Towards Sketch-based User Interaction with Integrated Software Development Environments

Sigurdur Gauti Samuelsson
siggigauti@hi.is
University of Iceland
Reykjavik, Iceland

Matthias Book
book@hi.is
University of Iceland
Reykjavik, Iceland

ABSTRACT

Powerful software tools, such as software development environments, often have complex graphical user interfaces (GUIs) that are not intuitive to handle, especially when performing complex, multi-step operations. We hypothesize that sketching could be a more intuitive way of expressing user intentions than navigating nested menus or memorizing keyboard shortcuts to accomplish complex operations. Enabling this vision requires software capable of both allowing the user to sketch anywhere on a GUI, and interpreting those sketches as specific commands to be performed within the integrated development environment (IDE). In this paper, we report on preliminary results of an elicitation study performed to gather insights into how developers would use a sketch-based interface.

CCS CONCEPTS

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

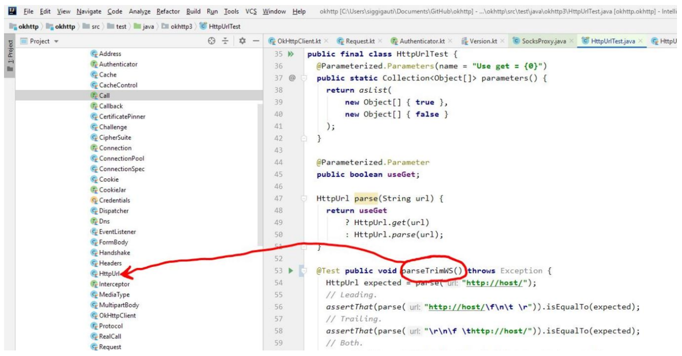
KEYWORDS

sketching, user interfaces, software development environments

ACM Reference Format:
Sigurdur Gauti Samuelsson and Matthias Book. 2020. Towards Sketch-based User Interaction with Integrated Software Development Environments. In *IEEE/ACM 42nd International Conference on Software Engineering Workshops (ICSEW'20), May 23–29, 2020, Seoul, Republic of Korea.* ACM, New York, NY, USA, 4 pages. https://doi.org/10.1145/3387940.3392231

1 INTRODUCTION

Integrated software development environments (IDEs) tend to have very feature-rich graphical user interfaces (GUIs) with a vast variety of menus, dialogs, window panes, buttons and other widgets, all tailored to specific tasks, and many unique to this particular tool. Executing non-trivial manipulations of the shown software code, or even navigating a complex project's code base, may require elaborate command sequences involving different menus, dialogs or keyboard shortcuts.


File Edit View Navigate Code Analyze Refactor Build Run Tools VCS Window Help
okhttp [C:\Users\siggigauti\Documents\GitHub\okhttp] - ...\okhttp\src\test\java\okhttp3\HttpUrlTest.java [okhttp.okhttp] - Intelli
okhttp okhttp src test java okhttp3 HttpUrlTest
Project
OkHttpClient.kt Request.kt Authenticator.kt Version.kt SocksProxy.java HttpUrlTest.java HttpU
Address
Authenticator
Cache
CacheControl
Call
Callback
CertificatePinner
Challenge
CipherSuite
Connection
ConnectionPool
ConnectionSpec
Cookie
CookieJar
Credentials
Dispatcher
Dns
EventListener
FormBody
Handshake
Headers
HttpUrl
Interceptor
MediaType
MultipartBody
OkHttpClient
Protocol
RealCall
Request

```java
public final class HttpUrlTest {
  @Parameterized.Parameters(name = "Use get = {0}")
  public static Collection<Object[]> parameters() {
    return asList(
        new Object[] { true },
        new Object[] { false }
    );
  }

  @Parameterized.Parameter
  public boolean useGet;

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

  @Test public void parseTrimWS() throws Exception {
    HttpUrl expected = parse( url: "http://host/");
    // Leading.
    assertThat(parse( url: "http://host/\f\n\t \r")).isEqualTo(expected);
    // Trailing.
    assertThat(parse( url: "\r\n\f \thttp://host/")).isEqualTo(expected);
    // Both.
```

Figure 1: Indicating the intent to move a method from one class to another in a sketch-enabled software IDE

Away from their computer, software engineers routinely use sketches (i.e. informal, ad-hoc drawings) to visualize and communicate complex ideas [5]. This prompted our vision of elevating sketches to a user interaction modality in software IDEs, enabling developers to perform complex operations by sketching directly on the software artefacts to be manipulated. [4].

For example, a software developer standing at a digital whiteboard, using an IDE to perform a code review of a project with a colleague, 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, the developer could simply use a pen to circle the method in the code editor, and draw an arrow to the desired class in the project browser (as shown in Fig. 1), to express the intended modification without having to interrupt the discussion and leave the whiteboard.

In this paper, we will explore the idea of sketching as an interaction modality for reviewing and refactoring existing code. After a brief overview of related work in Sect. 2, Sect. 3 presents preliminary results of an ongoing elicitation study whose focus is on observing how developers would trigger different operations in an IDE using a digital pen and touch screen. Section 4 concludes the paper and gives an outlook on our further research plans.

2 RELATED WORK

Our reason for turning to sketching as an additional input modality for software IDEs is that sketching has always been used by people as a way of externalizing thoughts and concepts [15]. In software engineering in particular, sketches are used to convey complex ideas in an informal style that takes less time to create and requires less cognitive effort to understand than more formal notations such as the Unified Modeling Language (UML) [13]. Sketching works well as a communication tool [8] and as an approach to solving complex design problems for those who practice it routinely [2, 3, 16].

In software engineering tools, digital sketching has so far however been restricted to drawing software models (i.e. creating content), rather than being employed as an input modality (i.e. conveying commands). For example, there have been sketch-based modeling systems [12] and interfaces, for example Teddy [9] and SKETCH [19], as well as numerous academic tools that emulate the classic whiteboard experience, such as Flatland [11], InkKit [6], and Calico [10], to name just a few.

While we are not aware of other works utilizing *sketching* as a command mechanism, performing *gestures* on touch screens [20] certainly is a related interaction modality. 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 page navigation, and three are used for system-level navigation such as switching between applications). Such simple, rather generic gestures, along with drag & drop, seem to be the extent of gesture control implemented 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 [18].

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 [1]. 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 [18], because sketch strokes are visualized persistently as they are being drawn, which could help to avoid errors due to malformed strokes, increase 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, Tu et al. showed that using a finger can be slightly faster than using a pen, but requires a larger amount of surface and is less accurate for complex gestures (e.g. with regard to shape and axial symmetry) [14]. Given that sketching on an IDE is bound to involve complex shapes, high accuracy in relation to background items, and possible handwritten input, we chose to focus on the pen as the only input medium. Users should still be able to use a finger on the touch screen to e.g. scroll or open menus, but since that functionality is already provided by the tool and operating system, it is not a subject of our research.

3 ELICITATION STUDY

Our current research focus is on exploring how users would interact with a sketch-based user interface; specifically, what they would sketch to express certain command intentions. Like gesture-based user interfaces, sketch-based user interfaces have very few intrinsic constraints on how the input could be shaped. Different users may therefore have quite different opinions on what is the most intuitive way to express a specific intended command through sketching.

To identify the "most agreeable" sketch-based expressions of common command intentions, we are currently performing an elicitation study that closely follows the structure of similar studies by Wobbrock et al. [17] and Good et al. [7]: We are observing professional software developers working at Icelandic software companies, asking them to perform sketches that should trigger a variety of operations in a hypothetical IDE.

3.1 Study Demographic

To date, 19 volunteers 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 25 and 35 years old.

The participants' mean time of 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 Study Setup

Each experiment started off with obtaining informed consent from the participant, letting them know that participation in the experiment is voluntary and all data would be anonymized. This was followed by a standard script describing the experiment. Each participant was provided with an electronic pen (stylus) and a 15-inch laptop with a touch screen folded to hide the keyboard and create a large tablet-like device. To help participants being in 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 Wizard-of-Oz-style slide show on the laptop, in which each of the slides contained a screenshot of an IDE and a written task prompt. 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. Each experiment comprised 16 prompts such as the following:

- How would you move the whole get() method to line 83?
- How would you add a comment between lines 71 and 72?
- How would you rename the *result* variable to ”value”?
- How would you undo an action?
- How would you trace the source of the *url* variable definition? (i.e. where it is defined)
- How would you create a new empty method *PushNotif()* within this class?
- How would you ask to see a brace's matching brace?
- How would you remove the ink of a past sketch?

The experiment always began with the same three simple, presumably straightforward tasks, to get the participant used to and comfortable with the novel concept of sketching commands. The bulk of the commands then followed in random order, to minimize any bias that the answer to one prompt might have on answers to future prompts. The four most complex prompts 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 answering 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 answered all prompts, the experiment was concluded with a questionnaire capturing various anonymous demographic information such as age group, gender, left- or right-handedness, work domain and specialization.

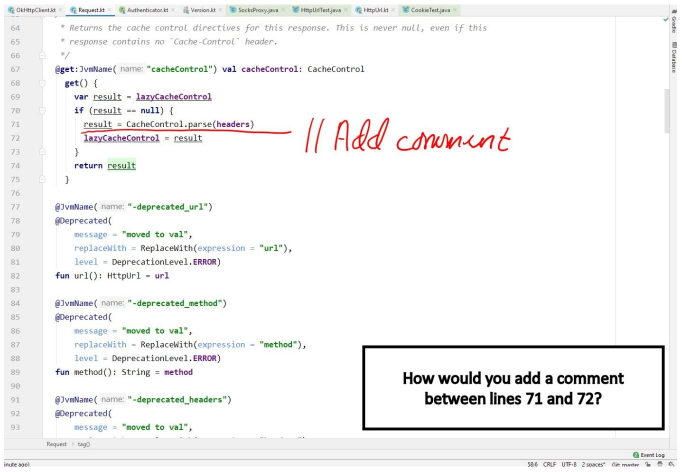
3.3 Preliminary Observations

Figure 2 shows participant responses to a selection of prompts. Every participant had a slightly different notion of what is possible and how to approach sketch-based input, resulting in different sketching styles. For example, while the participant's command in Fig. 2b is completely graphical, another participant's command was expressed in similar syntax as a Git command (Fig. 2c).

Participants who voiced think-aloud opinions offered useful suggestions for capabilities that a sketch recognition and interpretation interface would need. For example, editing tasks such as renaming variables and methods or moving lines of code were often treated like annotating a paper printout, using strike-through marks and margin notes. Another interesting observation was the use of known symbols. For example, when prompted to insert a comment between two lines, two participants used the iconic pair of forward slashes (“//”, as shown in Fig. 2a) to indicate that the following text should be a comment.

Regarding the lifetime of the command sketches, 18 out of 19 participants agreed 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.

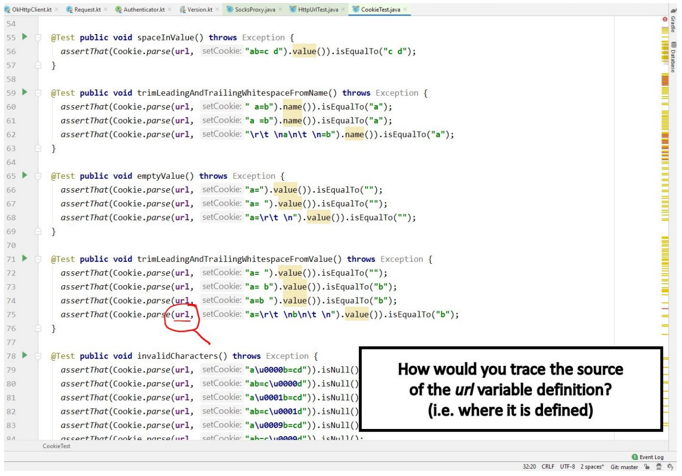
Some discussion revolved around possible cases of misinterpreted commands, which would leave the user surprised and probably frustrated by an unintended operation, whose cause (and maybe even effect) might not be obvious anymore after removal of the ink. Two possible solutions to this issue were discussed: Either the


OkHttpClient.kt Request.kt Authenticator.kt Version.kt SocksProxy.java HttpUrlTest.java HttpUrl.kt CookieTest.java
// Add comment
Request › tag()
Event Log
58:6 CRLF UTF-8 2 spaces* Git: master

```
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")
85     @Deprecated(
86         message = "moved to val",
87         replaceWith = ReplaceWith(expression = "method"),
88         level = DeprecationLevel.ERROR)
89     fun method(): String = method
90
91     @JvmName( name: "-deprecated_headers")
92     @Deprecated(
93         message = "moved to val",
```

How would you add a comment between lines 71 and 72?

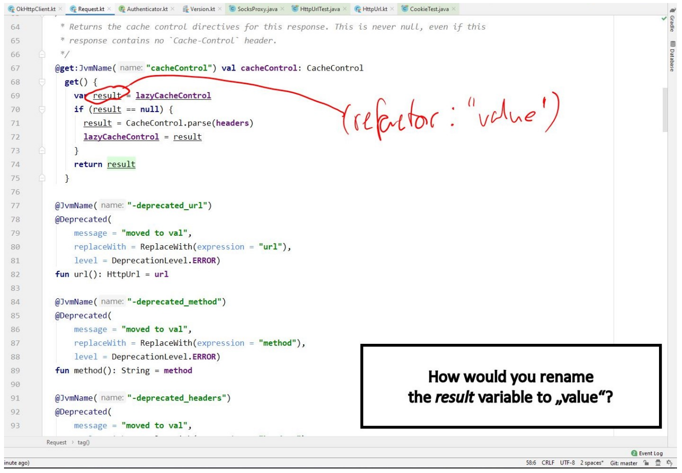
(a) Adding a comment between two lines of code


OkHttpClient.kt Request.kt Authenticator.kt Version.kt SocksProxy.java HttpUrlTest.java CookieTest.java
CookieTest
Event Log
32:20 CRLF UTF-8 2 spaces* Git: master

```
54
55   @Test public void spaceInValue() throws Exception {
56     assertThat(Cookie.parse(url, setCookie: "ab=c d").value()).isEqualTo("c d");
57   }
58
59   @Test public void trimLeadingAndTrailingWhitespaceFromName() throws Exception {
60     assertThat(Cookie.parse(url, setCookie: " a=b").name()).isEqualTo("a");
61     assertThat(Cookie.parse(url, setCookie: "a =b").name()).isEqualTo("a");
62     assertThat(Cookie.parse(url, setCookie: "\r\t \na\n\t \n=b").name()).isEqualTo("a");
63   }
64
65   @Test public void emptyValue() throws Exception {
66     assertThat(Cookie.parse(url, setCookie: "a=").value()).isEqualTo("");
67     assertThat(Cookie.parse(url, setCookie: "a= ").value()).isEqualTo("");
68     assertThat(Cookie.parse(url, setCookie: "a=\r\t \n").value()).isEqualTo("");
69   }
70
71   @Test public void trimLeadingAndTrailingWhitespaceFromValue() throws Exception {
72     assertThat(Cookie.parse(url, setCookie: "a= ").value()).isEqualTo("");
73     assertThat(Cookie.parse(url, setCookie: "a= b").value()).isEqualTo("b");
74     assertThat(Cookie.parse(url, setCookie: "a=b ").value()).isEqualTo("b");
75     assertThat(Cookie.parse(url, setCookie: "a=\r\t \nb\n\t \n").value()).isEqualTo("b");
76   }
77
78   @Test public void invalidCharacters() throws Exception {
79     assertThat(Cookie.parse(url, setCookie: "a\u0000b=cd")).isNull()
80     assertThat(Cookie.parse(url, setCookie: "ab=c\u0000d")).isNull()
81     assertThat(Cookie.parse(url, setCookie: "a\u0001b=cd")).isNull()
82     assertThat(Cookie.parse(url, setCookie: "ab=c\u0001d")).isNull()
83     assertThat(Cookie.parse(url, setCookie: "a\u0009b=cd")).isNull()
```

How would you trace the source of the *url* variable definition? (i.e. where it is defined)

(b) Tracing the source of a variable


OkHttpClient.kt Request.kt Authenticator.kt Version.kt SocksProxy.java HttpUrlTest.java HttpUrl.kt CookieTest.java
(refactor: "value")
Request › tag()
Event Log
58:6 CRLF UTF-8 2 spaces* Git: master

```
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")
85     @Deprecated(
86         message = "moved to val",
87         replaceWith = ReplaceWith(expression = "method"),
88         level = DeprecationLevel.ERROR)
89     fun method(): String = method
90
91     @JvmName( name: "-deprecated_headers")
92     @Deprecated(
93         message = "moved to val",
```

How would you rename the *result* variable to „value“?

(c) Renaming a variable

Figure 2: Responses to task prompts by different professional software developers

use of an interactive confirmation prompt that would describe the imminent command and ask for confirmation before execution, or the ability to easily undo any interpreted operation. The latter could likely rely on the IDE's built-in command stack, but still requires user awareness of what just happened, and whether that operation was intentional. Whether it would be more disruptive to explicitly confirm every sketch interpretation, or to stay alert for occasional misinterpreted operations in order to undo them, requires more in-depth study before deciding on an implementation.

One of the interaction patterns suggested by four participants was the use of a context menu that should appear upon touching or hovering over a particular word for a second or two. This is most likely influenced by experience with smart phones and other touch interfaces, where contextual operations are often called up by long-press gestures. While long-press or hover gestures fit naturally into a touch- and pen-based interaction environment, we will not focus on them in our research, for two reasons: Firstly, we want to focus on exploring novel *sketch-based* interactions, while long-pressing and hovering are already established *gesture-based* interactions. Secondly, these gestures are typically already handled by the operating system and the IDE itself, in order to bring up context menus. While we certainly do not want to discourage users from employing established gestures such as long-pressing, hovering, tapping, swiping, pinching etc., our research will focus on enabling additional interactions that rely on sketching alone.

Finally and perhaps unsurprisingly, a third of the participating software developers did not just envision how they would sketch a command for a given task prompt, but could not resist pondering how particular sketch commands should be recognized and interpreted by the hypothetical user interface logic. This may have biased their responses away from sketches inspired solely by intuition, and towards sketches whose interpretation they deemed technically feasible. While we appreciate the additional thought and input provided by these participants, this observation led us to plan the second phase of the elicitation study with university students, i.e. an audience that is familiar with the prompted operations, but does not yet have as much solution engineering experience.

Once we have collected responses from a broader base of users, we are going to formally evaluate the findings using a similar technique as Wobbrock et al. [17] to evaluate agreement scores between different causes for a specific effect, which will then inform the implementation of interpreters for individual sketches and their intended commands.

4 CONCLUSION

In this paper, we discussed preliminary findings of an elicitation study aimed at learning how developers would use sketching as an interaction medium to express command intentions.

Our next steps will be concluding the elicitation study and creating a taxonomy of sketch commands that should help with the formulation of a visual language for sketch-based user interaction with source code and possibly other content. In parallel, we will develop a prototype of a sketch recognition and interpretation layer for an IDE, as the basis for further studies on the usability and recognition accuracy of sketch-based user input.

ACKNOWLEDGMENTS

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

REFERENCES

[1] Caroline Appert and Shumin Zhai. 2009. Using Strokes As Command Shortcuts: Cognitive Benefits and Toolkit Support *(CHI '09)*. ACM, New York, NY, USA, 2289–2298. https://doi.org/10.1145/1518701.1519052

[2] 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

[3] Uday Athavankar and Arnab Mukherjee. 2003. *Blindfolded Classroom: Getting Design Students to Use Mental Imagery.* Springer Berlin Heidelberg, Berlin, Heidelberg, 111–120. https://doi.org/10.1007/978-3-662-07811-2_12

[4] 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.

[5] Mauro Cherubini, Gina Venolia, Rob DeLine, and Amy J. Ko. 2007. Let's Go to the Whiteboard: How and Why Software Developers Use Drawings *(CHI '07)*. Association for Computing Machinery, New York, NY, USA, 557–566. https://doi.org/10.1145/1240624.1240714

[6] 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

[7] 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

[8] Kathryn Henderson. 1991. Flexible Sketches and Inflexible Data Bases: Visual Communication, Conscription 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

[9] Takeo Igarashi, Takeo Igarashi, Satoshi Matsuoka, and Hidehiko Tanaka. 2007. Teddy: A Sketching Interface for 3D Freeform Design *(SIGGRAPH '07)*. ACM, New York, NY, USA, Article 21, 8 pages. https://doi.org/10.1145/1281500.1281532

[10] 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

[11] 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

[12] Shigeru Owada, Frank Nielsen, Kazuo Nakazawa, and Takeo Igarashi. 2003. A Sketching Interface for Modeling the Internal Structures of 3D Shapes. In *Smart Graphics*, Andreas Butz, Antonio Krüger, and Patrick Olivier (Eds.). Springer Berlin Heidelberg, 49–57. https://doi.org/10.1007/3-540-37620-8_5

[13] Beryl Plimmer and Mark Apperley. 2004. INTERACTING with Sketched Interface Designs: An Evaluation Study *(CHI EA '04)*. ACM, New York, NY, USA, 1337–1340. https://doi.org/10.1145/985921.986058

[14] Huawei Tu, Xiangshi Ren, and Shumin Zhai. 2012. A Comparative Evaluation of Finger and Pen Stroke Gestures *(CHI '12)*. ACM, New York, NY, USA, 1287–1296. https://doi.org/10.1145/2207676.2208584

[15] 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.

[16] 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

[17] Jacob O. Wobbrock, Meredith Ringel Morris, and Andrew D. Wilson. 2009. User-defined Gestures for Surface Computing *(CHI '09)*. ACM, New York, NY, USA, 1083–1092. https://doi.org/10.1145/1518701.1518866

[18] 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, Berlin, Heidelberg, 291–300. https://doi.org/10.1007/978-3-642-02577-8_32

[19] Robert C. Zeleznik, Kenneth P. Herndon, and John F. Hughes. 2007. SKETCH: An Interface for Sketching 3D Scenes *(SIGGRAPH '07)*. ACM, New York, NY, USA, Article 19, 6 pages. https://doi.org/10.1145/1281500.1281530

[20] S. Zhai, P. O. Kristensson, C. Appert, T. H. Andersen, and X. Cao. 2012. *Foundational Issues in Touch-Surface Stroke Gesture Design: An Integrative Review.* now. https://ieeexplore.ieee.org/document/8187096