Sketching with a Purpose: Moving from Supporting Modeling to Supporting Software Engineering Activities

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

André van der Hoek
University of California, Irvine
Irvine, California
andre@ics.uci.edu

ABSTRACT

With the advent of large interactive displays and the increasing ubiquity of touch-enabled devices, digital sketching tools are becoming viable alternatives to classic whiteboards for the design and engineering of software systems. In this paper, we argue that current sketching tools still focus largely on creating sketches as artifacts (mostly, models), rather than recognizing sketches as byproducts of engineers' thought processes as they tackle a broad variety of complex engineering activities. We therefore advocate a more activity- than artifact-oriented view of sketching and propose research on a number of ways in which digital sketching can support cognitively demanding software engineering activities more directly.

CCS CONCEPTS

• **Software and its engineering → Software development techniques**; Designing software; • **Human-centered computing →** *Interaction techniques*;

KEYWORDS

Sketching, software development, tools, interaction modality

ACM Reference Format:
Matthias Book and André van der Hoek. 2018. Sketching with a Purpose: Moving from Supporting Modeling to Supporting Software Engineering Activities. In *CHASE'18: CHASE'18:IEEE/ACM 11th International Workshop on Cooperative and Human Aspects of Software , May 27, 2018, Gothenburg, Sweden.* ACM, New York, NY, USA, Article 4, 4 pages. https://doi.org/10.1145/3195836.3195854

1 INTRODUCTION

People in all roles and professions rely on sketches to reason about complex or abstract ideas, to communicate concepts, and to externalize memory [19]. Software engineers draw sketches as well to understand, discuss and evolve concepts ranging from domain requirements to software designs and implementation details [3]. Such sketches are typically fairly simple, ad-hoc and informal drawings representing an architecture, some code, a user interface, a behavior, or more. Usually, the sketches evolve with additional detail and formalism as the developers work out their thoughts.

Aside from pen and paper, classic dry-erase whiteboards are the most ubiquitous sketching tool in practice, offering space for individual or collaborative work on detailed sketches. Crucially, dry-erase whiteboards are a highly intuitive sketching tool, enabling any project stakeholder to just pick up a pen and express their ideas or revise existing sketches, regardless of their technology affinity or formal training. The medium's intuitive physicality also has its drawbacks though: Sketches cannot be easily rearranged when running out of space, they cannot be conveniently stored off-medium for later retrieval or quick focus changes, and any collaboration must be local.

These limitations motivated the development of numerous digital sketching tools, buoyed considerably by the rapid emergence of touch-enabled devices of all sizes—from mobile phones, tablets and laptops to wall-mounted and tabletop interactive displays—over the course of the last decade. Typical features of these tools (beyond drawing digital ink strokes with a finger or pen) include manipulating sketches, zooming and panning the (possibly infinite) canvas, and storing and restoring sketches. More advanced features include the management of multiple sketches, drawing in (or converting to) more formal modeling notations, and remote collaboration [10].

Upon critical inspection, however, most of today's digital sketching solutions are not much more than touch-enabled versions of drawing or modeling tools offering a level of functionality that has been around for decades. In other words, with a few exceptions (e.g. the GUI prototyping tools mentioned in Section 2), current sketching tools unfortunately offer little functionality that is explicitly geared towards helping software engineers solve the particular problems for which they turn to sketching in the first place.

We hypothesize that this is because most current sketching tools are designed to produce sketches as (more or less formal) *models*, rather than designed to explicitly support the *activities* the engineers need to perform. While the former is certainly important, we believe the latter holds even more promise and is ripe for exploration. In this paper, we suggest areas where activity- rather than model-centric sketching could be beneficial, and outline opportunities for research towards enabling this vision.

After a brief, necessarily non-exhaustive overview of the state of the art in Section 2, we first discuss how and why engineers sketch in general (Section 3), and then suggest opportunities for supporting various software engineering activities through sketching (Section 4). We identify several cross-cutting research challenges that need to be resolved in order to facilitate our vision in Section 5 and conclude in Section 6 with a brief outlook at future work.

CHASE'18, May 27, 2018, Gothenburg, Sweden

ACM ISBN 978-1-4503-5725-8/18/05…$15.00
https://doi.org/10.1145/3195836.3195854

2 STATE OF THE ART

Digital sketching has been the focus of a large body of research in both the human-computer interaction and—more recently—the software engineering communities. Numerous tools focus on emulating the classic whiteboard experience, augmenting it with features such as histories (Flatland [15]), shape recognition (InkKit [4]), or shape rearrangement (Range [11]). A number of tools provide mechanisms for organizing and navigating multiple sketches. Solutions designed specifically for software engineers include DENIM's hierarchy of a site map, storyboards and sketches [16] or Calico's grids, scraps and palettes for fast focus switching and side-by-side views [14]. Remote collaboration of stakeholders has been implemented in TEAM STORM [7] and FlexiSketch Team [21], among others. Today, it is near ubiquitous in commercial products, such as SMART Bridgit, Google Drawings, draw.io, and Microsoft Whiteboard.

Most sketching tools that are designed for software engineers fall into two classes: They strive to bridge informal and formal modeling through support for sketching in or converting to the Unified Modeling Language (e.g., Knight [6], Tahuti [8], SUMLOW [2], OctoUML [20]), or they support user interface prototyping by converting interface sketches into executable storyboards (SILK [13], Sketchify [17], Active Story Touch [9]), domain-specific languages (SketchiXML [5]), or code (JavaSketchIt [1], Freeform [18]).

3 SKETCHING, MODELING AND ACTIVITIES

While "sketching" has become almost synonymous for everything that happens at the whiteboard, we argue that it is important to carefully delineate *sketching*, from *modeling*, from the software engineering *activities* that drive these two, in order to target tool support where it is most needed.

Sketching is the interaction mechanism through which people produce visual content. Its intuitiveness and ubiquity make it the prime modality of expression for visual content on any medium—designers have been observed to "sketch" even in mid-air.

Modeling is what developers often do while sketching (besides writing, doodling or annotating other elements). Models are abstractions that help engineers to understand and communicate particular aspects of the system or domain under consideration. Notably, models created at the whiteboard are typically much less formal than those produced in desktop tools, although designers may use more formal notation elements when needed.

Software engineering *activities* are what drive stakeholders to the whiteboard (or any sketching medium): activities that are cognitively challenging because they require domain understanding, knowledge transfer, or creative design—for example, a feature to be described, a set of architectural trade-offs to be considered, a data structure to be refactored, and more.

Using this explicit separation of sketching, modeling, and activities leads us to the key insight of this paper: The design of sketching tools has been driven more by a focus on facilitating modeling than a focus on facilitating the activities that lead to the modeling. While models are important, they are ultimately subservient to the reason people sketch: to solve problems. Sketching, and the modeling that takes place as part of it, should therefore be regarded as an *interaction modality* for tools that first and foremost should be designed to support the activities in which developers engage.

This insight is further supported by the observation that many sketches are neither evolving into more formal models, nor ever intended to become persistent artifacts. Rather, they are created, refined and referred to only as long as required to support a complex train of thought while solving a particular problem. The actual problem-solving still occurs in people's minds and conversations; sketches and the models contained in them are just the visible "tip of the cognitive iceberg" of reasoning about the domain or system.

Nevertheless, we would like to emphasize the importance of the prior work that went into making digital sketching tools as smooth and intuitive as sketching on a dry-erase board, enhancing the experience with manipulation, persistence and collaboration features, and striving to transfer this experience to more formal notations. Given that natural sketching occurs almost subconsciously as an extension of people's thought processes, any tool that relies on sketching as an input modality must provide a user experience that is as smooth and intuitive as possible, since any input inefficiency or inconvenience could disturb users' flow of thought and lead them to abandon the tool. This makes the prior work on intuitive digital sketching an essential foundation for the research on more activity-oriented support that we propose in the following section.

4 SKETCH-BASED ACTIVITY SUPPORT

Software engineers create sketches and models as part of a multitude of activities (e.g., stakeholder identification, domain engineering, programming, testing). Rather than discussing all these activities individually, we take a more general perspective by examining engineers' core interactions with information and artifacts (e.g., models, code, documents) that lie at the heart of all engineering activities. We consider three core interactions in which software engineers engage while sketching:

- *Sensemaking* of information and artifacts;
- *Annotating* artifacts to externalize implicit information; and
- *Manipulating* artifacts to progress in an activity.

In the following subsections, we first describe why the respective core interactions are common cognitive challenges in software projects. We then use examples of concrete software engineering activities to suggest novel ways in which sketching can be employed to support those activities. In Section 5, we then highlight research that needs to be done to enable this vision (e.g., development of suitable micro-notations, identification of suitable base layers etc.).

4.1 Sensemaking

Understanding software artifacts and their relationships becomes progressively more challenging as systems grow and mature (and in the process often deteriorate). Approaches to mediate this have been the focus of the fields of program comprehension and software visualization for a long time, but sensemaking is a more fundamental need permeating all aspects of software development—including making sense, for instance, of a test case and why it is not working, or of a DevOps strategy that needs improvement to address scalability. This is why sensemaking is an activity that often takes place at the whiteboard or on paper in the form of informal sketches representing the aspect of the system under consideration.

The question that arises is whether sketching tools can provide more explicit support for sensemaking than just facilitating the

drawing. We believe progress can be made along three dimensions: (1) populating sketches, (2) representing mental simulations, and (3) providing proactive guidance.

First, in terms of populating sketches, today sensemaking relies on the sketches drawn by developers themselves. These sketches are typically fairly minimal, but nonetheless still must be constructed—often from memory, involving some abstraction, and at times containing mistakes as a result. What if tools were available that could assist developers in creating sketches from existing artifacts, at just the right level of detail and abstraction, that are further explorable as desired? Existing tools that take in code and produce a sketchy UML diagram are one example, but what would an automatically created sketch of an application's security or scalability concerns look like, or its dynamic execution engine? The challenge lies in identifying interactive visualizations at a suitable abstraction level that can be automatically, or semi-automatically, provided.

Second, current sketching tools do little to help developers interpret what they see, leading to the typical "waving of hands in front of a sketch" in order to mentally simulate processes such as how a test case exercises some lines of code, how a message flows through the system, or how a requirement is addressed in various parts of the software. We believe there is room to assist developers in performing these kinds of analyses, for instance by allowing them to initiate sequences of actions in the model represented by the sketch that then visually trace through the sketch with non-permanent ink to show a slowly vanishing trail highlighting the sequence of interest. The challenge is to identify the kinds of mental simulations developers perform, and how these can be supported by suitable sketch-based input and output.

Finally, we believe interactive sketches could be augmented to provide proactive guidance to engineers, suggesting analysis steps or considerations that may be necessary to make sense of a certain situation. The tool could bring up related sketches with relevant parts highlighted automatically, it could subtly vary the intensity of parts of the sketch to guide the developer's eyes towards parts that the tool deems most pertinent (e.g. highlighting possible design flaws), or it could show where the latest updates were made to a sketch during the last session. Such proactive guidance requires requires considerable additional research, not least in anticipating user's intentions, and ensuring the tool neither just states the obvious nor leads users down wrong paths.

4.2 Annotating

Often, the most important knowledge about a system or aspect thereof may not be explicitly visible in a certain artifact (e.g., a model), but only exist implicitly in the minds of a few stakeholders. Consider, e.g., the knowledge that some part of a system represents a trade-off between different architectural drivers, or that a part of a design is still considered tenuous when it comes to scalability.

Allowing annotations to be made on software artifacts can help to make such knowledge visible and actionable. Indeed, freehand annotations have long been a part of sketching practice and can be found on almost any sketch. The question arises whether sketching tools can facilitate more powerful forms of annotations.

One example is provided by the Augmented Interaction Room (AugIR [12]), which offers a set of iconic annotations representing categories (such as value to stakeholders, risk drivers, uncertainty) that can be attached to any part of sketches created during domain analysis and high-level architecture design. The annotations help to elicit the concerns of all stakeholders, and provide a vocabulary for recording and resolving them.

Further research opportunities lie in identifying other activities where this approach may be useful. For instance, when discussing a particular bug using the aid of a sketch of the code, might it be possible to annotate parts of the code with icons representing levels of certainty regarding whether the code is correct or not? Or, when refactoring some particularly complex portion of the architecture, might it be possible to first annotate the existing architecture with icons that represent which components belong to which future, post-refactoring parts, to perform and examine the proposed refactoring hypothetically without actually implementing it yet?

The challenge in these approaches lies in figuring out what salient knowledge takes a prevailing role in the thought process of the engineers engaging with the artifacts under consideration, and understanding whether an underlying largely shared set of considerations exists that can be turned into a vocabulary of icons that then can be used to digitally annotate sketches.

4.3 Manipulating

Underneath many software engineering activities is the need to produce or manipulate some artifact, whether it is code, a specification, a test case, a user interface mockup or the like. The cognitive effort that often drives engineers to the whiteboard when creating and manipulating these artifacts stems from navigating the various layers of abstraction between requirements and implementation, keeping track of relations and dependencies between artifacts, and coming up with correct and efficient engineering solutions.

Moving to the whiteboard, however, comes at a cost: the disconnect between what is being sketched and the actual artifacts being represented. Whatever changes are made on the whiteboard must then be repeated in the artifacts themselves. What if, instead, we reconceptualized the sketch as being connected to the artifacts it represents? Current sketching tools that allow users to produce UML diagrams or wireframes already do so, but what if we pushed that idea further to other artifacts?

As one example, what if a test case could be specified in a sketchy manner, with automatic generation of appropriate test drivers, stubs, skeletons, and so on? What if, even more challenging, that test case could be derived from drawing over a piece of code to indicate the path the developer wants the test case to take? As another example, what if one could merge two variants of the same code base simply by drawing on the code being shown on a digital whiteboard? What if, instead of being shown the actual code, the developer would see a graphical representation that is shorthand for the code it represents, but is easier to grasp and manipulate by the developer (and can be unfolded as necessary to the full level of detail of the actual code)? As a final example, moving away from code-centric artifacts, what if one could sketch domain models and have those be translated into useful ontologies and knowledge to be available later?

Clearly, moving into these directions is not without its challenges, the most significant one of which is that it will likely be necessary to develop dedicated interactions that each support a specific purpose

(e.g., a set of sketching gestures for refactoring, a set of sketching gestures for execution tracing). It will be important to strike a balance between finding sufficiently precise means of expression and preserving the fluid, informal nature of sketching, since losing that natural feeling will likely result in unused features.

5 RESEARCH OPPORTUNITIES

Besides research on the specific architectural and user experience aspects of the activities suggested above, several larger, overarching research questions need to be answered that have not been a focus of previous work on more model-oriented sketching tools.

First and foremost is the question of suitable sketch-based representations of the artifacts (e.g., a test case, domain model) or properties thereof (e.g., scalability, correctness) that developers need to work with, together with suitable sketch-based expressions of the activities (i.e., forms of sensemaking, annotating, manipulating) that developers seek to perform. As sketching becomes an interaction modality for special purposes, it will likely be necessary to develop micro-notations that represent underlying artifacts in a sketchy, abbreviated, yet easily digested manner rather than in the more traditional formal notations to which most sketch tools simply apply a recognition engine. The design of these micro-notations and the interactions that go along with them represents perhaps the greatest research challenge as it enters uncharted territory to date. Great care also needs to be taken in this to ensure that the novel sketching techniques remain lightweight and intuitive, just as an engineer's informal sketch on a whiteboard would be.

Second, given that sketches as an input modality are almost by definition more transient than sketches as artifacts are, it is also likely that the boundary between sketch-based input and gesture-based input becomes fluid. Sketching tools can likely benefit from the prior work on gesture description and recognition performed by the HCI community, and sketchy micro-notations might serve as mnemonics that make even quite complex gestures memorable and thus viable as intuitive input commands.

Third, we need to consider what kinds of base layers are suitable to sketch on (existing models, auto-generated sketches, code or sketchy representations thereof, interface mockups, running applications, etc.). This includes both the question of the most intuitive starting points for different sensemaking, annotation and manipulation activities, and suitable tie-in points with the micro-notations mentioned above. Another challenge is that the sketching tool needs to be aware not just of the semantics of the users' sketches, but also of the elements of the underlying base layer, in order to correctly interpret the users' intentions.

Finally, much of what we have discussed thus far re-examines the role and architecture of sketching tools: They will likely morph into 'just' being an interaction layer that may well be integrated into other modeling and development tools, rather than remaining stand-alone. This, too, brings with it challenges in terms of how to preserve the nature of sketching in such feature-rich environments.

6 CONCLUSION

While one might think that with the current set of academic and industrial sketching tools available, a ceiling has been reached as to what is possible, we have argued in this position paper that this is only so if one continues to focus on *modeling* as the objective. As soon as one recognizes that it is the *activities* of software engineers that must be supported, a rich set of opportunities arises together with commensurate challenges that must be overcome. We hope that this position paper inspires a next generation of sketching tools to emerge that not only tackle the issues we outlined, but go further beyond them, and we invite our colleagues to join us in envisioning what this next generation might look like.

REFERENCES

[1] Anabela Caetano, Neri Goulart, Manuel Fonseca, and Joaquim Jorge. 2002. JavaSketchIt: Issues in sketching the look of user interfaces. In *AAAI Spring Symposium on Sketch Understanding*. 9–14.

[2] Qi Chen, John Grundy, and John Hosking. 2008. SUMLOW: Early design-stage sketching of UML diagrams on an E-whiteboard. *Software: Practice and Experience* 38, 9 (2008), 961–994.

[3] Mauro Cherubini, Gina Venolia, Rob DeLine, and Andrew J Ko. 2007. Let's go to the whiteboard: How and why software developers use drawings. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. ACM, 557–566.

[4] 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*. ACM, 29–30.

[5] Adrien Coyette, Jean Vanderdonckt, and Quentin Limbourg. 2006. SketchiXML: A design tool for informal user interface rapid prototyping. In *Intl. Workshop on Rapid Integration of Software Engineering Techniques*. Springer, 160–176.

[6] Christian Heide Damm, Klaus Marius Hansen, and Michael Thomsen. 2000. Tool support for cooperative object-oriented design: Gesture based modelling on an electronic whiteboard. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. ACM, 518–525.

[7] Joshua Hailpern, Erik Hinterbichler, Caryn Leppert, Damon 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 ACM SIGCHI Conference on Creativity & Cognition*. ACM, 193–202.

[8] Tracy Hammond and Randall Davis. 2006. Tahuti: A geometrical sketch recognition system for UML class diagrams. In *ACM SIGGRAPH 2006 Courses*. ACM, 25.

[9] Ali Hosseini-Khayat, Teddy Seyed, Chris Burns, and Frank Maurer. 2011. Low-fidelity prototyping of gesture-based applications. In *Proceedings of the 3rd ACM SIGCHI Symposium on Engineering Interactive Computing Systems*. ACM, 289–294.

[10] Gabe Johnson, Mark D Gross, Jason Hong, Ellen Yi-Luen Do, et al. 2009. Computational support for sketching in design: A review. *Foundations and Trends in Human-Computer Interaction* 2, 1 (2009), 1–93.

[11] Wendy Ju, Brian A Lee, and Scott R Klemmer. 2008. Range: Exploring implicit interaction through electronic whiteboard design. In *Proceedings of the 2008 ACM Conference on Computer Supported Cooperative Work*. ACM, 17–26.

[12] Markus Kleffmann, Matthias Book, and Volker Gruhn. 2014. Supporting collaboration of heterogeneous teams in an augmented team room. In *Proceedings of the 6th International Workshop on Social Software Engineering*. ACM, 9–16.

[13] James A Landay and Brad A Myers. 2001. Sketching interfaces: Toward more human interface design. *Computer* 34, 3 (2001), 56–64.

[14] Nicolas Mangano, Alex Baker, Mitch Dempsey, Emily Navarro, and André van der Hoek. 2010. Software design sketching with Calico. In *Proc. of the IEEE/ACM Intl. Conf. on Automated Software Engineering*. ACM, 23–32.

[15] 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*. ACM, 346–353.

[16] Mark W Newman, James Lin, Jason I Hong, and James A Landay. 2003. DENIM: An informal web site design tool inspired by observations of practice. *Human-Computer Interaction* 18, 3 (2003), 259–324.

[17] Željko Obrenovic and Jean-Bernard Martens. 2011. Sketching interactive systems with Sketchify. *ACM Trans. Computer-Human Interaction (TOCHI)* 18, 1 (2011), 4.

[18] 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*. ACM, 1337–1340.

[19] Barbara Tversky. 2002. What do sketches say about thinking. In *2002 AAAI Spring Symposium, Sketch Understanding Workshop, Stanford University, AAAI Technical Report SS-02-08*. 148–151.

[20] Boban Vesin, Rodi Jolak, and Michel R. V. Chaudron. 2017. OctoUML: An environment for exploratory and collaborative software design. In *Proc. of the 39th Intl. Conf. on Software Engineering (ICSE 2017) – Companion Volume*. ACM, 7–10.

[21] 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)*, Vol. 2. IEEE, 685–688.