table 2, with PAR1 indicating the style for denoting the method to be moved, and PAR2 indicating the style for denoting the destination line.

At first sight, the set of responses seems just as diverse, with nine different approaches being observed for this prompt. In this case, however, one of these approaches was chosen by about half of the participants. This more uniform approach by the participants is reflected in the higher agreement rate for prompt #3:

$$a_3 = \frac{\binom{10*9}{2} + \binom{2*1}{2} + \binom{1*0}{2} + \binom{1*0}{2} + \binom{1*0}{2} + \binom{1*0}{2} + \binom{1*0}{2} + \binom{1*0}{2} + \binom{1*0}{2}}{\binom{19*18}{2}} = 0.269$$

In the same way, we can classify the responses to all prompts and calculate their agreement rates, as shown in Fig. 7.

Table 4 presents the classification of the most frequently chosen sketching approach for those prompts with an agreement rate > 0.25. The other prompts had a lower agreement rate, and in

Table 4. Most popular sketching style for prompts with an agreement rate >0.25 .

Prompt no.	CMD	PAR1	PAR2	SEL1	SEL2	Occurrences
1	SYM	SEL	-	M-LINE	-	11
2	SEL	SEL	-	S-WORD	-	13
3	SEL	SEL	SEL	M-LINE	S-LINE	10
4	SYM	TXT	C-LOC	-	-	9
5	SEL	SEL	TXT	S-WORD	-	11
6	SEL	SEL	TXT	S-WORD	-	12
7	SYM	-	-	-	-	14
8	SEL	C-LOC	-	S-WORD	-	8
9	Insufficient agreement rate					7
10	SYM	-	-	-	-	12
11	Most common response not sketch-based					9
12	Insufficient agreement rate					4
13	Insufficient agreement rate					6
14	SEL	SEL	-	S-WORD	-	17
15	Insufficient agreement rate					6

prompt #11 the most frequently chosen approach was actually not sketch-based (i.e. “insufficient”), as discussed in more detail in Sect. 5.2.²

The stacked bar chart in Fig. 8 visualizes the number and sizes of the groups into which the structurally similar responses for each prompt were classified. Each bar corresponds to the responses to one prompt, and each bar segment represents a group of identically classified sketches, whose size indicates the number of participants who used the same sketching approach. The chart shows that for prompts with higher agreement rates, there are fewer and/or larger groups than for prompts with lower agreement rates.

4.3 Discussion

Our RQ1 examined if different software engineers use sketching in similar ways to express commands.

In response to our prompts, commands were expressed in symbolic form in about a third of the responses, and implied by sketched selections in another third of responses (Fig. 6a). Parameters were expressed through sketched selections in almost half of their occurrences (Fig. 6b). This indicates that a primarily visual expression of commands and their parameters was the preferred (even if not dominant) choice of our participants. We hypothesize that the share of explicit symbolic or textual expressions was higher for commands than for parameters because commands are more abstract concepts, while most parameters had some representation in the visible source code that participants could refer to. We assume that this also explains the much higher share of insufficiently expressed commands vs. parameters.

For the selection of various scopes, we observed quite homogeneous sketching styles in the responses to our prompts: About two-thirds of single-word selections were made by tapping on the word with the stylus, rather than underlining it or drawing similar markers (Fig. 6d). We hypothesize that the preference for briefly tapping on a single word to select it may stem from

²While the threshold of a 0.25 agreement rate is somewhat arbitrary, it correlates with at least half of the participants choosing a similar visual expression in our experiment.

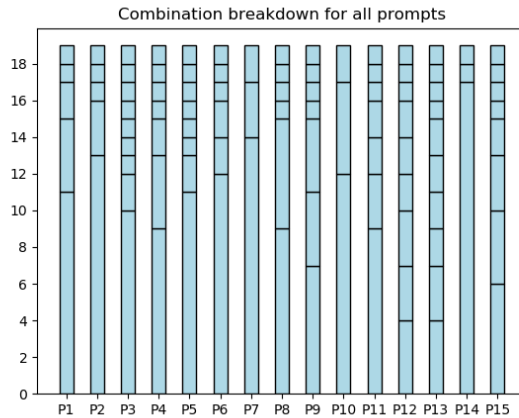


Fig. 8. Number and size of groups of structurally similar sketch responses to each prompt

participants’ preconditioning through mouse- and touch-based interfaces. Future usability studies will have to show if this interaction pattern will be integrated into the flow of sketching despite being “borrowed” from another modality, or if users will replace it with a sketched marker.

For the selection of single code lines, the clear dominance of drawing a pointing line or arrow (Fig. 6e) may be biased by the fact that single lines were only involved in our prompts as destinations of operations. Further elicitation studies will have to show if participants use other single-line selection expressions in operations that do not imply movement.

Finally, for the selection of multiple lines (Fig. 6f), the roughly even split between margin markers and enclosures was surprising to us, as we would have assumed enclosures to be participants’ preferred choice.

To answer RQ1 on a general level, we believe these results indicate that participants are comfortable with expressing commands and their parameters as sketches in general, but that they fall back to textual expressions or even emulate interaction patterns from mouse- or touch-based modalities where they struggle to express a command as a sketch.

These observations also encourage our belief that the interpretation of sketches should be independent of particular selection styles (i.e. a sketched command should be recognizable independently of how the user selects the subject(s) of the command).

On the level of our specific prompts, we saw considerable uniformity: For 10 out of 15 prompts, at least half of our participants independently came up with structurally identical sketch responses (i.e. they expressed commands, parameters and selections by the same means, as illustrated in Fig. 8).

Note that the sketches drawn by participants in the same group could still look quite diverse even if they were classified identically – for example, while many participants expressed the “undo” command symbolically, they still drew different symbols to express it, as shown in the sample of responses to prompt #7 in Fig. 9. However, as discussed in Sect. 3.4, our aim in RQ1 was not to identify a single, definitive sketch solution for a particular task, but to examine how uniform participants’ approaches to expressing a given task through a sketch would be.



Fig. 9. Two different symbols drawn by participants #13 and #16 for prompt #7 (undo an action).

Even if participants' precise visual expressions differed, their approaches were structurally clearly similar enough to give us confidence that common, intuitive visual expressions for IDE commands can be defined and recognized.

Those prompts where participants did not gravitate to a common solution tended to be the more abstract and complex ones. This leads us to RQ2, where we examined the most popular responses to each prompt, and considered why no clear favorite could be identified for some prompts.

5 RQ2 RESULTS

Our RQ2 examined what were the most common approaches for expressing common commands through sketches on the IDE, in preparation for the definition of a concrete visual sketching language and sketch recognition component in our future research.

Out of the 15 prompts that could be answered by sketching, 10 prompts (i.e. all except prompts #9, #11, #12, #13 and #15) were answered in the same style by at least half of the participants, as shown in Table 4. Figures 10–19 show representative examples of the types of visual expression suggested by the majority of participants for these prompts.

5.1 Most Common Types of Responses

To answer RQ2, we will briefly discuss the most popular approaches to each prompt with regard to their inclusion in a more formal visual sketching language and their possible adoption as actual commands in a sketch-enabled IDE:

- Prompt #1 asked how a participant would indent several lines of code. Most participants sketched some form of bracket to designate the code lines to edit, and expressed the indentation command symbolically, typically by a small arrow pointing right. In a sketch-enabled IDE, we would likely allow the user to use any form of selection style (e.g. also encircling the desired lines of code).
- Prompt #2 asked how a participant would collapse the display of a method's implementation. Most participants simply used the stylus to tap on the "collapse" widget provided by the IDE, emulating the mouse-based interaction pattern. While strictly not a sketch-based response, this observation suggests that sketch- and mouse-based interaction modalities should integrate well with each other, as one can perform most mouse-based interactions with the pen as well, without having to change tools (as would be the case when trying to combine e.g. mouse- and touch-based interactions).
- Prompt #3 asked how a participant would move a method implementation to a new location in the same class. Most users encircled the lines to move or marked them with a bracket in the margin, and then drew a long arrow to symbolize where to move the code. The distinguishing characteristic from prompt #1 (and the key visual element an IDE would need to use for disambiguation) was that the arrow pointed to the beginning of a code line outside the selected block, rather than to the space to the right of the selection.
- Prompt #4 asked how a participant would add a comment at specific location. Most participants solved this by drawing a common comment delimiter such as `"/"` or `"#"` to simultaneously indicate both the intended command and the desired location, followed by writing

```

65  * response contains no "Cache-Control" header.
66  */
67  @get:JvmName(name="cacheControl") val cacheControl: CacheControl
68  get() {
69      var result = lazyCacheControl
70      if (result == null) {
71          result = CacheControl.parse(headers)
72          lazyCacheControl = result
73      }
74      return result
75  }
76
77  @JvmName(name="--deprecated_url")
78  @Deprecated(
79      message = "moved to val",
80      replaceWith = ReplaceWith(expression = "url"),
81      level = DeprecationLevel.ERROR)
82  fun url(): HttpUrl = url
83
84  @JvmName(name="--deprecated_method")

```

Fig. 10. Most common type of response to prompt #1.

```

65  * response contains no "Cache-Control" header.
66  */
67  @get:JvmName(name="cacheControl") val cacheControl: CacheControl
68  get() {
69      var result = lazyCacheControl
70      if (result == null) {
71          result = CacheControl.parse(headers)
72          lazyCacheControl = result
73      }
74      return result
75  }
76
77  @JvmName(name="--deprecated_url")
78  @Deprecated(
79      message = "moved to val",
80      replaceWith = ReplaceWith(expression = "url"),
81      level = DeprecationLevel.ERROR)
82  fun url(): HttpUrl = url
83
84  @JvmName(name="--deprecated_method")

```

Fig. 11. Most common type of response to prompt #2.

```

64  * Returns the cache control directives for this response. This is never null, even if this
65  * response contains no "Cache-Control" header.
66  */
67  @get:JvmName(name="cacheControl") val cacheControl: CacheControl
68  get() {
69      var result = lazyCacheControl
70      if (result == null) {
71          result = CacheControl.parse(headers)
72          lazyCacheControl = result
73      }
74      return result
75  }
76
77  @JvmName(name="--deprecated_url")
78  @Deprecated(
79      message = "moved to val",
80      replaceWith = ReplaceWith(expression = "url"),
81      level = DeprecationLevel.ERROR)
82  fun url(): HttpUrl = url
83
84  @JvmName(name="--deprecated_method")

```

Fig. 12. Most common type of response to prompt #3.

```

65  * response contains no "Cache-Control" header.
66  */
67  @get:JvmName(name="cacheControl") val cacheControl: CacheControl
68  get() {
69      var result = lazyCacheControl
70      if (result == null) {
71          result = CacheControl.parse(headers)
72          lazyCacheControl = result
73      }
74      return result
75  }
76
77  @JvmName(name="--deprecated_url")
78  @Deprecated(
79      message = "moved to val",
80      replaceWith = ReplaceWith(expression = "url"),
81      level = DeprecationLevel.ERROR)
82  fun url(): HttpUrl = url
83
84  @JvmName(name="--deprecated_method")

```

Fig. 13. Most common type of response to prompt #4.

```

65  * response contains no "Cache-Control" header.
66  */
67  @get:JvmName(name="cacheControl") val cacheControl: CacheControl
68  get() {
69      var result = lazyCacheControl
70      if (result == null) {
71          result = CacheControl.parse(headers)
72          lazyCacheControl = result
73      }
74      return result
75  }
76
77  @JvmName(name="--deprecated_url")
78  @Deprecated(
79      message = "moved to val",
80      replaceWith = ReplaceWith(expression = "url"),
81      level = DeprecationLevel.ERROR)
82  fun url(): HttpUrl = url
83
84  @JvmName(name="--deprecated_method")

```

Fig. 14. Most common type of response to prompt #5.

```

@Parameterized.Parameter
public boolean useGet;

HttpUrl parse(String url) {
    return useGet
        ? HttpUrl.get(url)
        : HttpUrl.parse(url);
}

@Effect public void parse throwException() {
    HttpUrl expected = parse("http://host/");
    // Loading.
    assertThat(parse("http://host/"), isEqualTo(expected));
    // Trailing.
    assertThat(parse("http://host/"), isEqualTo(expected));
    // Both.
    assertThat(parse("http://host/"), isEqualTo(expected));
    // Both.
    assertThat(parse("http://host/"), isEqualTo(expected));
    assertThat(parse("http://host/").resolve("/"), isEqualTo(expected));
    assertThat(parse("http://host/").resolve(" ").resolve("/"), isEqualTo(expected));
}

```

Fig. 15. Most common type of response to prompt #6.

the comment as literal text. We would suggest that a sketch-enabled IDE following this convention allows any comment delimiter, independently of the edited code's programming language.

- Prompts #5 and #6 asked how a participant would rename a variable or a method, respectively. Most participants drew a line striking through the old name, which simultaneously served to identify the command and its subject. The new name was then written out as text. Regarding the recognition of this command in a sketch-enabled IDE, it is worth pondering whether

```

};

@Parameterized.Parameter
public boolean useGet;

HttpUrl parse(String url) {
    return useGet
        ? HttpUrl.get(url)
        : HttpUrl.parse(url);
}

```




Fig. 16. Most common type of response to prompt #7.

```

64 * Returns the cache control directives for this response. This is new
65 * response contains no 'Cache-Control' header.
66 */
67 @get:JvmName("cacheControl") val cacheControl: CacheControl
68 get() {
69     var result = lazyCacheControl
70     if (result == null) {
71         result = CacheControl.parse(headers)
72         lazyCacheControl = result
73     }
74     return result
75 }

```

Fig. 17. Most common type of response to prompt #8.

```

54
55 #test public void specialValue() throws Exception {
56     assertEquals(cookie.parse(cookie, HTTPCookie.Name.#3).value(), isQualTo("e #3"))
57 }
58
59 #test public void trailingTrailingTrailingTrailing() throws Exception {
60     assertEquals(cookie.parse(cookie, HTTPCookie.Name.#3).value(), isQualTo("e #3"))
61     assertEquals(cookie.parse(cookie, HTTPCookie.Name.#3).value(), isQualTo("e #3"))
62     assertEquals(cookie.parse(cookie, HTTPCookie.Name.#3).value(), isQualTo("e #3"))
63 }
64
65 #test public void emptyValue() throws Exception {
66     assertEquals(cookie.parse(cookie, HTTPCookie.Name.#3).value(), isQualTo("e #3"))
67     assertEquals(cookie.parse(cookie, HTTPCookie.Name.#3).value(), isQualTo("e #3"))
68     assertEquals(cookie.parse(cookie, HTTPCookie.Name.#3).value(), isQualTo("e #3"))
69 }
70
71 #test public void trailingTrailingTrailingTrailingTrailing() throws Exception {
72     assertEquals(cookie.parse(cookie, HTTPCookie.Name.#3).value(), isQualTo("e #3"))
73     assertEquals(cookie.parse(cookie, HTTPCookie.Name.#3).value(), isQualTo("e #3"))
74     assertEquals(cookie.parse(cookie, HTTPCookie.Name.#3).value(), isQualTo("e #3"))
75     assertEquals(cookie.parse(cookie, HTTPCookie.Name.#3).value(), isQualTo("e #3"))
76 }
77

```



Fig. 18. Most common type of response to prompt #10.

```

64 * Returns the cache control directives for this response. This is new
65 * response contains no 'Cache-Control' header.
66 */
67 @get:JvmName("cacheControl") val cacheControl: CacheControl
68 get() {
69     var result = lazyCacheControl
70     if (result == null) {
71         result = CacheControl.parse(headers)
72         lazyCacheControl = result
73     }
74     return result
75 }

```

Fig. 19. Most common type of response to prompt #14.

other selection styles (e.g. encircling or underlining) the name to be replaced should also be acceptable. While striking through a name is the most explicit expression of the “rename” command, we are leaning towards accepting other selection styles as well, since e.g. sloppy execution of the sketch may easily turn a strike-through into an underline.

- Prompt #7 asked how a participant would undo a prior operation. Most participants drew a symbolic expression mimicking the counter-clockwise arrow icon commonly used for “undo” buttons, but we also observed variations on this, such as a left-pointing arrow, a spiral, or a set of zig-zagging lines as if “scratching out” a mistake. For implementation in a sketch-enabled IDE, more concrete elicitation studies will have to reveal which symbol would be most intuitive for most users, or if several symbols could be accepted to trigger an undo operation, without conflicting with other command expressions.
- Prompt #8 asked how a participant would indicate that a variable should be “watched” during a debug run. Most participants suggested to simply mark the variable or its line with a dot in the margin, likely inspired by the practice of setting a breakpoint by clicking into the margin with the mouse. While this approach would be somewhat ambiguous in practice as it does not clearly express the command intention (it could also be interpreted as setting a breakpoint, or be part of a completely different command expression), the second-most popular approach used a symbol instead of the single margin dot (e.g. a “W” or stylized binoculars) to more explicitly declare the command intention. The participants’ struggle to express this command unambiguously tells us that particular care must be taken to design an intuitive visual expression for tasks that are more abstract than an immediate code manipulation.
- Prompt #10 asked how a participant would create a temporary “virtual whiteboard” for drawing arbitrary, ephemeral non-command sketches. Most participants sketched a box in an open area of the IDE to designate the desired location of the requested drawing area. While not required by the prompt, several participants mentioned that they deliberately drew the box without overlapping the code, to distinguish it from a selection. We believe however that

when designing a sketch-based command language, care should be taken to ensure that the language elements and recognition heuristics are sufficiently distinct to not require users to consciously disambiguate commands.

- Prompt #14 asked how a participant would find a given brace's matching brace. Most participants used the stylus to simply tap on one brace, explaining that they would expect the matching brace to be highlighted just as is done when one places the cursor on a brace using the mouse or keyboard. This mirrors our observation from prompt #2 that participants intuitively fell back to mouse-inspired interaction patterns where they seemed more straightforward than sketching their intention.
- Prompt #16 asked about a participant's preferred tool behavior for the removal of a sketch after successful execution of a command, and could therefore not be evaluated based on the taxonomy used for the other prompts. 18 out of 19 participants agreed however that the ink of a sketch should vanish automatically once the user had completed a sketch, and that sketch had successfully triggered an operation, to revert to an uncluttered user interface with room for the next command.

5.2 Discussion

The observations above show that for the majority of our prompts, common sketching approaches were chosen that lend themselves to a quite straightforward definition in a visual sketching language and interpretation by a sketch recognition component.

Besides the prompts for which we observed relatively uniform responses among many participants, we however also find it instructive to consider why no clear consensus emerged for the following five prompts, and develop hypotheses for what this means for the definition a broader sketch-based command vocabulary:

- Prompt #9 asked how a participant would trace the source of a variable, i.e. find the location where the variable is defined. Participants' responses varied between sketching symbols such as a question mark or magnifying glass on top of the variable name, writing out the command, or other approaches. At first glance, this diversity is somewhat surprising, given that we observed more solid agreement on the related operation of "watching" a variable (prompt #8). However, "watching" a variable can be considered a localized activity that developers might imagine happening right where they sketch it – they have a visible "anchor". Tracing the definition of a variable, in contrast, is a query about a remote, unknown location in the code that is not represented by a visible anchor, so participants might have been struggling to visualize the desired operation or its outcome in a straightforward sketch. This result also raises the question if manipulative operations may in general be more easily sketchable than observational or query operations, as the process and/or outcome of manipulative operations could be visualized more clearly – a question to explore more explicitly in future studies.
- Prompt #11 asked how a participant would specify a test case by tracing a path through a method. Even though the prompt already hinted at a sketch-based solution, only a few participants actually sketched something resembling an execution path over the method's code lines, while others tried a number of different approaches, most of which had to be classified as "insufficient" in terms of the command or parameter expression. Several participants also opted not to respond to this prompt, citing complexity and lack of understanding. It will be interesting to analyze this class of more "open" code comprehension and query tasks further, to determine where the limits to sketch-based expression of commands might lie.
- Prompt #12 asked how a participant would create a new method in the currently edited class. Participants sketched a very diverse set of responses, ranging from solutions involving

only text to combinations of symbols and text, elaborate selections, or semblances of UML diagrams. Despite the rather specific nature of the prompt, this may indicate that the existence of an “anchor” element in the editor pane that the sketch can relate to (as in most other prompts) helps users to come up with an intuitive (and thus common) sketching approach. Without such an “anchor”, participants were struggling to express even a rather common and specific operation.

- Prompt #13 asked how a participant would create a new method in a class that is not currently open in the editor. The vast majority of participants explained that they would first open the target class and then use the same approach as in prompt #12, rather than e.g. involving the project browser in the sketch, where the requested destination class was listed. While we envision our approach to allow developers to sketch across all panes of the IDE, the participants in our study mostly constrained themselves to sketches within the editor pane. This behavior may have been inspired by paper-based text annotation metaphors, which are also constrained to a single page. On the other hand, we would assume that users are quite used to operations spanning multiple panes or windows, given how deeply ingrained mouse and keyboard operations like drag & drop or copy & paste can be assumed to be – we therefore would have expected to see this pattern in pen-based interaction as well. However, our observations in this study might have been biased by the fact that the majority of our prompts involved only localized code manipulations, so participants were not encouraged to “think outside the box”. Future studies should examine cross-pane operations more explicitly, to examine how viable they are for expression through sketching.
- Prompt #15 asked how a participant would keep the implementation of one method visible while opening a different class. This prompt appeared to have been phrased too broadly, resulting in participants coming up with several different strategies for keeping a method implementation visible (split screens, floating screenshots, “peek overlays” etc.), which heavily influenced the structure of the sketches they proposed. In future elicitation studies, we may therefore split this task into separate prompts for different strategies.

In summary, the tasks that users struggled to express via sketching seem to fall into three classes:

- (1) Tasks that did not involve manipulation of code, but querying or observing more intangible aspects such as structural relationships or run-time behavior
- (2) Tasks for which the sketch could not be “anchored” at some existing element in the user interface, but required expressing the operation or its outcome from scratch
- (3) Tasks that might have been solved by including UI elements outside the editor pane in the sketch

In further studies, it will be interesting to examine these classes of tasks in particular, in order to establish where the limits to sketch-based expression of IDE operation indeed are. At the same time, we will pursue the further analysis of the large class of tasks that did yield common sketch responses, in order to understand how they can be defined and recognized most efficiently.

6 THREATS TO VALIDITY

In our elicitation study, we observed sketched command expressions that participants drew in response to prompts describing common code manipulation and comprehension tasks in an IDE, with the aim of examining how uniform participants’ responses would be (RQ1), and which sketching approaches would be favored by most participants (RQ2).

To ensure **construct validity**, we devised a taxonomy of sketch expressions that we used to code the participants’ sketched responses, and calculated agreement rates based on this classification. The level of detail we chose for the taxonomy is sufficient to distinguish broad classes of sketching

approaches (e.g. predominantly shape- vs. predominantly text-based responses, or implicit vs. explicit command expressions), but it does not capture more minute differences such as a code block being enclosed by a circle or a rectangle, an arrow being rooted inside or outside an enclosure, etc. A taxonomy that captures more details of the sketch expressions would have led to a more fragmented classification of responses and thus lower agreement scores. However, since our ultimate goal is to develop a visual language and recognition heuristics for sketched commands, we found our more coarse taxonomy more useful to examine the participants' responses at the level of flexibility that we feel a sketch-based user interface should exhibit.

With regard to ensuring **internal validity**, we strove to eliminate biases where possible by having participants complete the experiment separately, using a standardized script to introduce them to the experiment, and randomizing the order of most prompts, while maintaining a controlled order of introductory prompts to expose all participants to the same learning curve for this new modality. However, as discussed in Sect. 5, participants seem to have struggled with some prompts – possibly due to phrasing or possibly due to their larger scope. In addition, participants were not able to go back and change their answers after responding to a prompt, and so were not able to use the experience from later prompts to correct misunderstandings or sub-optimal responses they might have recognized in earlier prompts. Software developers who use a future sketch-enabled IDE routinely might be able to use the additional modality to a fuller potential than our first-time users. Further experiments are therefore suggested to determine if the low agreement scores observed for some of our prompts stem from misunderstandings of the tasks or deeper-rooted issues in expressing certain types of tasks through sketches. Given these considerations, we have more confidence in the validity of the high agreement rates than the validity of the low agreement rates.

Finally, in terms of **external validity**, we must point out that most of our prompts focused on relatively localized code manipulations. Since only a few prompts challenged participants to sketch larger structures, relationships or behaviors, the experiment may have biased participants to focus on finding sketch expressions for more localized commands rather than “big picture” operations. This means our findings cannot be generalized to assumptions about the sketchability of more complex tasks, but further elicitation studies should be conducted to focus specifically on those.

Also, while our participant cohort comprised individuals from the main user group of the envisioned software tool, and a broad range of specializations within that domain, it still counted only 19 participants, all from the same region, which means that our observations, and specifically the calculated agreement rates, cannot be generalized to software developers at large. In order to increase our sample and be less prone to individual participants' performance and preferences affecting the overall result, we therefore intend to conduct similar elicitation studies with larger groups of practitioners abroad, and with undergraduate students in Computer Science and Software Engineering at universities in different countries. While the relative homogeneity and inexperience of the student cohort may at first sight appear as a drawback, we speculate that they would be less biased by years of individual usage habits, project-specific working procedures and corporate culture, which might enable them to approach the tasks from a more analytical perspective and reduce the chance of outliers.

7 CONCLUSION

We have presented the results of an elicitation study designed to understand how software engineers would use sketching as an input modality to express code manipulation and comprehension commands in an integrated software development environment. We observed that for two-thirds of our task prompts, at least half of our participants used structurally similar sketch-based expressions of the commands and their parameters. We take the observation that most of our participants found sketch expressions for many prompts, and gravitated towards structurally similar expressions

for many of them, as an indication that sketches could be a viable modality for expressing IDE commands, which merits further research.

As suggested in the preceding sections, this future work will proceed in several steps, informed by the preferred sketching patterns we observed in this elicitation study:

First, we aim to design a visual language for the expression of various types of commands through sketches. To inform its design, we are planning to conduct further elicitation studies with larger, more diverse audiences, and with extended task sets (e.g. to examine how more of Fowler’s refactoring types [13] can be expressed through sketching). Additional studies will be useful to answer specific questions, e.g. about users’ preferred symbol choices, and about the sketchability of operations that have more of a “query” than an “edit” nature, that are not anchored in currently visible elements, or that involve multiple panes or windows.

Secondly, we plan to develop sketch recognition and interpretation heuristics to drive a sketch-enabled IDE prototype. This research branch will deal with the more technical and graphical aspects of processing pen-based user input (e.g. distinguishing command sketches from casual sketches without a command intention), deriving the users’ intentions (by associating semantics with the drawn shapes in the context of the widgets or artifacts they are drawn upon, as defined in the visual command language), and triggering the necessary operations in the software tool (with possible provisions for interactive execution, e.g. to parameterize a command or correct the effects of an unintended or misinterpreted command.)

Finally, we strive to evaluate the overall approach’s viability and usability in practice. Given a working IDE prototype that can recognize a variety of sketched command inputs, we will conduct user studies to assess the efficiency and usability of our approach vs. more traditional interaction modalities. This evaluation will need to include both synthetic experiments that focus on technical aspects such as recognition robustness, and user studies in realistic settings that examine the overall user experience from the perspectives of productivity and ergonomics.

If sketching proves viable as an additional command modality for software IDEs, we speculate that different sets of sketch-based commands could also be defined for intuitive interaction with other complex software tools in which data, documents or other artifacts are manipulated directly (e.g. working with data in spreadsheets, revising documents in word processors, visually interacting with information in data science applications, etc.), ultimately establishing sketching more firmly as an additional command modality for software tools.

ACKNOWLEDGMENTS

This work was supported by the Icelandic Research Fund (grant no. 196228).

REFERENCES

- [1] Robin Andersson, Jonas Berglund, Aykut Coskun, Morten Fjeld, and Mohammad Obaid. 2017. Defining Gestural Interactions for Large Vertical Touch Displays. In *Human-Computer Interaction - INTERACT 2017 - 16th IFIP TC 13 International Conference, Mumbai, India, September 25-29, 2017, Proceedings, Part I (Lecture Notes in Computer Science, Vol. 10513)*, Regina Bernhaupt, Girish Dalvi, Anirudha Joshi, Devanuj K. Balkrishnan, Jacki O’Neill, and Marco Winckler (Eds.). Springer, 36–55. https://doi.org/10.1007/978-3-319-67744-6_3
- [2] Caroline Appert and Shumin Zhai. 2009. Using Strokes As Command Shortcuts: Cognitive Benefits and Toolkit Support. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (Boston, MA, USA) (CHI ’09)*. ACM, New York, NY, USA, 2289–2298. <https://doi.org/10.1145/1518701.1519052>
- [3] Uday Athavankar. 1997. Mental Imagery as a Design Tool. *Cybernetics and Systems* 28, 1 (1997), 25–42. <https://doi.org/10.1080/019697297126236> arXiv:<https://doi.org/10.1080/019697297126236>
- [4] Uday Athavankar and Arnab Mukherjee. 2003. Blindfolded Classroom: Getting Design Students to Use Mental Imagery. In *Human Behaviour in Design: Individuals, Teams, Tools*. Springer, Berlin, Heidelberg, 111–120. https://doi.org/10.1007/978-3-662-07811-2_12

- [5] Matthias Book and André van der Hoek. 2018. Sketching with a purpose: moving from supporting modeling to supporting software engineering activities. In *2018 IEEE/ACM 11th International Workshop on Cooperative and Human Aspects of Software Engineering (CHASE)*. IEEE, 93–96.
- [6] Edwin Chan, Teddy Seyed, Wolfgang Stuerzlinger, Xing-Dong Yang, and Frank Maurer. 2016. User Elicitation on Single-hand Microgestures. In *Proceedings of the 2016 CHI Conference on Human Factors in Computing Systems, San Jose, CA, USA, May 7-12, 2016*, Jofish Kaye, Allison Druin, Cliff Lampe, Dan Morris, and Juan Pablo Hourcade (Eds.). ACM, 3403–3414. <https://doi.org/10.1145/2858036.2858589>
- [7] Xiaofan Chen and Beryl Plimmer. 2007. CodeAnnotator: digital ink annotation within Eclipse. In *Proceedings of the 2007 Australasian Computer-Human Interaction Conference, OZCHI 2007, Adelaide, Australia, November 28-30, 2007 (ACM International Conference Proceeding Series, Vol. 251)*, Bruce Thomas (Ed.). ACM, 211–214. <https://doi.org/10.1145/1324892.1324935>
- [8] Mauro Cherubini, Gina Venolia, Rob DeLine, and Amy 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 (San Jose, California, USA) (CHI '07)*. Association for Computing Machinery, New York, NY, USA, 557–566. <https://doi.org/10.1145/1240624.1240714>
- [9] Ronald Chung, Petrut Mirica, and Beryl Plimmer. 2005. InkKit: A Generic Design Tool for the Tablet PC. In *Proceedings of the 6th ACM SIGCHI New Zealand Chapter's International Conference on Computer-human Interaction: Making CHI Natural (CHINZ '05)*. ACM, New York, NY, USA, 29–30. <https://doi.org/10.1145/1073943.1073950>
- [10] Yasmin Felberbaum and Joel Lanir. 2018. Better Understanding of Foot Gestures: An Elicitation Study. In *Proceedings of the 2018 CHI Conference on Human Factors in Computing Systems, CHI 2018, Montreal, QC, Canada, April 21-26, 2018*, Regan L. Mandryk, Mark Hancock, Mark Perry, and Anna L. Cox (Eds.). ACM, 334. <https://doi.org/10.1145/3173574.3173908>
- [11] Justin W. Firestone, Rubi Quinones, and Brittany A. Duncan. 2019. Learning from Users: an Elicitation Study and Taxonomy for Communicating Small Unmanned Aerial System States Through Gestures. In *14th ACM/IEEE International Conference on Human-Robot Interaction, HRI 2019, Daegu, South Korea, March 11-14, 2019*. IEEE, 163–171. <https://doi.org/10.1109/HRI.2019.8673010>
- [12] Jonathan Fish and Stephen Scrivener. 1990. Amplifying the mind's eye: sketching and visual cognition. *Leonardo* 23, 1 (1990), 117–126.
- [13] Martin Fowler. 2018. *Refactoring: Improving the Design of Existing Code* (2nd ed.). Addison-Wesley Professional.
- [14] Michael D. Good, John A. Whiteside, Dennis R. Wixon, and Sandra J. Jones. 1984. Building a User-derived Interface. *Commun. ACM* 27, 10 (Oct. 1984), 1032–1043. <https://doi.org/10.1145/358274.358284>
- [15] Joshua M. Hailpern, Erik Hinterbichler, Caryn Leppert, Damon J. Cook, and Brian P. Bailey. 2007. TEAM STORM: demonstrating an interaction model for working with multiple ideas during creative group work. In *Proceedings of the 6th Conference on Creativity & Cognition, Washington, DC, USA, June 13-15, 2007*, Ben Shneiderman, Gerhard Fischer, Elisa Giaccardi, and Michael Eisenberg (Eds.). ACM, 193–202. <https://doi.org/10.1145/1254960.1254987>
- [16] Kathryn Henderson. 1991. Flexible Sketches and Inflexible Data Bases: Visual Communication, Conscriptio Devices, and Boundary Objects in Design Engineering. *Science, Technology, & Human Values* 16, 4 (1991), 448–473. <https://doi.org/10.1177/016224399101600402> arXiv:<https://doi.org/10.1177/016224399101600402>
- [17] Jochen Huber, Mohamed A. Sheik-Nainar, and Nada Matic. 2017. Force-enabled Touch Input on the Steering Wheel: An Elicitation Study. In *Adjunct Proceedings of the 9th International Conference on Automotive User Interfaces and Interactive Vehicular Applications, AutomotiveUI 2017, Oldenburg, Germany, September 24-27, 2017*, Susanne Boll, Andreas Löcken, Ronald Schroeter, Martin Baumann, Ignacio J. Alvarez, Lewis L. Chuang, Sebastian Feuerstack, Myoungsoon Jeon, Hanneke Hooft van Huysduynen, Nora Broy, Sebastian Osswald, Ioannis Politis, and David R. Large (Eds.). ACM, 168–172. <https://doi.org/10.1145/3131726.3131740>
- [18] Manolya Kavakli and John S Gero. 2001. Sketching as mental imagery processing. *Design studies* 22, 4 (2001), 347–364. [https://doi.org/10.1016/S0142-694X\(01\)00002-3](https://doi.org/10.1016/S0142-694X(01)00002-3)
- [19] James A. Landay and Brad A. Myers. 2001. Sketching Interfaces: Toward More Human Interface Design. *Computer* 34, 3 (2001), 56–64. <https://doi.org/10.1109/2.910894>
- [20] Jill H. Larkin and Herbert A. Simon. 1987. Why a Diagram is (Sometimes) Worth Ten Thousand Words. *Cogn. Sci.* 11, 1 (1987), 65–100. <https://doi.org/10.1111/j.1551-6708.1987.tb00863.x>
- [21] Makayla Lewis, Miriam Sturdee, Jason Alexander, Jelle van Dijk, Majken Kirkegaard Rasmussen, and Thuong N. Hoang. 2017. SketchingDIS: Hand-drawn sketching in HCI. In *DIS '17 Companion: Proceedings of the 2017 ACM Conference Companion Publication on Designing Interactive Systems (Edinburgh, Scotland)*. Association for Computing Machinery, New York, NY, USA, 356–359. <https://doi.org/10.1145/3064857.3064863>
- [22] Leonhard Lichtschlag and Jan Borchers. 2010. CodeGraffiti: communication by sketching for pair programmers. In *UIST '10: Adjunct proceedings of the 23rd annual ACM symposium on User interface software and technology*. ACM, 439–440. <https://doi.org/10.1145/1866218.1866260>

- [23] Leonhard Lichtschlag, Lukas Spychalski, and Jan Bochers. 2014. CodeGraffiti: Using hand-drawn sketches connected to code bases in navigation tasks. In *Proceedings of the 2014 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*. IEEE, 65–68. <https://doi.org/10.1109/VLHCC.2014.6883024>
- [24] Nicolas Mangano, Alex Baker, and André van der Hoek. 2008. Calico: A Prototype Sketching Tool for Modeling in Early Design. In *Proceedings of the 2008 International Workshop on Models in Software Engineering (MiSE '08)*. ACM, New York, NY, USA, 63–68. <https://doi.org/10.1145/1370731.1370747>
- [25] Nicolas Mangano, Thomas D. LaToza, Marian Petre, and André van der Hoek. 2015. How Software Designers Interact with Sketches at the Whiteboard. *IEEE Trans. Software Eng.* 41, 2 (2015), 135–156. <https://doi.org/10.1109/TSE.2014.2362924>
- [26] Catherine C. Marshall. 1997. Annotation: From Paper Books to the Digital Library. In *Proceedings of the Second ACM International Conference on Digital Libraries* (Philadelphia, Pennsylvania, USA) (DL '97). Association for Computing Machinery, New York, NY, USA, 131–140. <https://doi.org/10.1145/263690.263806>
- [27] Catherine C. Marshall. 1998. Toward an Ecology of Hypertext Annotation. In *HYPERTEXT '98. Proceedings of the Ninth ACM Conference on Hypertext and Hypermedia: Links, Objects, Time and Space - Structure in Hypermedia Systems, June 20-24, 1998, Pittsburgh, PA, USA*, Robert M. Akscyn (Ed.). ACM, 40–49. <https://doi.org/10.1145/276627.276632>
- [28] Gourav Modanwal and Kishor Sarawadekar. 2018. A Gesture Elicitation Study with Visually Impaired Users. In *HCI International 2018 - Posters' Extended Abstracts - 20th International Conference, HCI International 2018, Las Vegas, NV, USA, July 15-20, 2018, Proceedings, Part II (Communications in Computer and Information Science, Vol. 851)*, Constantine Stephanidis (Ed.). Springer, 54–61. https://doi.org/10.1007/978-3-319-92279-9_7
- [29] Soumik Mohian and Christoph Csallner. 2020. Doodle2App: Native App Code by Freehand UI Sketching. In *7th IEEE/ACM International Conference on Mobile Software Engineering and Systems (MOBILESoft)*.
- [30] Emerson Murphy-Hill, Chris Parnin, and Andrew P. Black. 2009. How We Refactor, and How We Know It. In *Proceedings of the 31st International Conference on Software Engineering (ICSE '09)*. IEEE Computer Society, USA, 287–297. <https://doi.org/10.1109/ICSE.2009.5070529>
- [31] Elizabeth D. Mynatt, Takeo Igarashi, W. Keith Edwards, and Anthony LaMarca. 1999. Flatland: New Dimensions in Office Whiteboards. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (CHI '99)*. ACM, New York, NY, USA, 346–353. <https://doi.org/10.1145/302979.303108>
- [32] Michael Nielsen, Moritz Störing, Thomas B. Moeslund, and Erik Granum. 2003. A Procedure for Developing Intuitive and Ergonomic Gesture Interfaces for HCI. In *Gesture-Based Communication in Human-Computer Interaction, 5th International Gesture Workshop, GW 2003, Genova, Italy, April 15-17, 2003, Selected Revised Papers (Lecture Notes in Computer Science, Vol. 2915)*, Antonio Camurri and Gualtiero Volpe (Eds.). Springer, 409–420. https://doi.org/10.1007/978-3-540-24598-8_38
- [33] Chris Parnin, Carsten Görg, and Spencer Rugaber. 2010. CodePad: interactive spaces for maintaining concentration in programming environments. In *Proceedings of the ACM 2010 Symposium on Software Visualization, Salt Lake City, UT, USA, October 25-26, 2010*, Alexandru Telea, Carsten Görg, and Steven P. Reiss (Eds.). ACM, 15–24. <https://doi.org/10.1145/1879211.1879217>
- [34] Roland A. Pfister and Martin J. Eppler. 2012. The Benefits of Sketching for Knowledge Management. *J. Knowl. Manag.* 16, 2 (2012), 372–382. <https://doi.org/10.1108/13673271211218924>
- [35] Beryl Plimmer and Mark Apperley. 2004. INTERACTING with Sketched Interface Designs: An Evaluation Study. In *CHI '04 Extended Abstracts on Human Factors in Computing Systems* (Vienna, Austria) (CHI EA '04). ACM, New York, NY, USA, 1337–1340. <https://doi.org/10.1145/985921.986058>
- [36] Patricia Pons and Javier Jaen. 2020. Interactive spaces for children: gesture elicitation for controlling ground mini-robots. *J. Ambient Intell. Humaniz. Comput.* 11, 6 (2020), 2467–2488. <https://doi.org/10.1007/s12652-019-01290-6>
- [37] Felix Raab. 2016. *Source Code Interaction on Touchscreens*. Ph.D. Dissertation. University of Regensburg. <https://nbn-resolving.org/urn:nbn:de:bvb:355-epub-331075>
- [38] Isabel Benavente Rodriguez and Nicolai Marquardt. 2017. Gesture Elicitation Study on How to Opt-in & Opt-out from Interactions with Public Displays. In *Proceedings of the Interactive Surfaces and Spaces, ISS 2017, Brighton, United Kingdom, October 17 - 20, 2017*, Sriram Subramanian, Jürgen Steimle, Raimund Dachselt, Diego Martínez Plasencia, and Tovi Grossman (Eds.). ACM, 32–41. <https://doi.org/10.1145/3132272.3134118>
- [39] Gary M Schumacher and Jane Gradwohl Nash. 1991. Conceptualizing and measuring knowledge change due to writing. *Research in the Teaching of English* (1991), 67–96.
- [40] Shaikh Shawon Arefin Shimon, Courtney Lutton, Zichun Xu, Sarah Morrison-Smith, Christina Boucher, and Jaime Ruiz. 2016. Exploring Non-touchscreen Gestures for Smartwatches. In *Proceedings of the 2016 CHI Conference on Human Factors in Computing Systems, San Jose, CA, USA, May 7-12, 2016*, Jofish Kaye, Allison Druin, Cliff Lampe, Dan Morris, and Juan Pablo Hourcade (Eds.). ACM, 3822–3833. <https://doi.org/10.1145/2858036.2858385>
- [41] M. Tovey, S. Porter, and R. Newman. 2003. Sketching, concept development and automotive design. *Design Studies* 24, 2 (March 2003), 135–153. [https://doi.org/10.1016/S0142-694X\(02\)00035-2](https://doi.org/10.1016/S0142-694X(02)00035-2)

- [42] Huawei Tu, Xiangshi Ren, and Shumin Zhai. 2012. A Comparative Evaluation of Finger and Pen Stroke Gestures. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems* (Austin, Texas, USA) (CHI '12). ACM, New York, NY, USA, 1287–1296. <https://doi.org/10.1145/2207676.2208584>
- [43] Barbara Tversky. 2002. What do sketches say about thinking. In *Sketch Understanding, papers from the 2002 AAAI Spring Symposium, March 25-27, 2002*. 148–151.
- [44] Jean Vanderdonckt, Nathan Magrofuoco, Suzanne Kieffer, Jorge Pérez, Ysabelle Rase, Paolo Roselli, and Santiago Villarreal. 2019. Head and Shoulders Gestures: Exploring User-Defined Gestures with Upper Body. In *Design, User Experience, and Usability. User Experience in Advanced Technological Environments - 8th International Conference, DUXU 2019, Held as Part of the 21st HCI International Conference, HCII 2019, Orlando, FL, USA, July 26-31, 2019, Proceedings, Part II (Lecture Notes in Computer Science, Vol. 11584)*, Aaron Marcus and Wentao Wang (Eds.). Springer, 192–213. https://doi.org/10.1007/978-3-030-23541-3_15
- [45] Radu-Daniel Vatavu. 2017. Improving Gesture Recognition Accuracy on Touch Screens for Users with Low Vision. In *Proceedings of the 2017 CHI Conference on Human Factors in Computing Systems, Denver, CO, USA, May 06-11, 2017*, Gloria Mark, Susan R. Fussell, Cliff Lampe, m. c. schraefel, Juan Pablo Hourcade, Caroline Appert, and Daniel Wigdor (Eds.). ACM, 4667–4679. <https://doi.org/10.1145/3025453.3025941>
- [46] Radu-Daniel Vatavu and Jacob O. Wobbrock. 2015. Formalizing Agreement Analysis for Elicitation Studies: New Measures, Significance Test, and Toolkit. In *Proceedings of the 33rd Annual ACM Conference on Human Factors in Computing Systems, CHI 2015, Seoul, Republic of Korea, April 18-23, 2015*, Bo Begole, Jinwoo Kim, Kori Inkpen, and Woontack Woo (Eds.). ACM, 1325–1334. <https://doi.org/10.1145/2702123.2702223>
- [47] Ilse M Verstijnen, Cees van Leeuwen, G Goldschmidt, Ronald Hamel, and JM Hennessey. 1998. Sketching and creative discovery. *Design studies* 19, 4 (1998), 519–546. [https://doi.org/10.1016/S0142-694X\(98\)00017-9](https://doi.org/10.1016/S0142-694X(98)00017-9)
- [48] Jagoda Walny, Bongshin Lee, Paul Johns, Nathalie Henry Riche, and Sheelagh Cpendale. 2012. Understanding Pen and Touch Interaction for Data Exploration on Interactive Whiteboards. *IEEE Trans. Vis. Comput. Graph.* 18, 12 (2012), 2779–2788. <https://doi.org/10.1109/TVCG.2012.275>
- [49] 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* (Boston, MA, USA) (CHI '09). Association for Computing Machinery, New York, NY, USA, 1083–1092. <https://doi.org/10.1145/1518701.1518866>
- [50] Joanna L. Wolfe. 2000. Effects of annotations on student readers and writers. In *Proceedings of the Fifth ACM Conference on Digital Libraries, June 2-7, 2000, San Antonio, TX, USA*. ACM, 19–26. <https://doi.org/10.1145/336597.336620>
- [51] Dustin Wüest, Norbert Seyff, and Martin Glinz. 2015. FLEXISKETCH TEAM: Collaborative Sketching and Notation Creation on the fly. In *37th IEEE / ACM International Conference on Software Engineering, ICSE 2015, Florence, Italy, May 16-24, 2015, Volume 2*, Antonia Bertolino, Gerardo Canfora, and Sebastian G. Elbaum (Eds.). IEEE Computer Society, 685–688. <https://doi.org/10.1109/ICSE.2015.223>
- [52] Wendy Yee. 2009. Potential Limitations of Multi-touch Gesture Vocabulary: Differentiation, Adoption, Fatigue. In *Human-Computer Interaction. Novel Interaction Methods and Techniques*, Julie A. Jacko (Ed.). Springer, Berlin, Heidelberg, 291–300. https://doi.org/10.1007/978-3-642-02577-8_32
- [53] Ionut-Alexandru Zaiti, Stefan Gheorghe Pentiuc, and Radu-Daniel Vatavu. 2015. On free-hand TV control: experimental results on user-elicited gestures with Leap Motion. *Pers. Ubiquitous Comput.* 19, 5-6 (2015), 821–838. <https://doi.org/10.1007/s00779-015-0863-y>
- [54] Shumin Zhai, Per Kristensson, Caroline Appert, Tue Andersen, and Xiang Cao. 2012. Foundational Issues in Touch-Surface Stroke Gesture Design: An Integrative Review. *Found. Trends Hum.-Comput. Interact.* 5, 2 (Feb. 2012), 97–205. <https://doi.org/10.1561/1100000012>

Received June 2020; revised August 2020; accepted September 2020